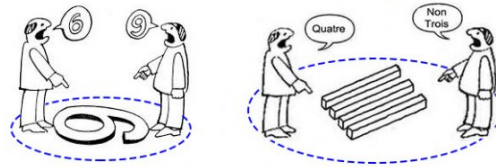


Récursivité



04 mars 2019

Damien Morard

Au cours de cette séance d'exercices, vous exécuterez et comprendrez les règles récursives, puis vous apprendrez à les transformer en LogicKit.

Exercice 1 : Exécuter les règles par inférence

Dans cet exercice, la seule chose dont vous avez besoin est de votre tête, avec un petit plus pour une feuille et un crayon ! Pour cette exercice, **évit**ez au **maximum** de programmer les exemples que nous verrons vu que nous le ferons dans le dernier exercice. De plus le but de cet exercice est de se détacher du code pour mieux comprendre comment fonctionne nos règles récursives. Nous allons inverser la manière de procéder, plutôt que de vous donner un problème et de directement l'implémenter, vous allez avoir à votre disposition des faits et des règles récursives et vous devrez calculer le résultat.

Toutes les règles que vous verrez sont écrites dans un langage similaire à LogicKit mais avec une syntaxe simplifiée. *Si la syntaxe vous bloque n'hésitez pas à poser des questions !* Vous verrez plus bas une liste de règles écrite dans un langage proche de LogicKit mais avec une syntaxe simplifiée.

Pour chacune des règles ci-dessous, **exécutez** les (à la main), trouvez le résultat demandé et dites ce qu'elles font. (Vous ne saurez pas ce qu'elles font avant de les exécuter)

1. Que vaut `res` dans `.fact("op1", 2, 1, res)` ?

```
.fact("op1", 0, y, y)
.rule("op1", succ(x), y, res) {
    .fact("op1", x, succ(y), res)
}
```

2. Que retourne le fait `.fact("op2", 3, 2)` ?

```
.fact("op2", succ(x), 0),  
.rule("op2", succ(x), succ(y)) {  
    .fact("op2", x, y)  
}
```

3. Que retourne les faits :
`.fact("op3", list, 4)` ?
`.fact("op3", list, 5)` ?

```
.fact("op3", cons(x, y), x),  
.rule("op3", cons(head, tail), x) {  
    .fact("op3", tail, x)  
}  
list = cons(3, cons(1, cons(5, empty)))
```

4. Que vaut `res` dans `.fact("op4", list, 0, res)` ?

```
.fact("op4", empty, x, cons(x, empty)),  
.rule("op4", cons(head, tail), x, cons(head, res)) {  
    .fact("op4", tail, x, res)  
},  
list = cons(3, cons(1, cons(5, empty)))
```

Exercice 2 : Comprendre les règles d'inférence

Dans le premier exercice vous avez appris à manipuler les règles d'inférences. Cependant les exemples que vous avez du appliquer vous ont aidé à comprendre les opérations des règles. Nous allons cette fois-ci voir des faits et des règles, à vous de deviner que font les règles présentées. Est-ce que les opérations portent sur des `Nat`, des `List` ou bien encore quel type nous retournons ?

1. Quelle opération décrit la règle `op5` ?

```
.fact("op5", empty, 0),  
.rule("op5", cons(head, tail), succ(res)) {  
    .fact("op5", tail, res)  
}
```

2. Quelle opération décrit la règle `op6` ?

```
.fact("op6", empty, x, cons(x, empty)),  
.rule("op6", cons(head, tail), x, cons(head, res)) {  
    .fact("op6", tail, x, res)  
}
```

3. Quelle opération décrit la règle `op7` ?

```
.fact("op7", cons(x, empty), x),  
.rule("op7", cons(head, tail), x) {  
    .fact("op7", tail, x) &&  
    x > head  
},  
.rule("op7", cons(head, tail), head) {  
    .fact("op7", tail, x) &&  
    head >= x  
}
```

Exercice 3 : LogicKit ! C'est pas si sorcier !

Le but des deux exercices précédents étaient de vous familiariser avec une logique toute nouvelle ! Sans parler spécialement d'un langage en particulier, nous avons pu voir le fonctionnement des règles et comment la récursivité fonctionne.

Finalement, la programmation est un simple outil qui n'est pas forcément nécessaire pour créer vos règles et les tester sur des cas simples. Si vous arrivez à écrire vos règles sur un papier, il ne vous reste plus qu'à les traduire et c'est ce que nous allons chercher à faire ! La syntaxe utilisée précédemment est déjà assez proche de celle de LogicKit, nous allons voir quels sont les éléments indispensables pour tout faire fonctionner.

Je vous présente ci-dessous la base dont vous aurez besoin 90% du temps, que vous pouvez copier/coller pour pouvoir ensuite ajouter vos règles et les tester :

```
import LogicKit
import LogicKitBuiltins
// Initial Term
// You can add other terms if needed

let list   : Term = .var("list")
let list1  : Term = .var("list1")
let list2  : Term = .var("list2")
let head   : Term = .var("head")
let tail   : Term = .var("tail")
let head1  : Term = .var("head1")
let tail1  : Term = .var("tail1")
let head2  : Term = .var("head2")
let tail2  : Term = .var("tail2")
let x      : Term = .var("x")
let y      : Term = .var("y")
let z      : Term = .var("z")
let res    : Term = .var("res")

// KnowledgeBase is a collection which contains all the things
// we know
var kb: KnowledgeBase = [

  // Complete the facts and rules in this knowledge base
  // HERE

]

// We merge your knowledge base with builtins types in LogicKit
// Also, you can use all operation on Nat and List
kb = kb + KnowledgeBase(knowledge: (List.axioms + Nat.axioms))
// Write your examples to test your rules below !

// Two ways to print results
// If you just want to know if there exists a result:
// ask = kb.ask(.fact(...))
// print(ask.next() != nil)
// If you want to print a value of your query (CARE, NAME OF
// YOUR VARIABLES WILL BE THE SAME IN YOUR BINDING)
// for binding in ask {
//   print("My result is:", binding["res"])
//}
```

Il nous manque maintenant les opérations classiques dont nous pouvons avoir besoin pour écrire nos règles. Les builtins types intègrent toutes ces opérations, voici la liste détaillée.

```
// 0
Nat.zero
// 1
Nat.succ(Nat.zero)
// 1 -> Nat.succ(Nat.zero)
Nat.from(1)
// Nat.succ(Nat.zero) -> 1
Nat.asSwiftInt(Nat.succ(Nat.zero))

Nat.add(x,y,res)
Nat.sub(x,y,res)
Nat.mul(x,y,res)
Nat.div(x,y,res)
Nat.mod(x,y,res)
Nat.greater(x,y)
Nat.greaterOrEqual(x,y)
Nat.smaller(x,y)
Nat.smallerOrEqual(x,y)

// []
List.empty
// [1]
List.cons(Nat.from(1), List.empty)
// [1,0]
List.cons(Nat.from(1), List.cons(Nat.zero, List.empty))
// List.from([0,1]) -> List.cons(Nat.zero,
    List.cons(Nat.from(1), List.empty))
List.from(elements: [0,1].map(Nat.from))
List.count(list: list, count: res)
List.contains(list: list, element: x)
List.concat(list1, list2, res)
```

Nous avons la plupart des bases pour faire toutes les opérations des exercices précédents.

1. Reprenez toutes les opérations des exercices précédents et transformez les en des opérations valides sous LogicKit. Vérifiez par la même occasion si vous obtenez les mêmes résultats.