Rapport synthétique

Interpréteur pour le lambda-calcul

Réalisé par :

Dano Matthieu
Fernandez Matéo
Froideveaux Loïc

Pour commencer, nous avons décidé d'écrire deux types énumérés pour les variables nommées et les indices de Bruijn :

```
/* --- Structures --- */
enum Named {
  def toStr: String = this match
    case Named.Var(x) => x
    case Named.Abs(x1, x2) => s"λ$x1.${x2.toStr}"
    case Named.App(x1, x2) => s"(${x1.toStr} ${x2.toStr})"

  case Var(x: String)
  case Abs(x: String, y: Named)
  case App(x: Named, y: Named)
}

enum DeBruijn {
  def toStr: String = this match
    case DeBruijn.FreeVar(name) => ""
    case DeBruijn.BoundVar(x) => x.toString
    case DeBruijn.Abs(x, y) => s"λ ${y.toStr}"
    case DeBruijn.App(x, y) => s"(${x.toStr} ${y.toStr})"

  case FreeVar(name: String)
  case BoundVar(x: Int)
  case Abs(x: String, y: DeBruijn)
  case App(x: DeBruijn, y: DeBruijn)
}
```

Les fonctions d'affichage sont intégrées dans les énumérations pour faciliter l'écriture. Au niveau des énumérations, pour de Bruijn nous avons ajouté un cas pour les variables libres et liées. Les applications prennent un String et un de Bruijn, le String est très important pour le passage d'un de Bruijn à une expression nommée.
Ensuite nous nous sommes attaqués au passage d'un lambda-termes avec indice de Bruijn et vice-versa :

```
/* --- Functions --- */
def namedToDeBruijn(expr: Named): DeBruijn = expr match {
  case Named.Var(x) => DeBruijn.FreeVar(x)
  case Named.Abs(x, y) => aux(y, Map(x -> 0))
  case Named.App(x, y) => DeBruijn.App(aux(x, Map.empty), aux(x, Map.empty))
}

def aux(expr: Named, map: Map[String, Int]): DeBruijn = expr match {
  case Named.Var(x) => if map.contains(x) then DeBruijn.Abs(x, DeBruijn.BoundVar(map(x))) else DeBruijn.FreeVar(x)
  case Named.Abs(name, y) => DeBruijn.Abs(name, aux(y, map.map((key, value) => (key, value + 1)).updated(name, 0)))
  case Named.App(x, y) => DeBruijn.App(
    aux(x, map.map((key, value) => (key, value + 1))),
    aux(x, map.map((key, value) => (key, value + 1))))
}

def deBruijnToNamed(expr: DeBruijn): Named = expr match {
  case DeBruijn.FreeVar(x) => Named.Var(x)
  case DeBruijn.BoundVar(dist) => Named.Var(dist.toString)
  case DeBruijn.Abs(name, y) => Named.Abs(name, deBruijnToNamed(y))
  case DeBruijn.App(x, y) => Named.App(deBruijnToNamed(x), deBruijnToNamed(y))
}
```

Alors pour le passage de nommé à indice celui-ci est presque fonctionnel sauf pour l'application. Notre passage repose sur la méthode récursive *aux()* qui transforme le reste de l'expression en indice.
Pour le passage de Bruijn à nommée, nous avons essayé de l'implémenter, mais nous n'avons pas réussi à parcourir l'arbre pour retrouver le lambda le plus proche pour l'associer à un Int.

Nous avons implémenté des tests unitaires :

```
class TestLambda extends AnyFlatSpec {

  "Named.toStr" should "be defined" in {
    Named.Var("x").toStr shouldBe "x"
    Named.App(Named.Abs("x", Named.Var("x")), Named.Var("y")).toStr shouldBe "(λx.x y)"
    Named.Abs("x", Named.App(Named.Var("x"), Named.Var("y"))).toStr shouldBe "λx.(x y)"
  }

  "DeBruijn.toStr" should "be defined" in {
    DeBruijn.FreeVar("x").toStr shouldBe ""
    DeBruijn.Abs("x", DeBruijn.BoundVar(0)).toStr shouldBe "λ 0"
    DeBruijn.Abs("x", DeBruijn.Abs("y", DeBruijn.BoundVar(1))).toStr shouldBe "λ λ 1"
  }

  "namedToDeBruijn" should "be defined" in {
    namedToDeBruijn(Named.Abs("x", Named.Abs("y", Named.Var("x")))).toStr shouldBe "λ λ 1"
    namedToDeBruijn(Named.App(
      Named.App(Named.Abs("x", Named.Abs("y", Named.Abs("z", Named.Var("x")))), Named.Var("z")),
      Named.App(Named.Var("y"), Named.Var("z")))).toStr shouldBe "λ λ λ 2 0 (1 0)"
  }

  "DeBruijnToNamed" should "be defined" in {
    deBruijnToNamed(DeBruijn.FreeVar("x")).toStr shouldBe ""
    deBruijnToNamed(DeBruijn.Abs("x", DeBruijn.BoundVar(0))).toStr shouldBe "λx.x"
    deBruijnToNamed(DeBruijn.App(
      DeBruijn.Abs("x", DeBruijn.BoundVar(0)),
      DeBruijn.Abs("y", DeBruijn.BoundVar(1)))).toStr shouldBe "(λx.x λy.x)"
  }

}
```