DeBruijn Notation, CES Machine, Continuation-Passing-Style Transformation, and the CE3R Machine

Mahnush Movahedi, Mahdi Zamani, Vlado Ovtcharov

May 5, 2012

In this project, we first implemented a small core lambda language, consisting of the lambda calculus with booleans and integers. Moreover, we implemented the nameless representation (DeBruijn notation), natural semantics with nameless representation, the CES machine, the Continuation-Passing-Style (CPS) transformation, and CE3R machine. Our first attempt for adding types to the CPS is based on [1]. We have also implemented the solution proposed in [2] as the second attempt to CPS transformation.

1 De Bruijn notation

The DeBruijn notation is a syntax for nameless representation of λ terms. It can be seen as a reversal of the usual syntax for the λ-calculus where the argument in an application is placed next to its corresponding binder in the function instead of after the latter's body. The goal here is to convert the core lambda language terms to De Bruijn terms, which consists of no variable names.

```
module DeBruijn where
import Data.List
import qualified AbstractSyntax as S
type Type = S.Type
type IntConst = Integer
data Term = Var Int
          | Abs Type Term
          | App Term Term
          | Fix Term
          | Let Term Term
          | Tru
          | Fls
          | If Term Term Term
          | IntConst IntConst
          | IntAdd Term Term
          | IntSub Term Term
```

```haskell
            | IntMul Term Term
            | IntDiv Term Term
            | IntNand Term Term
            | IntEq Term Term
            | IntLt Term Term
          deriving Eq
instance Show Term where
  show (Var i)            = "~" ++ show i
  show (Abs τ t)          = "abs(" ++ show t ++ ")"
  show (App t1 t2)        = "app(" ++ show t1 ++ "," ++ show t2 ++ ")"
  show (Fix t)            = "fix(" ++ show t ++ ")"
  show (Let t1 t2)        = "let " ++ show t1 ++
                            " in " ++ show t2
  show Tru                = "true"
  show Fls                = "false"
  show (If t1 t2 t3)      = "if " ++ show t1 ++ " then " ++ show t2 ++ " else "
                             ++ show t3 ++ " fi"
  show (IntConst t1)      = show t1
  show (IntAdd t1 t2)     = show t1 ++ "+" ++ show t2
  show (IntSub t1 t2)     = show t1 ++ "-" ++ show t2
  show (IntMul t1 t2)     = show t1 ++ "*" ++ show t2
  show (IntDiv t1 t2)     = show t1 ++ "/" ++ show t2
  show (IntNand t1 t2)    = show t1 ++ "|" ++ show t2
  show (IntEq t1 t2)      = show t1 ++ "=" ++ show t2
  show (IntLt t1 t2)      = show t1 ++ "<" ++ show t2
```

We define a function named *dbWorker*, which gets the original lambda term and a stack of strings for storing variable names. It then scans the input term recursively and adds variable names that appear in abstraction and let terms to the stack. Once a variable is seen, *dbWorker* returns the index of the variable in the stack replacing it with an integer.

```haskell
dbWorker :: S.Term -> [String] -> Maybe Term
dbWorker t s =
  case t of
    S.Tru          -> Just Tru
    S.Fls          -> Just Fls
    S.IntConst i   -> Just (IntConst i)
    S.Var x        -> do i <- elemIndex x s
                         Just (Var i)
    S.Abs x τ t1   -> do t1' <- dbWorker t1 (x : s)
                         Just (Abs τ t1')
    S.App t1 t2    -> do t1' <- dbWorker t1 s
```

```haskell
                        t2' ← dbWorker t2 s
                        Just (App t1' t2')
    S.Fix t           → do t' ← dbWorker t s
                        Just (Fix t')
    S.Let x t1 t2     → do t1' ← dbWorker t1 s
                        t2' ← dbWorker t2 (x : s)
                        Just (Let t1' t2')
    S.If t1 t2 t3     → do t1' ← dbWorker t1 s
                        t2' ← dbWorker t2 s
                        t3' ← dbWorker t3 s
                        Just (If t1' t2' t3')
    S.IntAdd t1 t2    → do t1' ← dbWorker t1 s
                        t2' ← dbWorker t2 s
                        Just (IntAdd t1' t2')
    S.IntSub t1 t2    → do t1' ← dbWorker t1 s
                        t2' ← dbWorker t2 s
                        Just (IntSub t1' t2')
    S.IntMul t1 t2    → do t1' ← dbWorker t1 s
                        t2' ← dbWorker t2 s
                        Just (IntMul t1' t2')
    S.IntDiv t1 t2    → do t1' ← dbWorker t1 s
                        t2' ← dbWorker t2 s
                        Just (IntDiv t1' t2')
    S.IntNand t1 t2   → do t1' ← dbWorker t1 s
                        t2' ← dbWorker t2 s
                        Just (IntNand t1' t2')
    S.IntEq t1 t2     → do t1' ← dbWorker t1 s
                        t2' ← dbWorker t2 s
                        Just (IntEq t1' t2')
    S.IntLt t1 t2     → do t1' ← dbWorker t1 s
                        t2' ← dbWorker t2 s
                        Just (IntLt t1' t2')
toDeBruijn :: S.Term → Term
toDeBruijn t = case dbWorker t [ ] of
  Just t → t
  otherwise → error "Cannot convert to De Bruijn notation!"
```

2 Natural semantics with nameless terms

Starting with an empty environment, the big-step machine scans the input nameless term for various subterm structures. Whenever a substitution is required (as in application, Fix, and Let), the value that substitues a variable is put on top of the environment and is used once the variable is accessed. Since the machine does big-step evaluation, everything inserted in the environment must be a value.

```
module NaturalSemanticsWithEnvironmentsClosuresAndDeBruijnIndices where
import Data.Maybe
import qualified DeBruijn as B
import qualified IntegerArithmetic as I
data Value = BoolVal Bool | IntVal Integer | Clo B.Term Env
instance Show Value where
  show (BoolVal b) = show b
  show (IntVal i) = show i
  show (Clo t e)  = "Function: Clo " ++ show t ++ " " ++ show e
type Env = [Value]
evalInEnv :: Env -> B.Term -> Maybe Value
evalInEnv e t =
  case t of
    B.Tru          -> Just (BoolVal True)
    B.Fls          -> Just (BoolVal False)
    (B.IntConst i) -> Just (IntVal i)
    (B.Var i)      -> if (length e > i)
                        then case e !! i of
                              (Clo t' e') -> evalInEnv e' t'
                              v -> Just v
                        else error ("Invalid environment!")
    a@(B.Abs _ _) -> Just (Clo a e)
    (B.If t1 t2 t3)  -> case evalInEnv e t1 of
                        Just (BoolVal True) -> evalInEnv e t2
                        Just (BoolVal False) -> evalInEnv e t3
    (B.App t1 t2)  -> case evalInEnv e t1 of
                        Just (Clo (B.Abs _ t12) e1) ->
                          case evalInEnv e t2 of
                            Just v -> evalInEnv (v : e1) t12
                            otherwise -> Nothing
                        otherwise -> Nothing
```

For evaluation of recursive statements in the form Fix λf.t, we need to substitute f with

Fix λf.t, so we need to put Fix λf.t in the environment to be used later when we access f. We cannot put a code in the environment unless we wrap it in a closure, here, in a Clo statement. Hence, *Fix t* is wapped in a closure and is stored in the environment. This is the only place in this machine where we put something different from a lambda in a closure:

$$
\begin{array}{ll}
a@(B.Fix\ t_1) & \to \textbf{case}\ evalInEnv\ e\ t_1\ \textbf{of} \\
 & \quad Just\ (Clo\ (B.Abs\ \_\ t_1')\ e1') \to evalInEnv\ ((Clo\ a\ e1') : e)\ t_1' \\
 & \quad otherwise \to Nothing \\
(B.Let\ t_1\ t_2) & \to \textbf{case}\ evalInEnv\ e\ t_1\ \textbf{of} \\
 & \quad Just\ v \to evalInEnv\ (v : e)\ t_2 \\
 & \quad otherwise \to Nothing \\
(B.IntAdd\ t_1\ t_2) & \to \textbf{do}\ (IntVal\ i1) \leftarrow evalInEnv\ e\ t_1 \\
 & \quad (IntVal\ i2) \leftarrow evalInEnv\ e\ t_2 \\
 & \quad Just\ (IntVal\ (I.intAdd\ i1\ i2)) \\
(B.IntSub\ t_1\ t_2) & \to \textbf{do}\ (IntVal\ i1) \leftarrow evalInEnv\ e\ t_1 \\
 & \quad (IntVal\ i2) \leftarrow evalInEnv\ e\ t_2 \\
 & \quad Just\ (IntVal\ (I.intSub\ i1\ i2)) \\
(B.IntMul\ t_1\ t_2) & \to \textbf{do}\ (IntVal\ i1) \leftarrow evalInEnv\ e\ t_1 \\
 & \quad (IntVal\ i2) \leftarrow evalInEnv\ e\ t_2 \\
 & \quad Just\ (IntVal\ (I.intMul\ i1\ i2)) \\
(B.IntDiv\ t_1\ t_2) & \to \textbf{do}\ (IntVal\ i1) \leftarrow evalInEnv\ e\ t_1 \\
 & \quad (IntVal\ i2) \leftarrow evalInEnv\ e\ t_2 \\
 & \quad Just\ (IntVal\ (I.intDiv\ i1\ i2)) \\
(B.IntNand\ t_1\ t_2) & \to \textbf{do}\ (IntVal\ i1) \leftarrow evalInEnv\ e\ t_1 \\
 & \quad (IntVal\ i2) \leftarrow evalInEnv\ e\ t_2 \\
 & \quad Just\ (IntVal\ (I.intNand\ i1\ i2)) \\
(B.IntLt\ t_1\ t_2) & \to \textbf{do}\ (IntVal\ i1) \leftarrow evalInEnv\ e\ t_1 \\
 & \quad (IntVal\ i2) \leftarrow evalInEnv\ e\ t_2 \\
 & \quad Just\ (BoolVal\ (I.intLt\ i1\ i2)) \\
(B.IntEq\ t_1\ t_2) & \to \textbf{do}\ (IntVal\ i1) \leftarrow evalInEnv\ e\ t_1 \\
 & \quad (IntVal\ i2) \leftarrow evalInEnv\ e\ t_2 \\
 & \quad Just\ (BoolVal\ (I.intEq\ i1\ i2))
\end{array}
$$

$$
\begin{array}{l}
eval :: B.Term \to Value \\
eval\ t = \textbf{case}\ (evalInEnv\ [\,]\ t)\ \textbf{of} \\
\quad Just\ v \to v \\
\quad otherwise \to error\ (\texttt{"Evaluation error."})
\end{array}
$$

3 CES compiler and virtual machine

CES stands for Code, Environment, Stack. The machine uses these three data structures for the evaluation of the input code. The first step is compilation: the source code of the program is compiled to an intermediate code in format of the Code data structure. The environment keeps track of the free variables of the code and the stack keeps track of the continuation of the code which will be evaluated next.

```
module CESMachine where
import qualified DeBruijn as S
import qualified IntegerArithmetic as I
data Inst = Int Integer
  | Bool Bool
  | Add
  | Sub
  | Mul
  | Div
  | Nand
  | Eq
  | Lt
  | Access Int
  | Close Code
  | Let
  | EndLet
  | Apply
  | Return
  | If
  | Fix
  deriving (Show, Eq)
type Code = [Inst]
data Value = BoolVal Bool | IntVal Integer | Clo Code Env | CloFix Code
  deriving Eq
instance Show Value where
  show (BoolVal b) = show b
  show (IntVal i)  = show i
  show (Clo c e)   = "Clo " ++ show c ++ " " ++ show e
  show (CloFix c) = "Clo " ++ show c
type Env = [Value]
data Slot = Value Value | Code Code | Env Env
  deriving Show
type Stack = [Slot]
```

$$
\begin{array}{l}
\textbf{type}\ \mathit{State} = (\mathit{Code}, \mathit{Env}, \mathit{Stack}) \\
\mathit{compile} :: \mathit{S.Term} \to \mathit{Code} \\
\mathit{compile}\ t = \\
\quad \textbf{case}\ t\ \textbf{of} \\
\qquad \begin{array}{ll}
\mathit{S.Var}\ n & \to [\mathit{Access}\ n] \\
\mathit{S.Tru} & \to [\mathit{Bool}\ \mathit{True}] \\
\mathit{S.Fls} & \to [\mathit{Bool}\ \mathit{False}] \\
\mathit{S.IntConst}\ i & \to [\mathit{Int}\ i] \\
\mathit{S.Abs}\ \_\ t & \to [\mathit{Close}\ (\mathit{compile}\ (t) +\!\!+ [\mathit{Return}])] \\
\mathit{S.App}\ t_1\ t_2 & \to (\mathit{compile}\ t_1) +\!\!+ \mathit{compile}\ (t_2) +\!\!+ [\mathit{Apply}]
\end{array}
\end{array}
$$

In the evaluation of **if** terms in the form $\mathit{If}\ t_1\ t_2\ t_3$, we have to first evaluate t_1 and then decide to evaluate t_2 or t_3. In other words, **if** should be evaluated in a lazy way. This is important because it requires to deal with **if** terms different from other terms in the compilation phase. To clarify this, note that compiling $\mathit{If}\ t_1\ t_2\ t_3$ as $\mathit{compile}\ (t_1) +\!\!+ \mathit{compile}\ (t_2) +\!\!+ \mathit{compile}\ (t_3) +\!\!+ [\mathit{If}]$ will not produce a lazy code because we first evaluate all three terms t_1, t_2, and t_3 and then, choose either t_2 or t_3. Instead, we put both t_2 and t_3 in a closure to postpone its evaluation. Once t_1 is evaluated, we bring either t_2 or t_3 out of the closure and evaluate it:

$$
\begin{array}{l}
\qquad \begin{array}{ll}
\mathit{S.If}\ t_1\ t_2\ t_3 & \to (\mathit{compile}\ t_1) +\!\!+ \\
 & \quad [\mathit{Close}\ (\mathit{compile}\ (t_2) +\!\!+ [\mathit{Return}])] +\!\!+ \\
 & \quad [\mathit{Close}\ (\mathit{compile}\ (t_3) +\!\!+ [\mathit{Return}])] +\!\!+ [\mathit{If}] \\
\mathit{S.Let}\ t_1\ t_2 & \to (\mathit{compile}\ t_1) +\!\!+ [\mathit{Let}] +\!\!+ (\mathit{compile}\ t_2) +\!\!+ [\mathit{EndLet}] \\
\mathit{S.Fix}\ t_1 & \to (\mathit{compile}\ t_1) +\!\!+ [\mathit{Fix}] \\
\mathit{S.IntAdd}\ t_1\ t_2 & \to (\mathit{compile}\ t_1) +\!\!+ (\mathit{compile}\ t_2) +\!\!+ [\mathit{Add}] \\
\mathit{S.IntSub}\ t_1\ t_2 & \to (\mathit{compile}\ t_1) +\!\!+ (\mathit{compile}\ t_2) +\!\!+ [\mathit{Sub}] \\
\mathit{S.IntMul}\ t_1\ t_2 & \to (\mathit{compile}\ t_1) +\!\!+ (\mathit{compile}\ t_2) +\!\!+ [\mathit{Mul}] \\
\mathit{S.IntDiv}\ t_1\ t_2 & \to (\mathit{compile}\ t_1) +\!\!+ (\mathit{compile}\ t_2) +\!\!+ [\mathit{Div}] \\
\mathit{S.IntNand}\ t_1\ t_2 & \to (\mathit{compile}\ t_1) +\!\!+ (\mathit{compile}\ t_2) +\!\!+ [\mathit{Nand}] \\
\mathit{S.IntLt}\ t_1\ t_2 & \to (\mathit{compile}\ t_1) +\!\!+ (\mathit{compile}\ t_2) +\!\!+ [\mathit{Lt}] \\
\mathit{S.IntEq}\ t_1\ t_2 & \to (\mathit{compile}\ t_1) +\!\!+ (\mathit{compile}\ t_2) +\!\!+ [\mathit{Eq}]
\end{array} \\
\mathit{step} :: \mathit{State} \to \mathit{Maybe}\ \mathit{State} \\
\mathit{step}\ \mathit{state} = \\
\quad \textbf{case}\ \mathit{state}\ \textbf{of} \\
\qquad ((\mathit{Int}\ v) : c, e, s) \quad \to \\
\qquad\quad \mathit{Just}\ (c, e, (\mathit{Value}\ (\mathit{IntVal}\ v)) : s) \\
\qquad ((\mathit{Bool}\ v) : c, e, s) \quad \to \\
\qquad\quad \mathit{Just}\ (c, e, (\mathit{Value}\ (\mathit{BoolVal}\ v)) : s) \\
\qquad ((\mathit{Access}\ n) : c, e, s) \to \\
\qquad\quad \textbf{case}\ e\ !!\ n\ \textbf{of}
\end{array}
$$

$$
\begin{aligned}
&\quad\quad \mathit{CloFix}\ t \rightarrow \mathit{Just}\ (t +\!\!+\ c, e, s)\\
&\quad\quad v \qquad\quad \rightarrow \mathit{Just}\ (c, e, (\mathit{Value}\ v) : s)\\
&((\mathit{Close}\ c') : c, e, s) \ \rightarrow\\
&\quad \mathit{Just}\ (c, e, (\mathit{Value}\ (\mathit{Clo}\ c'\ e)) : s)
\end{aligned}
$$

Environment is a local data structure. In other words, environment of subterms should not be used for the whole term so we do not need to push subterm environments to the term environment. This means that environment of a term must be used only for that term. In the following rules, we just get rid of the environment of the subterm (denoted by *e*) in the *Return* rule because the remaining code of the whole term is independent of the contents of *e*:

$$
\begin{aligned}
&(\mathit{Apply} : c, e, (\mathit{Value}\ v) : (\mathit{Value}\ (\mathit{Clo}\ c'\ e')) : s) \rightarrow\\
&\quad \mathit{Just}\ (c', v : e', (\mathit{Code}\ c) : (\mathit{Env}\ e) : s)\\
&(\mathit{Return} : c, e, v : (\mathit{Code}\ c') : (\mathit{Env}\ e') : s) \qquad \rightarrow\\
&\quad \mathit{Just}\ (c', e', v : s)
\end{aligned}
$$

Another example of this happens in the *Let* rules, where we do not need the value v anymore in the term when we finish evaluating the subterm (i.e. when we reach *EndLet*). So, we get rid of v and continue the evaluation:

$$
\begin{aligned}
&(\mathit{Let} : c, e, (\mathit{Value}\ v) : s) \rightarrow\\
&\quad \mathit{Just}\ (c, v : e, s)\\
&(\mathit{EndLet} : c, v : e, s) \qquad \rightarrow\\
&\quad \mathit{Just}\ (c, e, s)\\
&(\mathit{If} : c, e, (\mathit{Value}\ (\mathit{Clo}\ c3\ e3)) : (\mathit{Value}\ (\mathit{Clo}\ c2\ e2)) : (\mathit{Value}\ (\mathit{BoolVal}\ v)) : s) \rightarrow\\
&\quad \mathbf{if}\ (v \equiv \mathit{True})\ \mathbf{then}\\
&\quad\quad \mathit{Just}\ (c2, e2, (\mathit{Code}\ c) : (\mathit{Env}\ e) : s)\\
&\quad \mathbf{else}\\
&\quad\quad \mathit{Just}\ (c3, e3, (\mathit{Code}\ c) : (\mathit{Env}\ e) : s)
\end{aligned}
$$

In the Fix λf.t, we need to substitute f with Fix λ f, so we need to put Fix λ f.t in the environment. In the big-step machine, we explained that we cannot put a code in the environment unless we make a wrap it in a closure. But in the CES machine, this simply does not work. In fact, the closure for wrapping the fix term should be different from the one used for abstractions because we need to make sure Fix comes before the surrounding application. To address this problem, we define a different closure constructor named CloFix for fix terms. So, whenever a CloFix value is accessed in the environment, we do not put it in the stack as we do for redular Clo terms. Instead, we insert its content term in the front of the current code. This makes it possible to evaluate fix term before the surrounding application:

$$
\begin{aligned}
&(\mathit{Fix} : c, e, (\mathit{Value}\ (\mathit{Clo}\ (\mathit{Close}\ c' : c'')\ e')) : s) \qquad\qquad \rightarrow\\
&\quad \mathit{Just}\ (c, e, (\mathit{Value}\ (\mathit{Clo}\ (c' +\!\!+\ c'')\ ((\mathit{CloFix}\ (\mathit{Close}\ ((\mathit{Close}\ c' : c'')) : [\mathit{Fix}])) : (\mathit{skipFixEnvs}\ e)))) : s)
\end{aligned}
$$

$$
\begin{array}{l}
\quad\quad (Add : c, e, (Value\ (IntVal\ v2)) : (Value\ (IntVal\ v1)) : s) \rightarrow \\
\quad\quad\quad Just\ (c, e, (Value\ (IntVal\ (I.intAdd\ v1\ v2)) : s)) \\
\quad\quad (Sub : c, e, (Value\ (IntVal\ v2)) : (Value\ (IntVal\ v1)) : s) \rightarrow \\
\quad\quad\quad Just\ (c, e, (Value\ (IntVal\ (I.intSub\ v1\ v2)) : s)) \\
\quad\quad (Mul : c, e, (Value\ (IntVal\ v2)) : (Value\ (IntVal\ v1)) : s) \rightarrow \\
\quad\quad\quad Just\ (c, e, (Value\ (IntVal\ (I.intMul\ v1\ v2)) : s)) \\
\quad\quad (Div : c, e, (Value\ (IntVal\ v2)) : (Value\ (IntVal\ v1)) : s) \rightarrow \\
\quad\quad\quad Just\ (c, e, (Value\ (IntVal\ (I.intDiv\ v1\ v2)) : s)) \\
\quad\quad (Nand : c, e, (Value\ (IntVal\ v2)) : (Value\ (IntVal\ v1)) : s) \rightarrow \\
\quad\quad\quad Just\ (c, e, (Value\ (IntVal\ (I.intNand\ v1\ v2)) : s)) \\
\quad\quad (Lt : c, e, (Value\ (IntVal\ v2)) : (Value\ (IntVal\ v1)) : s) \rightarrow \\
\quad\quad\quad Just\ (c, e, (Value\ (BoolVal\ (I.intLt\ v1\ v2)) : s)) \\
\quad\quad (Eq : c, e, (Value\ (IntVal\ v2)) : (Value\ (IntVal\ v1)) : s) \rightarrow \\
\quad\quad\quad Just\ (c, e, (Value\ (BoolVal\ (I.intEq\ v1\ v2)) : s)) \\
\quad\quad otherwise \rightarrow Nothing \\
loop :: State \rightarrow State \\
loop\ state = \\
\quad \mathbf{case}\ step\ state\ \mathbf{of} \\
\quad\quad Just\ state' \rightarrow loop\ state' \\
\quad\quad Nothing \rightarrow state \\
eval :: S.Term \rightarrow Value \\
eval\ t = \mathbf{case}\ loop\ (compile\ t, [\,], [\,])\ \mathbf{of} \\
\quad (\_, \_, Value\ v : \_) \rightarrow v
\end{array}
$$

Now, consider the following recursive program in the core lambda language. Evaluation of the recursive calls results in creation of four environments (see Figure 1). The inner three environments correspond to the three recursive (fix) calls, each of which has an access to the outer environment variable y:

```
let y = 0
in
    app(
        fix (
            abs (f:->(Int,Int).
                abs (x:Int.
                    if =(x,y) then
                        1
                    else
                        *(x, app (f,-(x,1)))
                    fi
```

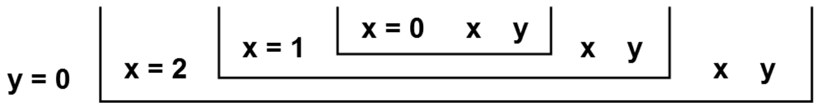

Figure 1: Evaluation of the recursive program results in creation of four environments. The inner three environments correspond to the three recursive (fix) calls, each of which has an access to the outer environment variable y.

```
                  )
              )
          ), 2
      )
end
```

In each of the three recursive calls, the environment should contain only the current value of x and the value of the global variable y otherwise the access to y is incorrectly made. Note that the contents of previous recursive calls are all contained in the current environment so y will not be accesses correctly. To fix this, we defined a function named *skipFixEnvs*, which skips all contents of the current environment related to previous recursive calls and returns the rest of contents related to global environment:

```
skipFixEnvs :: Env → Env
skipFixEnvs e = case reverse e of
  er → take (skipFixWorker er 0) er
skipFixWorker :: Env → Int → Int
skipFixWorker [ ] i = i
skipFixWorker (e : es) i = case e of
  (CloFix _) → i
  otherwise → skipFixWorker es (i + 1)
```

4 CPS transformation

Our first attempt for adding types to the CPS is based on [1]. We have also implemented the solution proposed in [2] as the second attempt.

4.1 CPS attempt 1

Through description of the rules defined in this solution can be found in section 4.2.

```
module CPS where
import AbstractSyntax
import qualified TypeCheck as T
toCPS :: Type → Term → Term
```

```
toCPS o t = case t of
  Abs x τ t   → Abs k (typeToCPS o (TypeArrow τ (typeOf t)))
                    (App (Var k) (Abs x τ (toCPS o t)))
                  where k = getFresh (fv t) "k"
  Var x       → Abs k o (App (Var k) t)
                  where k = getFresh [x] "k"
  IntConst x → Abs "k" o (App (Var "k") t)
  Tru         → Abs "k" o (App (Var "k") t)
  Fls         → Abs "k" o (App (Var "k") t)
  App (Fix (Abs f τ1 (Abs x τ2 t1))) t2 → (Abs k o
                                            (App
                                              (Abs v1 (typeToCPS o (typeOf t1))
                                                (App (toCPS o t2)
                                                  (Abs v2 (typeToCPS o (typeOf t2))
                                                    (App (App (Var v1) (Var v2)) (Var k))
                                                  )
                                                )
                                              ) (Fix (Abs f τ1 (Abs x τ2 (toCPS o t1))))
                                            )
                                          )
                                          where k = getFresh (fv t) "k"
                                                v1 = getFresh (fv t) "v1"
                                                v2 = getFresh (fv t) "v2"
  App t1 t2   → Abs k (typeToCPS o (typeOfResult t1))
                  (App (toCPS o t1)
                    (Abs v1 (typeToCPS o (typeOf t1))
                      (App (toCPS o t2)
                        (Abs v2 (typeToCPS o (typeOf t2))
                          (App (App (Var v1) (Var v2)) (Var k))))))
                  where k = getFresh (fv t) "k"
                    v1 = getFresh (fv t) "v1"
                    v2 = getFresh (fv t) "v2"
```

Note that we can define *Let* based on abstraction and application. To deal with *Let* in CE3R machine, we change the *Let* term to an application on an abstraction term in CPS transformation phase. So, we get rid of the *Let* without changing the semantics of the program:

$$Let\ s\ t_1\ t_2 \rightarrow toCPS\ o\ (App\ (Abs\ s\ (typeOf\ t_1)\ t_2)\ t_1)$$

Similar to CES machine, in the evaluation of *If* t_1 t_2 t_3 term, we need to postpone the evaluation of t_2 and t_3 after evaluating the t_1. Note that CPS itself is not an evaluator like

CES and it just produces an intermediate code. At first, it seems that handling *If* is simpler in CPS rather than in the evaluators. Since CPS machine wraps every terms in a lambda and passes a continuation to them, we can simply postpone the evaluation to the point we want by ignoring the continuation.

```
  If t1 t2 t3    → Abs k o
                     (App (toCPS o t1)
                        (Abs v1 o
                           (App (toCPS o t2)
                              (Abs v2 o
                                 (App (toCPS o t3)
                                    (Abs v3 o
                                       (App (Var k) (If (Var v1) (Var v2) (Var v3)))
                                    )
                                 )
                              )
                           )
                        )
                     )
                     where k = getFresh (fv t) "k"
                           v1 = getFresh (fv t) "v1"
                           v2 = getFresh (fv t) "v2"
                           v3 = getFresh (fv t) "v3"
  IntAdd a b  → toCPSBinOp o (IntAdd) a b
  IntSub a b  → toCPSBinOp o (IntSub)  a b
  IntMul a b  → toCPSBinOp o (IntMul) a b
  IntDiv a b  → toCPSBinOp o (IntDiv)  a b
  IntNand a b → toCPSBinOp o (IntNand) a b
  IntEq a b   → toCPSBinOp o (IntEq)   a b
  IntLt a b   → toCPSBinOp o (IntLt)   a b
toCPSBinOp o op t1 t2 = Abs k (typeToCPS o (typeToCPS o TypeInt))
  (App (toCPS o t1)
                      (Abs v1 (typeToCPS o (typeOf t1))
                         (App (toCPS o t2)
                            (Abs v2 (typeToCPS o (typeOf t2))
                               (App (Var k) (op (Var v1) (Var v2)))))))
                 where k = getFresh (fv t1 ++ fv t2) "k"
                       v1 = getFresh (fv t1 ++ fv t2) "v1"
                       v2 = getFresh (fv t1 ++ fv t2) "v2"
getFresh :: [String] → String → String
getFresh avoids x = if x ∈ avoids
```

```
                    then getFresh avoids ('a' : x)
                    else x
typeOf :: Term → Type
typeOf t = TypeBool
typeOfResult :: Term → Type
typeOfResult t = TypeBool
typeToCPS :: Type → Type → Type
typeToCPS o τ =
  case τ of
    TypeInt       → TypeArrow (TypeArrow TypeInt o) o
    TypeBool      → TypeArrow (TypeArrow TypeBool o) o
    TypeArrow a b → TypeArrow (TypeArrow (typeToCPS a o) (TypeArrow (typeToCPS b o) o)) o
toCPS' :: Term → Term
toCPS' t = toCPS (T.typeCheck t) t
```

4.2 CPS attempt 2

This CPS transformation works on tagged terms. The reasoning behind using these as opposed to the untagged terms is that during the typing of the continuations, we won't need to search for a term's type since it will be tagged on to the term. We also include the type checker to allow us to test that the CPS transform is properly typing the term.

```
module Syntax.CPS where
import qualified TypeCheck.TypeCheckB as T
import Syntax.SyntaxC
import qualified Syntax.SyntaxB
import Syntax.ExtendedTypes
```

As we proceed, we will often assign the type *TypeQ* to terms in the CPS transform. This is the type we use when we are not interested in the type of the term. We choose to type just the terms that are needed in the type checker. (The type checker needs just the abstractions to be typed, not applications or variables, etc.) Not having to worry about the type of some terms makes the transform simpler, but has the drawback that we cannot perform the CPS transform directly to the CPS term (since this term contains sub terms of *TypeQ* which propagate to terms that we do need for the type checker).

If we do want all of the terms to be typed, we can convert the tagged syntax to untagged and then to tagged again. This has the effect of removing the *TypeQ*'s and then retyping the terms correctly.

Before we start the CPS transform, we need to be able to generate fresh variables for our continuations. This is not a very smart way, but simple, we keep prefixing an initial guess with underscores until we find one that is fresh in a list of variables [*avoid*] .

We also define this behavior if we are given a term instead of a list of variables to avoid.

$$
\begin{aligned}
&\mathit{fresh} :: \mathit{String} \to [\mathit{String}] \to \mathit{String}\\
&\mathit{fresh}\ x\ \mathit{avoids} = \textbf{if}\ x \in \mathit{avoids}\ \textbf{then}\ \mathit{fresh}\ (\text{'\_'} : x)\ \mathit{avoids}\ \textbf{else}\ x\\
&\mathit{freshIn}\ i\ t = \mathit{fresh}\ i\ (\mathit{fv}\ t)
\end{aligned}
$$

During the CPS transform we will often need to access a variable, and do not care if we tag its type correctly, v does this for us given the variable name. We also will need to apply to terms together, where we don't care about the type of the result, $< * >$ does this.

$$
\begin{aligned}
&v\ x = V\ (q, x)\\
&t_1 < * > t_2 = \mathit{App}\ (q, t_1, t_2)\\
&\textbf{infixl}\ 5 < * >
\end{aligned}
$$

We also define $* >$ and $*\tilde{\ } >$ which are used in the typing of the CPS terms. These are related to the $*$ and $'$ operators in [1]. The author gives the type of the entire term rather than the continuations, however.

$$
\begin{aligned}
&a\ \tilde{\ } > b = \mathit{TypeArrow}\ a\ b\\
&\tau * > \mathit{ans} = (\mathit{typeStar}\ \mathit{ans}\ \tau)\\
&\tau *\tilde{\ } > \mathit{ans} = (\mathit{typeStar}\ \mathit{ans}\ \tau)\ \tilde{\ } > \mathit{ans}\\
&\textbf{infixl}\ 8 * >, *\tilde{\ } >\\
&\textbf{infixr}\ 5\ \tilde{\ } >\\
&\mathit{typeStar}\ o\ a = \textbf{case}\ a\ \textbf{of}\\
&\quad \mathit{TypeArrow}\ b\ c \to (\mathit{typeStar}\ o\ b)\ \tilde{\ } > ((\mathit{typeStar}\ o\ c)\ \tilde{\ } > o)\ \tilde{\ } > o\\
&\quad \mathit{otherwise} \to a
\end{aligned}
$$

Some shortcuts: *TypeQ* is the type we use when we are not interested in the type. *getTag* returns the type of its argument.

$$
\begin{aligned}
&q = \mathit{TypeQ}\\
&\mathit{tauOf} = \mathit{getTag}
\end{aligned}
$$

The rules we used for the CPS transform on terms:

$$
(\text{CPS-Const-Var})\frac{[[x :: \tau]]}{\lambda k :: \tau *\tilde{\ } > o \quad . \quad kx}
$$

$$
(\text{CPS-Abs})\frac{\lambda x :: \alpha \quad . \quad t :: \beta}{\lambda k :: (\alpha \Rightarrow \beta) *\tilde{\ } > o \quad . \quad k(\lambda \tilde{x} :: a * > o \quad . \quad [[t]])}
$$

$$
(\text{CPS-App})\frac{[[t_1 t_2]]}{\lambda k.[[t_1]](\lambda v_1.[[t_2]](\lambda v_2.v_1 v_2 k))}
$$

$$(\text{CPS-PrimOp})\frac{[[op(t_1 t_2)]]}{\lambda k.[[t_1]](\lambda v_1.[[t_2]](\lambda v_2.k(op(v_1 v_2))))}$$

$$(\text{CPS-If})\frac{[[\text{if } t_1 \text{ then } t_2 \text{ else } t_3]]}{\lambda k.\ [[t_1]]\ (\lambda v_1.\ \text{If } v_1 \text{ then } [[t_2]] \text{ else } [[t_3]]\ )}$$

$$(\text{CPS-Let})\frac{[[\text{Let } x = s \text{ in } t]]}{\lambda k.\ [[s]]\ (\lambda v_1.\ (\text{Let } x = v_1 \text{ in } [[t]])\ k)}$$

$$(\text{CPS-Let-Fix})\frac{[[\text{Let } y = \text{Fix}(\lambda f.\lambda x.N) \text{ in } M]]}{\lambda k.\text{Let } y = \text{Fix}(\ \lambda f.\lambda x.\lambda k_f.[[t_1]](\lambda v_1.k_f v_1)) \text{ in } [[M]] \text{ k}}$$

$$(\text{CPS-App-Fix})\frac{[[\ \text{Fix } (\lambda f.\lambda x.t_1)\ t_2]]}{\lambda k.(\lambda f.[[f t_2]]k)\text{Fix } (\lambda f.\lambda x.\lambda k_f.\ [[t_1]]\ (\lambda v_1.k_f v_1))}$$

The rules we used for the CPS transform on types:

$$(\text{Const x})\frac{answer :: o \quad x :: \tau}{k :: \tau *\tilde{\ } > o}$$

$$(\text{Var x})\frac{answer :: o \quad x :: \tau}{k :: \tau *\tilde{\ } > o}$$

$$(\text{Abs } \lambda x.t)\frac{answer :: o \quad x :: \alpha \quad t :: \beta}{k :: (\alpha \Rightarrow \beta) *\tilde{\ } > o \quad \tilde{x} :: \alpha * > o}$$

$$(\text{App } t_1\ t_2)\frac{answer :: o \quad t_1 :: \alpha \Rightarrow \beta \quad t_2 :: \alpha}{k :: \beta *\tilde{\ } > o \quad v_1 :: (\alpha \Rightarrow \beta) * > o \quad v_2 :: \alpha * > o}$$

$$(\text{PrimOp } t_1\ t_2)\frac{answer :: o \quad t_1 :: \alpha \quad t_2 :: \beta \quad (op \quad t_1 \quad t_2) :: \delta}{k :: \delta *\tilde{\ } > o \quad v_1 :: \alpha * > o \quad v_2 :: \beta * > o}$$

$$(\text{If } t_1\ t_2\ t_3)\frac{answer :: o \quad t_1 :: \alpha \quad t_2 :: \beta \quad t_3 :: \beta}{k :: \beta *\tilde{\ } > o \quad v_1 :: \alpha * > o}$$

$$(\text{Let } x = s \text{ in } t)\frac{answer :: o \quad t :: \alpha \quad s :: \beta}{k :: \alpha *\tilde{\ } > o \quad v_1 :: \beta *\tilde{\ } > o}$$

We only have *Fix* working for recursive functions applied to some argument. We also did not get fix to type correctly:

$toCPS :: Type \rightarrow Term \rightarrow Term$
$toCPS\ o\ t = \textbf{case}\ t\ \textbf{of}$

```
V (τ,x) →    Abs (q,k,k_tau,v k <∗> t)
               where k = ("kv_" ++ x) `freshIn` t
                     k_tau = τ ∗~> o
B (τ,x)  →   Abs (q,k,k_tau,v k <∗> t)
               where k = "kb_" ++ (show x)
                     k_tau = τ ∗~> o
I (τ,x)  →   Abs (q,k,k_tau,v k <∗> t)
               where k = "ki_" ++ (show x)
                     k_tau = τ ∗~> o
Abs (τ,x,x_tau,t) → Abs (q,k,k_tau
  ,              v k <∗> Abs (q,x,x_star,(toCPS o t)
  ))
                where k = ("ka_" ++ x) `freshIn` t
                          k_tau = τ ∗~> o
                          x_star = x_tau ∗> o
Let (let_tau,y,Fix (τ,Abs (a_f_tau,f,f_tau,Abs (a_x_tau,x,x_tau,t1))),t2)
  →
    Abs (q,k,k_tau,
      Let (y_tau,y,Fix (τ,
            Abs (a_f_tau,f,f_tau,
            Abs (a_x_tau,x,x_tau,
            Abs (q,k_f,k_f_tau,
            (toCPS o t1) <∗> Abs (q,v1,v1_tau,v k_f <∗> v v1)))))
         ,toCPS o t2 <∗> v k))
  where k = "k" `freshIn` t
        v1 = "v1" `freshIn` t
        k_f = "k_f" `freshIn` t
        f_star = f_tau ∗~> o
        k_tau = (tauOf t) ∗> o
        y_tau = τ ∗> o
        v1_tau = (tauOf t) ∗> o
        k_f_tau = (tauOf t) ∗> o
        app_tau = (modusPonens f_tau (tauOf t2)) ∗~> o
App (app_tau,Fix (τ,Abs (a_f_tau,f,f_tau,Abs (a_x_tau,x,x_tau,t1))),t2)
  →
    Abs (q,k,k_tau,
      Abs (q,f,f_star,toCPS o (App (app_tau,V (f_star,f),t2)) <∗> v k)
       <∗> Fix (τ,
        Abs (a_f_tau,f,f_tau,
        Abs (a_x_tau,x,x_tau,
```

```
            Abs (q, k_f, k_f_tau,
            (toCPS o t1) <*> Abs (q, v1, v1_tau, v k_f <*> v v1))))))
   where k = "k" `freshIn` t
            v1 = "v1" `freshIn` t
            k_f = "k_f" `freshIn` t
            f_star = f_tau ~> TypeBool
            k_tau = (tauOf t) *> o
            v1_tau = (tauOf t) *> o
            k_f_tau = (tauOf t) *> o
            app_tau = (modusPonens f_tau (tauOf t2)) *~> o
App (τ, t1, t2) → Abs (q, k, k_tau
               , (toCPS o t1) <*> Abs (q, v1, v1_tau
               , (toCPS o t2) <*> Abs (q, v2, v2_tau
               , V (q, v1) <*> V (q, v2) <*> V (q, k)
               )))
         where k = "k" `freshIn` t
               v1 = "v1" `freshIn` t
               v2 = "v2" `freshIn` t
               v1_tau = (tauOf t1 *> o)
               v2_tau = (tauOf t2 *> o)
               k_tau = (modusPonens (tauOf t1) (tauOf t2)) *~> o
PrimOp (τ, n, op, τs@(τ1 : τ2 : taur : tauss), (t1 : t2 : ts))
   → Abs (q, k, k_tau
   , (toCPS o t1) <*> Abs (q, v1, v1_tau
   , (toCPS o t2) <*> Abs (q, v2, v2_tau
   , v k                    <*> PrimOp (τ, n, op, τs, [v v1, v v2])
   )))
   where k = "k"       `freshIn` t
     v1 = "v1" `freshIn` t
     v2 = "v2" `freshIn` t
     v1_tau = τ1 *> o
     v2_tau = τ2 *> o
     k_tau = taur *~> o
If (τ, t1, t2, t3) → Abs (q, k, k_tau
            , (toCPS o t1) <*> Abs (q, v1, v1_tau
            , (toCPS o t2) <*> Abs (q, v2, v2_tau
            , (toCPS o t3) <*> Abs (q, v3, v3_tau
            , v k <*> If (q
                          , v v1
                          , v v2
```

```
                    ,v v3
                    )
              ))))
              where k = "k" `freshIn` t
                v1 = "v1" `freshIn` t
                v2 = "v2" `freshIn` t
                v3 = "v3" `freshIn` t
                k_tau = (tauOf t2) *~> o
                v1_tau = (tauOf t1) *> o
                v2_tau = (tauOf t2) *> o
                v3_tau = (tauOf t3) *> o
```

In many cases, when CPS transforming a term by hand it seems awkward to CPS transform the conditional that it checks. We can do this in this way:

```
If (τ, t1, t2, t3) → Abs (q, k, k_tau,
  If (τ
    , t1
    , (toCPS o t2) <*> v k
    , (toCPS o t3) <*> v k
    )
    )
  where k = "k" `freshIn` t
    k_tau = (tauOf t2) *~> o
```

We can also transform the conditional. This is the version used, and the one shown in the rules.

```
If (τ, t1, t2, t3) → Abs (q, k, k_tau
  , (toCPS o t1) <*> Abs (q, v1, v1_tau
  , If (q
    , v v1
    , (toCPS o t2) <*> v k
    , (toCPS o t3) <*> v k
    )
  ))
  where k = "k" `freshIn` t
    v1 = "v1" `freshIn` t
    k_tau = (tauOf t2) *~> o
    v1_tau = (tauOf t1) *> o
```

We can treat let like a macro and apply the substitution right away.

```
Let (τ, x, t2, t1) → toCPS o (subst x t2 t1)
```

A CPS transform for *Let* that keeps the *Let* in the term. This is the let transform that is used and in the rules.

```
Let (τ, x, s, t1) → Abs (q, k, k_tau
                       , (toCPS o s) <*> Abs (q, v1, v1_tau
                       , Let (x_tau, x, v v1, (toCPS o t1)) <*> v k
                       ))
  where k = "k" `freshIn` t
    v1 = "v1" `freshIn` t
    v2 = "v2" `freshIn` t
    v1_tau = (tauOf s) *> o
    x_tau = (tauOf s) *> o
    k_tau = (tauOf t1) *~> o
```

We can treat it like an applications with some of the redexes done.

```
Let (τ, x, t2, t1) → toCPS o (subst x t2 t1)
Let (τ, x, s, t1) → Abs (q, k, k_tau,
  (toCPS o s) <*> Abs (q, v1, v1_tau,
  Abs (q, x, x_tau, (toCPS o t1)) <*> v v1 <*> v k
  ))
  where k = "k" `freshIn` t
        v1 = "v1" `freshIn` t
        k_tau = (tauOf t1) *~> o
        v1_tau = (tauOf s) *> o
        x_tau = (tauOf s) *> o
```

This was the first attempt (which does not work) treating toCPS as if it commutes with let.

```
Let (τ, x, s, t1) → Abs (q, k, k_tau,
  Let (τ, x, (toCPS o s), (toCPS o t1)) <*> v k
  )
  where k = "k" `freshIn` t
        v1 = "v1" `freshIn` t
        v1_tau = (tauOf s) *> o
```

Here are some failed attempts of *Fix*. The first one tries to do it in the most obvious way. The second tries to do it like we did for applications. The third is an attempt of CPS transforming the fix point combinator. The fourth looks for the functions it is fixing, and replaces any call to this function with a call that applies it to the continuation first.

```
Fix (τ, t1)
  → Abs (q, k, k_tau,
```

```
    ((Fix (τ, toCPS o t1))) <*> v k)
  where k = "k" `freshIn` t
    k_tau = τ *~> o
Fix (τ, t1) → Abs (q, k, k_tau,
  (toCPS o t1) <*> Abs (q, v1, v1_tau,
  Abs (q, x, x_tau,
    Abs (q, k_f, k_f_tau,
  i Abs (q, v31, v31_tau,
    (v x <*> v x) <*> (v v31 <*> v v1 <*> v k_f))))
  Abs (q, x, x_tau,
    Abs (q, k_f, k_f_tau,
    Abs (q, v41, v41_tau, (
    v x <*> v x) <*> (v v41 <*> v v1 <*> v k_f)
  ))))
Fix (τ, Abs (a_f_tau, f, f_tau, Abs (a_x_tau, x, x_tau, t1)))
  →
    Fix (τ,
      Abs (a_f_tau, f, f_tau, Abs (a_x_tau, x_f, k_f_tau, vAbs (q, k, k_tau
      , k_f <*> (toCPS o t1')))
        ) <*> v k)
  where t1' = subst f (App (k_tau, (v f), (v k))) t1
    k = "k" `freshIn` t
    k_f = "k_f" `freshIn` t
    k_tau = (tauOf t1) *> o
    k_f_tau = o
```

Some functions to drive the CPS transform. *toCPS′* carries out the CPS transform of a term. *toCPSe* carries out the CPS transform and attaches the identity as the initial continuations.

Note: In case the entire program is a concrete type we apply it to the identity function in order to evaluate it. But if the program is a function, then we just return the CPS transform. If we do try to apply the identity in the same way we run into a type error. Since we also can't evaluate anything till we get further input, we leave it to the user to supply the correct type for the argument.

```
toCPS' t = case T.typing T.Empty t of
  Just τ → do tt ← tagTerm T.Empty t
    return $ toCPS τ tt
  Nothing → error $ "error \n" ++ show t
toCPSe t = case T.typing T.Empty t of
  Just τ@(TypeArrow α beta)
```

$\rightarrow$ **do** *tt* $\leftarrow$ *tagTerm T.Empty t*
return \$ *toCPS* τ *tt*
Just $\tau \rightarrow$ **do** *tt* $\leftarrow$ *tagTerm T.Empty t*
return \$ (*toCPS* τ *tt*) $< * >$ *Abs* (q, "a", τ, *V* (τ, "a"))

Finally, a function that checks that the CPS transform typed the term correctly follows:

checkType t = **do** $t' \leftarrow$ *toCPS'* *t*
T.typing T.Empty (*stripTags* t')

5 CE3R compiler and virtual machine

CE3R stands for Code, Environment and 3 Registers. Like the CES machine, the source code is first compiled to an intermediate code. The difference between the CES and CE3R machines is that CE3R is called on the CPS transformation. This transformation helps us reduce the amount of stack we need for evaluation and in fact, makes it possible to evaluate the intermediate code using only three registers. In CE3R, terms inside the Code data structure include integers that refer to the registers in which the result of the Code will be in.

```
module CE3RMachine where
import qualified CPS as C
import qualified DeBruijn as S
import qualified IntegerArithmetic as I
data Inst = Int Integer Integer
  | Bool Integer Bool
  | Add
  | Sub
  | Mul
  | Div
  | Nand
  | Eq
  | Lt
  | Access Integer Int
  | Close        Integer Code
  | App1
  | App2
  | If
  | Fix
  deriving Eq
instance Show Inst where
```

```
  show (Int j i)    = "Int" ++ show j ++ " " ++ show i
  show (Bool j b)   = "Bool" ++ show j ++ " " ++ show b
  show Add          = "Add"
  show Sub          = "Sub"
  show Mul          = "Mul"
  show Div          = "Div"
  show Nand         = "Nand"
  show Eq           = "Eq"
  show Lt           = "Lt"
  show (Access j i) = "~" ++ show j ++ " " ++ show i
  show (Close j c)  = "Close" ++ show j ++ " " ++ show c
  show App1         = "App1"
  show App2         = "App2"
  show If           = "If"
  show Fix          = "Fix"
type Code = [Inst]
data Value = BoolVal Bool
   | IntVal Integer
   | Clo Code Env
   | CloFix Code
   | Null
  deriving Eq
instance Show Value where
  show (BoolVal b) = show b
  show (IntVal i)  = show i
  show (Clo c e)   = "Clo " ++ show c ++ " " ++ show e
  show (CloFix c)  = "Clo " ++ show c
  show Null        = "Null"
type Env = [Value]
type Registers = (Value, Value, Value)
type State = (Code, Env, Registers)
compAtom :: Integer → S.Term → Code
compAtom j t = case t of
                 S.Tru                    → [Bool j True]
                 S.Fls                    → [Bool j False]
                 S.IntConst i             → [Int j i]
                 S.Var i                  → [Access j i]
                 S.Abs τ (S.Abs tau′ t)   → [Close j (compile t)]
                 S.Abs τ t                → [Close j (compile t)]
compile :: S.Term → Code
```

$$\begin{array}{l} \textit{compile } t = \\ \quad \textbf{case } t \textbf{ of} \end{array}$$

At first, it seems that we need four different registers to evaluate *App* (*If v1 v2 v3*) *k*, i.e. three registers for *If* and one for *k*. However, note that we do not need all registers at once, i.e. we can reuse the registers. So, we first calculate *If* using all three registers and we put the result in one of the used registers, then we put *k* in another register and finllay we do *App1*.

$$\begin{array}{l} \textit{S.App } k \ (\textit{S.If } t_1 \ t_2 \ t_3) \rightarrow \textit{compAtom } 1 \ t_1 +\!\!+ \textit{compAtom } 2 \ t_2 +\!\!+ \\ \qquad\qquad\qquad\qquad\qquad \textit{compAtom } 3 \ t_3 +\!\!+ [\textit{If}] +\!\!+ \textit{compAtom } 1 \ k +\!\!+ [\textit{App1}] \end{array}$$

In CPS, we change any operator that needs two arguments (like *app*, *IntAdd*, etc) to an operator that needs three arguments where the extra argument is the continuation. Thus, instead of looking for an operator, we look for the application of the operator on its arguments as well as the continuation. Therefore, we have rules like the following:

$$\begin{array}{lcl}
\textit{S.App } (\textit{S.App } t_1 \ t_2) \ t_3 & \rightarrow & \textit{compAtom } 1 \ t_1 +\!\!+ \textit{compAtom } 2 \ t_2 +\!\!+ \textit{compAtom } 3 \ t_3 +\!\!+ [\textit{App2}] \\
\textit{S.App } t_1 \ (\textit{S.IntAdd } t_2 \ t_3) & \rightarrow & \textit{compAtom } 1 \ t_1 +\!\!+ \textit{compAtom } 2 \ t_2 +\!\!+ \textit{compAtom } 3 \ t_3 +\!\!+ [\textit{Add}] \\
\textit{S.App } t_1 \ (\textit{S.IntSub } t_2 \ t_3) & \rightarrow & \textit{compAtom } 1 \ t_1 +\!\!+ \textit{compAtom } 2 \ t_2 +\!\!+ \textit{compAtom } 3 \ t_3 +\!\!+ [\textit{Sub}] \\
\textit{S.App } t_1 \ (\textit{S.IntMul } t_2 \ t_3) & \rightarrow & \textit{compAtom } 1 \ t_1 +\!\!+ \textit{compAtom } 2 \ t_2 +\!\!+ \textit{compAtom } 3 \ t_3 +\!\!+ [\textit{Mul}] \\
\textit{S.App } t_1 \ (\textit{S.IntDiv } t_2 \ t_3) & \rightarrow & \textit{compAtom } 1 \ t_1 +\!\!+ \textit{compAtom } 2 \ t_2 +\!\!+ \textit{compAtom } 3 \ t_3 +\!\!+ [\textit{Div}] \\
\textit{S.App } t_1 \ (\textit{S.IntNand } t_2 \ t_3) & \rightarrow & \textit{compAtom } 1 \ t_1 +\!\!+ \textit{compAtom } 2 \ t_2 +\!\!+ \textit{compAtom } 3 \ t_3 +\!\!+ [\textit{Nand}] \\
\textit{S.App } t_1 \ (\textit{S.IntEq } t_2 \ t_3) & \rightarrow & \textit{compAtom } 1 \ t_1 +\!\!+ \textit{compAtom } 2 \ t_2 +\!\!+ \textit{compAtom } 3 \ t_3 +\!\!+ [\textit{Eq}] \\
\textit{S.App } t_1 \ (\textit{S.IntLt } t_2 \ t_3) & \rightarrow & \textit{compAtom } 1 \ t_1 +\!\!+ \textit{compAtom } 2 \ t_2 +\!\!+ \textit{compAtom } 3 \ t_3 +\!\!+ [\textit{Lt}] \\
\textit{S.App } t_1 \ (\textit{S.Fix } t_2) & \rightarrow & \textit{compAtom } 1 \ t_1 +\!\!+ \textit{compAtom } 2 \ t_2 +\!\!+ [\textit{Fix}] +\!\!+ [\textit{App1}] \\
\textit{S.App } t_1 \ t_2 & \rightarrow & \textit{compAtom } 1 \ t_1 +\!\!+ \textit{compAtom } 2 \ t_2 +\!\!+ [\textit{App1}] \\
t & \rightarrow & \textit{compAtom } 1 \ t
\end{array}$$

$$\begin{array}{l}
\textit{step} :: \textit{State} \rightarrow \textit{Maybe State} \\
\textit{step state} = \\
\quad \textbf{case } \textit{state } \textbf{of}
\end{array}$$

$$\begin{array}{ll}
((\textit{Int } 1 \ i) : c, e, (_, v2, v3)) & \rightarrow \textit{Just } (c, e, (\textit{IntVal } i, v2, v3)) \\
((\textit{Int } 2 \ i) : c, e, (v1, _, v3)) & \rightarrow \textit{Just } (c, e, (v1, \textit{IntVal } i, v3)) \\
((\textit{Int } 3 \ i) : c, e, (v1, v2, _)) & \rightarrow \textit{Just } (c, e, (v1, v2, \textit{IntVal } i)) \\
((\textit{Bool } 1 \ b) : c, e, (_, v2, v3)) & \rightarrow \textit{Just } (c, e, (\textit{BoolVal } b, v2, v3)) \\
((\textit{Bool } 2 \ b) : c, e, (v1, _, v3)) & \rightarrow \textit{Just } (c, e, (v1, \textit{BoolVal } b, v3)) \\
((\textit{Bool } 3 \ b) : c, e, (v1, v2, _)) & \rightarrow \textit{Just } (c, e, (v1, v2, \textit{BoolVal } b)) \\
((\textit{Access } 1 \ i) : c, e, (_, v2, v3)) & \rightarrow \textit{Just } (c, e, (e \,!!\, i, v2, v3)) \\
((\textit{Access } 2 \ i) : c, e, (v1, _, v3)) & \rightarrow \textit{Just } (c, e, (v1, e \,!!\, i, v3)) \\
((\textit{Access } 3 \ i) : c, e, (v1, v2, _)) & \rightarrow \textit{Just } (c, e, (v1, v2, e \,!!\, i)) \\
((\textit{Close } 1 \ c') : c, e, (_, v2, v3)) & \rightarrow \textit{Just } (c, e, ((\textit{Clo } c' \ e), v2, v3)) \\
((\textit{Close } 2 \ c') : c, e, (v1, _, v3)) & \rightarrow \textit{Just } (c, e, (v1, (\textit{Clo } c' \ e), v3))
\end{array}$$

$$
\begin{array}{ll}
((\mathit{Close}\ 3\ c') : c, e, (v1, v2, \_)) & \rightarrow \mathit{Just}\ (c, e, (v1, v2, (\mathit{Clo}\ c'\ e))) \\
(\mathit{App1} : c, e, ((\mathit{Clo}\ c'\ e'), v2, \_)) & \rightarrow \mathit{Just}\ (c', v2 : e', (\mathit{Null}, \mathit{Null}, \mathit{Null})) \\
(\mathit{App2} : c, e, ((\mathit{Clo}\ c'\ e'), v2, v3)) & \rightarrow \mathit{Just}\ (c', v3 : v2 : e', (\mathit{Null}, \mathit{Null}, \mathit{Null})) \\
(\mathit{If} : c, e, (\mathit{BoolVal}\ v, t_2, t_3)) & \rightarrow \\
\quad \mathbf{if}\ v \equiv \mathit{True} & \\
\quad \mathbf{then}\ \mathit{Just}\ (c, e, (\mathit{Null}, t_2, \mathit{Null})) & \\
\quad \mathbf{else}\ \mathit{Just}\ (c, e, (\mathit{Null}, t_3, \mathit{Null})) &
\end{array}
$$

The rule for *Fix* in CE3R machine is similar to the one in the CES machine. We just put the *CloFix* part in one of the registers and the rest is the same:

$$
\begin{array}{ll}
(\mathit{Fix} : c, e, ((\mathit{Clo}\ c'\ e'), (\mathit{Clo}\ ((\mathit{Close}\ \_\ c1) : c1')\ e1), \_)) \rightarrow & \\
\quad \mathit{Just}\ (c, e, ((\mathit{Clo}\ c'\ e'), (\mathit{Clo}\ (c1 +\!\!+ c1')\ ((\mathit{CloFix}\ (\mathit{Close}\ 2\ ((\mathit{Close}\ 2\ c1 : c1')) : [\mathit{Fix}])) : (\mathit{skipFixEnvs}\ e & \\
(\mathit{Add} : c, e, ((\mathit{Clo}\ c'\ e'), \mathit{IntVal}\ v2, \mathit{IntVal}\ v3)) & \rightarrow \\
\quad \mathit{Just}\ (c', (\mathit{IntVal}\ (\mathit{I.intAdd}\ v2\ v3)) : e', (\mathit{Null}, \mathit{Null}, \mathit{Null})) & \\
(\mathit{Sub} : c, e, ((\mathit{Clo}\ c'\ e'), \mathit{IntVal}\ v2, \mathit{IntVal}\ v3)) & \rightarrow \\
\quad \mathit{Just}\ (c', (\mathit{IntVal}\ (\mathit{I.intSub}\ v2\ v3)) : e', (\mathit{Null}, \mathit{Null}, \mathit{Null})) & \\
(\mathit{Mul} : c, e, ((\mathit{Clo}\ c'\ e'), \mathit{IntVal}\ v2, \mathit{IntVal}\ v3)) & \rightarrow \\
\quad \mathit{Just}\ (c', (\mathit{IntVal}\ (\mathit{I.intMul}\ v2\ v3)) : e', (\mathit{Null}, \mathit{Null}, \mathit{Null})) & \\
(\mathit{Div} : c, e, ((\mathit{Clo}\ c'\ e'), \mathit{IntVal}\ v2, \mathit{IntVal}\ v3)) & \rightarrow \\
\quad \mathit{Just}\ (c', (\mathit{IntVal}\ (\mathit{I.intDiv}\ v2\ v3)) : e', (\mathit{Null}, \mathit{Null}, \mathit{Null})) & \\
(\mathit{Nand} : c, e, ((\mathit{Clo}\ c'\ e'), \mathit{IntVal}\ v2, \mathit{IntVal}\ v3)) & \rightarrow \\
\quad \mathit{Just}\ (c', (\mathit{IntVal}\ (\mathit{I.intNand}\ v2\ v3)) : e', (\mathit{Null}, \mathit{Null}, \mathit{Null})) & \\
(\mathit{Eq} : c, e, ((\mathit{Clo}\ c'\ e'), \mathit{IntVal}\ v2, \mathit{IntVal}\ v3)) & \rightarrow \\
\quad \mathit{Just}\ (c', (\mathit{BoolVal}\ (\mathit{I.intEq}\ v2\ v3)) : e', (\mathit{Null}, \mathit{Null}, \mathit{Null})) & \\
(\mathit{Lt} : c, e, ((\mathit{Clo}\ c'\ e'), \mathit{IntVal}\ v2, \mathit{IntVal}\ v3)) & \rightarrow \\
\quad \mathit{Just}\ (c', (\mathit{BoolVal}\ (\mathit{I.intLt}\ v2\ v3)) : e', (\mathit{Null}, \mathit{Null}, \mathit{Null})) & \\
\mathit{otherwise} & \rightarrow \mathit{Nothing}
\end{array}
$$

$$
\begin{array}{l}
\mathit{loop} :: \mathit{State} \rightarrow \mathit{State} \\
\mathit{loop}\ \mathit{state} = \\
\quad \mathbf{case}\ \mathit{step}\ \mathit{state}\ \mathbf{of} \\
\quad\quad \mathit{Just}\ \mathit{state}' \rightarrow \mathit{loop}\ \mathit{state}' \\
\quad\quad \mathit{Nothing} \rightarrow \mathit{state} \\
\mathit{eval} :: \mathit{S.Term} \rightarrow \mathit{Value} \\
\mathit{eval}\ t = \mathbf{case}\ \mathit{loop}\ (\mathit{compile}\ t, [\,], (\mathit{Null}, \mathit{Null}, \mathit{Null}))\ \mathbf{of} \\
\quad (\_, \_, (v, \_, \_)) \rightarrow v \\
\mathit{skipFixEnvs} :: \mathit{Env} \rightarrow \mathit{Env} \\
\mathit{skipFixEnvs}\ e = \mathbf{case}\ \mathit{reverse}\ e\ \mathbf{of} \\
\qquad\qquad \mathit{er} \rightarrow \mathit{take}\ (\mathit{skipFixWorker}\ \mathit{er}\ 0)\ \mathit{er} \\
\mathit{skipFixWorker} :: \mathit{Env} \rightarrow \mathit{Int} \rightarrow \mathit{Int}
\end{array}
$$

```haskell
skipFixWorker [] i = i
skipFixWorker (e : es) i = case e of
  (CloFix _) → i
  otherwise → skipFixWorker es (i + 1)
```

6 Main program

```haskell
module Main (
  main
) where
import TypeCheck
import AbstractSyntax
import System.Environment
import qualified DeBruijn as D
import qualified CESMachine as C
import qualified StructuralOperationalSemantics as S
import qualified NaturalSemanticsWithEnvironmentsClosuresAndDeBruijnIndices as N
import qualified Syntax.CPS as SP
import qualified CPS as P
import qualified Syntax.SyntaxB as B
import qualified Syntax.SyntaxC as X
import qualified CE3RMachine as R
main =
  do
    args ← System.Environment.getArgs
    let [sourceFile] = args
    source ← readFile sourceFile
    putStrLn ("---Input:---")
    putStrLn (source)
    let tokens = scan source
    let term = parse tokens
    putStrLn ("---Term:---")
    putStrLn (show term)
    let τ = typeCheck term
    putStrLn ("---Type:---")
    putStrLn (show τ)
    let deBruijnTerm = D.toDeBruijn term
    putStrLn ("---DeBruijn Notation:---")
    putStrLn (show (deBruijnTerm))
```

```
putStrLn ("---Normal form (Structural semantics):---")
putStrLn (show (S.eval term))
putStrLn ("---Normal form (Natural semantics with DeBruijn indices):---")
putStrLn (show (N.eval deBruijnTerm))
putStrLn ("---Normal form (CES machine):---")
putStrLn (show (C.eval deBruijnTerm))
let cpsTerm1 = P.toCPS' term
let cpsDeBruijnTerm1 = D.toDeBruijn (App cpsTerm1 (Abs "a" τ (Var "a")))
putStrLn ("---Normal form (CE3R machine on CPS1):---")
putStrLn (show (R.eval cpsDeBruijnTerm1))
let cpsTerm2 = case SP.toCPSe (B.fromSyntaxA term) of
  Just t → t
  otherwise → error "CPS error!"
putStrLn ("---CPS2 Type:---")
putStrLn (show (SP.checkType (X.stripTags cpsTerm2)))
let cpsDeBruijnTerm2 = D.toDeBruijn (B.toSyntaxA (X.stripTags cpsTerm2))
putStrLn ("---Normal form (CE3R machine on CPS2):---")
putStrLn (show (R.eval cpsDeBruijnTerm2))
```

7 Appendix: Test results

7.1 Test 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.abs(n."m"-"n"*"m"/"n")) in abs(k.0=app(app("mod", "
k"), 2)) in app("iseven", 7)
---Type:---
Bool
```

```
---DeBruijn Notation:---
let let abs(abs(~1-~0*~1/~0)) in abs(0=app(app(~1,~0),2)) in app(~0,7)
---Normal form (Structural semantics):---
False
---Normal form (Natural semantics with DeBruijn indices):---
False
---Normal form (CES machine):---
False
---Normal form (CE3R machine on CPS1):---
False
---CPS2 Type:---
Just ((Bool -> Bool) -> Bool)
---Normal form (CE3R machine on CPS2):---
False
```

7.2 Test 2

```
---Input:---
+(+(+(5,3),+(9,10)),+(+(15,13),14))
---Term:---
5+3+9+10+15+13+14
---Type:---
Int
---DeBruijn Notation:---
5+3+9+10+15+13+14
---Normal form (Structural semantics):---
69
---Normal form (Natural semantics with DeBruijn indices):---
69
---Normal form (CES machine):---
69
---Normal form (CE3R machine on CPS1):---
69
---CPS2 Type:---
Just ((Int -> Int) -> Int)
---Normal form (CE3R machine on CPS2):---
69
```

7.3 Test 3

```
---Input:---
app (
```

```
    abs (x: Int.
        if <(/(12,3),*(2,6)) then
            app( abs(x: Int . if <(x,10) then *(x,3) else *(x,4) fi), *(x,15))
        else 7
        fi
    ), 2
)
---Term:---
app(abs(x.if 12/3<2*6 then app(abs(x.if "x"<10 then "x"*3 else "x"*4 fi), "x"*15
) else 7 fi), 2)
---Type:---
Int
---DeBruijn Notation:---
app(abs(if 12/3<2*6 then app(abs(if ~0<10 then ~0*3 else ~0*4 fi),~0*15) else 7
fi),2)
---Normal form (Structural semantics):---
120
---Normal form (Natural semantics with DeBruijn indices):---
120
---Normal form (CES machine):---
120
---Normal form (CE3R machine on CPS1):---
120
---CPS2 Type:---
Just ((Int -> Int) -> Int)
---Normal form (CE3R machine on CPS2):---
120
```

7.4 Test 4

```
---Input:---
+(if <(5,3) then 4 else 6 fi,7)
---Term:---
if 5<3 then 4 else 6 fi+7
---Type:---
Int
---DeBruijn Notation:---
if 5<3 then 4 else 6 fi+7
---Normal form (Structural semantics):---
13
---Normal form (Natural semantics with DeBruijn indices):---
```

```
13
---Normal form (CES machine):---
13
---Normal form (CE3R machine on CPS1):---
13
---CPS2 Type:---
Just ((Int -> Int) -> Int)
---Normal form (CE3R machine on CPS2):---
13
```

7.5 Test 5

```
---Input:---
app(
   abs(x:Int.
      +(
        app(
            abs(z:Int.
               +(
                 app(
                    abs(x:Int.+(x,z)),
                    5
                 ),
                 app(
                    abs(y:Int.+(y,z)),
                    6
                 )
               )
            ), 7
        ), x
      )
   ),
   8
)
---Term:---
app(abs(x.app(abs(z.app(abs(x."x"+"z"), 5)+app(abs(y."y"+"z"), 6)), 7)+"x"), 8)
---Type:---
Int
---DeBruijn Notation:---
app(abs(app(abs(app(abs(~0+~1),5)+app(abs(~0+~1),6)),7)+~0),8)
---Normal form (Structural semantics):---
```

```
33
---Normal form (Natural semantics with DeBruijn indices):---
33
---Normal form (CES machine):---
33
---Normal form (CE3R machine on CPS1):---
33
---CPS2 Type:---
Just ((Int -> Int) -> Int)
---Normal form (CE3R machine on CPS2):---
33
```

7.6 Test 6

```
---Input:---
app(
    fix (
        abs (f:->(Int,Int).
            abs (x:Int.
                if =(x,0) then
                    1
                else
                    *(x, app (f,-(x,1)))
                fi
            )
        )
    ), 10
)
---Term:---
app(fix(abs(f.abs(x.if "x"=0 then 1 else "x"*app("f", "x"-1) fi))), 10)
---Type:---
Int
---DeBruijn Notation:---
app(fix(abs(abs(if ~0=0 then 1 else ~0*app(~1,~0-1) fi))),10)
---Normal form (Structural semantics):---
3628800
---Normal form (Natural semantics with DeBruijn indices):---
3628800
---Normal form (CES machine):---
3628800
---Normal form (CE3R machine on CPS1):---
```

```
Failed
---CPS2 Type:---
Failed
---Normal form (CE3R machine on CPS2):---
Failed
```

7.7 Test 7

```
---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.abs(x.if 0="x" then true else if 0="x"-1 then false else app("ie"
, "x"-2) fi fi))), 7)
---Type:---
Bool
---DeBruijn Notation:---
app(fix(abs(abs(if 0=~0 then true else if 0=~0-1 then false else app(~1,~0-2) fi
 fi))),7)
---Normal form (Structural semantics):---
False
---Normal form (Natural semantics with DeBruijn indices):---
False
---Normal form (CES machine):---
False
---Normal form (CE3R machine on CPS1):---
Failed
---CPS2 Type:---
Failed
---Normal form (CE3R machine on CPS2):---
Failed
```

7.8 Test 8

```
---Input:---
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 (c:->(Int,Int). abs (x: Int.
                if app (iseven, x) then app (c, /(x,2)) else
                  if =(x,1) then 1 else
                    app (c, +(*(3,x),1)) fi fi)))
  in
    app (collatz, 1000)
  end
end
---Term:---
let iseven = fix(abs(ie.abs(x.if 0="x" then true else if 1="x" then false else a
pp("ie", "x"-2) fi fi))) in let collatz = fix(abs(c.abs(x.if app("iseven", "x")
then app("c", "x"/2) else if "x"=1 then 1 else app("c", 3*"x"+1) fi fi))) in
app("collatz",1000)
---Type:---
Int
---DeBruijn Notation:---
let fix(abs(abs(if 0=~0 then true else if 1=~0 then false else app(~1,~0-2) fi f
i))) in let fix(abs(abs(if app(~2,~0) then app(~1,~0/2) else if ~0=1 then 1 else
 app(~1,3*~0+1) fi fi))) in app(~0,1000)
---Normal form (Structural semantics):---
1
---Normal form (Natural semantics with DeBruijn indices):---
1
---Normal form (CES machine):---
1
---Normal form (CE3R machine on CPS1):---
Failed
---CPS2 Type:---
Failed
---Normal form (CE3R machine on CPS2):---
Failed
```

7.9 Test 9

```
---Input:---
app(
  abs (x: Int .
      abs (y: Int .
          +(
             +(x,y),
```

```
          app (abs (x: Int . +(x,y)), 3)
        )
      )
  ),
  app (abs (x: Int . x), 5)
)
---Term:---
app(abs(x.abs(y."x"+"y"+app(abs(x."x"+"y"), 3))), app(abs(x."x"), 5))
---DeBruijn Notation:---
app(abs(abs(~1+~0+app(abs(~0+~1),3))),app(abs(~0),5))
---Normal form (Structural semantics):---
abs(y.5+"y"+app(abs(x."x"+"y"), 3))
---Normal form (Natural semantics with DeBruijn indices):---
Function: Clo abs(~1+~0+app(abs(~0+~1),3)) [5]
---Normal form (CES machine):---
Clo [Access 1,Access 0,Add,Close [Access 0,Access 1,Add,Return],Int 3,Apply,Add,
Return] [5]
---Normal form (CE3R machine on CPS1):---
Clo [Close1 [Close1 [~1 0,~2 5,App1],Close2 [Close1 [~1 0,~2 4,App1],Close2 [~1
2,~2 1,~3 0,Add],App1],App1],Close2 [Close1 [Close1 [~1 0,Close2 [Close1 [~1 0,~
2 2,App1],Close2 [Close1 [~1 0,~2 8,App1],Close2 [~1 2,~2 1,~3 0,Add],App1],App1
],App1],Close2 [Close1 [~1 0,Int2 3,App1],Close2 [~1 1,~2 0,~3 2,App2],App1],App
1],Close2 [~1 2,~2 1,~3 0,Add],App1],App1] [Clo [~1 0] [],5,Clo [Close1 [Close1
[~1 0,Close2 [~1 0,~2 1,App1],App1],Close2 [Close1 [~1 0,Int2 5,App1],Close2 [~1
 1,~2 0,~3 2,App2],App1],App1],Close2 [~1 1,~2 0,~3 2,App2],App1] [Clo [~1 0] []
],Clo [~1 0] []]
---CPS2 Type:---
Just (((((Int -> ((((Int -> (((Int -> ((Int -> (((Int -> ((Int -> (Int -> Int))
-> (Int -> Int))) -> (Int -> Int)) -> (Int -> Int))) -> (((Int -> ((Int -> (Int
-> Int)) -> (Int -> Int))) -> (Int -> Int)) -> (Int -> Int)))) -> (((Int -> ((In
t -> (Int -> Int)) -> (Int -> Int))) -> (Int -> Int)) -> (Int -> Int))) -> (((In
t -> ((Int -> (Int -> Int)) -> (Int -> Int))) -> (Int -> Int)) -> (Int -> Int)))
) -> (((Int -> ((Int -> (((Int -> ((Int -> (Int -> Int)) -> (Int -> Int))) -> (I
nt -> Int)) -> (Int -> Int))) -> (((Int -> ((Int -> (Int -> Int)) -> (Int -> Int
))) -> (Int -> Int)) -> (Int -> Int)))) -> (((Int -> ((Int -> (Int -> Int)) -> (
Int -> Int))) -> (Int -> Int)) -> (Int -> Int))) -> (((Int -> ((Int -> (Int -> I
nt)) -> (Int -> Int))) -> (Int -> Int)) -> (Int -> Int)))) -> (((Int -> ((Int ->
 (Int -> Int)) -> (Int -> Int))) -> (Int -> Int)) -> (Int -> Int))) -> (((Int ->
 ((Int -> (Int -> Int)) -> (Int -> Int))) -> (Int -> Int)) -> (Int -> Int)))) ->
 (((Int -> ((Int -> (((Int -> ((Int -> (Int -> Int)) -> (Int -> Int))) -> (Int ->
 Int)) -> (Int -> Int))) -> (((Int -> ((Int -> (Int -> Int)) -> (Int -> Int)))
```

```
-> (Int -> Int)) -> (Int -> Int)))) -> (((Int -> ((Int -> (Int -> Int)) -> (Int
-> Int))) -> (Int -> Int)) -> (Int -> Int))) -> (((Int -> ((Int -> (Int -> Int))
 -> (Int -> Int))) -> (Int -> Int)) -> (Int -> Int)))) -> (((Int -> ((Int -> (((
Int -> ((Int -> (Int -> Int)) -> (Int -> Int))) -> (Int -> Int)) -> (Int -> Int)
)) -> (((Int -> ((Int -> (Int -> Int)) -> (Int -> Int))) -> (Int -> Int)) -> (In
t -> Int)))) -> (((Int -> ((Int -> (Int -> Int)) -> (Int -> Int))) -> (Int -> In
t)) -> (Int -> Int))) -> (((Int -> ((Int -> (Int -> Int)) -> (Int -> Int))) -> (
Int -> Int)) -> (Int -> Int)))) -> (((Int -> ((Int -> (Int -> Int)) -> (Int -> I
nt))) -> (Int -> Int)) -> (Int -> Int))) -> (((Int -> ((Int -> (Int -> Int)) ->
(Int -> Int))) -> (Int -> Int)) -> (Int -> Int)))
---Normal form (CE3R machine on CPS2):---
Clo [Close1 [~1 0,Close2 [~1 0,Close2 [Close1 [Close1 [~1 0,~2 5,App1],Close2 [C
lose1 [~1 0,~2 4,App1],Close2 [~1 2,~2 1,~3 0,Add],App1],App1],Close2 [Close1 [C
lose1 [~1 0,Close2 [Close1 [~1 0,~2 2,App1],Close2 [Close1 [~1 0,~2 8,App1],Clos
e2 [~1 2,~2 1,~3 0,Add],App1],App1],App1],Close2 [Close1 [~1 0,Int2 3,App1],Clos
e2 [~1 1,~2 0,~3 2,App2],App1],App1],Close2 [~1 2,~2 1,~3 0,Add],App1],App1],App
1],App1],Close2 [Close1 [Close1 [~1 0,Close2 [~1 0,~2 1,App1],App1],Close2 [Clos
e1 [~1 0,Int2 5,App1],Close2 [~1 1,~2 0,~3 2,App2],App1],App1],Close2 [~1 1,~2 0
,~3 2,App2],App1],App1] []
```

References

[1] Timothy G. Griffin, A Formulae-as-Types Notion of Control, 1990: In Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, ACM Press.

[2] Olivier Danvy and Andrzej Filinski, Representing control: a study of the CPS transformation, 1992.