Productively recursing infinitely

Modelling evaluation of lambda calculus with coinduction in Agda

1. Background

Untyped **Lambda Calculus** consists of three components:

- **Functions:** A body and a parameter available in the body.
- **Application:** Binds a value to a parameter in a function
- **Variable reference:** Accesses this parameter.

(λx.λy.x)(10)

Defines a function that always returns 10

A common extension is **letrec** where a variable is defined that can recursively reference itself.

Agda is a total language usable as **proof assistant**.

Totality means all programs have to terminate successfully. Infinite or cyclic structures are not allowed generally.

Coinduction allows modelling infinite data within Agda

2. Research Questions

My research questions are as follows:

- What are the different ways to model evaluation of lambda calculus using cyclic data structures, and thus coinduction?
- How do the models compare to each other in terms of ease of implementation and their limitations?
- How suitable are these models to Agda and what are limitations Agda has that got in the way of evaluating lambda expressions?

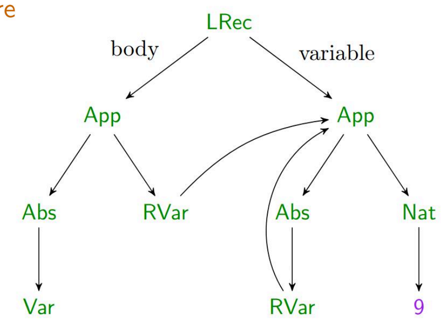
3. Encodings

There are different places where cycles appear in lambda calculus:

- **Variable bindings**: Variables refer back to the application where they were bound to a value. Due to time constraints not implemented.
- **Recursive variables**: Recursive variables refer back to the expression defined recursively.

```
record RTerm : Set where
  coinductive
  constructor RTermCtr
  field
    term : ITerm
```

```
data ITerm : Set where
  ITNat : Nat → ITerm
  ITVar : Nat → ITerm
  ITRVar : RTerm → ITerm
  ITAbs : ITerm → ITerm
  ITApp : ITerm → ITerm → ITerm
  ITLRec : RTerm → ITerm
```



4. Evaluation

- **Inductive:** Evaluator either needs disabled termination checker, or fuel parameter.

```
{-# NON_TERMINATING #-}
eval e (TVar x) = eval e (lookup e x)
eval e (LRec t t₁) = eval (t :: e) t₁
```

- **Coinductive:** Productive but needs inductive function to force result afterwards.

```
runIValFuel zero _ = nat 999
runIValFuel (suc f) (concrete x) = x
runIValFuel (suc f) (delay x) = runIValFuel f (rec x)
```

5. Agda Challenges

- **Unclear error reporting:** The error messages include unknow variables and highlighting is imprecise.

```
eval e (ITApp t a) = case eval e t of λ where
  (concrete (abs t)) → delay (mkRVal (a :: e) t)
  (delay x) → {! !}
```

- **Productivity checking:** Even trivial functions like id prevent guardedness.
- **Lacking documentation:** Not all language features explained. Code reading needed.

6. Limitations & Future work

- Some bugs in evaluator could be fixed.
- Evaluate lazily instead of with call-by-need.
- No dependent types, therefore no correctness guarantees.
- Combination between inductive and coinductive by having inductive expressions but a coinductive evaluator.

Sarah van de Noort (S.J.A.vandeNoort@student.tudelft.nl)

CSE3000 - Research Project

Supervisors: Jesper Cockx & Bohdan Liesnikov