December 12th, ‘24

Design of Programming Languages (CMSC 305),
Bard College

Adding recursive let to the Hasqtan language

Henry Wandover

Outline

- A little lambda calculus
- Considerations before implementation
- Typing rule
- Operational semantics
- Haskell implementation
- Examples

Full implementation available...

- https://github.com/hew02/hasqtan

The *real* basics of λ-calculus

- A lambda abstraction, or function definition, takes an input as a bound variable *x* and returns a body *M*: λx.M
- We will apply the **first item** (recursive function) to the **second item** (the input).
- Through this **self-application** we achieve recursion in the λ-calc, where a term acts both as the function and input.
- **General recursion** takes the form of:

```
rec f = f (rec f)
    = f (f (rec f))      (replace rec f with f (rec f))
    = f (f (f (...))...)      (infinite replacements can be made)
```

NOTE: Infinite in theory, your OS places a limit on a program's recursion depth

Tying the knot

Because of the laziness in Haskell, something like...

```haskell
cyclic = let x = 0 : y
             y = 1 : x
         in  x

main = do
        putStrLn $ show cyclic
```

...Evaluates to...

```
  cyclic
= x
= 0 : y
= 0 : 1 : x -- Knot! Back to the
beginning.
= 0 : 1 : 0 : y
= -- ad infintium
```

A note before attempting to implement

This implementation in *Hasqtan* requires the bound expression to be a lambda expression

The typing rule

$$\frac{(x, s \ \rightarrow \ t) : H \vdash E_1 :: s \ \rightarrow \ t \quad (x, s \ \rightarrow \ t) : H \vdash E_2 :: t}{H \vdash let \ x \ = \ E_1 \ in \ E_2 :: t}$$

Operational semantics of recursive let (lazily interpreted)

$$\frac{(x,\ (\ (x,\ Empty):\sigma[E_1])\ ):\sigma(E_2)\downarrow v_2}{\sigma(let\ x{=}E_1\ in\ E_2)\downarrow v_2}$$

Operational semantics of recursive let (eager interpretation)

$$\frac{(x, Empty){:}\sigma(E_1) \downarrow v_1 \qquad (x, v_1){:}\sigma(E_2) \downarrow v_2}{\sigma(let\ x = E_1\ in\ E_2) \downarrow v_2}$$

The implementation — normal let type check

```
-- Let expressions
typeChecker (Ok(Let x e1 e2)) env =
   let s = typeChecker (Ok e1) env
       env = (x, s) : env
   in
     typeChecker (Ok e2) env
```

The implementation (cont.) — recursive let type check

```
-- In a case of recursion where the x is unknown, adopt type from the
return of the lambda.
typeChecker (Ok(Let x (Lambda y body s t) e2)) env =
   let
     env' = (x, (Arrow s t)) : env -- Add func type to enivronment
     -- Check lambda body with both bindings
     bodyEnv = (y, s) : env'
     bodyType = typeChecker (Ok e2) bodyEnv
   in
     if bodyType == t -- Confirm return type of lambda matches the actual
body type.
     then typeChecker (Ok e2) env'
     else error $ "\x1b[1;31mRecursive function body type mismatch.
Expected: \x1b[0;0m" ++
          show t ++ " \x1b[1;31mbut got: \x1b[0;0m" ++ show bodyType
```

The implementation (cont.) — normal let interpretation

```
interpreter (Ok(Let x e1 e2)) env =
  let
    env = (x, e1) : env -- lazy
  in interpreter (Ok e2) env
```

The implementation (cont.) — recursive let interpretation

```
interpreter (Ok(Let x (Lambda y e1 s t) e2)) env =
  let
    env' = (x, Empty) : env
    env = (x, (Lambda y e1 s t) ) : env' -- lazy
  in interpreter (Ok e2) env
```

E.g.: Solving the 5 factorial (!5) ?

```
let f =
  (\x -> if x == 0 then 1 else f (x - 1) * x) :: Int -> Int
in f 5
```

First, bind recused calls

```
f x = if x == 0 then 1 else x * f (x - 1)

Apply f 5

Evaluate 5 == 0 = False thus 5 * f (5 - 1)

Apply f 4

Evaluate 4 * f (4 - 1)

Apply f 3

Evaluate 3 * f (3 - 1)

Apply f 2

Evaluate 2 * f (2 - 1)

Apply f 1

Evaluate 1 * f (1 - 1)

Apply f 0

Evaluate 1
```

Evaluate those lazy bindings, in reverse

```
fact 1 = 1 * (fact 0) = 1 * 1 = 1
fact 2 = 2 * (fact 1) = 2 * 1 = 2
fact 3 = 3 * (fact 2) = 3 * 2 = 6
fact 4 = 4 * (fact 3) = 4 * 6 = 24
fact 5 = 5 * (fact 4) = 5 * 24 = 120
```

E.g.: Calculating the 13th Fibonacci number.

```
let f =
  (\x -> if x <= 1 then x else f (x - 1) + f (x - 2)) :: Int -> Int
in f 13
```

References

Bernstein, Maxwell. 2017. “Writing a Lisp, Part 13: Let.” Max Bernstein. March 14, 2017. https://bernsteinbear.com/blog/lisp/13_let/.

Considerations on Codecrafting. 2020. “Subtype Inference by Example Part 1: Introducing CubiML.” Considerations on Codecrafting. July 4, 2020. https://blog.polybdenum.com/2020/07/04/subtype-inference-by-example-part-1-introducing-cubiml.html.

Reynaud, Alban, Gabriel Scherer, and Jeremy Yallop. 2021. “A Practical Mode System for Recursive Definitions.” Proceedings of the ACM on Programming Languages 5, no. POPL (January): 1–29. https://doi.org/10.1145/3434326.