La réduction
Les listes infinies

Titulaire : Dony Isabelle

Année 2025-2026

La réécriture

- Un programme fonctionnel consiste en une expression E (représentant l'algorithme et les entrées).
- Cette expression E est sujette à des **règles de réécriture** : la réduction consiste en un remplacement d'une partie de programme fonctionnel par une autre partie de programme selon une règle de réécriture bien définie. Ce processus de réduction sera répété jusqu'à obtention d'une **expression irréductible** (aucune partie ne peut être réécrite).
- L'expression ainsi obtenue est appelée **forme canonique** de E et constitue la sortie du programme.

Exemple

Exemple :

```
square n = n*n
square (3+2)
```

Observons : square n = n*n

EvaluationV1 :

suite de réductions

square (3+2)

-> square 5

-> 5*5

-> 25

EvaluationV2 :

autre suite de réductions

square (3+2)

-> (3+2)*(3+2)

-> 5*(3+2)

-> 5*5

-> 25

Expression réductible

- Une **expression réductible** est une expression pour laquelle s'appliquent des règles de réduction (pour laquelle il faut faire un travail pour l'évaluer)
- À l'opposé, une expression est dans sa **forme canonique** quand elle est non réductible

Considérons

```
mult (x,y) = x * y
mult (1+2,2+3)
```

Evaluation par l'intérieur (by value)

On évalue l'expression réductible qui ne contient pas d'autres expressions réductibles.

Si plusieurs expressions réductibles ont cette même propriété, on réduit alors l'expression réductible la plus à gauche.

Par exemple :

mult (1+2 ,2+3)
-> mult (3 ,2+3)
-> mult (3 ,5)
-> 3 * 5
-> 15

- Cette stratégie d'évaluation assure que les arguments des fonctions sont évalués avant la fonction même, c'est-à-dire ils sont **passés par valeur.**

Evaluation par l'extérieur (by name) (paresseuse ou lazy)

On évalue l'expression réductible qui n'est pas contenu dans d'autres expressions réductibles

Si plusieurs expressions réductibles ont cette même propriété, on réduit alors l'expression réductible la plus à gauche.

Donc :

mult (1+2 ,2+3)
-> 1+2 * 2+3
-> 3 * 2+3
-> 3 * 5
-> 15

Factorielle

```
fac :: Integer -> Integer
fac 1 = 1
fac x = x * fac (x-1)
```

Réductions lazy pour la factorielle

fac 5
5 * fac 4
5 * (4 * fac 3)
5 * (4 * (3 * fac 2))
5 * (4 * (3 * (2 * fac 1)))
5 * (4 * (3 * (2 * 1)))
5 * (4 * (3 * 2))
5 * (4 * 6)
5 * 24
120 -- il n’y a pas vraiment de pile en fait

Exemple montrant un intérêt du lazy

```
Prelude>  omega = omega + 1
Prelude> omega // infini
Interrupted..
```

omega

-> omega +1

-> (omega+ 1) + 1

-> ((omega + 1) + 1) +1

->…

Pas de forme canonique!

Intérêt de la stratégie « call by name »

```
Prelude> fst("haskell", omega)
"haskell"
```

- Haskell utilise une stratégie « call by name »
- l'appel `fst("haskell", omega)` ne devra pas évaluer omega et ne posera donc pas de problème.

```
fst (a, b) = a
fst ( "haskell", omega) se réduit à "haskell"
```

Exemple de génération de listes infinies

- En Haskell, il est facile de générer des listes infinies
- Exemples

La fonction from

```
from :: Integer -> [Integer]
from n = n:from (n+1)
```

La fonction fib

```
fib :: [Integer]
fib = 1:2:f 1 2
where f n m = (n+m):f m (n+m)
```

Intérêt des listes infinies

Une fonction qui produit une liste infinie peut être regardée comme un processus qui génère un flux infini de données à la demande de celui qui veut consommer les données.

Réalisé grâce à l'évaluation paresseuse

Exemple: la liste se construit à la demande

```
Main> take 10 fib
[1,2,3,5,8,13,21,34,55,89]
Main> last (take 500 fib)
225591516161936330872512695036072072046011324913758190588638
6641847462773868688340501598705279696849862 6
```

Réductions et liste infinie

Focalisons-nous sur `take 3 (from 1)`

La liste

- `take :: Int -> [a] -> [a]`

```
--paramètres:  take nb xs
--Précondition: xs non vide  et nb >= la taille de xs
--Résultat: les nb premiers éléments de xs
take 0 _ = []
take nb (x:xs) = x: take (nb-1) xs
```

Réductions et liste infinie

- take :: Int -> [a] -> [a]

--paramètres: take nb xs

--Précondition: xs non vide

et nb >= la taille de xs

--Résultat: les nb premiers éléments de xs

take 0 _ = []

take nb (x:xs) = x: take (nb-1) xs

- take 3 (from 1)
- take 3 (1:from 2)
- 1: take 2 (from 2)
- 1: take 2 (2:from 3)
- 1:2: take 1 (from 3)
- 1:2: take 1 (3: from 4)
- 1:2:3 : take 0 _
- 1:2:3:[]

Pas besoin de générer from 4!

Crible de Eratosthène

- https://www.math93.com/index.php/histoire-des-maths/notions-et-theoremes/186-crible-d-eratosthene
- Raisonner pour construire cette liste infinie de nombres premiers