- *Reduction* based normalization is defined by a set of reductions that are repeatedly applied inside a term.

$$\underline{(1+2)} + 2*(4+2)$$
$$3 + 2 * \underline{(4+2)}$$
$$3 + \underline{2*6}$$
$$\ldots$$

- *Evaluation* based normalization directly maps a term its normal form, for a given *environment*.

$$\frac{\dfrac{\ldots}{x = 2 \vdash 3 \mapsto 3} \qquad \dfrac{\ldots}{x = 2 \vdash 1 + x \mapsto 3}}{x = 2 \vdash 3 * (1 + x) \mapsto 9}$$

*) This is more complicated for λ-calculus.

Plan

1. Investigate how to implement reduction based normalization.
2. Generalize it.
3. Derive simple evaluation.
4. Optimize it further (compilation).

Lambda-terms

```
type Var    =  String
data Term   =  Var Var
            |  App Term Term
            |  Lam Var  Term
```

Free variables

```
freeVars :: Term -> [Var]
freeVars (Var x)      =  [x]
freeVars (App e1 e2)  =  freeVars e1 `union` freeVars e2
freeVars (Lam x e)    =  delete x (freeVars e)
```

State transformer monad

```
newtype S s a = S (s -> (a,s))

instance Monad (S s) where
   (S f) >>= k   =  S (\s ->  case f s of
                                (x,s') ->  case k x of
                                             S g -> g s')
   return x      =  S (\s ->  (x,s))

getS     ::  S s s
getS     =   S (\s -> (s,s))

setS     ::  s -> S s ()
setS x   =   S (\s -> ((),x))

runS     ::  S s a -> s -> (a,s)
runS (S f) s = f s
```

Generating new variables

```
newVar  ::  S Int Var
newVar  =   do  i <- getS
                setS (i+1)
                return ("x" ++ show i)
```

Substitution

```
subst :: Term -> (Var,Term) -> S Int Term
subst t (x,e) = subs t
  where fvs = freeVars e
        subs (Var y)   | x == y      = return e
                       | otherwise   = return (Var y)
        subs (App e1 e2)   = do e1' <- subs e1
                                e2' <- subs e2
                                return (App e1' e2')
        subs (Lam y e1)
             | x == y            =  return (Lam y e1)
             | notElem y fvs     =  do e1' <- subs e1
                                       return (Lam y e1')
             | otherwise    = do z    <- newVar
                                e1'  <- subst e1 (y,Var z)
                                e1''<- subs e1'
                                return (Lam z e1'')
```

Single-step reduction (applicative order)

```
reduA :: Term -> S Int (Maybe Term)
reduA (Var x) = return Nothing
reduA (Lam x e)
   =  do  me′ <- reduA e
          case me′ of
             Just e′   ->  return (Just (Lam x e′))
             Nothing   ->  return Nothing
```

Single-step reduction (applicative order)

```
reduA (App e1 e2)
  = do  me1 <- reduA e1
        case me1 of
          Just e1'  -> return (Just (App e1' e2))
          Nothing   ->
             do me2 <- reduA e2
                case me2 of
                  Just e2' -> return (Just (App e1 e2'))
                  Nothing  ->
                    case e1 of
                      Lam x e0  ->
                        do e <- subst e0 (x,e2)
                           return (Just e)
                      _         -> return Nothing
```

Single-step reduction (normal order)

```
reduN :: Term -> S Int (Maybe Term)
reduN (Var x) = return Nothing
reduN (Lam x e) = do me' <- reduN e
                     return (fmap (\e' -> Lam x e') me')
reduN (Lam x e1 `App` e2) = do e <- subst e1 (x,e2)
                               return (Just e)
reduN (App e1 e2)
  = do  me1 <- reduN e1
        case me1 of
          Just e1'  -> return (Just (App e1' e2))
          Nothing   ->
            do  me2 <- reduN e2
                return (fmap (\e2' -> App e1 e2') me2)
```

Generating reduction sequence

```
iterateSM :: (a -> S Int (Maybe a)) -> a -> S Int [a]
iterateSM f x = do y <- f x
                   case y of
                     Just y'   -> do ys <- iterateSM f y'
                                     return (x:ys)
                     Nothing   -> return [x]

reduceA   ::   Term -> S Int [Term]
reduceA   =    iterateSM reduA

reduceN   ::   Term -> S Int [Term]
reduceN   =    iterateSM reduN
```

Parametrised state transformer monad

```
newtype S m s a = S (s -> m (a,s))

instance Monad m => Monad (S m s) where
    return x      =  S (\s -> return (x,s))
    (S f) >>= k   =  S (\s -> do  (x,s') <- f s
                                  case k x of
                                     S g -> g s')
getS     ::  Monad m => S m s s
getS     =   S (\s -> return (s,s))

setS     ::  Monad m => s -> S m s ()
setS x   =   S (\s -> return ((),x))

runS     ::  Monad m => S m s a -> s -> m (a,s)
runS (S f) s = f s
```

Parametrised state transformer monad

```
instance MonadPlus m => MonadPlus (S m s) where
      mzero                  = S (\s -> mzero)
      (S f) `mplus` (S g)    = S (\s -> f s `mplus` g s)
```

New variables, substitution

```
type StM a = S Maybe Int a

newVar  ::  StM Var
newVar  =   ...

subst   ::  Term -> (Var,Term) -> StM Term
subst t (x,e) = ...
```

Single-step reduction (applicative order)

```
reduA :: Term -> StM Term
reduA (Var x)     = mzero
reduA (Lam x e)   = reduA e >>= \e' -> return (Lam x e')
reduA (App e1 e2)
    = (reduA e1 >>= \e1' -> return (App e1' e2)) `mplus`
      (reduA e2 >>= \e2' -> return (App e1 e2')) `mplus`
      (case e1 of
         Lam x e0  -> subst e0 (x,e2)
         _         -> mzero)
```

Single-step reduction (normal order)

```
reduN :: Term -> StM Term
reduN (Var x)     = mzero
reduN (Lam x e)   = reduN e >>= \e' -> return (Lam x e')
reduN (Lam x e1 `App` e2) = subst e1 (x,e2)
reduN (App e1 e2)
    = (reduN e1 >>= \e1' -> return (App e1' e2)) `mplus`
      (reduN e2 >>= \e2' -> return (App e1 e2'))
```

Generating reduction sequence

```
iterateStM :: (a -> StM a) -> a -> StM [a]
iterateStM f x =  (do ys <- f x >>= \y -> iterateStM f y
                      return (x : ys)) `mplus`
                   return [x]

reduceA   ::   Term -> StM [Term]
reduceA   =    iterateStM reduA

reduceN   ::   Term -> StM [Term]
reduceN   =    iterateStM reduN
```

- It is clear that this implements normalization according to the theory!

Naive evaluation might be wrong

```
evalN1 :: (Var -> Maybe Term) -> Term -> Maybe Term

evalN1 env (Var x) =
  case env x of
    Nothing -> return $ Var x
    Just e  -> evalN1 env e

evalN1 env (App e1 e2) = do
  e1' <- evalN1 env e1
  case e1' of
    Lam x b -> evalN1 (addEnv (x,e2) env) b
    e1'     -> do
      e2' <- evalN1 env e2
      return $ App e1' e2'

evalN1 env (Lam x e) = do
  e' <- evalN1 (removeEnv x env) e
  return $ Lam x e
```

Helper functions

```
startEnv :: Var -> Maybe a
startEnv        = \y -> Nothing

addEnv :: (Var, a) -> (Var -> Maybe a)
                   -> Var -> Maybe a
addEnv (x,e) f = \y -> if x==y then Just e  else f y

removeEnv :: Var -> (Var -> Maybe a)
                 -> Var -> Maybe a
removeEnv x f  = \y -> if x==y then Nothing else f y
```

Does not work :(

```
evalN1 startEnv (\x -> (\y -> (\ x -> y) A) x) B
```

Does not work :(

```
evalN1 startEnv (\x -> (\y -> (\ x -> y) A) x) B
evalN1 (addEnv x B startEnv) (\y -> (\ x -> y) A) x
```

Does not work :(

```
evalN1 startEnv (\x -> (\y -> (\ x -> y) A) x) B
evalN1 (addEnv x B startEnv) (\y -> (\ x -> y) A) x
evalN1 (addEnv y x $ addEnv x B startEnv) (\ x -> y) A
```

Does not work :(

```
evalN1 startEnv (\x -> (\y -> (\ x -> y) A) x) B
evalN1 (addEnv x B startEnv) (\y -> (\ x -> y) A) x
evalN1 (addEnv y x $ addEnv x B startEnv) (\ x -> y) A
evalN1 (addEnv x A $ addEnv y x $ addEnv x B startEnv) y
```

Does not work :(

```
evalN1 startEnv (\x -> (\y -> (\ x -> y) A) x) B
evalN1 (addEnv x B startEnv) (\y -> (\ x -> y) A) x
evalN1 (addEnv y x $ addEnv x B startEnv) (\ x -> y) A
evalN1 (addEnv x A $ addEnv y x $ addEnv x B startEnv) y
(addEnv x A $ addEnv y x $ addEnv x B startEnv) y
```

Does not work :(

```
evalN1 startEnv (\x -> (\y -> (\ x -> y) A) x) B
evalN1 (addEnv x B startEnv) (\y -> (\ x -> y) A) x
evalN1 (addEnv y x $ addEnv x B startEnv) (\ x -> y) A
evalN1 (addEnv x A $ addEnv y x $ addEnv x B startEnv) y
(addEnv x A $ addEnv y x $ addEnv x B startEnv) y
(addEnv y x $ addEnv x B startEnv) y
```

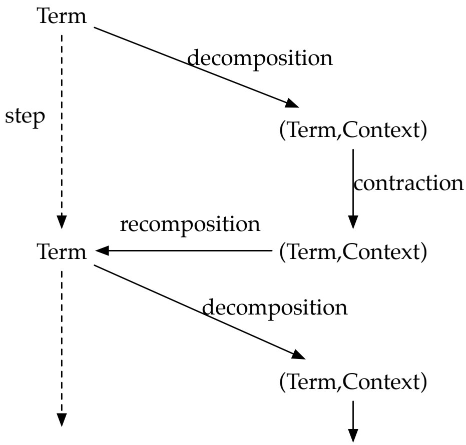
Does not work :(

```
evalN1 startEnv (\x -> (\y -> (\ x -> y) A) x) B
evalN1 (addEnv x B startEnv) (\y -> (\ x -> y) A) x
evalN1 (addEnv y x $ addEnv x B startEnv) (\ x -> y) A
evalN1 (addEnv x A $ addEnv y x $ addEnv x B startEnv) y
(addEnv x A $ addEnv y x $ addEnv x B startEnv) y
(addEnv y x $ addEnv x B startEnv) y
evalN1 (addEnv x A $ addEnv y x $ addEnv x B startEnv) x
```

Does not work :(

```
evalN1 startEnv (\x -> (\y -> (\ x -> y) A) x) B
evalN1 (addEnv x B startEnv) (\y -> (\ x -> y) A) x
evalN1 (addEnv y x $ addEnv x B startEnv) (\ x -> y) A
evalN1 (addEnv x A $ addEnv y x $ addEnv x B startEnv) y
(addEnv x A $ addEnv y x $ addEnv x B startEnv) y
(addEnv y x $ addEnv x B startEnv) y
evalN1 (addEnv x A $ addEnv y x $ addEnv x B startEnv) x
A
```

Derive from reductions



Types

```
type Decomposition = Term -> Maybe (Term, Context)
type Contraction   = (Term, Context) -> (Term, Context)
type Recomposition = (Term, Context) -> Term

type Context       = Term -> Term
```

Normal order decomposition

```
normalOrder :: Term -> Maybe (Term, Term -> Term)
normalOrder (Lam x e) = do
    (red,ctx) <- normalOrder e
    return (red, \z -> Lam x (ctx z))
normalOrder (App (Lam x e) y) =
  return (App (Lam x e) y, id)
normalOrder (App f y) =
  (normalOrder f >>= \(red, ctx) ->
      return (red, \ z -> App (ctx z) y)) `mplus`
  (normalOrder y >>= \(red, ctx) ->
      return (red, \ z -> App f (ctx z)))
normalOrder _ =
  mzero
```

Applicative order decomposition

```
appOrder :: Term -> Maybe (Term, Term -> Term)
appOrder (Lam x e) = do
    (red,ctx) <- appOrder e
    return (red, \z -> Lam x z)
appOrder (App f y) =
  (appOrder f >>= \(red, ctx) ->
      return (red, \ z -> App (ctx z) y)) `mplus`
  (appOrder y >>= \(red, ctx) ->
      return (red, \ z -> App f (ctx z))) `mplus`
  (case f of Lam x e -> return (App (Lam x e) y, id)
             _       -> mzero)
```

Recomposition and contraction

```
recompose :: (Term, Term -> Term) -> Term
recompose (x, f) = f x

reduce :: Term -> StM Term
reduce (App (Lam x e) y) = subst e (x, y)
reduce e = mzero

contraction :: (Term, Context) -> StM (Term, Context)
contraction (e,c) = do
  e' <- reduce e
  return (e', c)
```

The whole loop

```
normalize :: Decomposition -> Term -> StM Term
normalize decomp  e = loop e
  where
    loop e = do
        d <- liftStM decomp e
        c <- contraction d
        r <- return $ recompose c
        loop r
      `handle`
        return e
```

… where

```
--type StM a = S Maybe Int a

handle :: S Maybe a b -> S Maybe a b -> S Maybe a b
handle = mplus

liftStM :: Functor m => (a -> m b) -> a -> S m c b
liftStM f x = S (\ s -> fmap (\ y -> (y,s)) $ f x)
```

The reduction loop

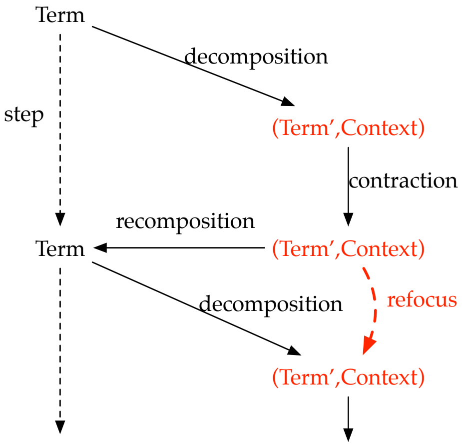
- Reduction is a state-monadic operation over a failure monad.
- Decomposition has a MonadPlus structure.
- Step = Recompose ∘ Contract ∘ Decompose

Two problems of reduction:

1. term can grow really big due to substitution
2. repeated recomposition is expensive

Solutions:

1. keep substitutions in a mapping instead
2. recompose using a stack (/the program stack)


Term
decomposition
step
(Term',Context)
contraction
recomposition
Term
(Term',Context)
refocus
decomposition
(Term',Context)

- Context is a stack
  - all information for recomposing the term
  - efficient to implement refocusing ($O(1)$)
- Term' is a pair (t, m) where
  - t is a sub-term of the whole program
  - m is mapping of applied substitutions (Var $\rightarrow$ $Term'$)

Normal order evaluation (almost Haskell)

```
type Term' = (Term, Var -> Maybe Term')

evalN :: Term' -> Maybe Term'

evalN (Var x, env) =
  case env x of
    Nothing -> return (Var x, env)
    Just e  -> evalN e

evalN (App e1 e2, env) = do
  (e1',env') <- evalN (e1, env)
  case e1' of
    Lam x b -> evalN (b, addEnv (x, (e2, env)) env')
    _       -> do (e2', env'') <- evalN (e2, env)
                  return (App (recomp e1' env')
                              (recomp e2' env''), env)

evalN e = return e
```

Recomposition after normaliztion

```
recomp :: Term -> (Var -> Maybe Term') -> Term
recomp (Var x)   env =
  case env x of
    Nothing          -> Var "x"
    Just (T (e,env')) -> recomp e env'
recomp (App f x) env =
  App (recomp f env) (recomp x env)
recomp (Lam x b) env =
  Lam x (recomp b env)
```

- Evaluators typically use a data-structure instead.
- (Compilers use a “standard constructor”.)

Normal order evaluation (Haskell)

```
newtype Term' = T (Term, Var -> Maybe Term')

evalN :: Term' -> Maybe Term'

evalN (T (Var x, env)) =
  case env x of
    Nothing -> return (T (Var x, env))
    Just e  -> evalN e

evalN (T (App e1 e2, env)) = do
 T (e1', env') <- evalN (T (e1, env))
 case e1' of
   Lam x b -> evalN (T (b, addEnv (x, T(e2, env)) env'))
   _       -> do T (e2', env'') <- evalN (T (e2, env))
                 return $ T (App (recomp e1' env')
                                 (recomp e2' env''),env)

evalN e = return e
```

Optimization and De Bruijn encoding

$$
\begin{aligned}
E ::= {} & N \\
\mid {} & (E_1\ E_2) \\
\mid {} & (\lambda\ E)
\end{aligned}
$$

Each value (t, m) of type Term' during evaluation:

- Terms t are just pointers into the full program AST.
- Environments m are just stacks of Term'-s.
- "Variable" n just picks the n-th value from the stack.

Compilation

Compilers can be generated from interpreters:

```
import Eval

program :: Term
program = ...       -- term containing free variables

main = maybe (putStrLn "Error") print $
         evalN (program, ... {- bindings for free vars.
 ↪   -})
```

- But this is *cheating* as Haskell is more complicated than λ-calculus.
- Also, performance implications are unclear (for Haskell).
- *We also do not want to generate x86 assembly!*

Key elements of the Von Neumann architecture

❶
```
-- instructions have small size
type Instr
```

Key elements of the Von Neumann architecture

1.
```
-- instructions have small size
type Instr
```

2.
```
-- from program state we can extract the next
 ↪  instruction
nextInstr :: ProgramState -> Instr
```

Key elements of the Von Neumann architecture

1.
```
-- instructions have small size
type Instr
```

2.
```
-- from program state we can extract the next
↪  instruction
nextInstr :: ProgramState -> Instr
```

3.
```
-- instructions are evaluated in constant time
evalInstr :: Instr -> Input -> ProgramState
                            -> (ProgramState,
↪  Output)
```

Key elements of the Von Neumann architecture

1.
```
-- instructions have small size
type Instr
```

2.
```
-- from program state we can extract the next
↪  instruction
nextInstr :: ProgramState -> Instr
```

3.
```
-- instructions are evaluated in constant time
evalInstr :: Instr -> Input -> ProgramState
                            -> (ProgramState,
↪  Output)
```

4. each cycle the next instruction is evaluated

Basic idea

1. Convert program into a sequence of instructions.
   - `compile :: Term -> [Instr]`
   - Sub-terms are sub-sequences of instructions
2. `eval(`t, e`) == evalC(compile(`t`))(`e`)`

- Pattern matching of terms is avoided.
- Compiled instructions can be optimized.
- Term $\equiv$ address of its first instructions.
  - Term' = (Int, Array Int Term')
- Program AST not in memory anymore.
- No code-gen outside of `compile(`t`)`

We saw …

- how to implement reduction based normalization,
- derivation of evaluation and the idea for compilation from reductions,
- performance implications from different approaches.