Prolog — TP Machine de Turing

July 22, 2022

On pourrait se demander si *Prolog* permet de faire autant de choses qu'un langage comme *Java*. Par faire "autant de choses", on entend calculer tout ce qui peut se calculer sur un ordinateur en écrivant les programmes en *Java*. En fait, il s'avère que tout ce qu'on peut calculer sur un ordinateur peut l'être par une machine de *Turing* (et vice-versa). Du coup, si on arrive à simuler une machine de *Turing* en *Prolog*, on aura montré que *Prolog* permet lui aussi de calculer tout ce qu'il est possible de calculer sur un ordinateur.

Je vous invite à regarder la page *Wikipédia* suivante : `https://fr.wikipedia.org/wiki/Machine_de_Turing` (jusqu'à l'exemple) qui résume le fonctionnement d'une machine de *Turing*. On reprendra l'exemple de programme de cet article qui se traduira dans notre implémentation par le terme suivant :

```
program(
           start,
           [stop],
           [delta(start, ' ', ' ', right, stop),
            delta(start, 1, ' ', right, s2),
            delta(s2, 1, 1, right, s2),
            delta(s2, ' ', ' ', right, s3),
            delta(s3, 1, 1, right, s3),
            delta(s3, ' ', 1, left, s4),
            delta(s4, 1, 1, left, s4),
            delta(s4, ' ', ' ', left, s5),
            delta(s5, 1, 1, left, s5),
            delta(s5, ' ', 1, right, start)]
      )
```

Notons que nous représentons le symbole blanc par `' '`.

Les prédicats suivants permettent d'accéder aux différents éléments du programme :

```
%initial_state(+, -): program * state
initial_state(program(InitialState, _, _), InitialState).

%final_states(+, -): program * state list
final_states(program(_, FinalStates, _), FinalStates).

%transitions(+, -): program * delta list
transitions(program(_, _, Deltas), Deltas).
```

Question 1 : écrire le prédicat suivant qui permet d'effectuer une transition :

```
%next(+, +, +, -, -, -): program * state * symbol * symbol * direction (left or right) * state
%next(Program, State0, Symbol0, Symbol1, Dir, State1)
```

En entrée de ce prédicat, on a le programme, l'état courant de la machine et le symbole sous la tête de lecture. On récupère le nouveau symbole, la direction de déplacement de la tête de lecture et le nouvel état.

Question 2 : on va représenter la bande par le terme :

`tape(Left, Right)`

Le symbole sous la tête de lecture est le premier élément de `Right`, ce qui est à gauche de la tête de lecture est dans `Left` et ce qui est à droite est dans le reste de `Right`. On s'assurera que les listes `Left` et `Right` ne soient jamais vides et contiennent au moins un symbole `' '`.

Écrire le prédicat :

```
%update_tape(+, +, +, -): tape * symbol * direction * tape
%update_tape(Tape, Symbol, Direction, UpdatedTape)
```

qui met à jour la bande de la machine. `Tape` est la bande de lecture, `Symbol` est le nouveau symbole à placer sous la tête de lecture, `Direction` est la direction de déplacement de la tête et `UpdatedTape` est la nouvelle bande.

Question 3 : écrire le prédicat :

```
%run_turing_machine(+, +, -, -): program * symbol list * symbol list * state
%run_turing_machine(Program, Input, Output, FinalState)
```

qui exécute le programme `program` sur l'entrée `Input` et rend la sortie `Output` (qui correspondra au contenu de la bande en fin d'exécution) et l'état final `FinalState`.

Question 4 : on voudrait pouvoir visualiser les différentes étapes de calculs effectuées par la machine de *Turing*. On veut récupérer une liste de couples `(State, Tape)` où `State` représente un état de la machine et `Tape` est le contenu de la bande. Vous pourrez ensuite créer un fichier de type *MetaPost* grâce au prédicat suivant qui vous est fourni :

```
%dump_to_mpost(+, +): string * (state * tape) list
%dump_to_mpost(Filename, Dump)
```

Ce prédicat permet d'écrire dans le fichier `Filename` une représentation des différentes étapes de l'exécution de la machine contenues dans la liste `Dump`. Une fois le fichier `Filename` généré, il vous faudra taper dans un shell :

```
> mpost Filename
> epstopdf Filename.1
```

Vous obtiendrez alors un fichier *pdf* de nom `Filename` qui vous permettra de visualiser les différentes étapes de l'exécution de votre programme.

Réécrire le prédicat `run_turing_machine` pour qu'il produise une liste représentant les différentes étapes de l'exécution de la machine :

```
%run_turing_machine(+, +, -, -, -):
%program * symbol list * symbol list * state * (state * tape) list
%run_turing_machine(Program, Input, Output, FinalState, Dump)
```

Question 5 (subsidiaire) : il est assez difficile d'écrire un programme dans le formalisme des machines de *Turing*. On va donc essayer, pour de petits programmes, de les faire écrire par *Prolog*. On vous donne les prédicats suivants :

```
%make_pairs(+, +, -): 'a list * 'a list * ('a * 'a) list
%make_pairs(L1, L2, PairList)

%complete_list(+, +, +, +, -):
%(state * symbol) list * symbol list * direction list * state list * delta list
%complete_list(StateSymbolList, SymbolList, DirectionList, StateList, DeltaList)
```

Le prédicat `make_pairs` permet de créer une liste contenant le produit cartésien des listes `L1` et `L2`. Par exemple :

```
?- make_pairs([s1, s2, s3], [' ', 1], Res).
Res = [(s1, ' '), (s1, 1), (s2, ' '), (s2, 1), (s3, ' '), (s3, 1)]
Yes
```

Le prédicat `complete_list` va réussir autant de fois qu'il est possible de compléter les paires contenues dans la liste `StateSymbolList` en utilisant les symboles de `SymbolList`, les directions de `DirectionList` et les états de `StateList` pour obtenir une liste de transitions qui permettront d'écrire un programme. Par exemple :

```
?- complete_list([(s1, ' '), (s1, 1)], [' ', 1], [left, right], [init, s1, stop], Res).
Res = [delta(s1, ' ', ' ', left, init), delta(s1, 1, ' ', left, init)]
Yes
Res = [delta(s1, ' ', ' ', left, init), delta(s1, 1, ' ', left, s1)]
Yes
Res = [delta(s1, ' ', ' ', left, init), delta(s1, 1, ' ', left, stop)]
Yes
Res = [delta(s1, ' ', ' ', left, init), delta(s1, 1, ' ', right, init)]
Yes
...
```

Vous allez écrire un prédicat qui réussit autant de fois qu'il y a de programmes possibles. S'il y a deux états non terminaux `init` et `s1` et deux symboles `' '` et `1` par exemple, le programme comportera 4 transitions (celles correspondantes à `(init, ' ')`, `(init, 1)`, `(s1, ' ')` et `(s1, 1)`). Donc, s'il y a n états non terminaux et m symboles, le programme comportera $n \times m$ transitions. Écrire le prédicat suivant :

```
%all_programs(+, +, +, +, -): state * state list * symbol list * state list * program
%all_programs(Init, StateList, SymbolList, FinalStateList, Program)
```

où `Init` est l'état initial de la machine, `StateList` correspond aux états non terminaux sans l'état `Init`, `SymbolList` est la liste de symboles, et `FinalStateList` est la liste des états terminaux. Par exemple :

```
?- all_programs(init, [s2, s3], [' ', 1], [left, right], [stop], Program).
Program = program(init, [stop],
[delta(init, ' ', ' ', left, init),
 delta(init, 1, ' ', left, init),
 delta(s2, ' ', ' ', left, init),
 delta(s2, 1, ' ', left, init),
 delta(s3, ' ', ' ', left, init),
 delta(s3, 1, ' ', left, init)])
Yes
...
```

On va maintenant essayer de trouver un programme permettant de calculer la valeur d'une fonction pour une entrée donnée. Cette fonction s'appelle la fonction du *castor affairé*. Je vous invite à lire une partie de l'article suivant : `https://fr.wikipedia.org/wiki/Castor_affair%C3%A9`. Nous allons essayer de faire découvrir à *Prolog* un programme permettant de produire la valeur de cette fonction pour $n = 2$ et $n = 3$.

Comme parmi les programmes que l'on va générer certains vont faire boucler indéfiniment la machine de *Turing*, il va nous falloir modifier le prédicat `run_turing_machine` pour qu'il prenne en paramètre le nombre d'étapes maximum lors de l'exécution de la machine de *Turing* pour le programme donné. Écrire le prédicat :

```
%run_turing_machine(+, +, -, -, -, +, +):
%program * symbol list * symbol list * state * (state * tape) list * int
%run_turing_machine(Program, Input, Output, FinalState, Dump, NbMaxIter)
```

Vous allez maintenant pouvoir écrire le prédicat suivant :

```
%find_busy_beaver(+, +, +, +, +, +, -):
%state * state list * state list * symbol list * int * int * program
%find_busy_beaver(Init, States, FinalStates, Symbols, NbMaxIter, BusyBeaverNumber, Program)
```

où `Init` est l'état initial de la machine, `States` est l'ensemble des états non terminaux sans l'état `init`, `FinalStates` est l'ensemble des états terminaux, `NbMaxIter` est le nombre maximum de pas d'exécution d'une machine, `BusyBeaverNumber` est la valeur de la fonction du castor affairé et `Program` est un programme correspondant au castor affairé. En faisant l'appel suivant :

```
?- find_busy_beaver(init, [s1], [stop], [' ', 1], 6, 4, Program).
```

vous obtiendrez les programmes correspondant au castor affairé pour $n = 2$. Notez que normalement on ne devrait pas connaître le nombre d'itérations maximal et la valeur n à l'avance, mais ce n'est pas trop difficile d'adapter pour pouvoir s'en passer.