

# Dictionaries: lazy or eager type class witnesses?

Tom Schrijvers

Katholieke Universiteit Leuven, Belgium  
TOM.SCHRIJVERS@CS.KULEUVEN.BE

May 15, 2009





- 1 Ad-hoc Overloading
- 2 Type Classes
- 3 Implementation
- 4 Lazy or Strict Witnesses
- 5 Conclusion



- 1 Ad-hoc Overloading
- 2 Type Classes
- 3 Implementation
- 4 Lazy or Strict Witnesses
- 5 Conclusion

```
iMax :: Int -> Int -> Int
iMax x y = if x 'intGE' y then x
           else y

intGE = ... -- built-in >= for integers
```

```
iMax :: Int -> Int -> Int
iMax x y = if x 'intGE' y then x
           else y
```

```
intGE = ... -- built-in >= for integers
```

```
fMax :: Float -> Float -> Float
fMax x y = if x 'floatGE' y then x
           else y
```

```
floatGE = ... -- built-in >= for floats
```

```
hoMax :: (a -> a -> Bool) -> a -> a -> a
hoMax (>=) x y = if x >= y then x
                  else y

iMax = hoMax intGE
fMax = hoMax floatGE
```

- ▶ greater flexibility
- ▶ extra higher-order parameter (>=)

Now build on hoMax:

```
hoMaximum :: (a -> a -> Bool) -> [a] -> a
hoMaximum _ [x] = x
hoMaximum (>=) (x:y:ys) =
    hoMaximum (>=) (hoMax (>=) x y : ys)

main = hoMaximum intGE [5,7,42,0]
```

Now build on hoMax:

```
hoMaximum :: (a -> a -> Bool) -> [a] -> a
hoMaximum _ [x] = x
hoMaximum (>=) (x:y:ys) =
    hoMaximum (>=) (hoMax (>=) x y : ys)

main = hoMaximum intGE [5,7,42,0]
```

- ▶ extra parameter (>=) spreads

Now build on hoMax:

```
hoMaximum :: (a -> a -> Bool) -> [a] -> a
hoMaximum _ [x] = x
hoMaximum (>=) (x:y:ys) =
    hoMaximum (>=) (hoMax (>=) x y : ys)

main = hoMaximum intGE [5,7,42,0]
```

- ▶ extra parameter (>=) spreads
- ▶ overly verbose

Now build on hoMax:

```
hoMaximum :: (a -> a -> Bool) -> [a] -> a
hoMaximum _ [x] = x
hoMaximum (>=) (x:y:ys) =
    hoMaximum (>=) (hoMax (>=) x y : ys)

main = hoMaximum intGE [5,7,42,0]
```

- ▶ extra parameter (>=) spreads
- ▶ overly verbose
- ▶ must be chosen  
extra opportunity for bugs

Now build on hoMax:

```
hoMaximum :: (a -> a -> Bool) -> [a] -> a
hoMaximum _ [x] = x
hoMaximum (>=) (x:y:ys) =
    hoMaximum (>=) (hoMax (>=) x y : ys)

main = hoMaximum intGE [5,7,42,0]
```

- ▶ extra parameter (>=) spreads
- ▶ overly verbose
- ▶ must be chosen  
extra opportunity for bugs
- ▶ must be passed explicitly  
extra opportunity for bugs



- 1 Ad-hoc Overloading
- 2 **Type Classes**
- 3 Implementation
- 4 Lazy or Strict Witnesses
- 5 Conclusion

Type classes allow you to write:

```
max x y = if x >= y then x
          else y

maximum [x]      = x
maximum (x:y:ys) = maximum (max x y : ys)

main1 = maximum [5,7,42,0]
main2 = maximum [5.0,0.7,4.2,0.0]
```

- ▶ don't pass the comparison function explicitly
- ▶ just use the **default** implementation (whatever that may be)
- ▶ default implementation chosen once and forall

Declare overloaded methods and their implementations

```
class Ord a where  
  (>=) :: a -> a -> Bool
```

Declare overloaded methods and their implementations

```
class Ord a where  
  (>=) :: a -> a -> Bool
```

```
instance Ord Int where  
  (>=) = intGE
```

```
instance Ord Float where  
  (>=) = floatGE
```

Declare overloaded methods and their implementations

```
class Ord a where
  (>=) :: a -> a -> Bool

instance Ord Int where
  (>=) = intGE

instance Ord Float where
  (>=) = floatGE

instance Ord a => Ord [a] where
  []      >= []      = True
  []      >= _      = False
  (x:xs) >= []      = True
  (x:xs) >= (y:ys) = x >= y && xs >= ys
```

## Type class constraint in signature

- ▶ instance (i.e. implementation) for this type must exist!
- ▶ ad-hoc overloading
  - ▶ not parametric polymorphism
  - ▶ not data-type genericity

```
max :: Ord a => a -> a -> a
max x y = if x >= y then x
          else y
```

## Type class constraint in signature

- ▶ instance (i.e. implementation) for this type must exist!
- ▶ ad-hoc overloading
  - ▶ not parametric polymorphism
  - ▶ not data-type genericity

```
max :: Ord a => a -> a -> a
max x y = if x >= y then x
          else y
```

```
maximum :: Ord a => [a] -> a
maximum [x] = x
maximum (x:y:ys) = maximum (max x y : ys)
```

```
*Main> maximum [5,7,42,0]
```

```
42
```

```
*Main> maximum [5.0,0.7,4.2,0.0]
```

```
5.0
```

```
*Main> maximum [5,7,42,0]
```

```
42
```

```
*Main> maximum [5.0,0.7,4.2,0.0]
```

```
5.0
```

```
*Main> maximum [\a b -> a,\a b -> b]
```

```
*Main> maximum [5,7,42,0]
42
*Main> maximum [5.0,0.7,4.2,0.0]
5.0
*Main> maximum [\a b -> a,\a b -> b]
```

No instance for (Ord (t -> t -> t))  
arising from a use of 'maximum'

Possible fix:

add an instance declaration for (Ord (t -> t -> t))

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

  -- Minimal complete definition:
  --      (==) or (/=)
x /= y    = not (x == y)
x == y    = not (x /= y)
```

Illustrates:

- ▶ multiple methods
- ▶ default implementation



- 1 Ad-hoc Overloading
- 2 Type Classes
- 3 Implementation**
- 4 Lazy or Strict Witnesses
- 5 Conclusion

## Transformation-based implementation:

- 1 Specialization
- 2 Runtime witnesses: type representation
- 3 Runtime witnesses: dictionaries

## Low-level implementation:

- ▶ (type tags)

- ▶ compile-time specialization for each type the code is called with
- ▶ similar to C++ templates

### Specialized for Int

```
iMax :: Int -> Int -> Int
```

```
iMax x y = if x 'intGE' y then x  
          else y
```

```
iMaximum :: [Int] -> Int
```

```
iMaximum [x] = x
```

```
iMaximum (x:y:ys) = iMaximum (iMax x y : ys)
```

- ▶ code explosion hazard
- ▶ over-eager specialization hazard: non-termination of compiler
- ▶ C++ templates: template instantiation limit
- ▶ cannot handle unbounded instantiation depth, non-uniform recursion

```
data Seq a = Nil | Cons a (Seq (a,a))
```

```
same Nil Nil          = True
```

```
same (Cons x xs) (Cons y ys) =
```

```
  x == y && same xs ys
```

```
-- requires, e.g.
```

```
-- Eq Int, Eq (Int,Int), Eq ((Int,Int),(Int,Int)), ...
```

*Provide witness that can point us to the method implementation.*

```
max w x y = if (>=) w x y then x
           else y
```

```
maximum w [x] = x
```

```
maximum w (x:y:ys) = maximum w (max w x y : ys)
```

- ▶ responsibility of type checker to add witness parameter
- ▶ allow generic code (single copy)
- ▶ allow witnesses to be built dynamically, as needed
- ▶ expressible in Haskell-without-type-classes

Now what can we use as witness?

- ▶ witness = reification of type as a value

```
data TypeRep = IntType
              | FloatType
              | ListType TypeRep

(>=) :: TypeRep -> a -> a -> Bool
(>=) w x y =
  case w of
    IntType      -> x 'intGE'    y
    FloatType    -> x 'floatGE'  y
    ListType we  -> listGE we x y

main = maximum IntType [5,7,42,0]
```

- ▶ witness = reification of type as a value

```
data TypeRep = IntType
              | FloatType
              | ListType TypeRep

(>=) :: TypeRep -> a -> a -> Bool
(>=) w x y =
  case w of
    IntType      -> x 'intGE'    y
    FloatType    -> x 'floatGE'  y
    ListType we  -> listGE we x y

main = maximum IntType [5,7,42,0]
```

**Issue 1:** Not well-typed in Haskell'98 / System F!

- fix: GADT for type coercion

```
data TypeRep a where
  IntType    :: TypeRep Int
  FloatType  :: TypeRep Float
  ListType   :: TypeRep a -> TypeRep [a]

(>=) :: TypeRep a -> a -> a -> Bool
(>=) w x y =
  case w of
    IntType    -> x 'intGE'    y
    FloatType  -> x 'floatGE'  y
    ListType we -> listGE we x y
```

Well-typed in GHC / System  $F_C$

**Issue 2:** Non-modular!

Adding a new instance requires

- ▶ adding a new constructor
- ▶ adding a branch to the case expression

```
data TypeRep a where
  IntType    :: TypeRep Int
  FloatType  :: TypeRep Float
  ListType   :: TypeRep a -> TypeRep [a]

(>=) :: TypeRep a -> a -> a -> Bool
(>=) w x y =
  case w of
    IntType    -> x 'intGE'    y
    FloatType  -> x 'floatGE'  y
    ListType we -> listGE we x y
```

- ▶ witness = dictionary (record) of method implementations

```
data OrdDict a = OD { (>=) :: a -> a -> Bool }
```

- ▶ witness = dictionary (record) of method implementations

```
data OrdDict a = OD { (>=) :: a -> a -> Bool }
```

```
intOrd :: OrdDict Int
```

```
intOrd = OD { (>=) = intGE }
```

```
floatOrd :: OrdDict Float
```

```
floatOrd = OD { (>=) = floatGE }
```

```
listOrd :: OrdDict a -> OrdDict [a]
```

```
listOrd ed = OD { (>=) = listGE ed }
```

- ▶ witness = dictionary (record) of method implementations

```
data OrdDict a = OD { (>=) :: a -> a -> Bool }
```

```
intOrd :: OrdDict Int
```

```
intOrd = OD { (>=) = intGE }
```

```
floatOrd :: OrdDict Float
```

```
floatOrd = OD { (>=) = floatGE }
```

```
listOrd :: OrdDict a -> OrdDict [a]
```

```
listOrd ed = OD { (>=) = listGE ed }
```

```
main = maximum intOrd [5,7,42,0]
```

- ▶ witness = dictionary (record) of method implementations

```
data OrdDict a = OD { (>=) :: a -> a -> Bool }
```

```
intOrd :: OrdDict Int
```

```
intOrd = OD { (>=) = intGE }
```

```
floatOrd :: OrdDict Float
```

```
floatOrd = OD { (>=) = floatGE }
```

```
listOrd :: OrdDict a -> OrdDict [a]
```

```
listOrd ed = OD { (>=) = listGE ed }
```

```
main = maximum intOrd [5,7,42,0]
```

- ▶ **Issue 1:** well-typed in Haskell 98 / System F
- ▶ **Issue 2:** modularly extensible

- ▶ field selectors generated for record syntax

```
data OrdDict a = OD { (>=) :: a -> a -> Bool }
```

```
==>
```

```
(>=) :: OrdDict a -> a -> a -> Bool
```

```
(>=) d x y = case d of  
             OD m -> m x y
```

- ▶ field selectors generated for record syntax

```
data OrdDict a = OD { (>=) :: a -> a -> Bool }
```

```
==>
```

```
(>=) :: OrdDict a -> a -> a -> Bool
```

```
(>=) d x y = case d of
              OD m -> m x y
```

```
data EqDict a = ED { (==) :: a -> a -> Bool
                    , (/=) :: a -> a -> Bool }
```

```
==>
```

```
(==) :: EqDict a -> a -> a -> Bool
```

```
(==) d x y = case d of
              ED m1 m2 -> m1 x y
```



- 1 Ad-hoc Overloading
- 2 Type Classes
- 3 Implementation
- 4 Lazy or Strict Witnesses**
- 5 Conclusion

- ▶ In Haskell, all values are lazy (by default).

- ▶ *declaratively* :

```
data Lazy a = Lifted a | Bottom
```

- ▶ *operationally* :

```
type Lazy a = () -> a
```

- ▶ In Haskell, all values are lazy (by default).

- ▶ *declaratively* :

```
data Lazy a = Lifted a | Bottom
```

- ▶ *operationally* :

```
type Lazy a = () -> a
```

- ▶ Lazy values must be forced to inspect them:

```
force :: Lazy a -> a
```

```
case b of
  True  -> ...
  False -> ...
==>
case force b of
  True  -> ...
  False -> ...
```

- ▶ In Haskell, all values are lazy.
- ▶ Dictionaries are values.
- ▶ Ergo dictionaries are lazy.

```
(>=) d = case d of
          OD m  -> m
```

==>

```
(>=) d = case force d of
          OD m  -> m
```

- ▶ In Haskell, all values are lazy.
- ▶ Dictionaries are values.
- ▶ Ergo dictionaries are lazy.

```
(>=) d = case d of
         OD m  -> m
```

==>

```
(>=) d = case force d of
         OD m  -> m
```

...but can dictionaries be Bottom?

What does a bottom dictionary mean?

- ▶ Declaratively: no dictionary is available.
- ▶ Operationally: non-terminating computation/error.

What does a bottom dictionary mean?

- ▶ Declaratively: no dictionary is available.
- ▶ Operationally: non-terminating computation/error.

Can this happen? Who is to blame?

- ▶ The type checker checks whether a type class instance exists.
- ▶ The type checker generates the dictionaries.

What does a bottom dictionary mean?

- ▶ Declaratively: no dictionary is available.
- ▶ Operationally: non-terminating computation/error.

Can this happen? Who is to blame?

- ▶ The type checker checks whether a type class instance exists.
- ▶ The type checker generates the dictionaries.

The type checker does not lie! It **never** generates bottom dictionaries.

The type checker guarantees the existence of a dictionary:

- ▶ Dictionaries are **never** Bottom in Haskell'98!
- ▶ Dictionaries don't have to be **forced**.

Optimization waiting to be exploited:

- ▶ use **case** without **force** to get a speed-up

```
(>=) d = case d of
        OD m  -> m
```

==>

```
(>=) d = case d of
        OD m  -> m
```

## Generalized Algebraic Data Types (GADTs)

- ▶ construction captures type class constraint
- ▶ pattern match releases type class constraint

```
data T a where
  K :: Ord a => T a
  f :: T a -> a -> a -> Bool
  f t x y = case t of
             K -> x >= y
main = f K 1 2
```

## Generalized Algebraic Data Types (GADTs)

- ▶ construction captures type class constraint
- ▶ pattern match releases type class constraint

```
data T a where
  K :: Ord a => T a
  f :: T a -> a -> a -> Bool
  f t x y = case t of
             K -> x >= y
main = f K 1 2
```

==>

```
data T a where
  K :: OrdDict a -> T a
  f :: T a -> a -> a -> Bool
  f t x y = case force t of
             K w -> (>=) w x y
main = f (K intOrd) 1 2
```

## Generalized Algebraic Data Types (GADTs)

- ▶ construction captures type class constraint
- ▶ pattern match releases type class constraint

Dictionaries are eager because:

- ▶ type checker tucks eager dictionary in GADT
- ▶ forced GADT pattern match exposes eager dictionary
- ▶ dictionary only used after forced GADT pattern match

*Recursive (Co-inductive) Dictionaries*

[Lämmel&amp;Peyton Jones]

- ▶ two or more dictionaries with cyclic dependencies

```

data NatF n = Z | S n
data Fix f = Fix (f (Fix f))
type Nat = Fix Nat
zero, one :: Nat
zero = Fix Z                                one = Fix (S (Fix Z))

instance Eq n => Eq (Nat n) where
  Z      == Z      = True
  (S x) == (S y)  = x == y
  _      == _      = x == y
instance Eq (f (Fix f))=> Eq (Fix f) where
  (Fix x) == (Fix y) = x == y

main = zero == one

```

*Recursive (Co-inductive) Dictionaries*

- ▶ two or more dictionaries with cyclic dependencies

```
instance Eq n => Eq (Nat n) where
  Z      == Z      = True
  (S x) == (S y)  = x == y
  _      == _      = x == y
instance Eq (f (Fix f))=> Eq (Fix f) where
  (Fix x) == (Fix y) = x == y
```

```
main = zero == one
```

```
==>
```

```
main = let w1 = fixEq w2
          w2 = natfEq w1
        in (==) w1 zero one
```

*Recursive* (Co-inductive) Dictionaries

- ▶ two or more dictionaries with cyclic dependencies

*Recursive (Co-inductive) Dictionaries*

- ▶ two or more dictionaries with cyclic dependencies
- ▶ requires laziness to build the cycle  
strict pure functional languages can't build cycles  
(impure ones can!)

*Recursive* (Co-inductive) Dictionaries

- ▶ two or more dictionaries with cyclic dependencies
- ▶ requires laziness to build the cycle  
strict pure functional languages can't build cycles  
(impure ones can!)

Ok, so the dictionary has to be lazy, right?

*Recursive* (Co-inductive) Dictionaries

- ▶ two or more dictionaries with cyclic dependencies
- ▶ requires laziness to build the cycle  
strict pure functional languages can't build cycles  
(impure ones can!)

Ok, so the dictionary has to be lazy, right?

- ▶ yes, but ...

*Recursive (Co-inductive) Dictionaries*

- ▶ two or more dictionaries with cyclic dependencies
- ▶ requires laziness to build the cycle  
strict pure functional languages can't build cycles  
(impure ones can!)

Ok, so the dictionary has to be lazy, right?

- ▶ yes, but ...
- ▶ the constructor doesn't depend on the other dictionary!

```
fixEq w1 = EqDict { (==) = fixe w2
                  , (/=) = fixneq w2 }
natfEq w2 = EqDict { (==) = natfeq w1
                   , (/=) = natfneq w1 }
```

(relies on dictionary fields being non-strict)



- 1 Ad-hoc Overloading
- 2 Type Classes
- 3 Implementation
- 4 Lazy or Strict Witnesses
- 5 Conclusion**

You've seen

- ▶ an overview of type classes
- ▶ possible implementations of type classes
- ▶ why dictionaries don't have to be lazy
- ▶ eager dictionaries scale beyond Haskell'98

# Questions & Remarks?