Revisiting Combinators

Edward Kmett

A Toy Lambda Calculus

```haskell
data LC a
  = Var a
  | App (LC a) (LC a)
  | Lam (LC (Maybe a))
  deriving (Functor, Foldable, Traversable)

instance Applicative LC where pure = Var; (<*>) = ap

instance Monad LC where
  Var a >>= f = f a
  App l r >>= f = App (l >>= f) (r >>= f)
  Lam k >>= f = Lam $ k >>= \case
    Nothing -> pure Nothing
    Just a -> Just <$> f a

lam :: Eq a => a -> LC a -> LC a
lam v b = Lam $ bind v <$> b where
  bind v u = u <$ guard (u /= v)
```

What is a Combinator?

A combinator is a function that has no free variables.

Introducing Combinators

- Moses Shönfinkel (Моисей Исаевич Шейнфинкель), **“Über die Bausteine der mathematischen Logik”** 1924
- Haskell Curry, **“Grundlagen der Kombinatorischen Logik”** 1930
- David Turner, **“Another Algorithm for Bracket Abstraction”** 1979
- Smullyan **“To Mock a Mockingbird”** 1985
- Sabine Broda and Lúis Damas **“Bracket abstraction in the combinator system Cl(K)”** 1987(?)

A Toy Combinatory Logic

```haskell
data CL a = V a | A (CL a) (CL a) | S | K | I | B | C
  deriving (Functor, Foldable, Traversable)

instance Applicative CL where pure = V; (<*>) = ap

instance Monad CL where
  V a >>= f = f a
  A l r >>= f = A (l >>= f) (r >>= f)
  S >>= _ = S
  K >>= _ = K
  I >>= _ = I
  B >>= _ = B
  C >>= _ = C
```

A Field Guide

```
S x y z = (x z) (y z) -- (<*>)
K x y = x -- const
I x = x -- id
B x y z = x (y z) -- (.)
C x y z = x z y -- flip
Y f = f (Y f) = let x = f x in x -- fix
```

Abstraction Elimination

a la Shönfinkel

```
infixl 1 %
(%) = A

compile :: LC a -> CL a
compile (Var a)   = V a
compile (App l r) = compile l % compile r
compile (Lam b)   = compileLam b

compileLam :: LC (Maybe a) -> CL a
compileLam (Var Nothing)) = I
compileLam (Var (Just y)) = K % V y

compileLam (Ap l r) = case (sequence l, sequence r) of
  (Nothing, Nothing) -> S % compileLam l % compileLam r
  (Nothing, Just r') -> C % compileLam l % compile r
  (Just l', Nothing) -> B % compile l' % compileLam r
  (Just l', Just r') -> K % (compile l' % compile r')

compileLam (Lam b) = join $ compileLam $ dist $ compileLam b where
  dist :: C (Maybe a) -> L (Maybe (C a))
  dist (A l r) = App (dist l) (dist r)
  dist xs = Var (sequence xs)
```

Play with this at http://pointfree.io/ or with @pl on lambdabot

Supercombinators

- John Hughes, **“Super Combinators: A New Implementation Method for Applicative Languages”** 1982
- Lennart Augustsson, **“A compiler for Lazy ML”** 1984
- Simon Peyton Jones **“Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine”** 1992
- Simon Marlow, Alexey Rodriguez Yakushev and Simon Peyton Jones **“Faster laziness using dynamic pointer tagging”** 2007

Unknown Thunk Evaluation

jmp %eax

Every Thunk in GHC is a SPECTRE v2 Vulnerability

SPMD-on-SIMD Evaluation

- Small evaluator that immediately recovers coherence between instructions.
- Scales with the product of the SIMD-width and # of cores, rather than one or the other
- This bypasses the spectre issues.
- This generalizes to GPU compute shading

Why are combinator terms bigger?

- S x y z = (x z) (y z) — application with environment
- K x y = x — ignores environment
- I x — uses environment

- These all work “one variable at a time”

Sketching an “improved” abstraction elimination algorithm

We need to be able to build and manipulate ‘partial’ environments in sublinear time.

Some references:

- Abadi, Cardelli, Curien, Lévy **“Explicit Substitutions”** 1990
- Lescanne, **“From λσ to λv a journey through calculi of explicit substitutions”** 1994
- Mazzoli, **“A Well-Typed Suspension Calculus”** 2017

Evaluation by Collection

fst (a,b) ~> a

snd (a,b) ~> b

- John Hughes, **“The design and implementation of programming languages”** 1983
- Philip Wadler, **“Fixing some space leaks with a garbage collector”** 1987
- Christina von Dorrien, **“Stingy Evaluation”** 1989

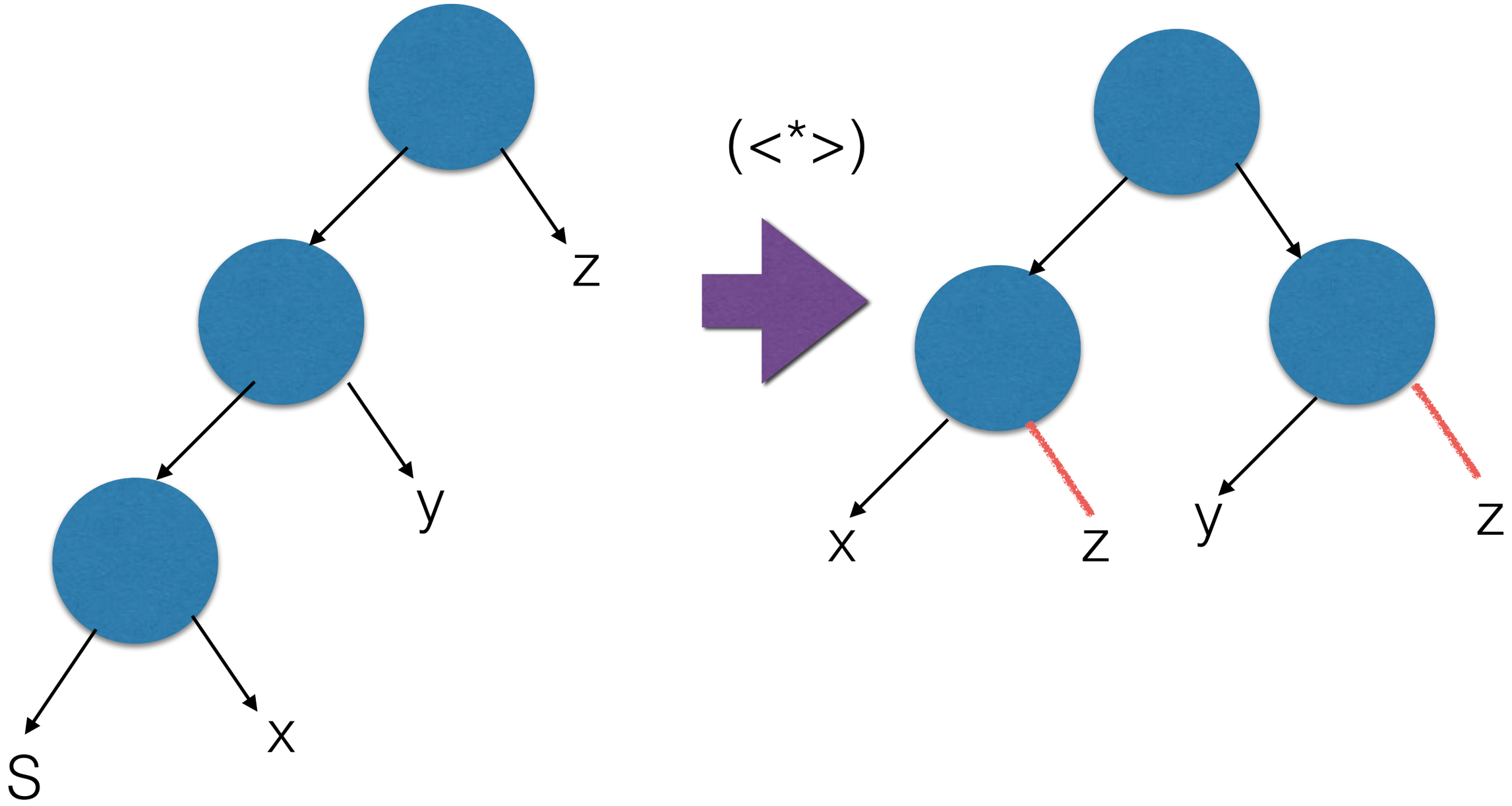
Some extra stingy combinator evaluation rules:

```
  A (A K x) _           -> Just x -- def K
  A I x                 -> Just x -- def I
  A S K                 -> Just K_ -- Hudak 1
  A S K_                -> Just I -- Hudak 2
  A S k@(A K (A K _)) -> Just k -- Hudak 3
  A (A S (A K x)) I     -> Just x -- Hudak 4, Turner p35 2
  A Y k@(A K _)         -> Just k -- Hudak 9
```

(Be careful, not all of Hudak's rules are stingy!)

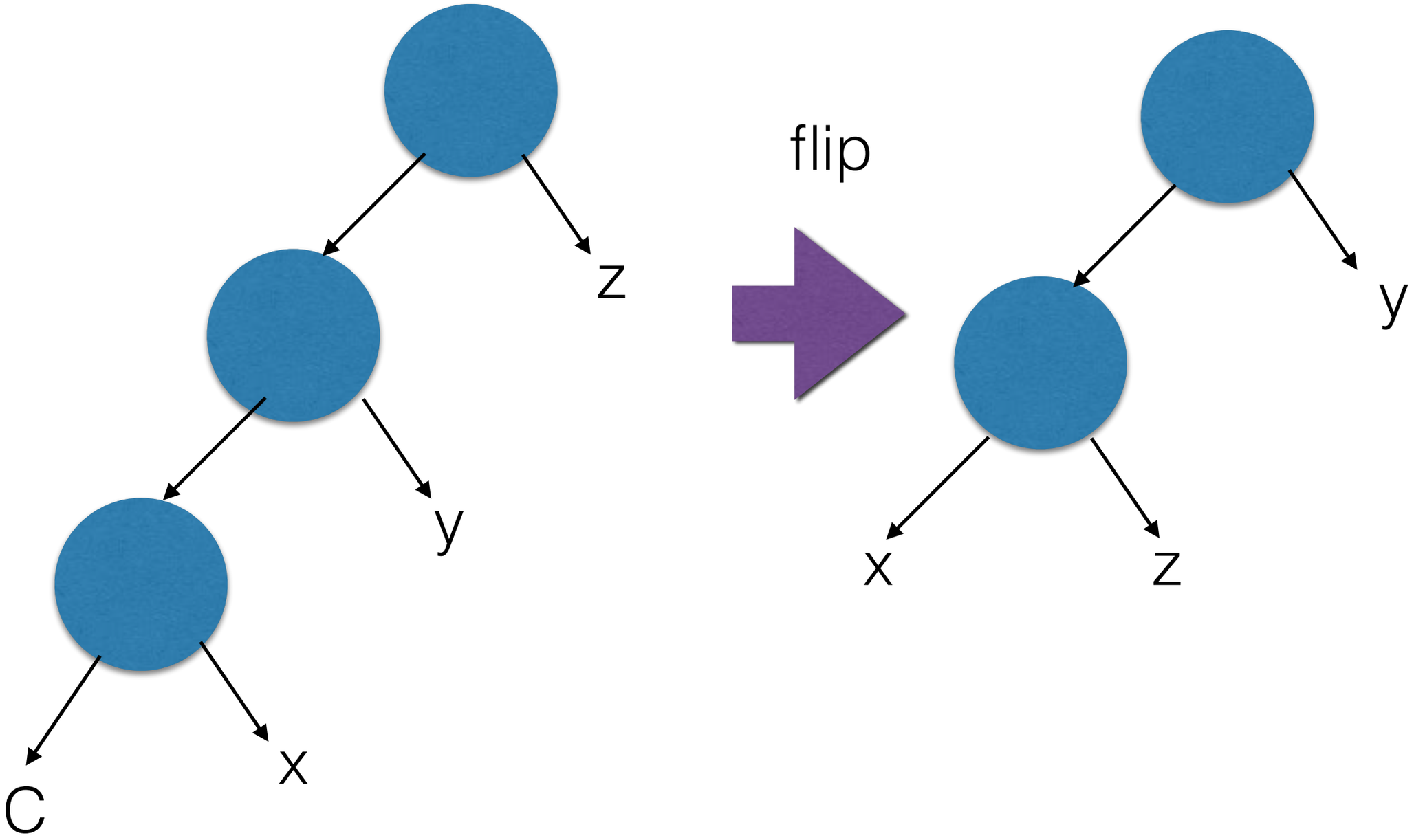
- Paul Hudak and David Kranz, **"A Combinator-based Compiler for a Functional Language"** 1984

One Bit Reference Counting

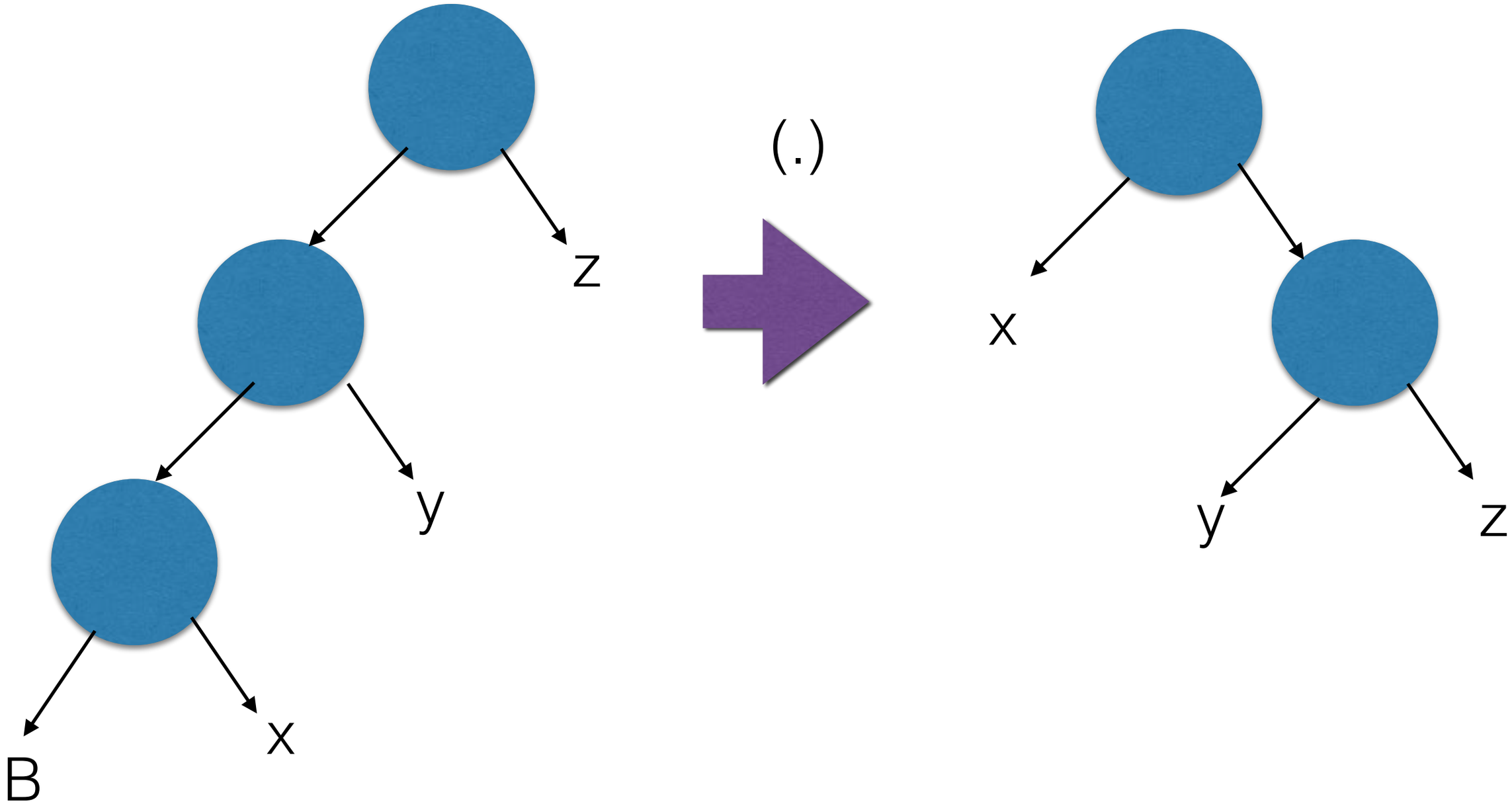


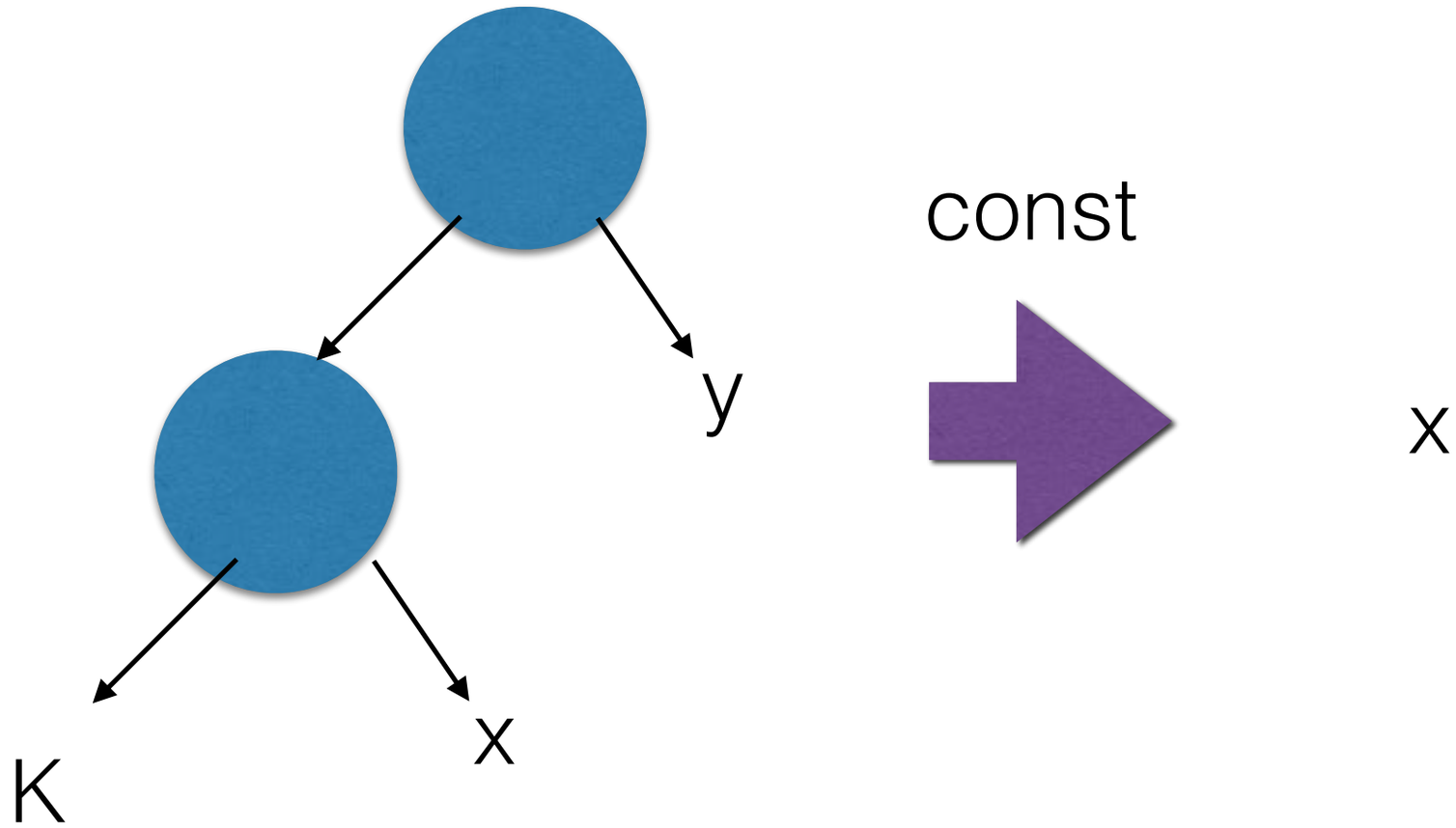
W.R. Stoye, T.J.W. Clarke and A.C. Norman **“Some Practical Methods for Rapid Combinator Reduction”** 1984

One Bit Reference Counting



One Bit Reference Counting




K
x
y
const
x

Extra reductions that require “uniqueness” to be stingy:

```
  A (A (A C f) x) y          -> Just $ A (A f y) x
  A (A (A B f) g) x          -> Just $ A f (A g x)
  A (A S (A K x)) (A K y) -> Just $ A K (A x y) -- Hudak 5, Turner p35 1
  A (A S (A K x)) y          -> Just $ A (A B x) y -- Turner p35 3
  A (A S x) (A K y)          -> Just $ A (A C x) y -- Turner p35 4
```

Can we reduce `fst (x, y) ~> x` by stingy evaluation **without** special casing Wadler's rules?

```
fst p = p (\xy.x)

pair x y z = z x y

snd p = p (\xy.y)
```

```
fst = CIK

snd = CI(KI)

pair = BC(CI)
```

Optionally Acyclic Heaps

- None of the S,B,C,K,I… rules introduce new cycles on the heap except Y.

- Hash-consing!

- Eiichi Goto **“Monocopy and associative algorithms in extended Lisp”** 1974

- Eelco Dolstra, **“Maximal Laziness”** 2008

Why I Care?

- Compilers for dependently typed languages are slow. Hash consign is usually bolted in as an afterthought. I'd like an efficient default evaluator that automatically memoizes and addresses unification and substitution concerns.

- Daniel Dougherty **"Higher-order unification via combinators"** 1993