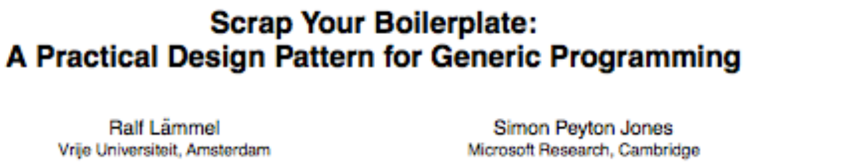
Generics via Lisp-Like Primitives

go fold-free

Matt Roberts

[plrg] macquarie

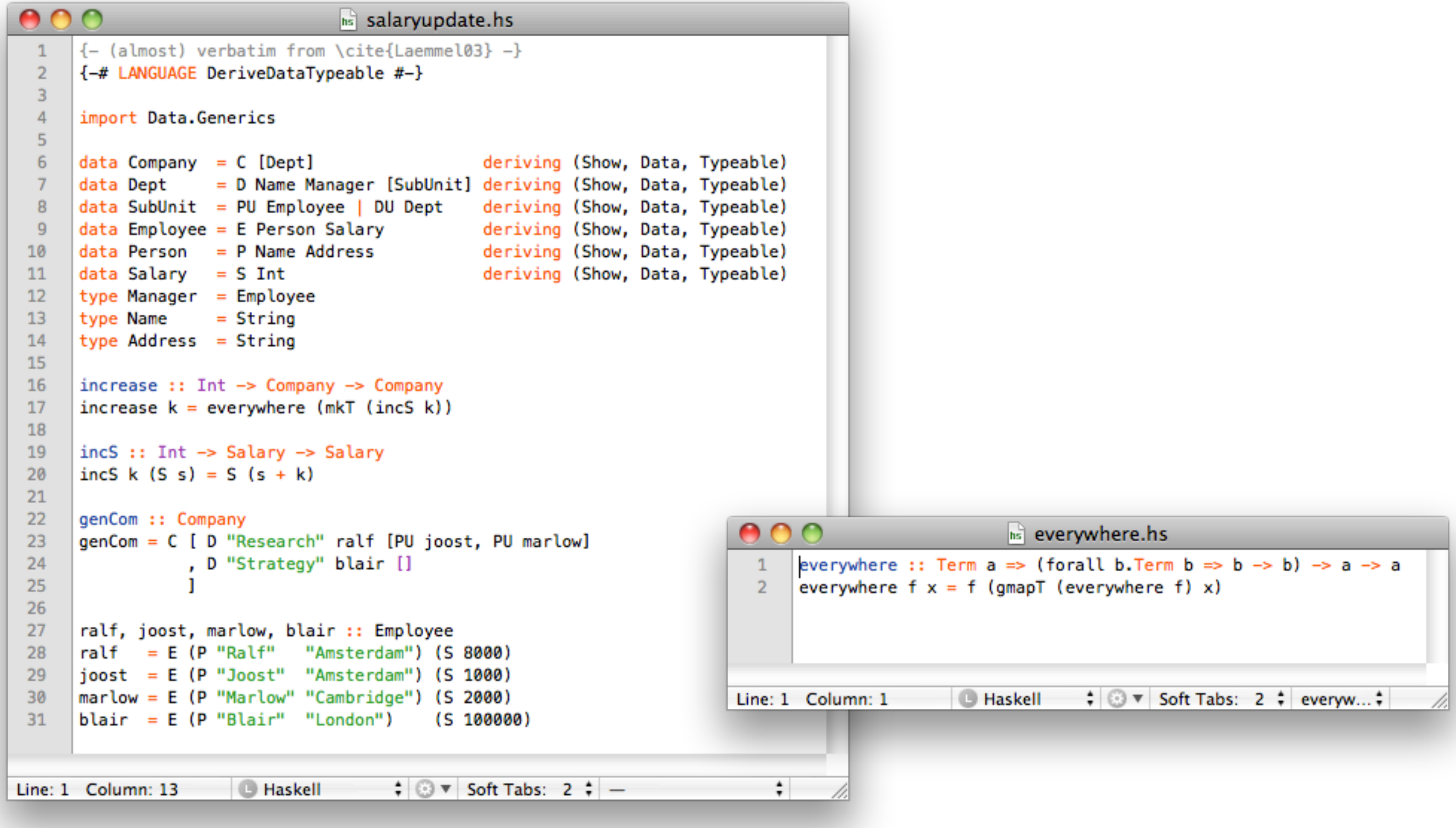
Datatype Generic is Nice



Scrap Your Boilerplate:
A Practical Design Pattern for Generic Programming

Ralf Lämmel
Vrije Universiteit, Amsterdam

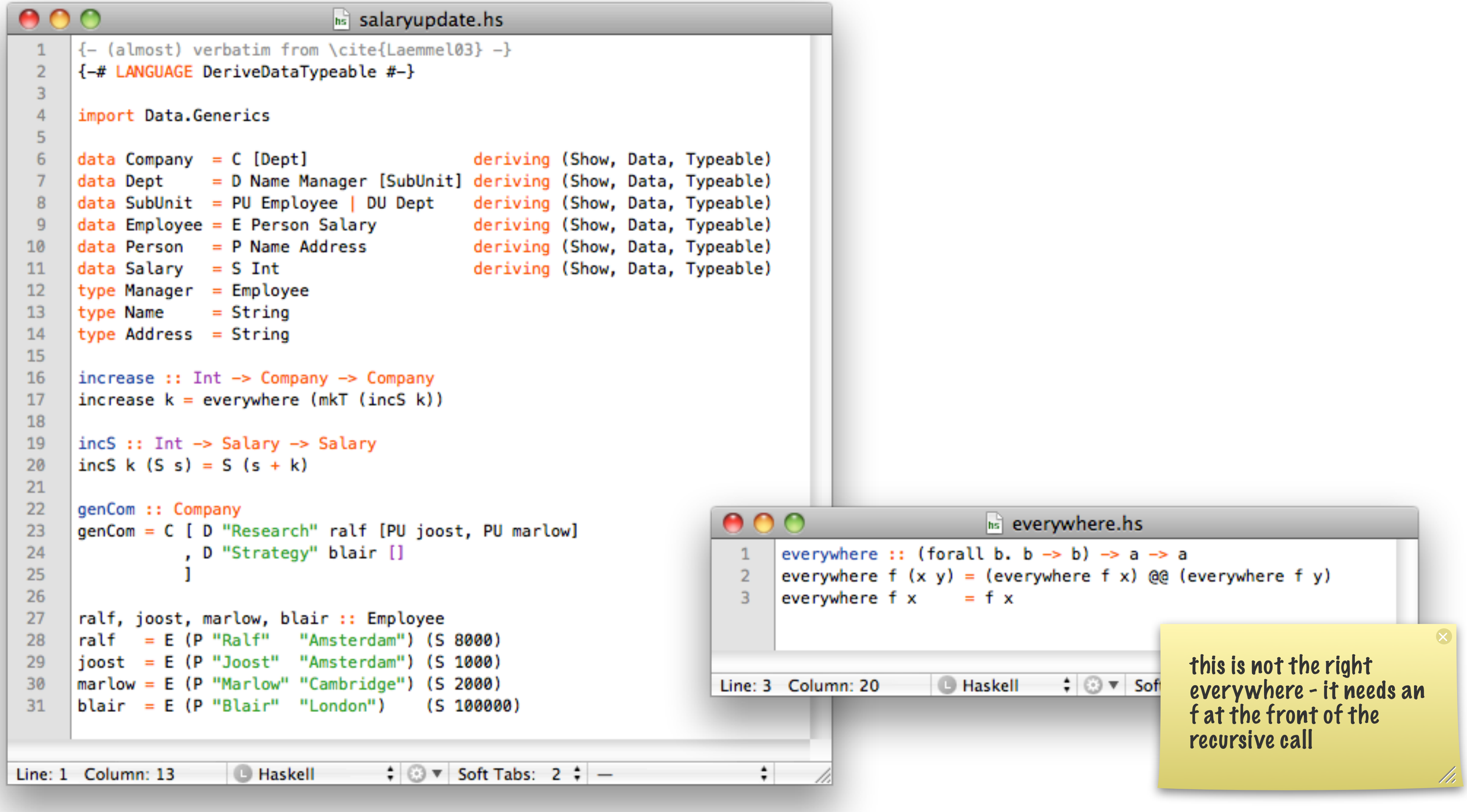
Simon Peyton Jones
Microsoft Research, Cambridge



```haskell
{- (almost) verbatim from \cite{Laemmel03} -}
{-# LANGUAGE DeriveDataTypeable #-}

import Data.Generics

data Company  = C [Dept]                   deriving (Show, Data, Typeable)
data Dept     = D Name Manager [SubUnit]   deriving (Show, Data, Typeable)
data SubUnit  = PU Employee | DU Dept      deriving (Show, Data, Typeable)
data Employee = E Person Salary            deriving (Show, Data, Typeable)
data Person   = P Name Address             deriving (Show, Data, Typeable)
data Salary   = S Int                      deriving (Show, Data, Typeable)
type Manager  = Employee
type Name     = String
type Address  = String

increase :: Int -> Company -> Company
increase k = everywhere (mkT (incS k))

incS :: Int -> Salary -> Salary
incS k (S s) = S (s + k)

genCom :: Company
genCom = C [ D "Research" ralf [PU joost, PU marlow]
           , D "Strategy" blair []
           ]

ralf, joost, marlow, blair :: Employee
ralf   = E (P "Ralf"   "Amsterdam") (S 8000)
joost  = E (P "Joost"  "Amsterdam") (S 1000)
marlow = E (P "Marlow" "Cambridge") (S 2000)
blair  = E (P "Blair"  "London")    (S 100000)
```

```haskell
everywhere :: Term a => (forall b.Term b => b -> b) -> a -> a
everywhere f x = f (gmapT (everywhere f) x)
```

Explicit Spine View is Nicer



```haskell
{- (almost) verbatim from \cite{Laemmel03} -}
{-# LANGUAGE DeriveDataTypeable #-}

import Data.Generics

data Company  = C [Dept]                  deriving (Show, Data, Typeable)
data Dept     = D Name Manager [SubUnit]  deriving (Show, Data, Typeable)
data SubUnit  = PU Employee | DU Dept     deriving (Show, Data, Typeable)
data Employee = E Person Salary           deriving (Show, Data, Typeable)
data Person   = P Name Address            deriving (Show, Data, Typeable)
data Salary   = S Int                     deriving (Show, Data, Typeable)
type Manager  = Employee
type Name     = String
type Address  = String

increase :: Int -> Company -> Company
increase k = everywhere (mkT (incS k))

incS :: Int -> Salary -> Salary
incS k (S s) = S (s + k)

genCom :: Company
genCom = C [ D "Research" ralf [PU joost, PU marlow]
           , D "Strategy" blair []
           ]

ralf, joost, marlow, blair :: Employee
ralf   = E (P "Ralf"   "Amsterdam") (S 8000)
joost  = E (P "Joost"  "Amsterdam") (S 1000)
marlow = E (P "Marlow" "Cambridge") (S 2000)
blair  = E (P "Blair"  "London")    (S 100000)
```

Line: 1 Column: 13 Haskell Soft Tabs: 2

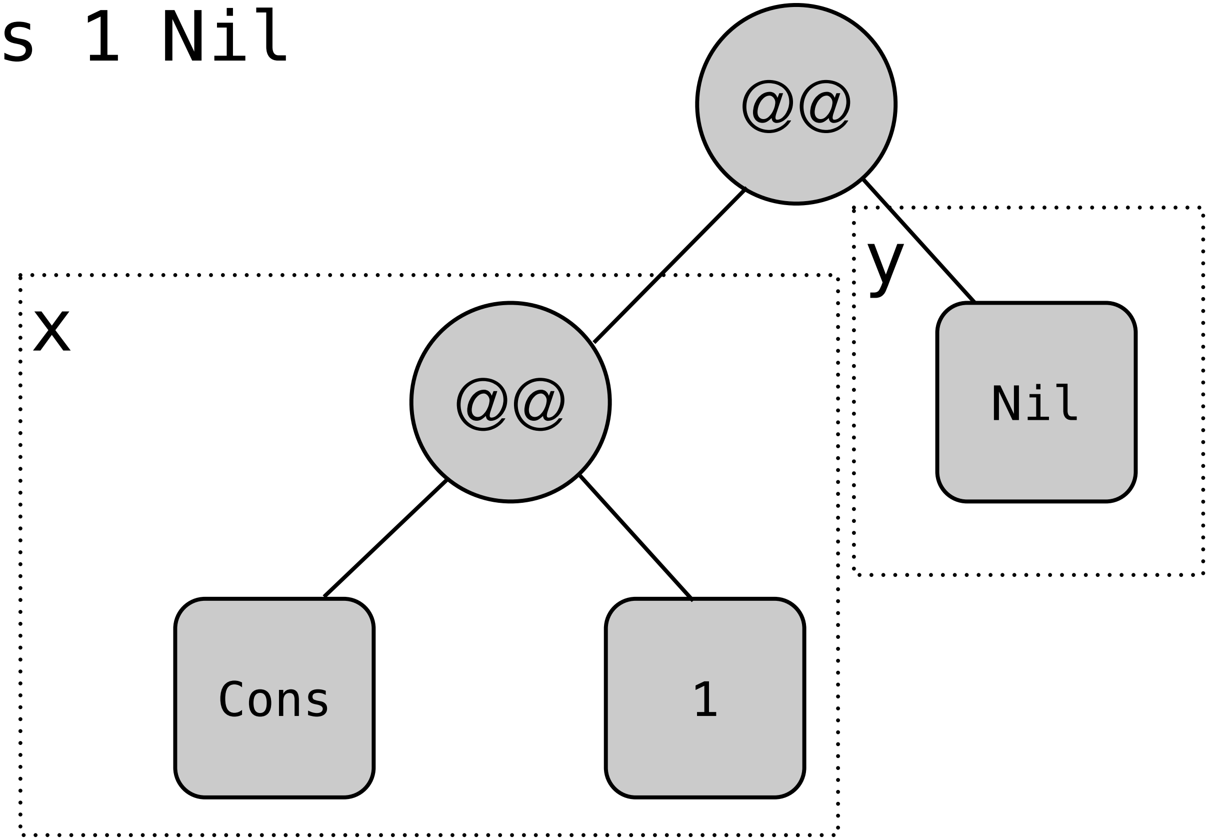
```haskell
everywhere :: (forall b. b -> b) -> a -> a
everywhere f (x y) = (everywhere f x) @@ (everywhere f y)
everywhere f x     = f x
```

Line: 3 Column: 20 Haskell Sof

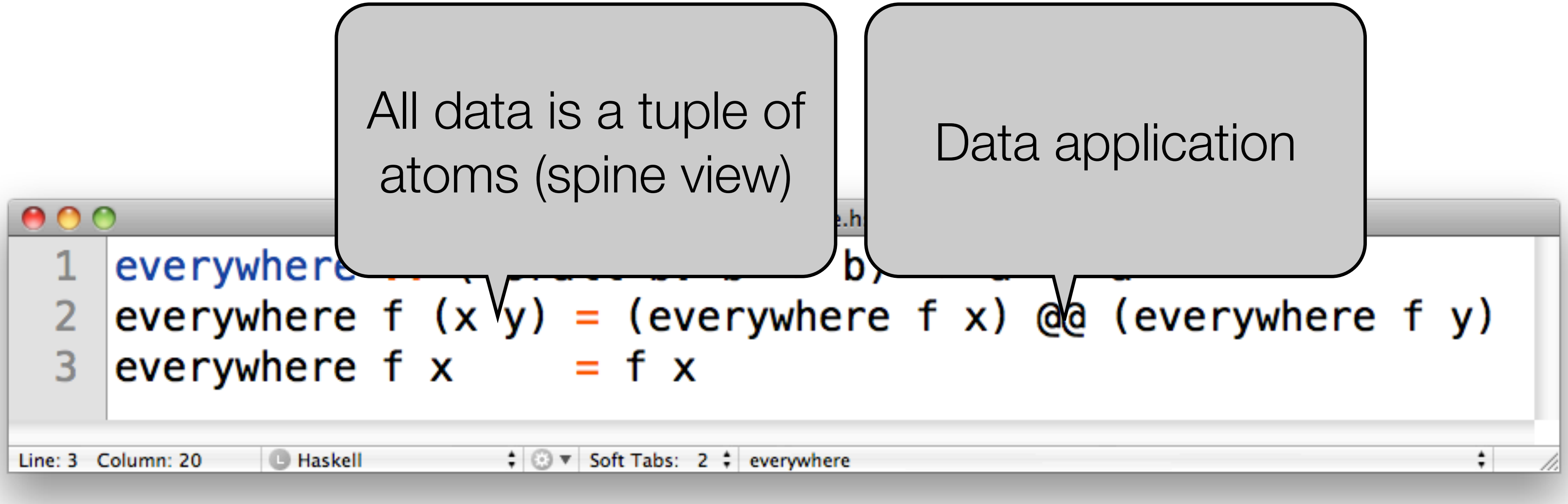
this is not the right everywhere - it needs an f at the front of the recursive call

Spine view (tuples of atoms)

```
Cons 1 Nil
```



But this does not exist



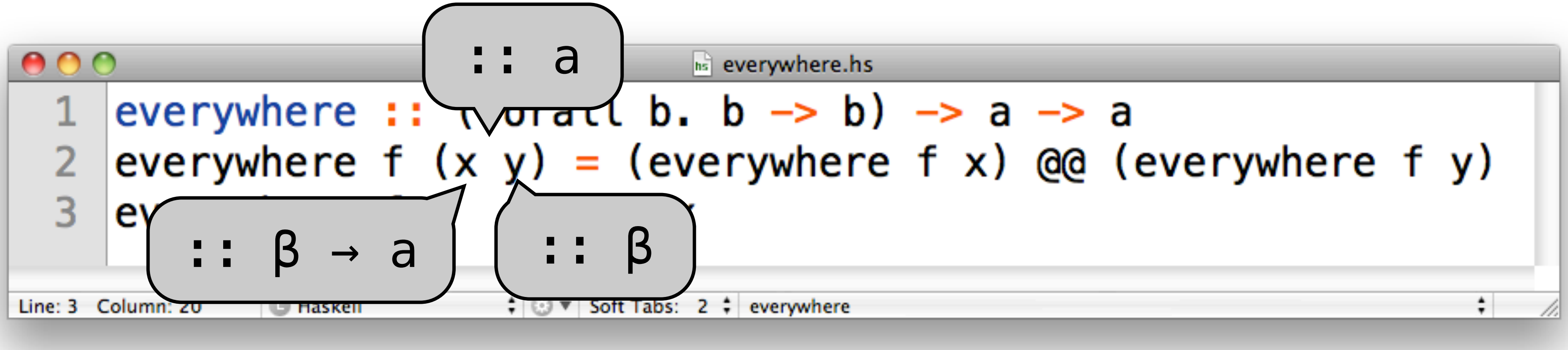
All data is a tuple of atoms (spine view)

Data application

```haskell
everywhere :: (forall b. b -> b) -> a -> a
everywhere f (x y) = (everywhere f x) @@ (everywhere f y)
everywhere f x     = f x
```

Line: 3 Column: 20 Haskell Soft Tabs: 2 everywhere

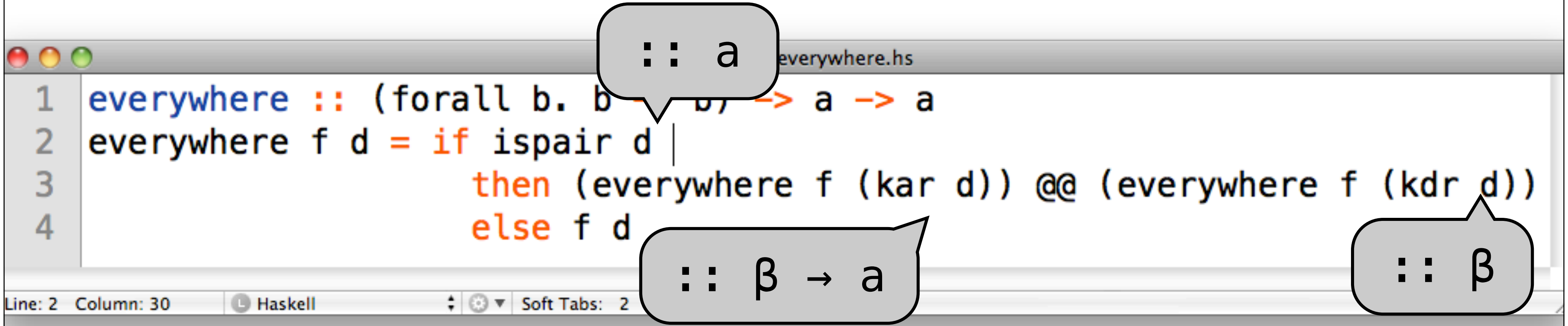
Types!



```haskell
everywhere :: (forall b. b -> b) -> a -> a
everywhere f (x y) = (everywhere f x) @@ (everywhere f y)
ev
```

The βs must be the same, but there is no link between them

Compiled version



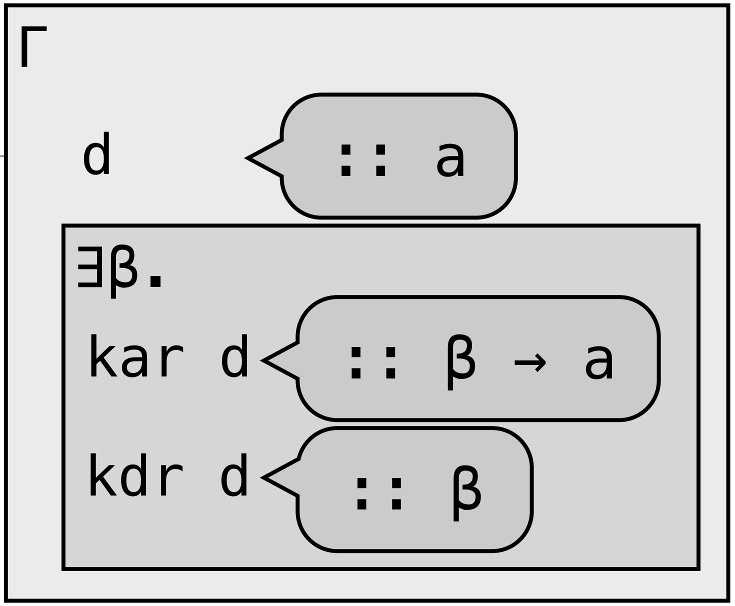
```haskell
everywhere :: (forall b. b -> b) -> a -> a
everywhere f d = if ispair d
                    then (everywhere f (kar d)) @@ (everywhere f (kdr d))
                    else f d
```

Line: 2 Column: 30 Haskell Soft Tabs: 2

The βs must be the same, we link them at the ispair

Type algorithm

every kar,kdr witnessed by an ispair



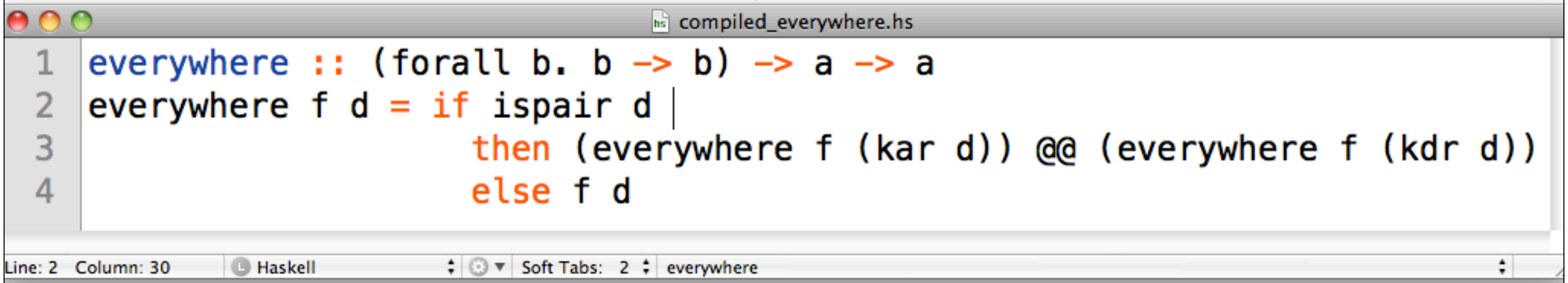
Γ

d :: a

∃β.

kar d :: β → a

kdr d :: β



compiled_everywhere.hs

```haskell
everywhere :: (forall b. b -> b) -> a -> a
everywhere f d = if ispair d
                    then (everywhere f (kar d)) @@ (everywhere f (kdr d))
                    else f d
```

Line: 2 Column: 30 Haskell Soft Tabs: 2 everywhere

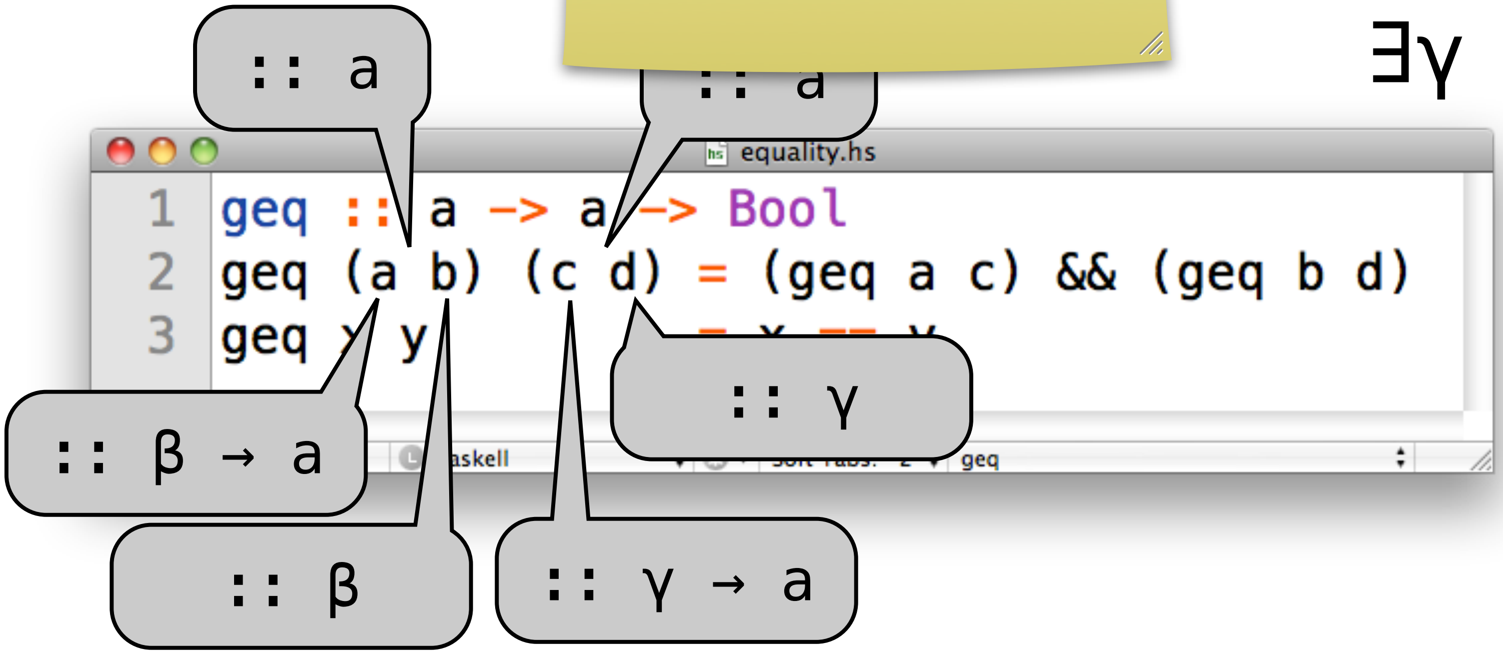
the triple, ispair, kar and kdr are acting like a single function (a fold) for the type system.

Generic Equality

This slide is incorrect and should be ignored.

e same but
on is lost

∃α
∃γ



```
geq :: a -> a -> Bool
geq (a b) (c d) = (geq a c) && (geq b d)
geq x y = x == y
```

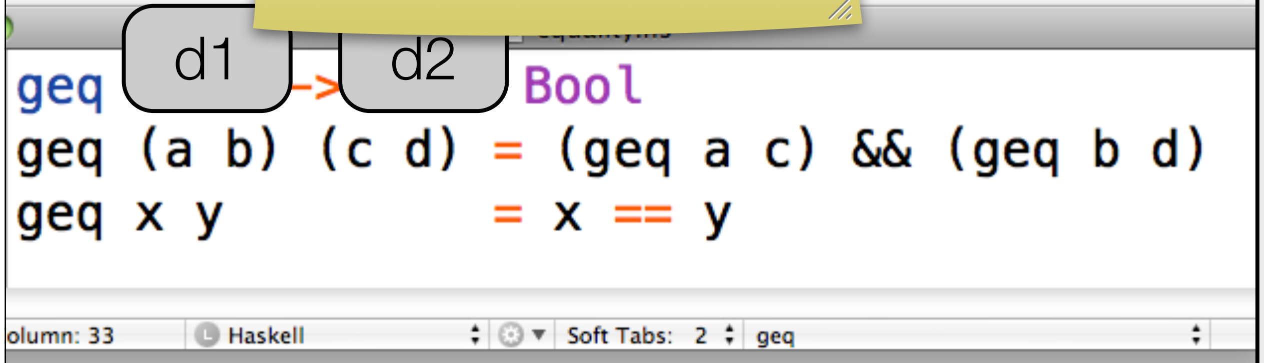
Existentially quantified variables have to be treated as constants - they can only unify with themselves.

Type algorithm

This slide is incorrect and should be ignored.

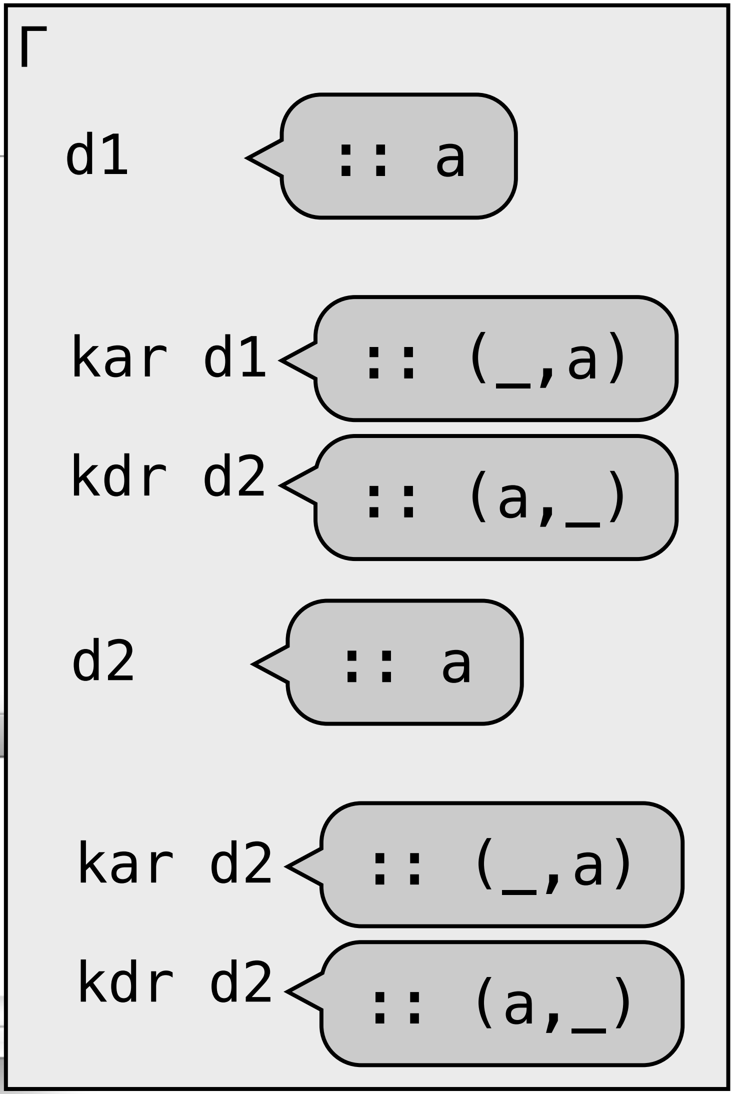
w he

info iously



```
geq        ->      Bool
geq (a b) (c d) = (geq a c) && (geq b d)
geq x y         = x == y
```

olumn: 33 | Haskell | Soft Tabs: 2 | geq



Γ

d1 :: a

kar d1 :: (_,a)

kdr d2 :: (a,_)

d2 :: a

kar d2 :: (_,a)

kdr d2 :: (a,_)

the triple, ispair, kar and kdr are acting like a single function (a fold) for the type system.

Opens up a whole world

We don’t have time to cover more here, but you will find all these details and more in my upcoming dissertation

▽ warning! you need to learn a new syntax

You can play with dgen at http://dgen.science.mq.edu.au:8080