

# Ad-hoc Polymorphism and Dynamic Typing in a Statically Typed Functional Language

Thomas van Noort

joint work with  
Peter Achten    Rinus Plasmeijer

Institute for Computing and Information Sciences  
Radboud University Nijmegen

Workshop on Generic Programming  
September 26, 2010

# Introduction

Static typing is the cornerstone of Clean, Haskell, and ML:

- ▶ It prevents most erroneous run-time behaviour
- ▶ It provides opportunities for compile-time optimisations

# Introduction

Static typing is the cornerstone of Clean, Haskell, and ML:

- ▶ It prevents most erroneous run-time behaviour
- ▶ It provides opportunities for compile-time optimisations

But sometimes, types are not known until run time:

- ▶ Exchanging values between applications by (de)serialisation
- ▶ Receiving input provided by a user
- ▶ Obtaining values from a network connection
- ▶ ...

Typically, when interacting with the 'outside' world

## An ideal world

Dynamic typing defers type unification until run time:

- ▶ Values are wrapped in a uniform black box
- ▶ Values are unwrapped by providing an expected type
- ▶ Type safety is still guaranteed

## An ideal world

Dynamic typing defers type unification until run time:

- ▶ Values are wrapped in a uniform black box
- ▶ Values are unwrapped by providing an expected type
- ▶ Type safety is still guaranteed

Combination of static and dynamic typing is best, as phrased by:

*“Static typing where possible, dynamic typing when needed”*  
(Meijer and Drayton, 2004)

## An ideal world

Dynamic typing defers type unification until run time:

- ▶ Values are wrapped in a uniform black box
- ▶ Values are unwrapped by providing an expected type
- ▶ Type safety is still guaranteed

Combination of static and dynamic typing is best, as phrased by:

*“Static typing where possible, dynamic typing when needed”*  
(Meijer and Drayton, 2004)

Ideally, statically typed language is the starting point, where:

- ▶ Language provides an escape to type values dynamically
- ▶ Dynamic typing system is orthogonal to the static system
- ▶ No restrictions on values or types that can be dynamic

# The current state of affairs of dynamic typing

The most common type concepts are supported:

# The current state of affairs of dynamic typing

The most common type concepts are supported:

- ▶ Monomorphism, e.g., (un)wrapping values 'a' or [*False*]
  - ▶ Haskell (Baars and Swierstra, 2002; Cheney and Hinze, 2002)
  - ▶ Clean and ML (Abadi et al., 1991; Pil, 1997)

# The current state of affairs of dynamic typing

The most common type concepts are supported:

- ▶ Monomorphism, e.g., (un)wrapping values 'a' or [*False*]
  - ▶ Haskell (Baars and Swierstra, 2002; Cheney and Hinze, 2002)
  - ▶ Clean and ML (Abadi et al., 1991; Pil, 1997)
- ▶ Parametric polymorphism, e.g., (un)wrapping values  $\perp$  or *id*
  - ▶ Clean and ML (Abadi et al., 1994; Leroy and Mauny, 1993)

# The current state of affairs of dynamic typing

The most common type concepts are supported:

- ▶ Monomorphism, e.g., (un)wrapping values 'a' or [*False*]
  - ▶ Haskell (Baars and Swierstra, 2002; Cheney and Hinze, 2002)
  - ▶ Clean and ML (Abadi et al., 1991; Pil, 1997)
- ▶ Parametric polymorphism, e.g., (un)wrapping values  $\perp$  or *id*
  - ▶ Clean and ML (Abadi et al., 1994; Leroy and Mauny, 1993)

## Shameless plug

Haskell '10 paper on using Clean's dynamic typing in Haskell:

*Exchanging Sources Between Clean and Haskell -  
A Double-Edged Front End for the Clean Compiler*

# The current state of affairs of dynamic typing

The most common type concepts are supported:

- ▶ Monomorphism, e.g., (un)wrapping values 'a' or [*False*]
  - ▶ Haskell (Baars and Swierstra, 2002; Cheney and Hinze, 2002)
  - ▶ Clean and ML (Abadi et al., 1991; Pil, 1997)
- ▶ Parametric polymorphism, e.g., (un)wrapping values  $\perp$  or *id*
  - ▶ Clean and ML (Abadi et al., 1994; Leroy and Mauny, 1993)

## Shameless plug

Haskell '10 paper on using Clean's dynamic typing in Haskell:

*Exchanging Sources Between Clean and Haskell -  
A Double-Edged Front End for the Clean Compiler*

Ad-hoc polymorphism is a type concept that is missing:

- ▶ Type classes in Clean and Haskell, modules in ML
- ▶ Usual suspects: equality and sorting

# Goal

Explore their interactions in a statically typed functional language:

# Goal

Explore their interactions in a statically typed functional language:

- ▶ Dynamic typing in ad-hoc polymorphism
  - ▶ e.g., sorting a dynamically typed list of values

# Goal

Explore their interactions in a statically typed functional language:

- ▶ Dynamic typing in ad-hoc polymorphism
  - ▶ e.g., sorting a dynamically typed list of values
- ▶ Ad-hoc polymorphism in dynamic typing
  - ▶ e.g., (de)serialisation of a sorting function

# Goal

Explore their interactions in a statically typed functional language:

- ▶ Dynamic typing in ad-hoc polymorphism
  - ▶ e.g., sorting a dynamically typed list of values
- ▶ Ad-hoc polymorphism in dynamic typing
  - ▶ e.g., (de)serialisation of a sorting function

Examples are defined using Clean syntax with curried types:

- ▶ Ad-hoc polymorphism by 'simple' type classes
- ▶ Dynamic typing by Clean's extensive system

## Preliminaries - Ad-hoc polymorphism

## Ad-hoc polymorphism - Type classes

Type classes abstract over behaviour:

```
class Ord  $\alpha$  | Eq  $\alpha$  where
```

```
lt ::  $\alpha \rightarrow \alpha \rightarrow \text{Bool}$ 
```

## Ad-hoc polymorphism - Type classes

Type classes abstract over behaviour:

```
class Ord  $\alpha$  | Eq  $\alpha$  where
```

```
  lt ::  $\alpha \rightarrow \alpha \rightarrow \text{Bool}$ 
```

```
instance Ord Int where
```

```
  lt  $x$   $y$  = ltInt  $x$   $y$ 
```

## Ad-hoc polymorphism - Type classes

Type classes abstract over behaviour:

```
class Ord  $\alpha$  | Eq  $\alpha$  where
```

```
  lt ::  $\alpha \rightarrow \alpha \rightarrow \text{Bool}$ 
```

```
instance Ord Int where
```

```
  lt x y = ltInt x y
```

```
instance Ord [ $\alpha$ ] | Ord  $\alpha$  where
```

```
  lt x y = lt (length x) (length y)  
     $\vee$  or (zipWith lt x y)
```

## Ad-hoc polymorphism - Type classes

Type classes abstract over behaviour:

```
class Ord  $\alpha$  | Eq  $\alpha$  where
```

```
  lt ::  $\alpha \rightarrow \alpha \rightarrow \text{Bool}$ 
```

```
instance Ord Int where
```

```
  lt x y = ltInt x y
```

```
instance Ord [ $\alpha$ ] | Ord  $\alpha$  where
```

```
  lt x y = lt (length x) (length y)  
    ∨ or (zipWith lt x y)
```

Ad-hoc polymorphic values require additional context:

```
sort ::  $\forall \alpha . [\alpha] \rightarrow [\alpha]$  | Ord  $\alpha$ 
```

```
let x = [1..10]
```

```
in eq
```

```
  (sort x) x
```

## Ad-hoc polymorphism - Dictionary-passing style

At compile time, type classes are translated to dictionaries:

$$:: \text{DictOrd } \alpha = \{ \text{lt} :: \alpha \rightarrow \alpha \rightarrow \text{Bool}, \text{dictEq} :: \text{DictEq } \alpha \}$$

## Ad-hoc polymorphism - Dictionary-passing style

At compile time, type classes are translated to dictionaries:

$:: DictOrd\ \alpha = \{ lt :: \alpha \rightarrow \alpha \rightarrow Bool, dictEq :: DictEq\ \alpha \}$

$dictOrdInt :: DictOrd\ Int$

$dictOrdInt = \{ lt = \lambda x\ y \rightarrow ltInt\ x\ y, dictEq = dictEqInt \}$

## Ad-hoc polymorphism - Dictionary-passing style

At compile time, type classes are translated to dictionaries:

$$:: \text{DictOrd } \alpha = \{ \text{lt} :: \alpha \rightarrow \alpha \rightarrow \text{Bool}, \text{dictEq} :: \text{DictEq } \alpha \}$$
$$\text{dictOrdInt} :: \text{DictOrd Int}$$
$$\text{dictOrdInt} = \{ \text{lt} = \lambda x y \rightarrow \text{ltInt } x y, \text{dictEq} = \text{dictEqInt} \}$$
$$\text{dictOrdList} :: \forall \alpha . \text{DictOrd } \alpha \rightarrow \text{DictOrd } [\alpha]$$
$$\text{dictOrdList } da = \{ \text{lt} = \lambda x y \rightarrow \text{dictOrdInt.lt } (\text{length } x) (\text{length } y) \\ \quad \vee \text{ or } (\text{zipWith } da.\text{lt } x y) \\ \quad , \text{dictEq} = \text{dictEqList } da.\text{dictEq} \}$$

## Ad-hoc polymorphism - Dictionary-passing style

At compile time, type classes are translated to dictionaries:

```
DictOrd  $\alpha = \{ \text{lt} :: \alpha \rightarrow \alpha \rightarrow \text{Bool}, \text{dictEq} :: \text{DictEq } \alpha \}$   
dictOrdInt :: DictOrd Int  
dictOrdInt = { lt =  $\lambda x y \rightarrow \text{ltInt } x y$ , dictEq = dictEqInt }  
dictOrdList ::  $\forall \alpha . \text{DictOrd } \alpha \rightarrow \text{DictOrd } [\alpha]$   
dictOrdList da = { lt =  $\lambda x y \rightarrow \text{dictOrdInt.lt } (\text{length } x) (\text{length } y)$   
                   $\vee \text{ or } (\text{zipWith } \text{da.lt } x y)$   
                  , dictEq = dictEqList da.dictEq }
```

Ad-hoc polymorphic values are provided additional arguments:

```
sort ::  $\forall \alpha . \text{DictOrd } \alpha \rightarrow [\alpha] \rightarrow [\alpha]$   
let x = [1..10]  
in (dictEqList dictEqInt).eq  
      (sort (dictOrdList dictOrdInt) x) x
```

## Preliminaries - Dynamic typing

## Dynamic typing - Monomorphism

Values are wrapped using the keyword **dynamic**:

*wrapInt* :: *Int* → *Dynamic*

*wrapInt* x = **dynamic** x :: *Int*

## Dynamic typing - Monomorphism

Values are wrapped using the keyword **dynamic**:

$$\begin{aligned} \text{wrapInt} &:: \text{Int} \rightarrow \text{Dynamic} \\ \text{wrapInt } x &= \mathbf{dynamic} \ x :: \text{Int} \end{aligned}$$

Values are unwrapped using the `::` annotation in a pattern match:

$$\begin{aligned} \text{unwrapInt} &:: \text{Dynamic} \rightarrow \text{Int} \\ \text{unwrapInt } (x :: \text{Int}) &= x \\ \text{unwrapInt } (x :: \text{String}) &= \text{stringToInt } x \\ \text{unwrapInt } \_ &= \perp \end{aligned}$$

Dynamic pattern matches can fail due to run-time type unification

## Dynamic typing - Parametric polymorphism

Parametric polymorphic values are wrapped similarly:

$wrapFun :: (\forall \alpha . \alpha \rightarrow \alpha) \rightarrow Dynamic$

$wrapFun f = \mathbf{dynamic} f$

## Dynamic typing - Parametric polymorphism

Parametric polymorphic values are wrapped similarly:

$$\begin{aligned} \text{wrapFun} &:: (\forall \alpha . \alpha \rightarrow \alpha) \rightarrow \text{Dynamic} \\ \text{wrapFun } f &= \mathbf{dynamic } f \end{aligned}$$

Then, dynamic pattern matches include universal quantification:

$$\begin{aligned} \text{unwrapFun} &:: \text{Dynamic} \rightarrow (\forall \alpha . \alpha \rightarrow \alpha) \\ \text{unwrapFun } (f :: \forall \alpha . \alpha \rightarrow \alpha) &= f \\ \text{unwrapFun } \_ &= \perp \end{aligned}$$

Note the different binding sites for the (duplicate) type variable  $\alpha$

## Dynamic typing - Type dependencies

The process of (un)wrapping values can depend on the context:

$$\begin{aligned} \text{wrap} &:: \forall \alpha . \alpha \rightarrow \text{Dynamic} \mid \text{TC } \alpha \\ \text{wrap } x &= \mathbf{dynamic} \ x \end{aligned}$$

## Dynamic typing - Type dependencies

The process of (un)wrapping values can depend on the context:

$$\begin{aligned} \text{wrap} &:: \forall \alpha . \alpha \rightarrow \text{Dynamic} \mid TC \alpha \\ \text{wrap } x &= \mathbf{dynamic} \ x \end{aligned}$$

Use the  $\wedge$  notation to refer to context-dependent type variables:

$$\begin{aligned} \text{unwrap} &:: \forall \alpha . \text{Dynamic} \rightarrow \alpha \mid TC \alpha \\ \text{unwrap } (x :: \alpha^\wedge) &= x \\ \text{unwrap } \_ &= \perp \end{aligned}$$

## Dynamic typing - Type dependencies

The process of (un)wrapping values can depend on the context:

$$\begin{aligned} \text{wrap} &:: \forall \alpha . \alpha \rightarrow \text{Dynamic} \mid TC \alpha \\ \text{wrap } x &= \mathbf{dynamic} \ x \end{aligned}$$

Use the  $\wedge$  notation to refer to context-dependent type variables:

$$\begin{aligned} \text{unwrap} &:: \forall \alpha . \text{Dynamic} \rightarrow \alpha \mid TC \alpha \\ \text{unwrap } (x :: \alpha^\wedge) &= x \\ \text{unwrap } \_ &= \perp \end{aligned}$$

The built-in type class  $TC$  provides type codes:

- ▶ Value representation of types, available for non-opaque types
- ▶ Required when (un)wrapped value is context dependent

## Dynamic typing - Pattern variables

Instead of enumerating every possible type, use pattern variables:

$$\begin{aligned} \text{dynApp} &:: \text{Dynamic} \rightarrow \text{Dynamic} \rightarrow \text{Dynamic} \\ \text{dynApp } (f :: a \rightarrow b) (x :: a) &= \mathbf{dynamic} (f \ x) \\ \text{dynApp } \_ \quad \quad \quad \_ &= \perp \end{aligned}$$

## Dynamic typing - Pattern variables

Instead of enumerating every possible type, use pattern variables:

$$\begin{aligned} \text{dynApp} &:: \text{Dynamic} \rightarrow \text{Dynamic} \rightarrow \text{Dynamic} \\ \text{dynApp } (f :: a \rightarrow b) (x :: a) &= \mathbf{dynamic} (f \ x) \\ \text{dynApp } \_ \quad \quad \quad \_ &= \perp \end{aligned}$$

This enforces type unification between the two arguments:

$$\begin{aligned} \text{dynApp } (\mathbf{dynamic} \text{ inc}) \quad (\mathbf{dynamic} \ 1) &\rightsquigarrow \mathbf{dynamic} \ 2 \\ \text{dynApp } (\mathbf{dynamic} \ \text{length}) \ (\mathbf{dynamic} \ [1 \dots 10]) &\rightsquigarrow \mathbf{dynamic} \ 10 \\ \text{dynApp } (\mathbf{dynamic} \ \text{length}) \ (\mathbf{dynamic} \ 1) &\rightsquigarrow \perp \end{aligned}$$

## Dynamic typing - Pattern variables

Instead of enumerating every possible type, use pattern variables:

$$\begin{aligned} \text{dynApp} &:: \text{Dynamic} \rightarrow \text{Dynamic} \rightarrow \text{Dynamic} \\ \text{dynApp } (f &:: a \rightarrow b) (x &:: a) = \mathbf{dynamic} (f \ x) \\ \text{dynApp } \_ & \quad \quad \quad \_ = \perp \end{aligned}$$

This enforces type unification between the two arguments:

$$\begin{aligned} \text{dynApp } (\mathbf{dynamic} \text{ inc}) \quad (\mathbf{dynamic} \ 1) & \quad \rightsquigarrow \mathbf{dynamic} \ 2 \\ \text{dynApp } (\mathbf{dynamic} \ \text{length}) \ (\mathbf{dynamic} \ [1 \dots 10]) & \rightsquigarrow \mathbf{dynamic} \ 10 \\ \text{dynApp } (\mathbf{dynamic} \ \text{length}) \ (\mathbf{dynamic} \ 1) & \quad \rightsquigarrow \perp \end{aligned}$$

Dynamics preserve lazy behaviour of functional programs:

$$\text{dynApp } (\mathbf{dynamic} \ \text{head}) \ (\mathbf{dynamic} \ [1 \dots]) \quad \rightsquigarrow \mathbf{dynamic} \ 1$$

## Act 1 - Dynamic typing in ad-hoc polymorphism

## Dynamic typing in ad-hoc polymorphism

Goal: use ad-hoc polymorphic values in a dynamic setting:

$\text{dynSort} :: \text{Dynamic} \rightarrow \text{Dynamic}$

$\text{dynSort } (x :: [a]) = \mathbf{dynamic} (\text{sort } x)$

$\text{dynSort } \_ = \perp$

# Dynamic typing in ad-hoc polymorphism

Goal: use ad-hoc polymorphic values in a dynamic setting:

$$\text{dynSort} :: \text{Dynamic} \rightarrow \text{Dynamic}$$
$$\text{dynSort } (x :: [a]) = \mathbf{dynamic} (\text{sort } x)$$
$$\text{dynSort } \_ = \perp$$

This poses an evident challenge:

- ▶ Ad-hoc polymorphism is resolved at compile time
- ▶ Critical type information becomes apparent only at run time

## Dynamic typing in ad-hoc polymorphism

A possible (non)solution enumerates all possible types:

$$\begin{aligned} \text{dynSort} &:: \text{Dynamic} \rightarrow \text{Dynamic} \\ \text{dynSort } (x :: [\text{Int}]) &= \mathbf{dynamic} (\text{sort } x) \\ \text{dynSort } (x :: [[\text{Int}]] &= \mathbf{dynamic} (\text{sort } x) \\ \text{dynSort } (x :: [[[ \text{Int} ]]]) &= \mathbf{dynamic} (\text{sort } x) \\ \dots & \\ \text{dynSort } \_ &= \perp \end{aligned}$$

This approach is cumbersome, error prone, and does not scale

## Dynamic typing in ad-hoc polymorphism

A possible (non)solution enumerates all possible types:

```
dynSort :: Dynamic → Dynamic  
dynSort (x :: [Int])    = dynamic (sort x)  
dynSort (x :: [[Int]]) = dynamic (sort x)  
dynSort (x :: [[[Int]]]) = dynamic (sort x)  
...  
dynSort _                = ⊥
```

This approach is cumbersome, error prone, and does not scale

We discuss two complementary approaches:

- ▶ Container datatypes (producer is responsible)
- ▶ Dynamic dictionary composition (consumer is responsible)

## Container datatypes - Including instances

A well-known form is the existential datatype:

$$:: EContOrdList = \exists \alpha . EContOrdList ([\alpha] \mid Ord \alpha)$$

## Container datatypes - Including instances

A well-known form is the existential datatype:

$$:: EContOrdList = \exists \alpha . EContOrdList ([\alpha] \mid Ord \alpha)$$

A more permissive form also exposes the type of the content:

$$:: ContOrdList \alpha = ContOrdList ([\alpha] \mid Ord \alpha)$$

## Container datatypes - Including instances

A well-known form is the existential datatype:

$$:: EContOrdList = \exists \alpha . EContOrdList ([\alpha] \mid Ord \alpha)$$

A more permissive form also exposes the type of the content:

$$:: ContOrdList \alpha = ContOrdList ([\alpha] \mid Ord \alpha)$$

Construction includes the appropriate instance silently:

```
x :: ContOrdList Int
x = ContOrdList [1..10]
```

## Container datatypes - Making instances available

Use a container datatype to enable sorting in a dynamic setting:

$\text{dynSort} :: \text{Dynamic} \rightarrow \text{Dynamic}$

$\text{dynSort} (\text{ContOrdList } x :: \text{ContOrdList } a) = \mathbf{dynamic} (\text{sort } x)$

$\text{dynSort } \_ = \perp$

## Container datatypes - Making instances available

Use a container datatype to enable sorting in a dynamic setting:

$$\text{dynSort} :: \text{Dynamic} \rightarrow \text{Dynamic}$$
$$\text{dynSort} (\text{ContOrdList } x :: \text{ContOrdList } a) = \mathbf{\text{dynamic}} (\text{sort } x)$$
$$\text{dynSort } \_ = \perp$$

In dictionary-passing style, the dictionary is simply passed on:

$$:: \text{ContOrdList } \alpha = \text{ContOrdList } [\alpha] (\text{DictOrd } \alpha)$$
$$\text{dynSort} :: \text{Dynamic} \rightarrow \text{Dynamic}$$
$$\text{dynSort} (\text{ContOrdList } x \text{ da} :: \text{ContOrdList } a) = \mathbf{\text{dynamic}} (\text{sort } \text{da } x)$$
$$\text{dynSort } \_ = \perp$$

## Container datatypes - Making instances available

Use a container datatype to enable sorting in a dynamic setting:

$$\text{dynSort} :: \text{Dynamic} \rightarrow \text{Dynamic}$$
$$\text{dynSort} (\text{ContOrdList } x :: \text{ContOrdList } a) = \mathbf{\text{dynamic}} (\text{sort } x)$$
$$\text{dynSort } \_ = \perp$$

In dictionary-passing style, the dictionary is simply passed on:

$$:: \text{ContOrdList } \alpha = \text{ContOrdList } [\alpha] (\text{DictOrd } \alpha)$$
$$\text{dynSort} :: \text{Dynamic} \rightarrow \text{Dynamic}$$
$$\text{dynSort} (\text{ContOrdList } x \text{ da} :: \text{ContOrdList } a) = \mathbf{\text{dynamic}} (\text{sort } \text{da } x)$$
$$\text{dynSort } \_ = \perp$$

- + More of a static approach
- + Minimal run-time overhead
- ± Producer determines the included contexts
- Requires additional plumbing of (type) constructors
- Ambiguities quickly arise with multiple containers

Better suited for situations where more rigidity is required

## Dynamic dictionary composition - Requiring instances

Come up with the appropriate instance dynamically:

$\text{dynSort} :: \text{Dynamic} \rightarrow \text{Dynamic}$

$\text{dynSort } (x :: [a] \mid \text{Ord } a) = \mathbf{\text{dynamic}} (\text{sort } x)$

$\text{dynSort } \_ = \perp$

## Dynamic dictionary composition - Requiring instances

Come up with the appropriate instance dynamically:

$$\text{dynSort} :: \text{Dynamic} \rightarrow \text{Dynamic}$$
$$\text{dynSort } (x :: [a] \mid \text{Ord } a) = \mathbf{dynamic} (\text{sort } x)$$
$$\text{dynSort } \_ = \perp$$

Translates to definition that uses conventional mechanisms:

$$\text{dynSort} :: \text{Dynamic} \rightarrow \text{Dynamic}$$
$$\text{dynSort } (x :: [a]) \mid \text{gda} = \mathbf{dynamic} (\text{sort } \text{da } x)$$
$$\text{dynSort } \_ = \perp$$

## Dynamic dictionary composition - Requiring instances

Come up with the appropriate instance dynamically:

```
dynSort :: Dynamic → Dynamic  
dynSort (x :: [a] | Ord a) = dynamic (sort x)  
dynSort _ = ⊥
```

Translates to definition that uses conventional mechanisms:

```
dynSort :: Dynamic → Dynamic  
dynSort (x :: [a]) | gda = dynamic (sort da x)  
  where (gda, da) = guards (genDictOrd (dynamic ⊥ :: a))  
dynSort _ = ⊥
```

## Dynamic dictionary composition - Requiring instances

Come up with the appropriate instance dynamically:

```
dynSort :: Dynamic → Dynamic  
dynSort (x :: [a] | Ord a) = dynamic (sort x)  
dynSort _ = ⊥
```

Translates to definition that uses conventional mechanisms:

```
dynSort :: Dynamic → Dynamic  
dynSort (x :: [a]) | gda = dynamic (sort da x)  
  where (gda, da) = guards (genDictOrd (dynamic ⊥ :: a))  
dynSort _ = ⊥
```

Invariant: given type  $\alpha$ , construct dictionary *DictOrd*  $\alpha$  if present

```
genDictOrd :: Dynamic → Maybe Dynamic  
guards ::  $\forall \alpha . \text{Maybe Dynamic} \rightarrow (\text{Bool}, \alpha) \mid \text{TC } \alpha$ 
```

## Dynamic dictionary composition - Generating instances

Arms of the generator function follow from available instances:

*genDictOrd* :: *Dynamic* → *Maybe Dynamic*

## Dynamic dictionary composition - Generating instances

Arms of the generator function follow from available instances:

*genDictOrd* :: *Dynamic* → *Maybe Dynamic*

*genDictOrd* (*\_* :: *Int*) = *Just* (**dynamic** *dictOrdInt*)

## Dynamic dictionary composition - Generating instances

Arms of the generator function follow from available instances:

*genDictOrd* :: *Dynamic* → *Maybe Dynamic*

*genDictOrd* (*\_* :: *Int*) = *Just* (**dynamic** *dictOrdInt*)

*genDictOrd* (*\_* :: [*a*]) = **do** *da* ← *genDictOrd* (**dynamic** *⊥* :: *a*)  
*Just* (**dynamic** (*dictOrdList* (*unwrap da*)))

## Dynamic dictionary composition - Generating instances

Arms of the generator function follow from available instances:

*genDictOrd* :: *Dynamic* → *Maybe Dynamic*

*genDictOrd* (*\_* :: *Int*) = *Just* (**dynamic** *dictOrdInt*)

*genDictOrd* (*\_* :: [*a*]) = **do** *da* ← *genDictOrd* (**dynamic** *⊥* :: *a*)  
*Just* (**dynamic** (*dictOrdList* (*unwrap da*)))

*genDictOrd* *\_* = *Nothing*

## Dynamic dictionary composition - Generating instances

Arms of the generator function follow from available instances:

$genDictOrd :: Dynamic \rightarrow Maybe Dynamic$

$genDictOrd (\_ :: Int) = Just (\mathbf{dynamic} dictOrdInt)$

$genDictOrd (\_ :: [a]) = \mathbf{do} da \leftarrow genDictOrd (\mathbf{dynamic} \perp :: a)$   
 $Just (\mathbf{dynamic} (dictOrdList (unwrap da)))$

$genDictOrd \_ = Nothing$

- More of a dynamic approach
- More compile-time and run-time overhead
- ± Consumer determines the required contexts
- + Does not require additional plumbing of (type) constructors
- + Does not suffer from ambiguity problems

Better suited for situations where more flexibility is required

## Act 2 - Ad-hoc polymorphism in dynamic typing

## Ad-hoc polymorphism in dynamic typing

Goal: (un)wrap ad-hoc polymorphic values:

*wrappedSort* :: *Dynamic*

*wrappedSort* = *wrap sort*

*dynAppOrd* ::  $\forall \alpha . \text{Dynamic} \rightarrow [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha$

*dynAppOrd* (*f* ::  $\forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha$ ) *x* = *f x*

*dynAppOrd* \_ =  $\perp$

## Ad-hoc polymorphism in dynamic typing

Goal: (un)wrap ad-hoc polymorphic values:

*wrappedSort* :: *Dynamic*

*wrappedSort* = *wrap sort*

*dynAppOrd* ::  $\forall \alpha . \text{Dynamic} \rightarrow [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha$

*dynAppOrd* (*f* ::  $\forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha$ ) *x* = *f x*

*dynAppOrd* \_ =  $\perp$

This poses two challenges:

- ▶ Come up with appropriate type codes
- ▶ Extend existing dynamic pattern match semantics

## Ad-hoc polymorphism in dynamic typing

Goal: (un)wrap ad-hoc polymorphic values:

*wrappedSort* :: *Dynamic*

*wrappedSort* = *wrap sort*

*dynAppOrd* ::  $\forall \alpha . \text{Dynamic} \rightarrow [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha$

*dynAppOrd* (*f* ::  $\forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha$ ) *x* = *f x*

*dynAppOrd* \_ =  $\perp$

This poses two challenges:

- ▶ Come up with appropriate type codes
- ▶ Extend existing dynamic pattern match semantics

We discuss two different approaches:

- ▶ Dictionary-passing types ('erase' ad-hoc polymorphism)
- ▶ Type code extension (facilitate ad-hoc polymorphism)

## Dictionary-passing types

Include the type code for the dictionary-passing type instead:

$$\text{sort} :: \forall \alpha . \text{DictOrd } \alpha \rightarrow [\alpha] \rightarrow [\alpha]$$

## Dictionary-passing types

Include the type code for the dictionary-passing type instead:

$$\text{sort} :: \forall \alpha . \text{DictOrd } \alpha \rightarrow [\alpha] \rightarrow [\alpha]$$

Include the dictionary-passing type in dynamic pattern matches:

$$\begin{aligned} \text{dynAppOrd} &:: \forall \alpha . \text{DictOrd } \alpha \rightarrow \text{Dynamic} \rightarrow [\alpha] \rightarrow [\alpha] \\ \text{dynAppOrd } da &(f :: \forall \alpha . \text{DictOrd } \alpha \rightarrow [\alpha] \rightarrow [\alpha]) x = f \text{ da } x \\ \text{dynAppOrd } da \_ & \_ = \perp \end{aligned}$$

## Dictionary-passing types

Include the type code for the dictionary-passing type instead:

$$\text{sort} :: \forall \alpha . \text{DictOrd } \alpha \rightarrow [\alpha] \rightarrow [\alpha]$$

Include the dictionary-passing type in dynamic pattern matches:

$$\begin{aligned} \text{dynAppOrd} &:: \forall \alpha . \text{DictOrd } \alpha \rightarrow \text{Dynamic} \rightarrow [\alpha] \rightarrow [\alpha] \\ \text{dynAppOrd } da &(f :: \forall \alpha . \text{DictOrd } \alpha \rightarrow [\alpha] \rightarrow [\alpha]) x = f \text{ da } x \\ \text{dynAppOrd } da \_ & \_ = \perp \end{aligned}$$

- + Lightweight approach, reuses the dictionary-passing style
- ± Pattern matching does not take superclasses into account
- Backdoor to the internal dictionary by **dynamic** ( $\lambda da x \rightarrow x$ )

## Type code extension

Extend type codes to include constructs for ad-hoc polymorphism

## Type code extension

Extend type codes to include constructs for ad-hoc polymorphism

Extend type unification semantics in dynamic pattern matches:

$$\begin{aligned} \text{dynAppOrd} &:: \forall \alpha . \text{Dynamic} \rightarrow [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha \\ \text{dynAppOrd } (f &:: \forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha) \ x = f \ x \\ \text{dynAppOrd } \_ & \qquad \qquad \qquad \_ = \perp \end{aligned}$$

## Type code extension

### Pop quiz!

A rank-2 polymorphic analogue of *dynAppOrd* is defined as:

*rank2AppOrd* ::

$\forall \alpha . (\forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha) \rightarrow [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha$   
*rank2AppOrd* *f* *x* = *f* *x*

## Type code extension

### Pop quiz!

A rank-2 polymorphic analogue of *dynAppOrd* is defined as:

*rank2AppOrd* ::

$\forall \alpha . (\forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha) \rightarrow [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha$   
*rank2AppOrd* *f* *x* = *f* *x*

Which of the following are well-typed arguments of *rank2AppOrd*?

*sort* ::  $\forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha$

*nub* ::  $\forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Eq } \alpha$

*reverse* ::  $\forall \alpha . [\alpha] \rightarrow [\alpha]$

- A. *sort*
- B. *sort* and *nub*
- C. *sort* and *reverse*
- D. *sort*, *nub*, and *reverse*

## Type code extension

### Pop quiz!

A rank-2 polymorphic analogue of *dynAppOrd* is defined as:

*rank2AppOrd* ::

$\forall \alpha . (\forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha) \rightarrow [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha$   
*rank2AppOrd* *f* *x* = *f* *x*

Which of the following are well-typed arguments of *rank2AppOrd*?

*sort* ::  $\forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha$

*nub* ::  $\forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Eq } \alpha$

*reverse* ::  $\forall \alpha . [\alpha] \rightarrow [\alpha]$

- A. *sort*
- B. *sort* and *nub*
- C. *sort* and *reverse*
- D. *sort*, *nub*, and *reverse*

## Type code extension

Extend type codes to include constructs for ad-hoc polymorphism

Extend type unification semantics in dynamic pattern matches:

$$\begin{aligned} \text{dynAppOrd} &:: \forall \alpha . \text{Dynamic} \rightarrow [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha \\ \text{dynAppOrd } (f &:: \forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha) \ x = f \ x \\ \text{dynAppOrd } \_ & \qquad \qquad \qquad \_ = \perp \end{aligned}$$

## Type code extension

Extend type codes to include constructs for ad-hoc polymorphism

Extend type unification semantics in dynamic pattern matches:

$$\begin{aligned} \text{dynAppOrd} &:: \forall \alpha . \text{Dynamic} \rightarrow [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha \\ \text{dynAppOrd } (f &:: \forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha) \ x = f \ x \\ \text{dynAppOrd } \_ & \qquad \qquad \qquad \_ = \perp \end{aligned}$$

- Heavyweight, extension of type codes and type unification
- + Takes superclass relations into account, and also the other:

$$\begin{aligned} \text{distinguish} &:: \text{Dynamic} \rightarrow \dots \\ \text{distinguish } (f &:: \forall \alpha . [\alpha] \rightarrow [\alpha]) & \qquad \qquad \qquad = \dots \\ \text{distinguish } (f &:: \forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Eq } \alpha) & = \dots \\ \text{distinguish } (f &:: \forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha) & = \dots \\ \text{distinguish } \_ & \qquad \qquad \qquad \qquad \qquad \qquad = \perp \end{aligned}$$

- + Does not provide a backdoor to internal dictionaries

## Conclusion

Different approaches to both sides of the interaction:

# Conclusion

Different approaches to both sides of the interaction:

- ▶ Dynamic typing in ad-hoc polymorphism, approaches:
  - ▶ Rigid: container datatypes
  - ▶ Flexible: dynamic dictionary composition

# Conclusion

Different approaches to both sides of the interaction:

- ▶ Dynamic typing in ad-hoc polymorphism, approaches:
  - ▶ Rigid: container datatypes
  - ▶ Flexible: dynamic dictionary composition
- ▶ Ad-hoc polymorphism in dynamic typing, approaches:
  - ▶ Lightweight: dictionary-passing types
  - ▶ Heavyweight: type code extension

# Conclusion

Different approaches to both sides of the interaction:

- ▶ Dynamic typing in ad-hoc polymorphism, approaches:
  - ▶ Rigid: container datatypes
  - ▶ Flexible: dynamic dictionary composition
- ▶ Ad-hoc polymorphism in dynamic typing, approaches:
  - ▶ Lightweight: dictionary-passing types
  - ▶ Heavyweight: type code extension

The paper discusses the effects of several type class extensions:

- ▶ Multi-parameter type classes
- ▶ Flexible contexts
- ▶ Flexible instances

# Conclusion

Different approaches to both sides of the interaction:

- ▶ Dynamic typing in ad-hoc polymorphism, approaches:
  - ▶ Rigid: container datatypes
  - ▶ Flexible: dynamic dictionary composition
- ▶ Ad-hoc polymorphism in dynamic typing, approaches:
  - ▶ Lightweight: dictionary-passing types
  - ▶ Heavyweight: type code extension

The paper discusses the effects of several type class extensions:

- ▶ Multi-parameter type classes
- ▶ Flexible contexts
- ▶ Flexible instances

Future work:

- ▶ Continue experimentation with implementation in Clean
- ▶ More formal approach to the interactions