CS 555

Exercise 2

Mark Mitchell, Doug Keating

2.1 Enriching the Core Lambda Language

We updated our parser and interpreters so that it can handle let expressions and general recursion using the fix operator.

Key for Evaluation and Typing Rules

Terms :	$a, b ::= \lambda x.\, a \mid a_1 a_2 \mid x \mid c \mid bool \mid \text{if } a_1 a_2 a_3 \mid op(a_1, a_2)$
Constants :	$c ::= \mid -4294967296 \mid ... \mid -1 \mid 0 \mid 1 \mid ... \mid 4294967296 \mid$
Booleans :	$bool ::= tru \mid fls$
Operators :	$op ::= + \mid - \mid * \mid / \mid Nand \mid == \mid <$
OpNum :	$opN ::= + \mid - \mid * \mid / \mid Nand$
OpBool :	$opB ::= eq \mid lt$
Values :	$v ::= c \mid bool \mid \lambda x.\, a$
Types :	$T ::= T \to T \mid Int \mid Bool$
Environments :	$e ::= [] \mid e, x : T$

(Small-Step) Structural Operational Semantics Rules

Additional rules for let and fix have been added to the core lambda language.

Small-Step Evaluation Rules

T-App1 $$\frac{a \to a'}{ab \to a'b}$$

T-App2 $$\frac{b \to b'}{vb \to vb'}$$

T-Abs $$\frac{}{(\lambda x.\, a)v \to [x \mapsto v]a}$$

T-If1 $$\frac{a \to a'}{\text{if } a\ b_1\ b_2 \ \to \text{if } a'\ b_1\ b_2}$$

T-IfTru $$\frac{}{\text{if } fls\ a\ b\ \to a}$$

T-IfFls $$\frac{}{\text{if } fls\ a\ b\ \to b}$$

T-Op1 $$\frac{a \to a'}{op(a, b) \to op(a', b)}$$

T-Op2 $$\frac{b \to b'}{op(v, b) \to op(v, b')}$$

T-Op3 $$\frac{}{op(c_1, c_2) \to v}$$

T-Let1 $$\frac{a \to a'}{\text{Let } x = a \text{ in } b \to \text{Let } x = a' \text{ in } b}$$

T-Let2 $$\frac{}{\text{Let } x = v \text{ in } b \to [x \mapsto v]b}$$

T-Fix1 $$\frac{a \to a'}{\text{Fix } a \to \text{Fix } a'}$$

T-Fix2 $$\frac{}{\text{Fix } (\lambda x.\, a) \to [x \mapsto \text{Fix } (\lambda x.\, a)](\lambda x.\, a)}$$

(Big-Step)Natural Semantics Rules

The natural semantics rules have been extended to include the let and fix cases.

Big-Step Evaluation Rules

T-TermValue $a \Rightarrow v$

T-Constant $c \Rightarrow c$

T-Abstraction $\lambda x.\, a \Rightarrow \lambda x.\, a$

T-Application
$$\frac{a \Rightarrow \lambda x.\, a' \quad b \Rightarrow v' \quad [x \mapsto v']a' \Rightarrow v}{ab \Rightarrow v}$$

T-IfTruN
$$\frac{a \Rightarrow tru \quad b_1 \Rightarrow v}{\text{if } a\ b_1\ b_2\ \Rightarrow v}$$

T-IfFlsN
$$\frac{a \Rightarrow fls \quad b_2 \Rightarrow v}{\text{if } a\ b_1\ b_2\ \Rightarrow v}$$

T-OpN
$$\frac{a_1 \Rightarrow v_1 \quad a_2 \Rightarrow v_2 \quad v = \tilde{op}(v_1, v_2)}{op(a_1, a_2) \Rightarrow v}$$

T-LetN
$$\frac{a \Rightarrow v' \quad [x \mapsto v']b \Rightarrow v}{\text{Let } x = a \text{ in } b \Rightarrow v}$$

T-FixN
$$\frac{a \Rightarrow (\lambda x.\, a') \quad \text{Fix } (\lambda x.\, a') \Rightarrow [x \mapsto \ \text{Fix } (\lambda x.\, a')](\lambda x.\, a')}{\text{Fix } a \Rightarrow [x \mapsto \ \text{Fix } (\lambda x.\, a')](\lambda x.\, a')}$$

Formal Typing Rules

These typing rules include the core lambda calculus and the extended terms let and fix.

Typing Rules

Type-Base	$e \vdash t : T$
Type-Var	$\dfrac{x : T \text{ 'member' } e}{e \vdash x : T}$
Type-Abs	$\dfrac{e,\ x : T \vdash t : T' \ x \text{ 'not a member' } dom(e)}{e \vdash \lambda x : T.t : T \to T'}$
Type-App	$\dfrac{e \vdash t_1 : T \to T',\ e \vdash t_2 : T}{e \vdash t_1\ t_2\ : T'}$
Type-Int	$\dfrac{}{e \vdash c : Int}$
Type-BoolTrue	$\dfrac{}{e \vdash true : Bool}$
Type-BoolFalse	$\dfrac{}{e \vdash false : Bool}$
Type-Conditional	$\dfrac{e \vdash t_1 : Bool,\ e \vdash t_2 : T,\ e \vdash t_3 : T}{e \vdash \text{ if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$
Type-OpNum	$\dfrac{e \vdash t_1 : Int,\ e \vdash t_2 : Int}{e \vdash opN(t_1, t_2)}$ (where opN is $+ \| - \| * \| / \| Nand$)
Type-OpBool	$\dfrac{e \vdash t_1 : Int,\ e \vdash t_2 : Int}{e \vdash opB(t_1, t_2) : Bool}$ (where opB is $eq \| lt$)
Type-Let	$\dfrac{e \vdash t_1 : T,\ e,\ x : T\ \vdash t2 : T' \ x \text{ 'not a member' } dom(e)}{e \vdash \text{ let } x = t_1 \text{ in } t_2 : T'}$
Type-Fix	$\dfrac{e \vdash t : T \to T}{e \vdash \text{ fix } t : T}$

Parsing

This is the parser updated to read this language:

Source Code:

```haskell
module AbstractSyntax where
import System.Environment
import Data.Char
data Token = TkArr | TkLPar | TkComma | TkRPar | TkBool | TkIntWrd | TkVarId String
 | TkAbs | TkColon | TkFullStop | TkApp | TkTrue | TkFalse | TkIf | TkThen | TkElse
 | TkFi | TkIntLit Integer | TkPlus | TkMinus | TkMul | TkDiv | TkNand | TkEq | TkLt
 | TkFix | TkLet | TkIn | TkEnd
deriving (Show, Eq)
data Type = TypeArrow Type Type | TypeBool | TypeInt deriving (Eq)
type Var = String
data Term = Var Var | Abs Var Type Term | App Term Term | Tru | Fls
 | If Term Term Term | IntConst Integer | IntAdd Term Term | IntSub Term Term
 | IntMul Term Term | IntDiv Term Term | IntNand Term Term | IntEq Term Term
 | IntLt Term Term | ParTerm Term | Fix Term | Let Var Term Term
deriving (Eq)
instance Show Type where
  show (TypeArrow x y) = "->(" ++ (show x) ++ "," ++ (show y) ++ ")"
  show TypeBool = "Bool"
  show TypeInt  = "Int"
instance Show Term where
  show (Var x)        = x
  show (Abs x y z)    = "abs(" ++ x ++ ":" ++ (show y) ++ "." ++ (show z) ++ ")"
  show (App x y)      = "app(" ++ (show x) ++ "," ++ (show y) ++ ")"
  show Tru            = "true"
  show Fls            = "false"
  show (If x y z)     = "if " ++ (show x) ++ " then " ++ (show y) ++ " else " ++ (show z)
                        ++ " fi"
  show (Fix x)        = "fix " ++ (show x)
  show (Let x y z)    = "let " ++ x ++ " = " ++ (show y) ++ " in " ++ (show z) ++ " end"
  show (IntConst x)   = show x
  show (IntAdd x y)   = "+(" ++ (show x) ++ "," ++ (show y) ++ ")"
  show (IntSub x y)   = "-(" ++ (show x) ++ "," ++ (show y) ++ ")"
  show (IntMul x y)   = "*(" ++ (show x) ++ "," ++ (show y) ++ ")"
  show (IntDiv x y)   = "/(" ++ (show x) ++ "," ++ (show y) ++ ")"
  show (IntNand x y)  = "^(" ++ (show x) ++ "," ++ (show y) ++ ")"
  show (IntEq x y)    = "=(" ++ (show x) ++ "," ++ (show y) ++ ")"
  show (IntLt x y)    = "<(" ++ (show x) ++ "," ++ (show y) ++ ")"
  show (ParTerm x)    = "(" ++ (show x) ++ ")"
commaSeperated :: [Token]
commaSeperated = [TkArr, TkApp, TkPlus, TkMinus, TkMul, TkDiv, TkNand, TkEq, TkLt]
  -- Take a string and converts it to tokens
makeTokens :: [Char] → [Token]
makeTokens cs = makeTokenList [ ] cs
  -- Builds token list, ignores spaces
makeTokenList :: [Token] → [Char] → [Token]
makeTokenList ls [ ] = reverse ls
makeTokenList ls (x : xs) | (x ≡ '-') ∧ ((head xs) ≡ '>')
       = makeTokenList (TkArr : ls) (tail xs)
   | x ≡ '(' = makeTokenList (TkLPar : ls) xs
```

```
  | x ≡ ',' = makeTokenList (TkComma : ls) xs
  | x ≡ ')' = makeTokenList (TkRPar : ls) xs
  | x ≡ ':' = makeTokenList (TkColon : ls) xs
  | x ≡ '.' = makeTokenList (TkFullStop : ls) xs
  | x ≡ '+' = makeTokenList (TkPlus : ls) xs
  | x ≡ '-' = makeTokenList (TkMinus : ls) xs
  | x ≡ '*' = makeTokenList (TkMul : ls) xs
  | x ≡ '/' = makeTokenList (TkDiv : ls) xs
  | x ≡ '^' = makeTokenList (TkNand : ls) xs
  | x ≡ '=' = makeTokenList (TkEq : ls) xs
  | x ≡ '<' = makeTokenList (TkLt : ls) xs
  | isSpace x = makeTokenList ls xs
  | isAlphaNum x = let alphas = takeWhile isAlphaNum (x : xs)
                       rest = dropWhile isAlphaNum (x : xs)
    in makeTokenList
    ((matchToken alphas) : ls) rest
  | otherwise = error ("unrecognized character "
    ++ show x)
  -- Multiple char cases such as key words are handled by this function
matchToken :: [Char] → Token
matchToken "Bool"  = TkBool
matchToken "Int"   = TkIntWrd
matchToken "abs"   = TkAbs
matchToken "app"   = TkApp
matchToken "true"  = TkTrue
matchToken "false" = TkFalse
matchToken "if"    = TkIf
matchToken "then"  = TkThen
matchToken "else"  = TkElse
matchToken "fi"    = TkFi
matchToken "fix"   = TkFix
matchToken "let"   = TkLet
matchToken "in"    = TkIn
matchToken "end"   = TkEnd
matchToken x       = if (isInt x) then TkIntLit (read x :: Integer) else TkVarId x
isInt :: [Char] → Bool
isInt xs = foldr (λx y → (elem x ['0' .. '9']) ∧ y) True xs
  -- returns type from tokens
tokenType :: [Token] → Type
tokenType [TkIntWrd] = TypeInt
tokenType [TkBool]   = TypeBool
tokenType [TkArr]    = error ("Improper Token used for type")
tokenType (TkArr : xs) = if ((head xs) ≡ TkLPar) then typeArrHelp (xs)
    else error ("Improper Token used for type")
tokenType xs         = error ("Improper Token used for type")
  -- Gets the String that is designated as the variable
getVar :: Token → Var
getVar (TkVarId x) = x
getVar x = error ("Improper variable " ++ show x)
  -- Creates an arrow type from a list of tokens
```

```haskell
typeArrHelp :: [Token] → Type
typeArrHelp xs =
  let arrTypes = (caseHelper (parRemove xs) commaSeperated TkComma 0 [ ])
    in TypeArrow (tokenType (fst arrTypes)) (tokenType (snd arrTypes))
  -- Takes tokens that return terminal terms that do not contain other terms
makeSingleTerm :: Token → Term
makeSingleTerm TkTrue = Tru
makeSingleTerm TkFalse = Fls
makeSingleTerm (TkVarId x) = Var x
makeSingleTerm (TkIntLit x) = IntConst x
makeSingleTerm x = error ("Undefined Term for token " ++ show x)
  -- Takes a list of tokens to return a single term
buildTerm :: [Token] → Term
buildTerm [ ] = error ("No tokens provided")
buildTerm [x] = makeSingleTerm x
buildTerm (x : xs)
   | x ≡ TkAbs = absTerm xs
   | x ≡ TkApp = appTerm xs
   | x ≡ TkIf   = ifTerm xs
   | x ≡ TkLPar = ParTerm (parCase xs)
   | (x ≡ TkFix) ∧ ((head xs) ≡ TkLPar) = Fix (parCase (tail xs))
   | x ≡ TkLet = letTerm xs
   | (x ≡ TkPlus) ∧ ((head xs) ≡ TkLPar) =
     let operands = (caseHelper (parRemove xs) commaSeperated TkComma 0 [ ])
       in IntAdd (buildTerm (fst operands)) (buildTerm (snd operands))
   | (x ≡ TkMinus) ∧ ((head xs) ≡ TkLPar) =
     let operands = (caseHelper (parRemove xs) commaSeperated TkComma 0 [ ])
       in IntSub (buildTerm (fst operands)) (buildTerm (snd operands))
   | (x ≡ TkMul) ∧ ((head xs) ≡ TkLPar) =
     let operands = (caseHelper (parRemove xs) commaSeperated TkComma 0 [ ])
       in IntMul (buildTerm (fst operands)) (buildTerm (snd operands))
   | (x ≡ TkDiv) ∧ ((head xs) ≡ TkLPar) =
     let operands = (caseHelper (parRemove xs) commaSeperated TkComma 0 [ ])
       in IntDiv (buildTerm (fst operands)) (buildTerm (snd operands))
   | (x ≡ TkNand) ∧ ((head xs) ≡ TkLPar) =
     let operands = (caseHelper (parRemove xs) commaSeperated TkComma 0 [ ])
       in IntNand (buildTerm (fst operands)) (buildTerm (snd operands))
   | (x ≡ TkEq) ∧ ((head xs) ≡ TkLPar) =
     let operands = (caseHelper (parRemove xs) commaSeperated TkComma 0 [ ])
       in IntEq (buildTerm (fst operands)) (buildTerm (snd operands))
   | (x ≡ TkLt) ∧ ((head xs) ≡ TkLPar) =
     let operands = (caseHelper (parRemove xs) commaSeperated TkComma 0 [ ])
       in IntLt (buildTerm (fst operands)) (buildTerm (snd operands))
   | otherwise = error ("Undefined Term on token " ++ show x)
  -- Helper function for the abs case
absTerm :: [Token] → Term
absTerm xs =
  let absCase = caseHelper (parRemove xs) [TkAbs] TkColon 0 [ ]
    in let colDot = caseHelper (snd absCase) [TkAbs] TkFullStop 0 [ ]
      in Abs (getVar (head (fst absCase))) (tokenType (fst colDot))
        (buildTerm (snd colDot))
```

```haskell
-- Helper function for the app case
appTerm :: [Token] → Term
appTerm xs =
  let appCase = caseHelper (parRemove xs) commaSeperated TkComma 0 [ ]
    in App (buildTerm (fst appCase)) (buildTerm (snd appCase))

-- Function removes the first and last parenthese of term or group of terms
parRemove :: [Token] → [Token]
parRemove xs = if (((head xs) ≡ TkLPar) ∧ ((last xs) ≡ TkRPar))
    then (tail (init xs)) else error ("Mismatched parentheses")

-- Helper function for the let case
letTerm :: [Token] → Term
letTerm (x : xs) =
  let letCase = caseHelper (letFix xs) [TkLet] TkIn 0 [ ]
    in Let (getVar x) (buildTerm (fst letCase)) (buildTerm (snd letCase))

-- Function checks for closing end term for let functions
letFix :: [Token] → [Token]
letFix (x : xs) = if ((x ≡ TkEq) ∧ (last xs ≡ TkEnd)) then (init xs)
  else error ("missing end word")
letFix _ = error ("Improper use of let expression")

-- Helper function for the if case
ifTerm :: [Token] → Term
ifTerm xs =
  let ifCase = caseHelper (ifFix xs) [TkIf] TkThen 0 [ ]
    in let thenElse = caseHelper (snd ifCase) [TkIf] TkElse 0 [ ]
      in If (buildTerm (fst ifCase)) (buildTerm (fst thenElse))
        (buildTerm (snd thenElse))

-- Function checks for closing fi term for if functions
ifFix :: [Token] → [Token]
ifFix xs = if (last xs ≡ TkFi) then (init xs) else error ("missing fi")

-- Expression that contain multiple terms or types needs to be broken up
-- at key points i.e. if cases seperate terms at the then and else tokens,
-- however nested if cases are ignored via a counter such that the tokens are
-- not partitioned at this point inside nested terms.
caseHelper :: [Token] → [Token] → Token → Int → [Token] → ([Token], [Token])
caseHelper [ ] tk0 _ _ _ = error ("Incomplete " ++ (show (head tk0)) ++ " statement")
caseHelper (x : xs) tk0 tk n ys | ((n ≡ 0) ∧ (x ≡ tk)) = (ys, xs)
   | ((¬ (n ≡ 0)) ∧ (x ≡ tk)) =
     caseHelper xs tk0 tk (n − 1) (ys ++ [x])
   | x ∈ tk0 = caseHelper xs tk0 tk (n + 1) (ys ++ [x])
   | otherwise = caseHelper xs tk0 tk n (ys ++ [x])

-- Handles the ParTerm case
parCase :: [Token] → Term
parCase xs = if (last xs ≡ TkRPar) then (buildTerm (init xs))
    else error ("missing parentheses")

fv :: Term → [Var]
fv (Var x)         = [x]
fv (Abs x _ t)     = [y | y ← (fv t), x ≢ y]
fv (If t1 t2 t3)   = (fv t1) ++ (fv t2) ++ (fv t3)
fv (ParTerm t)     = fv t
fv (App t1 t2)     = (fv t1) ++ (fv t2)
```

```
fv (IntAdd t1 t2) = (fv t1) ++ (fv t2)
fv (IntSub t1 t2) = (fv t1) ++ (fv t2)
fv (IntMul t1 t2) = (fv t1) ++ (fv t2)
fv (IntDiv t1 t2) = (fv t1) ++ (fv t2)
fv (IntNand t1 t2) = (fv t1) ++ (fv t2)
fv (IntEq t1 t2)  = (fv t1) ++ (fv t2)
fv (IntLt t1 t2)  = (fv t1) ++ (fv t2)
fv (Fix t)        = fv t
fv (Let x t1 t2)  = (fv t1) ++ [y | y <- (fv t2), x /= y]
fv _              = []
  -- Checking the free variables prevents overwriting an inner abstraction
  -- variable name when that variable is set from its innermost lambda
subst :: Var -> Term -> Term -> Term
subst x s t = if (elem x (fv t)) then subHelper x s t else t
  -- Abs case already checked for free variables,
subHelper :: Var -> Term -> Term -> Term
subHelper x s (Var y)        = if (y == x) then s else (Var y)
subHelper x s (Abs y tp t)   = Abs y tp (subHelper x s t)
subHelper x s (App t1 t2)    = App (subst x s t1) (subst x s t2)
subHelper x s (If t1 t2 t3)  = If (subst x s t1) (subst x s t2) (subst x s t3)
subHelper x s (IntAdd t1 t2) = IntAdd (subst x s t1) (subst x s t2)
subHelper x s (IntSub t1 t2) = IntSub (subst x s t1) (subst x s t2)
subHelper x s (IntMul t1 t2) = IntMul (subst x s t1) (subst x s t2)
subHelper x s (IntDiv t1 t2) = IntDiv (subst x s t1) (subst x s t2)
subHelper x s (IntNand t1 t2) = IntNand (subst x s t1) (subst x s t2)
subHelper x s (IntEq t1 t2) = IntEq (subst x s t1) (subst x s t2)
subHelper x s (IntLt t1 t2)  = IntLt (subst x s t1) (subst x s t2)
subHelper x s (ParTerm t)    = ParTerm (subHelper x s t)
subHelper x s (Fix t)        = Fix (subHelper x s t)
subHelper x s (Let y t1 t2)  =
  Let y (subst x s t1) (if x == y then t2 else (subst x s t2))
subHelper _ _ z              = z
isValue :: Term -> Bool
isValue (Abs _ _ _) = True
isValue Tru         = True
isValue Fls         = True
isValue (IntConst _) = True
isValue _           = False
```

Updated Structural Operational Semantics

This is the code used to express the small-step semantics:

Source Code:

```
module StructuralOperationalSemantics where
```

```haskell
import Data.List
import qualified AbstractSyntax as S
import qualified IntegerArithmetic as I
termToInt :: S.Term → Integer
termToInt (S.IntConst x) = x
termToInt _ = error ("Non integer in arithmetic application")
justVal :: Maybe S.Term → S.Term
justVal (Just x) = x
justVal _ = error ("incorrect use of justVal")
eval1 :: S.Term → Maybe S.Term
eval1 t = case t of
  S.App (S.Abs x τ11 t12) t2
     | S.isValue t2 → Just (S.subst x t2 t12)
     | otherwise →
       let newT2 = (eval1 t2)
         in if (newT2 ≡ Nothing) then Nothing
           else Just (S.App (S.Abs x τ11 t12) (justVal newT2))
  S.App t1 t2
     | S.isValue t1 → Nothing
     | otherwise → let newT1 = (eval1 t1)
         in if (newT1 ≡ Nothing) then Nothing
           else Just (S.App (justVal newT1) t2)
  S.If t1 t2 t3
     | ¬ (S.isValue t1) →
       let newT1 = (eval1 t1)
         in if (newT1 ≡ Nothing) then Nothing
           else Just (S.If (justVal newT1) t2 t3)
     | t1 ≡ S.Tru → Just t2
     | t1 ≡ S.Fls → Just t3
     | otherwise → Nothing
  S.Fix (S.Abs x y t1) → Just (S.subst x (S.Fix (S.Abs x y t1)) t1)
  S.Fix t → if ((eval1 t) ≡ Nothing) then Nothing
           else Just (S.Fix (justVal (eval1 t)))
  S.Let x t1 t2
     | S.isValue t1 → Just (S.subst x t1 t2)
     | otherwise → let newT1 = (eval1 t1)
         in if (newT1 ≡ Nothing) then Nothing
           else Just (S.Let x (justVal newT1) t2)
  S.IntAdd t1 t2
     | ¬ (S.isValue t1) →
       let newT1 = (eval1 t1)
         in if (newT1 ≡ Nothing) then Nothing
           else Just (S.IntAdd (justVal newT1) t2)
     | ¬ (S.isValue t2) →
       let newT2 = (eval1 t2)
         in if (newT2 ≡ Nothing) then Nothing
           else Just (S.IntAdd t1 (justVal newT2))
     | otherwise → Just (S.IntConst (I.intAdd (termToInt t1) (termToInt t2)))
  S.IntSub t1 t2
     | ¬ (S.isValue t1) →
       let newT1 = (eval1 t1)
```

```
        in if (newT1 ≡ Nothing) then Nothing
          else Just (S.IntSub (justVal newT1) t2)
  | ¬ (S.isValue t2) →
      let newT2 = (eval1 t2)
        in if (newT2 ≡ Nothing) then Nothing
          else Just (S.IntSub t1 (justVal newT2))
  | otherwise → Just (S.IntConst (I.intSub (termToInt t1) (termToInt t2)))
S.IntMul t1 t2
  | ¬ (S.isValue t1) →
      let newT1 = (eval1 t1)
        in if (newT1 ≡ Nothing) then Nothing
          else Just (S.IntMul (justVal newT1) t2)
  | ¬ (S.isValue t2) →
      let newT2 = (eval1 t2)
        in if (newT2 ≡ Nothing) then Nothing
          else Just (S.IntMul t1 (justVal newT2))
  | otherwise → Just (S.IntConst (I.intMul (termToInt t1) (termToInt t2)))
S.IntDiv t1 t2
  | ¬ (S.isValue t1) →
      let newT1 = (eval1 t1)
        in if (newT1 ≡ Nothing) then Nothing
          else Just (S.IntDiv (justVal newT1) t2)
  | ¬ (S.isValue t2) →
      let newT2 = (eval1 t2)
        in if (newT2 ≡ Nothing) then Nothing
          else Just (S.IntDiv t1 (justVal newT2))
  | otherwise → Just (S.IntConst (I.intDiv (termToInt t1) (termToInt t2)))
S.IntNand t1 t2
  | ¬ (S.isValue t1) →
      let newT1 = (eval1 t1)
        in if (newT1 ≡ Nothing) then Nothing
          else Just (S.IntNand (justVal newT1) t2)
  | ¬ (S.isValue t2) →
      let newT2 = (eval1 t2)
        in if (newT2 ≡ Nothing) then Nothing
          else Just (S.IntNand t1 (justVal newT2))
  | otherwise → Just (S.IntConst (I.intNand (termToInt t1) (termToInt t2)))
S.IntEq t1 t2
  | ¬ (S.isValue t1) →
      let newT1 = (eval1 t1)
        in if (newT1 ≡ Nothing) then Nothing
          else Just (S.IntEq (justVal newT1) t2)
  | ¬ (S.isValue t2) →
      let newT2 = (eval1 t2)
        in if (newT2 ≡ Nothing) then Nothing
          else Just (S.IntEq t1 (justVal newT2))
  | otherwise → if (I.intEq (termToInt t1) (termToInt t2))
        then Just S.Tru else Just S.Fls
S.IntLt t1 t2
  | ¬ (S.isValue t1) →
      let newT1 = (eval1 t1)
```

```
            in if (newT1 ≡ Nothing) then Nothing
               else Just (S.IntLt (justVal newT1) t2)
        | ¬ (S.isValue t2) →
          let newT2 = (eval1 t2)
            in if (newT2 ≡ Nothing) then Nothing
               else Just (S.IntLt t1 (justVal newT2))
        | otherwise → if (I.intLt (termToInt t1) (termToInt t2))
            then Just S.Tru else Just S.Fls
    S.ParTerm t → (eval1 t)
    S.Tru        → Nothing
    S.Fls        → Nothing
    S.IntConst _ → Nothing
    S.Var _      → Nothing
    S.Abs _ _ _  → Nothing
eval :: S.Term → S.Term
eval t =
  case eval1 t of
    Just t′ → eval t′
    Nothing → t
```

Updated Natural Semantics

The natural semantics are very similar to the structural operational semantics, but instead of using single step evaluation, it uses big step evaluation to allow for terms to be reduced without returning the program to the root of the evaluation tree. Effectively natural semantics allow for evaluation to occur nearby to where the most work has been done. We implemented the following code for Natural Semantics:

Source Code:

```
module NaturalSemantics where
import Data.List
import qualified AbstractSyntax as S
import qualified IntegerArithmetic as I
eval :: S.Term → S.Term
eval t = if (S.isValue t) then t
  else evalPattern t
evalPattern :: S.Term → S.Term
evalPattern (S.ParTerm t) = eval t
evalPattern (S.If t1 t2 t3)  = if ((eval t1) ≡ S.Tru) then eval t2 else eval t3
evalPattern (S.App t1 t2)   = appHelp (eval t1) t2
evalPattern (S.Fix t)       = fixHelp (eval t) (S.Fix t)
evalPattern (S.Let x t1 t2) = eval (S.subst x (eval t1) t2)
evalPattern x               = binaryOp x
binaryOp :: S.Term → S.Term
binaryOp (S.IntAdd t1 t2) = S.IntConst (I.intAdd (termToInt t1) (termToInt t2))
binaryOp (S.IntSub t1 t2)  = S.IntConst (I.intSub (termToInt t1) (termToInt t2))
binaryOp (S.IntMul t1 t2) = S.IntConst (I.intMul (termToInt t1) (termToInt t2))
```

```
binaryOp (S.IntDiv t1 t2)  = S.IntConst (I.intDiv (termToInt t1) (termToInt t2))
binaryOp (S.IntNand t1 t2) = S.IntConst (I.intNand (termToInt t1) (termToInt t2))
binaryOp (S.IntEq t1 t2)   =
  if (I.intEq (termToInt t1) (termToInt t2)) then S.Tru else S.Fls
binaryOp (S.IntLt t1 t2)   =
  if (I.intLt (termToInt t1) (termToInt t2)) then S.Tru else S.Fls
binaryOp x                 = x
  -- Type is not correct so it returns x since it is stuck
appHelp :: S.Term → S.Term → S.Term
appHelp (S.Abs x _ t1) t2 = eval (S.subst x (eval t2) t1)
apphelp x y = S.App x y
  -- Type is not correct so it returns S.App x y since it is stuck
  -- Just like apphelp but don't evalute the fix term inside right away
fixHelp :: S.Term → S.Term → S.Term
fixHelp (S.Abs x _ t1) t2 = eval (S.subst x t2 t1)
fixHelp _ x = x
  -- Type is not correct so it returns x since it is stuck
termToInt :: S.Term → Integer
termToInt (S.IntConst x) = x
termToInt x =
  let newX = eval x
    in if x ≢ newX then termToInt (eval newX)
      else error ("Non integer in arithmetic application")
```

Integer Arithmetic

The code used for the IntegerArithmetic module is identical to that used in Exercise 1.

Source Code:

```
module IntegerArithmetic where
import Data.Bits
intRestrictRangeAddMul :: Integer → Integer
intRestrictRangeAddMul m
  | m ⩾ 0 = m `mod` 4294967296
  | otherwise = −((−m) `mod` 4294967296)
intAdd :: Integer → Integer → Integer
intAdd m n = intRestrictRangeAddMul (m + n)
intSub :: Integer → Integer → Integer
intSub m n = intRestrictRangeAddMul (m − n)
intMul :: Integer → Integer → Integer
intMul m n = intRestrictRangeAddMul (m ∗ n)
intDiv :: Integer → Integer → Integer
intDiv m n = if n ≡ 0 then error "integer division by zero"
             else intRestrictRangeAddMul (m `div` n)
```

```
intNand :: Integer → Integer → Integer
intNand m n = intRestrictRangeAddMul (complement (m .&. n))
intEq :: Integer → Integer → Bool
intEq   m n = m ≡ n
intLt :: Integer → Integer → Bool
intLt   m n = m < n
```

Type Checker

We used the provided code for type checking for the core lambda language and then extended it for the let and fix operations.

Source Code:

```
module Typing where
import Data.Maybe
import Data.List
import qualified AbstractSyntax as S
data Context = Empty
             | Bind Context S.Var S.Type
             deriving Eq
instance Show Context where
  show Empty          = "<>"
  show (Bind Γ x τ) = show Γ ++ "," ++ x ++ ":" ++ show τ
contextLookup :: S.Var → Context → Maybe S.Type
contextLookup x Empty = Nothing
contextLookup x (Bind Γ y τ)
   | x ≡ y      = Just τ
   | otherwise = contextLookup x Γ
typing :: Context → S.Term → Maybe S.Type
typing Γ t = case t of
  S.Var x → contextLookup x Γ
  S.Abs x τ1 t2 → do τ2 ← typing (Bind Γ x τ1) t2; Just (S.TypeArrow τ1 τ2)
  S.App t1 t2 → do S.TypeArrow τ11 τ12 ← typing Γ t1
    τ ← typing Γ t2
    if τ ≡ τ11 then Just τ12 else Nothing
  S.Tru → Just S.TypeBool
  S.Fls → Just S.TypeBool
  S.If t1 t2 t3 → do S.TypeBool ← typing Γ t1
    τ ← typing Γ t2
    tau' ← typing Γ t3
    if tau' ≡ τ then Just τ else Nothing
  S.Let x t1 t2 → do τ1 ← typing Γ t1
    typing (Bind Γ x τ1) t2
  S.Fix t → do (S.TypeArrow τ1 τ2) ← typing Γ t
```

```
            Just τ2
    S.IntConst _ → Just S.TypeInt
    S.IntAdd t1 t2 → arith t1 t2
    S.IntSub t1 t2 → arith t1 t2
    S.IntMul t1 t2 → arith t1 t2
    S.IntDiv t1 t2 → arith t1 t2
    S.IntNand t1 t2 → arith t1 t2
    S.IntEq t1 t2 → rel t1 t2
    S.IntLt t1 t2 → rel t1 t2
    where
      arith t1 t2 = do S.TypeInt ← typing Γ t1; S.TypeInt ← typing Γ t2; Just S.TypeInt
      rel t1 t2 = do S.TypeInt ← typing Γ t1; S.TypeInt ← typing Γ t2; Just S.TypeBool
typeCheck :: S.Term → S.Type
typeCheck t =
  case typing Empty t of
    Just τ → τ
    _ → error "type error"
```

Main Program

Our main program reads from a text file a string, this is sent to the parser to be tokenized and converted to a term. This term is type checked to see if the entire program can be interpreted. Then the term is sent to the updated small-step and big-step evaluation modules, following this the ReductionSemantics, CCMachine, SCCMachine, CKMachine, and the CEKMachine are called. The reduction semantics, CC Machine and SCC machine all use a context to handle reductions. All of these modules produce the same term in the test cases provided at the bottom.

The reduction semantics returns an updated term and rebuilds its context for every reduction. The CC and SCC machines carry the context along with the current term to be reduced, this allows the reductions to be handled down inside the context tree. However the context tree must be recursed through when it is updated so that a new context will reflect the most recent reduction. The CK and CEK machines do not need to do this sort of recursive decent through the program. By using the continuation structure all reductions are effectively done inside of the context of whatever structure is at the head of a list of program instructions. This is far more effective as the program can be updated in constant time once a term is reduced.

Source Code:

```
import System.Environment
import Data.Char
import qualified Typing as T
import qualified AbstractSyntax as S
import qualified StructuralOperationalSemantics as O
import qualified NaturalSemantics as N
  -- import qualified ReductionSemantics as R
import qualified CCMachine as C
```

```
import qualified SCCMachine as D
import qualified CKMachine as K
import qualified CEKMachine as L
main = do
  args ← getArgs
  str ← mapM readFile args
  putStrLn "---Input:---"
  putStrLn (head str)
  putStrLn "---Term:---"
  let tokens = (S.makeTokens (head str))
  let term = S.buildTerm tokens
  putStrLn $ show term
  putStrLn "---Type:---"
  let exprType = T.typeCheck term
  putStrLn $ show exprType
  putStrLn "---Structural Semantics - Normal form:---"
  let newTerm1 = O.eval term
  putStrLn $ show newTerm1
  putStrLn "---Natural Semantics - Normal form:---"
  let newTerm2 = N.eval term
  putStrLn $ show newTerm2
    -- putStrLn "—Reduction Semantics - Normal form:—"
    -- let newTerm3 = R.textualMachineEval term
    -- putStrLn $ show newTerm3
  putStrLn "---CCMachine - Normal form:---"
  let newTerm4 = C.ccMachineEval term
  putStrLn $ show newTerm4
  putStrLn "---SCCMachine - Normal form:---"
  let newTerm5 = D.sccMachineEval term
  putStrLn $ show newTerm5
  putStrLn "---CKMachine - Normal form:---"
  let newTerm6 = K.ckMachineEval term
  putStrLn $ show newTerm6
  putStrLn "---CEKMachine - Normal form:---"
  let newTerm7 = L.cekMachineEval term
  putStrLn $ show newTerm7
```

2.2 Reduction Semantics

2.2.1 Evaluation Contexts

The following code was used to implement evaluation contexts:

Source Code:

```
module EvaluationContext where
import qualified AbstractSyntax as S
data Context = □
    | AppT  Context S.Term
    | AppV  S.Term Context
    | If      Context S.Term S.Term
    | IntAddT   Context S.Term
    | IntAddV   S.Term Context
    | IntSubT   Context S.Term
    | IntSubV   S.Term Context
    | IntMulT   Context S.Term
    | IntMulV   S.Term Context
    | IntDivT   Context S.Term
    | IntDivV   S.Term Context
    | IntNandT  Context S.Term
    | IntNandV  S.Term Context
    | IntEqT Context S.Term
    | IntEqV    S.Term Context
    | IntLtT    Context S.Term
    | IntLtV    S.Term Context
    | ParTerm   Context
    | Fix     Context
    | LetT    S.Var Context S.Term
    | LetV    S.Var S.Term Context
fillWithTerm :: Context → S.Term → S.Term
fillWithTerm c t = case c of
  □               → t
  AppT c1 t2      → S.App (fillWithTerm c1 t) t2
  AppV t1 c2      → S.App t1 (fillWithTerm c2 t)
  If c1 t2 t3     → S.If (fillWithTerm c1 t) t2 t3
  IntAddT c1 t2   → S.IntAdd (fillWithTerm c1 t) t2
  IntAddV t1 c2   → S.IntAdd t1 (fillWithTerm c2 t)
  IntSubT c1 t2   → S.IntSub (fillWithTerm c1 t) t2
  IntSubV t1 c2   → S.IntSub t1 (fillWithTerm c2 t)
  IntMulT c1 t2   → S.IntMul (fillWithTerm c1 t) t2
  IntMulV t1 c2   → S.IntMul t1 (fillWithTerm c2 t)
  IntDivT c1 t2   → S.IntDiv (fillWithTerm c1 t) t2
  IntDivV t1 c2   → S.IntDiv t1 (fillWithTerm c2 t)
  IntNandT c1 t2  → S.IntNand (fillWithTerm c1 t) t2
  IntNandV t1 c2  → S.IntNand t1 (fillWithTerm c2 t)
  IntEqT c1 t2    → S.IntEq (fillWithTerm c1 t) t2
  IntEqV t1 c2    → S.IntEq t1 (fillWithTerm c2 t)
  IntLtT c1 t2    → S.IntLt (fillWithTerm c1 t) t2
  IntLtV t1 c2    → S.IntLt t1 (fillWithTerm c2 t)
  ParTerm c1      → S.ParTerm (fillWithTerm c1 t)
  Fix c1          → S.Fix (fillWithTerm c1 t)
  LetT v1 c2 t3   → S.Let v1 (fillWithTerm c2 t) t3
  LetV v1 t2 c3   → S.Let v1 t2 (fillWithTerm c3 t)
fillWithContext :: Context → Context → Context
```

$$
\begin{array}{l}
\mathit{fillWithContext}\ c\ c' = \mathbf{case}\ c\ \mathbf{of} \\
\quad \begin{array}{ll}
\Box & \to c' \\
\mathit{AppT}\ \mathit{c1}\ t_2 & \to \mathit{AppT}\ (\mathit{fillWithContext}\ \mathit{c1}\ c')\ t_2 \\
\mathit{AppV}\ t_1\ \mathit{c2} & \to \mathit{AppV}\ t_1\ (\mathit{fillWithContext}\ \mathit{c2}\ c') \\
\mathit{If}\ \mathit{c1}\ t_2\ t_3 & \to \mathit{If}\ (\mathit{fillWithContext}\ \mathit{c1}\ c')\ t_2\ t_3 \\
\mathit{IntAddT}\ \mathit{c1}\ t_2 & \to \mathit{IntAddT}\ (\mathit{fillWithContext}\ \mathit{c1}\ c')\ t_2 \\
\mathit{IntAddV}\ t_1\ \mathit{c2} & \to \mathit{IntAddV}\ t_1\ (\mathit{fillWithContext}\ \mathit{c2}\ c') \\
\mathit{IntSubT}\ \mathit{c1}\ t_2 & \to \mathit{IntSubT}\ (\mathit{fillWithContext}\ \mathit{c1}\ c')\ t_2 \\
\mathit{IntSubV}\ t_1\ \mathit{c2} & \to \mathit{IntSubV}\ t_1\ (\mathit{fillWithContext}\ \mathit{c2}\ c') \\
\mathit{IntMulT}\ \mathit{c1}\ t_2 & \to \mathit{IntMulT}\ (\mathit{fillWithContext}\ \mathit{c1}\ c')\ t_2 \\
\mathit{IntMulV}\ t_1\ \mathit{c2} & \to \mathit{IntMulV}\ t_1\ (\mathit{fillWithContext}\ \mathit{c2}\ c') \\
\mathit{IntDivT}\ \mathit{c1}\ t_2 & \to \mathit{IntDivT}\ (\mathit{fillWithContext}\ \mathit{c1}\ c')\ t_2 \\
\mathit{IntDivV}\ t_1\ \mathit{c2} & \to \mathit{IntDivV}\ t_1\ (\mathit{fillWithContext}\ \mathit{c2}\ c') \\
\mathit{IntNandT}\ \mathit{c1}\ t_2 & \to \mathit{IntNandT}\ (\mathit{fillWithContext}\ \mathit{c1}\ c')\ t_2 \\
\mathit{IntNandV}\ t_1\ \mathit{c2} & \to \mathit{IntNandV}\ t_1\ (\mathit{fillWithContext}\ \mathit{c2}\ c') \\
\mathit{IntEqT}\ \mathit{c1}\ t_2 & \to \mathit{IntEqT}\ (\mathit{fillWithContext}\ \mathit{c1}\ c')\ t_2 \\
\mathit{IntEqV}\ t_1\ \mathit{c2} & \to \mathit{IntEqV}\ t_1\ (\mathit{fillWithContext}\ \mathit{c2}\ c') \\
\mathit{IntLtT}\ \mathit{c1}\ t_2 & \to \mathit{IntLtT}\ (\mathit{fillWithContext}\ \mathit{c1}\ c')\ t_2 \\
\mathit{IntLtV}\ t_1\ \mathit{c2} & \to \mathit{IntLtV}\ t_1\ (\mathit{fillWithContext}\ \mathit{c2}\ c') \\
\mathit{ParTerm}\ \mathit{c1} & \to \mathit{ParTerm}\ (\mathit{fillWithContext}\ \mathit{c1}\ c') \\
\mathit{Fix}\ \mathit{c1} & \to \mathit{Fix}\ (\mathit{fillWithContext}\ \mathit{c1}\ c') \\
\mathit{LetT}\ \mathit{v1}\ \mathit{c2}\ t_3 & \to \mathit{LetT}\ \mathit{v1}\ (\mathit{fillWithContext}\ \mathit{c2}\ c')\ t_3 \\
\mathit{LetV}\ \mathit{v1}\ t_2\ \mathit{c3} & \to \mathit{LetV}\ \mathit{v1}\ t_2\ (\mathit{fillWithContext}\ \mathit{c3}\ c')
\end{array}
\end{array}
$$

2.2.2 Standard Reduction

When forming the evaluation contexts, if a term is a redex, then it should be the next thing reduced. Otherwise the first subterm that is not a value should be searched for a redex. In our implementation this is accomplished by returning the evaluation context (Term, E.Hole) if the term is a redexe. Otherwise, the first non-value subterm is passed to a helper function that uses makeEvalContext and E.fillWithContext to recursively search for a redex within that subterm to reduce. makeContractum is responsible for reducing the redex once it is found.

The textual machine recursively evaluates a term, splitting it into an evaluation context and a subterm. If this subterm is a redex, it is reduced to obtain a contractum, and the evaluation context is filled with this contractum. This machine is inherently inefficient, since it essentially returns to the root of the tree and then searches for the location of the next redex. This does not provide any means of localization that could enhance efficiency.

Source Code:

module *ReductionSemantics* **where**
import *qualified AbstractSyntax as S*
import *qualified EvaluationContext as E*
import *qualified IntegerArithmetic as I*

$\mathit{makeEvalContext} :: \mathit{S.Term} \to \mathit{Maybe}\ (\mathit{S.Term}, \mathit{E.Context})$

```
makeEvalContext t = case t of
  S.App (S.Abs x τ11 t12) t2
     | S.isValue t2  → Just (t, E.□)
  S.App t1 t2
     | S.isValue t1        → nextEC (t2, (E.AppV t1 E.□))
     | otherwise           → nextEC (t1, (E.AppT E.□ t2))
  S.If (S.Fls) t2 t3 → Just (t, E.□)
  S.If (S.Tru) t2 t3 → Just (t, E.□)
  S.If t1 t2 t3        → nextEC (t1, (E.If E.□ t2 t3))
  S.IntAdd (S.IntConst t1) (S.IntConst t2) → Just (t, E.□)
  S.IntAdd t1 t2
     | S.isValue t1        → nextEC (t2, (E.IntAddV t1 E.□))
     | otherwise           → nextEC (t1, (E.IntAddT E.□ t2))
  S.IntSub (S.IntConst t1) (S.IntConst t2) → Just (t, E.□)
  S.IntSub t1 t2
     | S.isValue t1        → nextEC (t2, (E.IntSubV t1 E.□))
     | otherwise           → nextEC (t1, (E.IntSubT E.□ t2))
  S.IntMul (S.IntConst t1) (S.IntConst t2) → Just (t, E.□)
  S.IntMul t1 t2
     | S.isValue t1        → nextEC (t2, (E.IntMulV t1 E.□))
     | otherwise           → nextEC (t1, (E.IntMulT E.□ t2))
  S.IntDiv (S.IntConst t1) (S.IntConst t2) → Just (t, E.□)
  S.IntDiv t1 t2
     | S.isValue t1        → nextEC (t2, (E.IntDivV t1 E.□))
     | otherwise           → nextEC (t1, (E.IntDivT E.□ t2))
  S.IntNand (S.IntConst t1) (S.IntConst t2) → Just (t, E.□)
  S.IntNand t1 t2
     | S.isValue t1        → nextEC (t2, (E.IntNandV t1 E.□))
     | otherwise           → nextEC (t1, (E.IntNandT E.□ t2))
  S.IntEq (S.IntConst t1) (S.IntConst t2) → Just (t, E.□)
  S.IntEq t1 t2
     | S.isValue t1        → nextEC (t2, (E.IntEqV t1 E.□))
     | otherwise           → nextEC (t1, (E.IntEqT E.□ t2))
  S.IntLt (S.IntConst t1) (S.IntConst t2) → Just (t, E.□)
  S.IntLt t1 t2
     | S.isValue t1        → nextEC (t2, (E.IntLtV t1 E.□))
     | otherwise           → nextEC (t1, (E.IntLtT E.□ t2))
  S.Let x t1 t2
     | S.isValue t1  → Just (t, E.□)
     | otherwise →       nextEC (t1, (E.LetT x E.□ t2))
  S.Fix (S.Abs x τ11 t12) → Just (t, E.□)
  S.Fix t → Just (t, E.Fix E.□)
  _                    → Nothing
nextEC :: (S.Term, E.Context) → Maybe (S.Term, E.Context)
nextEC (t, c) = do
  (t′, c′) ← makeEvalContext t
  return (t′, E.fillWithContext c c′)
makeContractum :: S.Term → S.Term
```

```
makeContractum t = case t of
  S.App (S.Abs x τ11 t12) t2                    → S.subst x t2 t12
  S.If (S.Tru) t2 t3                            → t2
  S.If (S.Fls) t2 t3                            → t3
  S.IntAdd (S.IntConst n1) (S.IntConst n2)      → S.IntConst (I.intAdd n1 n2)
  S.IntSub (S.IntConst n1) (S.IntConst n2)      → S.IntConst (I.intSub n1 n2)
  S.IntMul (S.IntConst n1) (S.IntConst n2)      → S.IntConst (I.intMul n1 n2)
  S.IntDiv (S.IntConst n1) (S.IntConst n2)      → S.IntConst (I.intDiv n1 n2)
  S.IntNand (S.IntConst n1) (S.IntConst n2)     → S.IntConst (I.intNand n1 n2)
  S.IntLt (S.IntConst n1) (S.IntConst n2)       → if (I.intLt n1 n2) then S.Tru else S.Fls
  S.IntEq (S.IntConst n1) (S.IntConst n2)       → if (I.intEq n1 n2) then S.Tru else S.Fls
  S.Let x t1 t2                                 → S.subst x t1 t2
  S.Fix (S.Abs x τ11 t12)                       → S.subst x (S.Fix (S.Abs x τ11 t12)) t12
textualMachineStep :: S.Term → Maybe S.Term
textualMachineStep t = case makeEvalContext t of
  Just (t′, c)     → Just (E.fillWithTerm c (makeContractum t′))
  _                → Nothing
textualMachineEval :: S.Term → S.Term
textualMachineEval t = case textualMachineStep t of
  Just t′          → textualMachineEval t′
  _                → t
```

2.3 Abstract Register Machines

2.3.1 CCMachine

This machine builds a context tree which is a copy of the program with a hole located where the current reduction is being handled. Once the term is reduced its relative context will have further terms extracted and then reduced, and the context tree is rebuilt to reflect the structure surrounding these reductions. Updating the context after a reduction is a little tricky because the context tree will be almost the same one level above the current reduction, inside this level a new context will be added as a child or a hole will be used because the next reduction will be handled on this level or a different branch of this level.

```
module CCMachine where
import qualified AbstractSyntax as S
import qualified EvaluationContext as E
import qualified IntegerArithmetic as I
ccMachineStep :: (S.Term, E.Context) → Maybe (S.Term, E.Context)
ccMachineStep (t, c) = case t of
  S.App t1 t2
    | ¬ (S.isValue t1)
```

```
      → Just (t1, E.fillWithContext c (E.AppT E.□ t2))  {-cc1 -}
    | S.isValue t1 ∧ ¬ (S.isValue t2)
      → Just (t2, E.fillWithContext c (E.AppV t1 E.□))  {-cc2 -}
  S.App (S.Abs x _ t12) t2 → Just (S.subst x t2 t12, c)      {-ccbeta -}
  S.If (S.Tru) t1 t2     → Just (t1, c)
  S.If (S.Fls) t1 t2     → Just (t2, c)
  S.If t1 t2 t3          → Just (t1, E.fillWithContext c (E.If E.□ t2 t3))
  S.Fix (S.Abs x y t1) → Just ((S.subst x t t1), c)
  S.Fix t1             → Just (t1, E.fillWithContext c (E.Fix E.□))
  S.Let x t1 t2
    | S.isValue t1 → Just ((S.subst x t1 t2), c)
    | otherwise    → Just (t1, E.fillWithContext c (E.LetT x E.□ t2))
  S.IntAdd (S.IntConst t1) (S.IntConst t2)
    → Just (S.IntConst (I.intAdd t1 t2), c)
  S.IntAdd t1 t2
    | S.isValue t1 → Just (t2, E.fillWithContext c (E.IntAddV t1 E.□))
    | otherwise    → Just (t1, E.fillWithContext c (E.IntAddT E.□ t2))
  S.IntSub (S.IntConst t1) (S.IntConst t2)
    → Just (S.IntConst (I.intSub t1 t2), c)
  S.IntSub t1 t2
    | S.isValue t1 → Just (t2, E.fillWithContext c (E.IntSubV t1 E.□))
    | otherwise    → Just (t1, E.fillWithContext c (E.IntSubT E.□ t2))
  S.IntMul (S.IntConst t1) (S.IntConst t2)
    → Just (S.IntConst (I.intMul t1 t2), c)
  S.IntMul t1 t2
    | S.isValue t1 → Just (t2, E.fillWithContext c (E.IntMulV t1 E.□))
    | otherwise    → Just (t1, E.fillWithContext c (E.IntMulT E.□ t2))
  S.IntDiv (S.IntConst t1) (S.IntConst t2)
    → Just (S.IntConst (I.intDiv t1 t2), c)
  S.IntDiv t1 t2
    | S.isValue t1 → Just (t2, E.fillWithContext c (E.IntDivV t1 E.□))
    | otherwise    → Just (t1, E.fillWithContext c (E.IntDivT E.□ t2))
  S.IntNand (S.IntConst t1) (S.IntConst t2)
    → Just (S.IntConst (I.intNand t1 t2), c)
  S.IntNand t1 t2
    | S.isValue t1 → Just (t2, E.fillWithContext c (E.IntNandV t1 E.□))
    | otherwise    → Just (t1, E.fillWithContext c (E.IntNandT E.□ t2))
  S.IntEq (S.IntConst t1) (S.IntConst t2)
    → Just ((if (I.intEq t1 t2) then S.Tru else S.Fls), c)
  S.IntEq t1 t2
    | S.isValue t1 → Just (t2, E.fillWithContext c (E.IntEqV t1 E.□))
    | otherwise    → Just (t1, E.fillWithContext c (E.IntEqT E.□ t2))
  S.IntLt (S.IntConst t1) (S.IntConst t2)
    → Just ((if (I.intLt t1 t2) then S.Tru else S.Fls), c)
  S.IntLt t1 t2
    | S.isValue t1 → Just (t2, E.fillWithContext c (E.IntLtV t1 E.□))
    | otherwise    → Just (t1, E.fillWithContext c (E.IntLtT E.□ t2))
  otherwise → if (S.isValue t) then (fillTermHelper1 (t, c) E.□) else Nothing
fillTermHelper1 :: (S.Term, E.Context) → E.Context → Maybe (S.Term, E.Context)
```

$$
\begin{array}{l}
\textit{fillTermHelper1}\ (t, c)\ c1 = \textbf{case}\ c\ \textbf{of} \\
\quad E.\Box \rightarrow \textit{Nothing} \\
\quad E.AppT\ t_1\ t_2 \quad \rightarrow fTH2\ (t, c, c1, t_1, (E.AppT\ E.\Box\ t_2)) \\
\quad E.AppV\ t_1\ t_2 \quad \rightarrow fTH2\ (t, c, c1, t_2, (E.AppV\ t_1\ E.\Box)) \\
\quad E.If\ t_1\ t_2\ t_3 \quad \rightarrow fTH2\ (t, c, c1, t_1, (E.If\ E.\Box\ t_2\ t_3)) \\
\quad E.Fix\ t_1 \quad \rightarrow fTH2\ (t, c, c1, t_1, (E.Fix\ E.\Box)) \\
\quad E.LetT\ x\ t_1\ t_2 \quad \rightarrow fTH2\ (t, c, c1, t_1, (E.LetT\ x\ E.\Box\ t_2)) \\
\quad E.IntAddV\ t_1\ t_2 \rightarrow fTH2\ (t, c, c1, t_2, (E.IntAddV\ t_1\ E.\Box)) \\
\quad E.IntAddT\ t_1\ t_2 \rightarrow fTH2\ (t, c, c1, t_1, (E.IntAddT\ E.\Box\ t_2)) \\
\quad E.IntSubV\ t_1\ t_2 \rightarrow fTH2\ (t, c, c1, t_2, (E.IntSubV\ t_1\ E.\Box)) \\
\quad E.IntSubT\ t_1\ t_2 \rightarrow fTH2\ (t, c, c1, t_1, (E.IntSubT\ E.\Box\ t_2)) \\
\quad E.IntMulV\ t_1\ t_2 \rightarrow fTH2\ (t, c, c1, t_2, (E.IntMulV\ t_1\ E.\Box)) \\
\quad E.IntMulT\ t_1\ t_2 \rightarrow fTH2\ (t, c, c1, t_1, (E.IntMulT\ E.\Box\ t_2)) \\
\quad E.IntDivV\ t_1\ t_2 \rightarrow fTH2\ (t, c, c1, t_2, (E.IntDivV\ t_1\ E.\Box)) \\
\quad E.IntDivT\ t_1\ t_2 \rightarrow fTH2\ (t, c, c1, t_1, (E.IntDivT\ E.\Box\ t_2)) \\
\quad E.IntNandV\ t_1\ t_2 \rightarrow fTH2\ (t, c, c1, t_2, (E.IntNandV\ t_1\ E.\Box)) \\
\quad E.IntNandT\ t_1\ t_2 \rightarrow fTH2\ (t, c, c1, t_1, (E.IntNandT\ E.\Box\ t_2)) \\
\quad E.IntEqV\ t_1\ t_2 \quad \rightarrow fTH2\ (t, c, c1, t_2, (E.IntEqV\ t_1\ E.\Box)) \\
\quad E.IntEqT\ t_1\ t_2 \quad \rightarrow fTH2\ (t, c, c1, t_1, (E.IntEqT\ E.\Box\ t_2)) \\
\quad E.IntLtV\ t_1\ t_2 \quad \rightarrow fTH2\ (t, c, c1, t_2, (E.IntLtV\ t_1\ E.\Box)) \\
\quad E.IntLtT\ t_1\ t_2 \quad \rightarrow fTH2\ (t, c, c1, t_1, (E.IntLtT\ E.\Box\ t_2)) \\
\quad \textit{otherwise} \quad \rightarrow \textit{Nothing}
\end{array}
$$

-- This function takes a term t and a context c, this context may fill a hole in
-- context c1, context c2 is the nested context in c if c2 is not a hole to be
-- filled by t then we recur and c3 pererves the structure of c at that level,
-- with a hole where c2 is taken from.

$$
\begin{array}{l}
fTH2 :: (S.Term, E.Context, E.Context, E.Context, E.Context) \rightarrow \textit{Maybe}\ (S.Term, E.Context) \\
fTH2\ (t, c, c1, c2, c3) = \textbf{if}\ (\textit{isHole}\ c2)\ \textbf{then}\ \textit{Just}\ (E.\textit{fillWithTerm}\ c\ t, c1) \\
\quad \textbf{else}\ (\textit{fillTermHelper1}\ (t, c2)\ (E.\textit{fillWithContext}\ c1\ c3)) \\
\textit{isHole} :: E.Context \rightarrow \textit{Bool} \\
\textit{isHole}\ E.\Box = \textit{True} \\
\textit{isHole}\ \_ = \textit{False} \\
\textit{ccMachineEvalHelp} :: (S.Term, E.Context) \rightarrow (S.Term, E.Context) \\
\textit{ccMachineEvalHelp}\ (t, c) = \\
\quad \textbf{case}\ \textit{ccMachineStep}\ (t, c)\ \textbf{of} \\
\qquad \textit{Just}\ t' \rightarrow \textit{ccMachineEvalHelp}\ t' \\
\qquad \textit{Nothing} \rightarrow (t, c) \\
\textit{ccMachineEval} :: S.Term \rightarrow S.Term \\
\textit{ccMachineEval}\ t = \textit{fst}\ (\textit{ccMachineEvalHelp}\ (t, E.\Box))
\end{array}
$$

2.3.2 SCCMachine

Very similar to the CCMachine, this machine however will look at the context once a term is reduced to a value in order to decide what to do next. This allows for applications and operations to be done immediately once the terms needed are reduced to values. This does the same thing as the CCMachine but it combines steps which make its evaluation much simpler.

```haskell
module SCCMachine where
import qualified AbstractSyntax as S
import qualified EvaluationContext as E
import qualified IntegerArithmetic as I
sccMachineStep :: (S.Term, E.Context) → Maybe (S.Term, E.Context)
sccMachineStep (t, c) = case t of
  S.App t1 t2       → Just (t1, E.fillWithContext c (E.AppT E.□ t2))
  S.If t1 t2 t3     → Just (t1, E.fillWithContext c (E.If E.□ t2 t3))
  S.Fix t1          → Just (t1, E.fillWithContext c (E.Fix E.□))
  S.Let x t1 t2     → Just (t1, E.fillWithContext c (E.LetT x E.□ t2))
  S.IntAdd t1 t2    → Just (t1, E.fillWithContext c (E.IntAddT E.□ t2))
  S.IntSub t1 t2    → Just (t1, E.fillWithContext c (E.IntSubT E.□ t2))
  S.IntMul t1 t2    → Just (t1, E.fillWithContext c (E.IntMulT E.□ t2))
  S.IntDiv t1 t2    → Just (t1, E.fillWithContext c (E.IntDivT E.□ t2))
  S.IntNand t1 t2   → Just (t1, E.fillWithContext c (E.IntNandT E.□ t2))
  S.IntEq t1 t2     → Just (t1, E.fillWithContext c (E.IntEqT E.□ t2))
  S.IntLt t1 t2     → Just (t1, E.fillWithContext c (E.IntLtT E.□ t2))
  otherwise → if (S.isValue t) then (fillTermHelper1 (t, c) E.□) else Nothing
  -- Similar to the one in the CC machine but this handles cases of flipping
  -- contexts like E.AppT E.Hole t2 for E.AppV t1 E.Hole where t1 is a value this
  -- also handles cases where terms can be applied such as addition or application
  -- this context is removed and used to create a new value, leaving a E.Hole in
  -- since that particular redux is complete.
fillTermHelper1 :: (S.Term, E.Context) → E.Context → Maybe (S.Term, E.Context)
fillTermHelper1 (t, c) c1 = case c of
  E.□ → Nothing
  E.AppT E.□ t2 → Just (t2, E.fillWithContext c1 (E.AppV t E.□))
  E.AppT t1 t2      → fTH2 (t, c, c1, t1, (E.AppT E.□ t2))
  E.AppV (S.Abs x _ t12) E.□
    → Just ((S.subst x t t12), c1)
  E.AppV _ E.□      → Nothing
  E.AppV t1 t2      → fTH2 (t, c, c1, t2, (E.AppV t1 E.□))
  E.If E.□ t2 t3
    | t ≡ S.Tru     → Just (t2, c1)
    | t ≡ S.Fls     → Just (t3, c1)
    | otherwise     → Nothing
  E.If t1 t2 t3     → fTH2 (t, c, c1, t1, (E.If E.□ t2 t3))
  E.Fix E.□         → fixHelper (S.Fix t, c1)
  E.Fix t1          → fTH2 (t, c, c1, t1, (E.Fix E.□))
  E.LetT x E.□ t2
                    → Just ((S.subst x t t2), c1)
  E.LetT x t1 t2    → fTH2 (t, c, c1, t1, (E.LetT x E.□ t2))
  E.IntAddV t1 E.□
                    → binaryOpHelp ((S.IntAdd t1 t), c1)
  E.IntAddV t1 t2   → fTH2 (t, c, c1, t2, (E.IntAddV t1 E.□))
  E.IntAddT E.□ t2
                    → Just (t2, E.fillWithContext c1 (E.IntAddV t E.□))
  E.IntAddT t1 t2   → fTH2 (t, c, c1, t1, (E.IntAddT E.□ t2))
  E.IntSubV t1 E.□
                    → binaryOpHelp ((S.IntSub t1 t), c1)
  E.IntSubV t1 t2   → fTH2 (t, c, c1, t2, (E.IntSubV t1 E.□))
```

```
    E.IntSubT E.□ t2
                      → Just (t2, E.fillWithContext c1 (E.IntSubV t E.□))
    E.IntSubT t1 t2   → fTH2 (t, c, c1, t1, (E.IntSubT E.□ t2))
    E.IntMulV t1 E.□
                      → binaryOpHelp ((S.IntMul t1 t), c1)
    E.IntMulV t1 t2   → fTH2 (t, c, c1, t2, (E.IntMulV t1 E.□))
    E.IntMulT E.□ t2
                      → Just (t2, E.fillWithContext c1 (E.IntMulV t E.□))
    E.IntMulT t1 t2   → fTH2 (t, c, c1, t1, (E.IntMulT E.□ t2))
    E.IntDivV t1 E.□
                      → binaryOpHelp ((S.IntDiv t1 t), c1)
    E.IntDivV t1 t2   → fTH2 (t, c, c1, t2, (E.IntDivV t1 E.□))
    E.IntDivT E.□ t2
                      → Just (t2, E.fillWithContext c1 (E.IntDivV t E.□))
    E.IntDivT t1 t2   → fTH2 (t, c, c1, t1, (E.IntDivT E.□ t2))
    E.IntNandV t1 E.□
                      → binaryOpHelp ((S.IntNand t1 t), c1)
    E.IntNandV t1 t2  → fTH2 (t, c, c1, t2, (E.IntNandV t1 E.□))
    E.IntNandT E.□ t2
                      → Just (t2, E.fillWithContext c1 (E.IntNandV t E.□))
    E.IntNandT t1 t2  → fTH2 (t, c, c1, t1, (E.IntNandT E.□ t2))
    E.IntEqV t1 E.□
                      → binaryOpHelp ((S.IntEq t1 t), c1)
    E.IntEqV t1 t2    → fTH2 (t, c, c1, t2, (E.IntEqV t1 E.□))
    E.IntEqT E.□ t2
                      → Just (t2, E.fillWithContext c1 (E.IntEqV t E.□))
    E.IntEqT t1 t2    → fTH2 (t, c, c1, t1, (E.IntEqT E.□ t2))
    E.IntLtV t1 E.□
                      → binaryOpHelp ((S.IntLt t1 t), c1)
    E.IntLtV t1 t2    → fTH2 (t, c, c1, t2, (E.IntLtV t1 E.□))
    E.IntLtT E.□ t2
                      → Just (t2, E.fillWithContext c1 (E.IntLtV t E.□))
    E.IntLtT t1 t2    → fTH2 (t, c, c1, t1, (E.IntLtT E.□ t2))
    otherwise         → Nothing
    -- This function takes a term t and a context c, this context may fill a hole in
    -- context c1, context c2 is the nested context in c if c2 is not a hole to be
    -- filled by t then we recur and c3 pererves the structure of c at that level,
    -- with a hole where c2 is taken from.
fTH2 :: (S.Term, E.Context, E.Context, E.Context, E.Context) → Maybe (S.Term, E.Context)
fTH2 (t, c, c1, c2, c3) = if (isHole c2) then Just (E.fillWithTerm c t, c1)
   else (fillTermHelper1 (t, c2) (E.fillWithContext c1 c3))

fixHelper :: (S.Term, E.Context) → Maybe (S.Term, E.Context)
fixHelper (t@(S.Fix (S.Abs x y t1)), c) = Just ((S.subst x t t1), c)
fixHelper (_, c) = Nothing

binaryOpHelp :: (S.Term, E.Context) → Maybe (S.Term, E.Context)
binaryOpHelp (t, c) = case t of
   S.IntAdd (S.IntConst t1) (S.IntConst t2)
      → Just (S.IntConst (I.intAdd t1 t2), c)
   S.IntSub (S.IntConst t1) (S.IntConst t2)
      → Just (S.IntConst (I.intSub t1 t2), c)
   S.IntMul (S.IntConst t1) (S.IntConst t2)
```

```
        → Just (S.IntConst (I.intMul t1 t2), c)
    S.IntDiv (S.IntConst t1) (S.IntConst t2)
        → Just (S.IntConst (I.intDiv t1 t2), c)
    S.IntNand (S.IntConst t1) (S.IntConst t2)
        → Just (S.IntConst (I.intNand t1 t2), c)
    S.IntEq (S.IntConst t1) (S.IntConst t2)
        → Just ((if (I.intEq t1 t2) then S.Tru else S.Fls), c)
    S.IntLt (S.IntConst t1) (S.IntConst t2)
        → Just ((if (I.intLt t1 t2) then S.Tru else S.Fls), c)
    otherwise
        → Nothing
isHole :: E.Context → Bool
isHole E.□ = True
isHole _ = False
sccMachineEvalHelp :: (S.Term, E.Context) → (S.Term, E.Context)
sccMachineEvalHelp (t, c) =
    case sccMachineStep (t, c) of
      Just t′ → sccMachineEvalHelp t′
      Nothing → (t, c)
sccMachineEval :: S.Term → S.Term
sccMachineEval t = fst (sccMachineEvalHelp (t, E.□))
```

2.3.3 CKMachine

This machine does something similar to the CC and SCC machines but this uses a continuation structure rather then a context tree. Allowing for the machine to use the head of the continuation structure when a reduction is done, instead of requiring that the context tree be traversed in order for the programs state to be updated.

```
module CKMachine where
import qualified AbstractSyntax as S
import qualified IntegerArithmetic as I
data Cont = ⊙
          | Fun    S.Term Cont   -- where Term is a value
          | Arg    S.Term Cont
          | If     S.Term S.Term Cont
          | Fix    Cont
          | Let    S.Var S.Term Cont
          | Plus1  S.Term Cont
          | Plus2  S.Term Cont
          | Minus1 S.Term Cont
          | Minus2 S.Term Cont
          | Times1 S.Term Cont
          | Times2 S.Term Cont
          | Div1   S.Term Cont
          | Div2   S.Term Cont
```

```
             | Nand1  S.Term Cont
             | Nand2  S.Term Cont
             | Eq1    S.Term Cont
             | Eq2    S.Term Cont
             | Lt1    S.Term Cont
             | Lt2    S.Term Cont
ckMachineStep :: (S.Term, Cont) → Maybe (S.Term, Cont)
ckMachineStep (t, k) = case t of
  S.App t1 t2       → Just (t1, Arg t2 k)
  S.If t1 t2 t3     → Just (t1, If t2 t3 k)
  S.Fix t1          → Just (t1, Fix k)
  S.Let x t1 t2     → Just (t1, Let x t2 k)
  S.IntAdd t1 t2    → Just (t1, Plus1 t2 k)
  S.IntSub t1 t2    → Just (t1, Minus1 t2 k)
  S.IntMul t1 t2    → Just (t1, Times1 t2 k)
  S.IntDiv t1 t2    → Just (t1, Div1 t2 k)
  S.IntNand t1 t2   → Just (t1, Nand1 t2 k)
  S.IntEq t1 t2     → Just (t1, Eq1 t2 k)
  S.IntLt t1 t2     → Just (t1, Lt1 t2 k)
  otherwise → if (S.isValue t) then ckContHelp (t, k) else Nothing
ckContHelp :: (S.Term, Cont) → Maybe (S.Term, Cont)
ckContHelp (t, k) = case k of
  ⊙ → Nothing
  Fun (S.Abs x _ t12) k'
                  → Just ((S.subst x t t12), k')
  Arg t2 k'       → Just (t2, Fun t k')
  If t2 t3 k'
     | t ≡ S.Tru → Just (t2, k')
     | t ≡ S.Fls → Just (t3, k')
     | otherwise → Nothing
  Fix k'          → fixHelper (S.Fix t, k')
  Let x t2 k'     → Just ((S.subst x t t2), k')
  Plus1 t2 k'     → Just (t2, Plus2 t k')
  Plus2 v1 k'     → binaryOpHelp ((S.IntAdd v1 t), k')
  Minus1 t2 k'    → Just (t2, Minus2 t k')
  Minus2 v1 k'    → binaryOpHelp ((S.IntSub v1 t), k')
  Times1 t2 k'    → Just (t2, Times2 t k')
  Times2 v1 k'    → binaryOpHelp ((S.IntMul v1 t), k')
  Div1 t2 k'      → Just (t2, Div2 t k')
  Div2 v1 k'      → binaryOpHelp ((S.IntDiv v1 t), k')
  Nand1 t2 k'     → Just (t2, Nand2 t k')
  Nand2 v1 k'     → binaryOpHelp ((S.IntNand v1 t), k')
  Eq1 t2 k'       → Just (t2, Eq2 t k')
  Eq2 v1 k'       → binaryOpHelp ((S.IntEq v1 t), k')
  Lt1 t2 k'       → Just (t2, Lt2 t k')
  Lt2 v1 k'       → binaryOpHelp ((S.IntLt v1 t), k')
  otherwise       → Nothing
fixHelper :: (S.Term, Cont) → Maybe (S.Term, Cont)
fixHelper (t@(S.Fix (S.Abs x y t1)), k) = Just ((S.subst x t t1), k)
fixHelper (_, k) = Nothing
binaryOpHelp :: (S.Term, Cont) → Maybe (S.Term, Cont)
```

```
binaryOpHelp (t, k) = case t of
  S.IntAdd (S.IntConst t1) (S.IntConst t2)
    → Just (S.IntConst (I.intAdd t1 t2), k)
  S.IntSub (S.IntConst t1) (S.IntConst t2)
    → Just (S.IntConst (I.intSub t1 t2), k)
  S.IntMul (S.IntConst t1) (S.IntConst t2)
    → Just (S.IntConst (I.intMul t1 t2), k)
  S.IntDiv (S.IntConst t1) (S.IntConst t2)
    → Just (S.IntConst (I.intDiv t1 t2), k)
  S.IntNand (S.IntConst t1) (S.IntConst t2)
    → Just (S.IntConst (I.intNand t1 t2), k)
  S.IntEq (S.IntConst t1) (S.IntConst t2)
    → Just ((if (I.intEq t1 t2) then S.Tru else S.Fls), k)
  S.IntLt (S.IntConst t1) (S.IntConst t2)
    → Just ((if (I.intLt t1 t2) then S.Tru else S.Fls), k)
  otherwise
    → Nothing
ckMachineEvalHelp :: (S.Term, Cont) → (S.Term, Cont)
ckMachineEvalHelp (t, k) =
  case ckMachineStep (t, k) of
    Just t′ → ckMachineEvalHelp t′
    Nothing → (t, k)
ckMachineEval :: S.Term → S.Term
ckMachineEval t = fst (ckMachineEvalHelp (t, ⊙))
```

2.3.4 CEKMachine

The CEKMachine builds upon the CKMachine by adding closure and environment this allows for variables to be updated as the terms are being reduced rather than calling a recursive substitution function. The environment fuctions as a lookup table that is only called when a value is required for a variable.

```
module CEKMachine where
import qualified AbstractSyntax as S
import qualified IntegerArithmetic as I
newtype Closure = Cls (S.Term, Environment)
                  deriving Show
newtype Environment = Env [(S.Var, Closure)]
  deriving Show
emptyEnv :: Environment
emptyEnv = Env [ ]
lookupEnv :: Environment → S.Var → Closure
lookupEnv (e@(Env [ ])) x =
  error ("variable " ++ x ++ " not bound in environment " ++ show e)
lookupEnv (Env ((v, c) : t)) x
   | x ≡ v      = c
   | otherwise = lookupEnv (Env t) x
```

```
data Cont = ⊙
          | Fun    Closure Cont     -- where Closure is a value
          | Arg    Closure Cont
          | If     Closure Closure Cont    -- lazy
          | Fix    Cont
          | Let    S.Var Closure Cont
          | Plus1  Closure Cont
          | Plus2  Closure Cont
          | Minus1 Closure Cont
          | Minus2 Closure Cont
          | Times1 Closure Cont
          | Times2 Closure Cont
          | Div1   Closure Cont
          | Div2   Closure Cont
          | Nand1  Closure Cont
          | Nand2  Closure Cont
          | Eq1    Closure Cont
          | Eq2    Closure Cont
          | Lt1    Closure Cont
          | Lt2    Closure Cont
cekMachineStep :: (Closure, Cont) → Maybe (Closure, Cont)
cekMachineStep (cl, k) = case cl of
  Cls (S.App t1 t2, e)     → Just (Cls (t1, e), Arg (Cls (t2, e)) k)
  Cls (S.If t1 t2 t3, e)   → Just (Cls (t1, e),
                               If (Cls (t2, e)) (Cls (t3, e)) k)
  Cls (S.Fix t1, e)        → Just (Cls (t1, e), Fix k)
  Cls (S.Let x t1 t2, e)   → Just (Cls (t1, e), Let x (Cls (t2, e)) k)
  Cls (S.IntAdd t1 t2, e)  → Just (Cls (t1, e), Plus1 (Cls (t2, e)) k)
  Cls (S.IntSub t1 t2, e)  → Just (Cls (t1, e), Minus1 (Cls (t2, e)) k)
  Cls (S.IntMul t1 t2, e)  → Just (Cls (t1, e), Times1 (Cls (t2, e)) k)
  Cls (S.IntDiv t1 t2, e)  → Just (Cls (t1, e), Div1 (Cls (t2, e)) k)
  Cls (S.IntNand t1 t2, e) → Just (Cls (t1, e), Nand1 (Cls (t2, e)) k)
  Cls (S.IntEq t1 t2, e)   → Just (Cls (t1, e), Eq1 (Cls (t2, e)) k)
  Cls (S.IntLt t1 t2, e)   → Just (Cls (t1, e), Lt1 (Cls (t2, e)) k)
  Cls (S.Var x, e)         → Just (lookupEnv e x, k)
  Cls (t, e)               → if (S.isValue t) then cekHelper (cl, k)
                               else Nothing
cekHelper :: (Closure, Cont) → Maybe (Closure, Cont)
cekHelper (cl@(Cls (t, e)), k) = case k of
  ⊙                           → Nothing
  Fun (Cls ((S.Abs x _ t12), (Env e′))) k′
                              → Just (Cls (t12, Env ((x, cl) : e′)), k′)
  Arg (Cls (t2, e′)) k′       → Just (Cls (t2, e′), Fun cl k′)
  If (Cls (t2, e1)) (Cls (t3, e2)) k′
     | t ≡ S.Tru              → Just (Cls (t2, e1), k′)
     | t ≡ S.Fls              → Just (Cls (t3, e2), k′)
     | otherwise              → Nothing
  Fix k′                      → fixHelper (Cls (S.Fix t, e), k′)
  Let x (Cls (t2, (Env e′))) k′ → Just (Cls (t2, Env ((x, cl) : e′)), k′)
  Plus1 (Cls (t2, e′)) k′     → Just (Cls (t2, e′), Plus2 cl k′)
  Plus2 (Cls (v1, e′)) k′     → biOpHelp (Cls ((S.IntAdd v1 t), e′), k′)
```

$Minus1\ (Cls\ (t_2, e'))\ k' \quad \to Just\ (Cls\ (t_2, e'), Minus2\ cl\ k')$
$Minus2\ (Cls\ (v1, e'))\ k' \quad \to biOpHelp\ (Cls\ ((S.IntSub\ v1\ t), e'), k')$
$Times1\ (Cls\ (t_2, e'))\ k' \quad \to Just\ (Cls\ (t_2, e'), Times2\ cl\ k')$
$Times2\ (Cls\ (v1, e'))\ k' \quad \to biOpHelp\ (Cls\ ((S.IntMul\ v1\ t), e'), k')$
$Div1\ (Cls\ (t_2, e'))\ k' \quad \to Just\ (Cls\ (t_2, e'), Div2\ cl\ k')$
$Div2\ (Cls\ (v1, e'))\ k' \quad \to biOpHelp\ (Cls\ ((S.IntDiv\ v1\ t), e'), k')$
$Nand1\ (Cls\ (t_2, e'))\ k' \quad \to Just\ (Cls\ (t_2, e'), Nand2\ cl\ k')$
$Nand2\ (Cls\ (v1, e'))\ k' \quad \to biOpHelp\ (Cls\ ((S.IntNand\ v1\ t), e'), k')$
$Eq1\ (Cls\ (t_2, e'))\ k' \quad \to Just\ (Cls\ (t_2, e'), Eq2\ cl\ k')$
$Eq2\ (Cls\ (v1, e'))\ k' \quad \to biOpHelp\ (Cls\ ((S.IntEq\ v1\ t), e'), k')$
$Lt1\ (Cls\ (t_2, e'))\ k' \quad \to Just\ (Cls\ (t_2, e'), Lt2\ cl\ k')$
$Lt2\ (Cls\ (v1, e'))\ k' \quad \to biOpHelp\ (Cls\ ((S.IntLt\ v1\ t), e'), k')$
$otherwise \quad \to Nothing$

$fixHelper :: (Closure, Cont) \to Maybe\ (Closure, Cont)$
$fixHelper\ (t@(Cls\ (S.Fix\ (S.Abs\ x\ y\ t_1), (Env\ e))), k) =$
$\quad Just\ (Cls\ (t_1, Env\ ((x, t) : e)), k)$
$fixHelper\ (\_, k) = Nothing$

$biOpHelp :: (Closure, Cont) \to Maybe\ (Closure, Cont)$
$biOpHelp\ (Cls\ (t, e), k) =$ **case** t **of**
$\quad S.IntAdd\ (S.IntConst\ t_1)\ (S.IntConst\ t_2)$
$\qquad \to Just\ (Cls\ (S.IntConst\ (I.intAdd\ t_1\ t_2), e), k)$
$\quad S.IntSub\ (S.IntConst\ t_1)\ (S.IntConst\ t_2)$
$\qquad \to Just\ (Cls\ (S.IntConst\ (I.intSub\ t_1\ t_2), e), k)$
$\quad S.IntMul\ (S.IntConst\ t_1)\ (S.IntConst\ t_2)$
$\qquad \to Just\ (Cls\ (S.IntConst\ (I.intMul\ t_1\ t_2), e), k)$
$\quad S.IntDiv\ (S.IntConst\ t_1)\ (S.IntConst\ t_2)$
$\qquad \to Just\ (Cls\ (S.IntConst\ (I.intDiv\ t_1\ t_2), e), k)$
$\quad S.IntNand\ (S.IntConst\ t_1)\ (S.IntConst\ t_2)$
$\qquad \to Just\ (Cls\ (S.IntConst\ (I.intNand\ t_1\ t_2), e), k)$
$\quad S.IntEq\ (S.IntConst\ t_1)\ (S.IntConst\ t_2)$
$\qquad \to Just\ ((\mathbf{if}\ (I.intEq\ t_1\ t_2)\ \mathbf{then}\ Cls\ (S.Tru, e)\ \mathbf{else}\ Cls\ (S.Fls, e)), k)$
$\quad S.IntLt\ (S.IntConst\ t_1)\ (S.IntConst\ t_2)$
$\qquad \to Just\ ((\mathbf{if}\ (I.intLt\ t_1\ t_2)\ \mathbf{then}\ Cls\ (S.Tru, e)\ \mathbf{else}\ Cls\ (S.Fls, e)), k)$
$\quad otherwise$
$\qquad \to Nothing$

$cekMachineEvalHelp :: (Closure, Cont) \to (Closure, Cont)$
$cekMachineEvalHelp\ (cl, k) =$
$\quad$ **case** $cekMachineStep\ (cl, k)$ **of**
$\qquad Just\ clk' \to cekMachineEvalHelp\ clk'$
$\qquad Nothing \to (cl, \odot)$

$cekMachineEval :: S.Term \to S.Term$
$cekMachineEval\ t =$
$\quad$ **let** $Cls\ (t', k) = fst\ (cekMachineEvalHelp\ (Cls\ (t, emptyEnv), \odot))$
$\qquad$ **in** t'

Test Cases

Test Case 1:

```
——Input:———
let
        iseven =
                let
                        mod = abs (m:Int. abs (n:Int. −(m,∗(n,/(m,n)))))
                in
                        abs (k:Int. =(0, app(app(mod,k),2)))
                end
in
        app (iseven, 7)
end

——Term:———
let iseven = let mod = abs(m:Int.abs(n:Int.−(m,∗(n,/(m,n)))))
in abs(k:Int.=(0,app(app(mod,k),2))) end in app(iseven,7) end

——Type:———
Bool

——Structural Semanitcs − Normal form:———
false
——Natural Semanitcs − Normal form:———
false
——Reduction Semantics − Normal form:———
false
——CCMachine − Normal form:———
false
——SCCMachine − Normal form:———
false
——CKMachine − Normal form:———
false
——CEKMachine − Normal form:———
false
```

Test Case 2:

```
——Input:———
app (fix (abs (ie:−>(Int,Bool). abs (x:Int. if =(0,x) then true else
if =(0, −(x,1)) then false else app (ie, −(x,2)) fi fi))), 7)

——Term:———
```

```
app(fix abs(ie:->(Int,Bool).abs(x:Int.if =(0,x) then true else if
 =(0,-(x,1)) then false else app(ie,-(x,2)) fi fi)),7)
```

——Type:———
```
Bool
```

——Structural Semanitcs – Normal form:———
```
false
```
——Natural Semanitcs – Normal form:———
```
false
```
——Reduction Semantics – Normal form:———
```
false
```
——CCMachine – Normal form:———
```
false
```
——SCCMachine – Normal form:———
```
false
```
——CKMachine – Normal form:———
```
false
```
——CEKMachine – Normal form:———
```
false
```

Test Case 3:

——Input:———
```
app (app (fix (abs (e:->(Int,->(Int,Int)). abs (x:Int. abs (y: Int.
if =(0,y) then 1 else *(x,app(app(e,x),-(y,1))) fi)))), 2), 3)
```

——Term:———
```
app(app(fix abs(e:->(Int,->(Int,Int)).abs(x:Int.abs(y:Int.if =(0,y)
then 1 else *(x,app(app(e,x),-(y,1))) fi))),2),3)
```

——Type:———
```
Int
```

——Structural Semanitcs – Normal form:———
8
——Natural Semanitcs – Normal form:———
8
——Reduction Semantics – Normal form:———
8
——CCMachine – Normal form:———
8
——SCCMachine – Normal form:———
8
——CKMachine – Normal form:———
8
——CEKMachine – Normal form:———
8

Test Case 4:

```
——Input:———
app (fix (abs (f:->(Int,Int). abs (x: Int. if =(0,x) then 1 else
*(x, app(f, -(x,1))) fi))), app (fix (abs (f:->(Int,Int).
abs (x: Int. if =(0,x) then 1 else *(x, app(f, -(x,1))) fi))), 3))

——Term:———
app(fix abs(f:->(Int,Int).abs(x:Int.if =(0,x) then 1 else *(x,app(f,-(x,1))) fi)),
app(fix abs(f:->(Int,Int).abs(x:Int.if =(0,x) then 1 else *(x,app(f,-(x,1))) fi)),3))

——Type:———
Int

——Structural Semanitcs - Normal form:———
720
——Natural Semanitcs - Normal form:———
720
——Reduction Semantics - Normal form:———
720
——CCMachine - Normal form:———
720
——SCCMachine - Normal form:———
720
——CKMachine - Normal form:———
720
——CEKMachine - Normal form:———
720
```

Test Case 5:

```
let
        iseven = fix (abs (ie:->(Int,Bool). abs (x:Int.
                if =(0,x) then true else
                 if =(1,x) then false else
                  app (ie, -(x,2)) fi fi)))
in
        let
                collatz = fix (abs (collatz:->(Int,Int). abs (x: Int.
                 if app (iseven, x) then app (collatz, /(x,2)) else
                  if =(x,1) then 1 else
                   app (collatz, +(*(3,x),1)) fi fi)))
        in
          app (collatz, 1000)
        end
end
```

```
——Term:———
let iseven = fix abs(ie:−>(Int,Bool).abs(x:Int.if =(0,x) then true
 else if =(1,x) then false else app(ie,−(x,2)) fi fi)) in
  let collatz = fix abs(collatz:−>(Int,Int).abs(x:Int.if app(iseven,x)
   then app(collatz,/(x,2)) else if =(x,1) then 1 else
    app(collatz,+(∗(3,x),1)) fi fi)) in app(collatz,1000) end end

——Type:———
Int

——Structural Semanitcs − Normal form:———
1
——Natural Semanitcs − Normal form:———
1
——Reduction Semantics − Normal form:———
1
——CCMachine − Normal form:———
1
——SCCMachine − Normal form:———
1
——CKMachine − Normal form:———
1
——CEKMachine − Normal form:———
1
```