IA pour les Jeux
Licence Informatique et Vidéoludisme

Datalog

Nicolas Jouandeau

n@up8.edu

2022

Datalog

- langage de requêtes pour les bases de données déductives (BDD)
- première spécification en 1977 (H. Gallaire et J. Minker)
- programmation déclarative = résoudre par déclaration (des faits, de règles, de données, ...)
  - ensemble de faits et de règles = programme datalog
  - possibilités de requêtes sur un programme
  - prédicats intentionnels = spécifier les propriétés communes à plusieurs éléments
  - prédicats extensionnels = énumérer tous les éléments
- programme Datalog interprété dans DrRacket
  `https://docs.racket-lang.org/datalog/index.html`
- en Datalog en forme parenthésée préfixée avec DrRacket
  un programme Datalog avec une syntaxe proche de Racket
- en combinant Datalog en forme parenthésée préfixée et Racket
  des assertions en Datalog et des fonctions en Racket
  (fonctions Racket qui utilisent des résultats de requêtes Datalog)

Datalog

- une variable est une séquence (de minuscules, majuscules, digits et underscore) commençant par une majuscule
- un booleen est vrai avec `true` et est faux avec `false`
- le test d'égalité avec = et le test d'inégalité avec `!=`
- l'évaluation avec `:-`
  (ce qui permet d'accéder à tous les opérateurs aritmétiques de Racket) dont les opérateurs préfixes d'arité 2
  - la soustraction avec − et l'addition avec +
  - la multiplication avec * et la division avec /
  - les opérateurs de comparaison < et > (qui retournent un booléen)

  (et également les opérateurs d'arité 1 classiques `sqrt`, `log` ...)
- ⚠ les listes ne sont pas supportées (sous-ensemble de Prolog)
- une réponse fausse (en Prolog) est une absence de réponse (en Datalog)
- les clauses se terminent par un .
- les questions se terminent par un `?`

premier exemple, avec 3 faits et 2 règles

```
#lang datalog
f(1,1).
g(1,2).
g(1,3).
h(X,Y):- f(X,Y).
h(X,Y):- g(X,Y).
```

exécution

```
> f(1,X)?
f(1, 1).
> g(X,2)?
g(1, 2).
> f(2,X)?

>
```

exécution

```
> h(1,X)?
h(1, 1).
h(1, 2).
h(1, 3).
```

deuxième exemple

```
#lang datalog
a(X,Y):- Y:- +(X,1).
b(X,Y):- Y:- -(X,1).
c(X,Y):- Y:- *(X,2).
d(X,Y):- Y:- /(X,2).
e(X,Y):- d(X,Y1), Y2:- >(Y1,1), Y2=true, Y=1.
f(X,Y):- Y:- log(X).
g(X,Y):- Y:- sqrt(X).
h(X,Y):- Y:- expt(X,3).
```

exécution

```
> a(1,X)?
a(1, 2).
> b(1,X)?
b(1, 0).
> c(1,X)?
c(1, 2).
> d(10,X)?
d(10, 5).
>
```

exécution

```
> e(10,X)?
e(10, 1).
> e(1,X)?

> f(10,X)?
f(10, 2.302585092994046).
> g(10,X)?
g(10, 3.162277660168379 5).
> h(10,X)?
h(10, 1000).
>
```

poser les questions dans le programme Datalog

```
#lang datalog
a(X,Y):- Y:- +(X,1).
b(X,Y):- Y:- -(X,1).
c(X,Y):- Y:- *(X,2).
d(X,Y):- Y:- /(X,2).
e(X,Y):- d(X,Y1), Y2:- >(Y1,1), Y2=true, Y=1.
a(1,X)?
b(1,X)?
c(1,X)?
d(10,X)?
e(10,X)?
e(1,X)?
e(11,X)?
```

exécution

```
a(1, 2).
b(1, 0).
c(1, 2).
d(10, 5).
e(10, 1).

e(11, 1).
```

exemple de fonctions classiques avec évaluation

```
#lang datalog
f1(X,Y,Z):- Z:- expt(X,Y).
f2(X,Y):- Y:- sqrt(X).
f3(X,Y,Z):- Z:- quotient(X,Y).
f4(X,Y,Z):- Z:- remainder(X,Y).
f5(X,Y,Z):- Z:- modulo(X,Y).
f6(X,Y):- Y:- add1(X).
f7(X,Y):- Y:- sub1(X).
f8(X,Y,Z):- Z:- max(X,Y).
f9(X,Y,Z):- Z:- min(X,Y).
f10(X,Y):- Y:- random(X).
f1(3,3,X)?
f2(3,X)?
f3(13,3,X)?
f4(13,3,X)?
f5(13,3,X)?
f6(10,X)?
f7(10,X)?
f8(10,11,X)?
f9(10,11,X)?
f10(10,X)?
```

exécution

```
f1(3, 3, 27).
f2(3, 1.732050807568772).
f3(13, 3, 4).
f4(13, 3, 1).
f5(13, 3, 1).
f6(10, 11).
f7(10, 9).
f8(10, 11, 11).
f9(10, 11, 10).
f10(10, 6).
```

définir un prédicat `is_even/2` en version récursive

- le prototype est `is_even(X,Y)` dans lequel
  - `X` est le nombre testé
  - pour `X`= 0, `Y` est vrai
  - pour `X`= 1, `Y` est faux
  - pour `X`> 1, `X` est défini en fonction de `X`−2

```
#lang datalog
is_even(0,true).
is_even(1,false).
is_even(X,Y):- X != 0,X != 1, X1:- -(X,2),is_even(X1,Y).
is_even(100,X)?
is_even(101,X)?
```

exécution

```
is_even(100, #t).
is_even(101, #f).
```

solution `is_even/2` avec l'opérateur de comparaison `<`

```
#lang datalog
is_even(0,true).
is_even(1,false).
is_even(X,Y):- C:- >(X,1), C=true, X1:- -(X,2),is_even(X1,Y).
is_even(100,X)?
is_even(101,X)?
```

solution `is_even/2` avec `modulo`

```
#lang datalog
is_even(X,Y):- X1:- modulo(X,2),X1=0,Y=true.
is_even(X,Y):- X1:- modulo(X,2),X1=1,Y=false.
is_even(100,X)?
is_even(101,X)?
```

exécution

```
is_even(100, #t).
is_even(101, #f).
```

définir un prédicat `integer_sum/2`

- le prototype est `integer_sum(X,Y)` dans lequel
  - `Y` est la somme des valeurs de `1` à `X`

```
#lang datalog
...
...
integer_sum(10,X)?
integer_sum(100,X)?
```

exécution

```
integer_sum(10, 55).
integer_sum(100, 5050).
```

définir un prédicat `integer_sum/2`

- le prototype est `integer_sum(X,Y)` dans lequel
  - `Y` est la somme des valeurs de `1` à `X`

```
#lang datalog
integer_sum(1,1).
integer_sum(X,Y):- X != 1, X1:- -(X,1),
  integer_sum(X1,R1), Y:- +(X,R1).
integer_sum(10,X)?
integer_sum(100,X)?
```

exécution

```
integer_sum(10, 55).
integer_sum(100, 5050).
```

définir un prédicat `square_sum/2`

- le prototype est `square_sum(X,Y)` dans lequel
  - `Y` est la somme des carrés des valeurs de `1` à `X`

```
#lang datalog
...
...
square_sum(10,X)?
square_sum(100,X)?
```

exécution

```
square_sum(10, 385).
square_sum(100, 338350).
```

définir un prédicat `square_sum/2`

- le prototype est `square_sum(X,Y)` dans lequel
  - `Y` est la somme des carrés des valeurs de `1` à `X`

```
#lang datalog
square_sum(1,1).
square_sum(X,Y):- X != 1, X1:- -(X,1),
  square_sum(X1,R1), X2:- *(X,X), Y:- +(X2,R1).
square_sum(10,X)?
square_sum(100,X)?
```

exécution

```
square_sum(10, 385).
square_sum(100, 338350).
```

définir un prédicat correspondant à $1/log(x)^2$

- le prototype est `f(X,Y)` dans lequel
  - pour `X` > 0, `Y` $= 1/log(X)^2$

```
#lang datalog
...
...
f1(10,X)?
f1(-10,X)?
```

exécution

```
f1(10, 0.1886116970116138 7).
```

définir un prédicat correspondant à $1/log(x)^2$

- le prototype est `f(X,Y)` dans lequel
  - pour `X` > 0, `Y` $= 1/log(X)^2$

```
#lang datalog
f(1,1).
f(X,Y):- P:- >(X,0),P = true,
  Y1 :- log(X), Y2:- *(Y1,Y1), Y:- /(1,Y2).
f(10,X)?
f(-10,X)?
```

exécution

```
f(10, 0.1886116970116138 7).
```

suite de Fibonacci

- le prototype est `fibo(X,Y)` dans lequel
  - `Y` est le `X`$^{\text{ième}}$ terme de la suite de Fibonacci

```
#lang datalog
fibo(0, 0).
fibo(1, 1).
...
fibo(30, F)?
```

exécution

```
fibo(30, 832040).
```

suite de Fibonacci

- le prototype est `fibo(X,Y)` dans lequel
  - `Y` est le $X^{\text{ième}}$ terme de la suite de Fibonacci

```
#lang datalog
fibo(0, 0).
fibo(1, 1).
fibo(N, F):- N != 1, N != 0, N1:- -(N, 1), N2 :- -(N, 2),
  fibo(N1, F1), fibo(N2, F2), F:- +(F1, F2).
fibo(30, F)?
```

exécution

```
fibo(30, 832040).
```

Datalog en forme parenthésée préfixée

- ajouter des assertions (faits ou règles) avec `(! ... )`
- retirer des assertions avec `(~ ... )`
- poser des questions avec `(? ... )`
- notation parenthésée préfixée avec espace comme séparateur

datalog

```
#lang datalog
f(1,1).
g(1,2).
g(1,3).
h(X,Y):- f(X,Y).
h(X,Y):- g(X,Y).
f(1,X)?
g(1,X)?
```

exécution

```
f(1, 1).
g(1, 2).
g(1, 3).
```

datalog parenthésé préfixé

```
#lang datalog/sexp
(! (f 1 1))
(! (g 1 2))
(! (g 1 3))
(! (:- (h X Y) (f X Y)))
(! (:- (h X Y) (g X Y)))
(? (f 1 X))
(? (g 1 X))
```

exécution

```
f(1, 1).
g(1, 2).
g(1, 3).
```

Datalog parenthésé préfixé

- `Z :- +(X, Y)` devient `(+ X Y :- Z)`

`is_even/2` en version récursive

```
#lang datalog/sexp
(! (is_even 0 true))
(! (is_even 1 false))
(! (:- (is_even X Y)
       (!= X 0) (!= X 1) (- X 2 :- X1) (is_even X1 Y)
   ))
(? (is_even 10 Y))
(? (is_even 11 Y))
```

exécution

```
is_even(10, true).
is_even(11, false).
```

is_even/2 avec l'opérateur de comparaison >

```
#lang datalog/sexp
(! (is_even 0 true))
(! (is_even 1 false))
(! (:- (is_even X Y)
  (> X 1 :- C) (= C #t) (- X 2 :- X1) (is_even X1 Y)
))
(? (is_even 10 Y))
(? (is_even 11 Y))
```

is_even/2 utilisant modulo

```
#lang datalog/sexp
(! (:- (is_even X Y)
     (modulo X 2 :- X1) (= X1 0) (= Y true)
   ))
(! (:- (is_even X Y)
     (modulo X 2 :- X1) (= X1 1) (= Y false)
   ))
(? (is_even 10 Y))
(? (is_even 11 Y))
```

exécution

```
is_even(10, true).
is_even(11, false).
```

integer_sum/2

```
#lang datalog/sexp
(! (integer_sum 1 1))
(! (:- (integer_sum X Y)
     (!= X 1) (- X 1 :- X1) (integer_sum X1 R1) (+ X R1 :- Y)
   ))
(? (integer_sum 10 X))
(? (integer_sum 100 X))
```

exécution

```
integer_sum(10, 55).
integer_sum(100, 5050).
```

suite de Fibonacci

```
#lang datalog/sexp
(! (fibo 0 0))
(! (fibo 1 1))
(! (:- (fibo X Y)
     (!= X 0) (!= X 1) (- X 1 :- X1) (- X 2 :- X2)
     (fibo X1 R1) (fibo X2 R2) (+ R1 R2 :- Y)
   ))
(? (fibo 30 X))
```

exécution

```
fibo(30, 832040).
```

retour des listes ☺

- `(list X)` crée une liste avec une valeur `X`
- `(cons X Y)` ajoute un élément `X` en tête quand `Y` est une liste et crée un tuple `(X . Y)` quand `Y` est un élément

exemple avec `list` et `cons`

```
#lang datalog/sexp
(! (f1 (cons 1 2)))
(! (:- (f2 X Y Z) (cons X Y :- Z)))
(! (f3 0 (list 0)))
(! (:- (f3 X L)
  (> X 0 :- A)(= A #t)(- X 1 :- X1)(f3 X1 L1)(cons X L1 :- L)))
(? (f1 X))
(? (f2 1 2 X))
(? (f2 3 (list 4) X))
(? (f2 3 '(4) X))
(? (f3 10 X))
```

exécution

```
f1('(1 . 2)).
f2(1, 2, '(1 . 2)).
f2(3, '(4), '(3 4)).
f2(3, '(4), '(3 4)).
f3(10, '(10 9 8 7 6 5 4 3 2 1 0)).
```

signification des faits et des règles

- définir un prédicat `(all-1 X Y V)` avec `V` vrai ssi `X` et `X` sont égaux à `-1`

prédicat `all-1`

```
#lang datalog/sexp
;;; on déclare le fait suivant
(! (:- (all-1 -1 -1 #t)))

;;; V est faux si la première valeur est différente de -1,
;;; et quelquesoit la deuxième valeur
(! (:- (all-1 A B #f) (!= A -1) (= B B)))

;;; V est faux si la deuxième valeur est différente de -1,
;;; et quelquesoit la première valeur
(! (:- (all-1 A B #f) (= A A) (!= B -1)))

(? (all-1 -1 -1 X))
(? (all-1 2 -1 X))
(? (all-1 -1 23 X))
(? (all-1 17 32 X))
```

exécution

```
all-1(-1, -1, #t).
all-1(2, -1, #f).
all-1(-1, 23, #f).
all-1(17, 32, #f).
```

utiliser Datalog parenthésé préfixé dans un programme Racket

- utiliser `(define AAA (make-theory))` pour définir une BDD `AAA`
- utiliser `(datalog! AAA (! ...))` pour ajouter des assertions
- utiliser `(datalog AAA (? ...))` pour poser des questions

programme Racket avec base de données déductive

```
#lang racket
(require datalog)
(define fun (make-theory))
(datalog! fun
 (! (f 1 1))
 (! (g 1 2))
 (! (g 1 3))
 (! (:- (h X Y) (f X Y)))
 (! (:- (h X Y) (g X Y)))
)
(datalog fun (? (f 1 Y)))
(datalog fun (? (h 1 Y)))
```

exécution

```
'(#hasheq((Y . 1)))
'(#hasheq((Y . 1)) #hasheq((Y . 2)) #hasheq((Y . 3)))
```

utiliser Datalog parenthésé préfixée dans un programme Racket (suite)

- exemple d'une fonction `nb-h` qui compte le nombre de `Y` vérifiant `(h X Y)` pour une valeur de `X` fixée
- exemple d'une fonction `is-h` qui retourne vrai si `X` vérifie `h` et qui retourne faux sinon

programme Racket

```
#lang racket
(require datalog)
(define fun (make-theory))
(datalog! fun
 (! (f 1 1))
 (! (g 1 2))
 (! (g 1 3))
 (! (:- (h X Y) (f X Y)))
 (! (:- (h X Y) (g X Y)))
)
(define (nb-h X) (length (datalog fun (? (h X Y)))))
(define (is-h X Y) (not (empty? (datalog fun (? (h X Y))))))
(nb-h 1)
(is-h 1 0)
(is-h 1 2)
```

exécution

```
3
#f
#t
```