Haskell Overloading (1)

What is the type of (==)?

E.g. the following both work:

\[
\begin{align*}
1 & \ == \ 2 \\
' a' & \ == \ ' b'
\end{align*}
\]

I.e., (==) can be used to compare both numbers and characters.
Haskell Overloading (1)

What is the type of (==)?

E.g. the following both work:

\[
1 \ == \ 2 \\
'a' \ == \ 'b'
\]

I.e., (==) can be used to compare both numbers and characters.

Maybe (==) :: a -> a -> Bool?
What is the type of (==)?

E.g. the following both work:

```
1 == 2
'a' == 'b'
```

I.e., (==) can be used to compare both numbers and characters.

Maybe (==) :: a -> a -> Bool?

**No!!! Cannot work uniformly for arbitrary types!**
A function like the identity function

\[
id :: a \rightarrow a \quad \text{id} \quad x = x
\]

is **polymorphic** precisely because it works uniformly for all types: there is no need to “inspect” the argument.
Haskell Overloading (2)

A function like the identity function

\[ \text{id :: } a \rightarrow a \quad \text{id } x = x \]

is **polymorphic** precisely because it works uniformly for all types: there is no need to “inspect” the argument.

In contrast, to compare two “things” for equality, they very much have to be inspected, and an **appropriate method of comparison** needs to be used.
Haskell Overloading (3)

Moreover, some types do not in general admit a decidable equality. E.g. functions (when domain infinite).
Moreover, some types do not in general admit a decidable equality. E.g. functions (when domain infinite).

Similar remarks apply to many other types. E.g.:
Moreover, some types do not in general admit a decidable equality. E.g. functions (when domain infinite).

Similar remarks apply to many other types. E.g.:

- We may want to be able to add numbers of any kind
Moreover, some types do not in general admit a decidable equality. E.g. functions (when domain infinite).

Similar remarks apply to many other types. E.g.:

- We may want to be able to add numbers of any kind
- But to add properly, we must understand what we are adding
Moreover, some types do not in general admit a decidable equality. E.g. functions (when domain infinite).

Similar remarks apply to many other types. E.g.:

- We may want to be able to add numbers of any kind
- But to add properly, we must understand what we are adding
- Not every type admits addition
Idea:
Haskell Overloading (4)

Idea:

- Introduce the notion of a *type class*: a set of types that support certain related operations.
Haskell Overloading (4)

Idea:

- Introduce the notion of a **type class**: a set of types that support certain related operations.
- **Constrain** those operations to **only** work for types belonging to the corresponding class.
Haskell Overloading (4)

Idea:

- Introduce the notion of a **type class**: a set of types that support certain related operations.
- **Constrain** those operations to **only** work for types belonging to the corresponding class.
- Allow a type to be **made an instance of** (added to) a type class by providing **type-specific implementations** of the operations of the class.
The Type Class \texttt{Eq}

class Eq a where

\hspace{1cm} (==) :: a \to a \to \text{Bool}

\texttt{(==)} is not a function, but a \textbf{method} of the \textbf{type class} \texttt{Eq}. It's type signature is:

\hspace{1cm} (==) :: Eq a \Rightarrow a \to a \to \text{Bool}

\texttt{Eq a} is a \textbf{class constraint}. It says that that the equality method works for any type belonging to the type class \texttt{Eq}.
Instances of \texttt{Eq} (1)

Various types can be made instances of a type class like \texttt{Eq} by providing implementations of the class methods for the type in question:

\begin{verbatim}
instance Eq Int where
  x == y = primEqInt x y

instance Eq Char where
  x == y = primEqChar x y
\end{verbatim}
Suppose we have a data type:

```haskell
data Answer = Yes | No | Unknown
```

We can make `Answer` an instance of `Eq` as follows:

```haskell
instance Eq Answer where
  Yes    == Yes    = True
  No     == No     = True
  Unknown == Unknown = True
  _      == _      = False
```
Instances of \textbf{Eq} (3)

Consider:

\begin{verbatim}
data Tree a = Leaf a
    | Node (Tree a) (Tree a)
\end{verbatim}

Can \texttt{Tree} be made an instance of \texttt{Eq}?
Instances of \textbf{Eq} (4)

Yes, for any type \(a\) that is already an instance of \textbf{Eq}:

\begin{verbatim}
instance (Eq a) => Eq (Tree a) where
    Leaf a1 == Leaf a2 = a1 == a2
    Node t1l t1r == Node t2l t2r = t1l == t2l && t1r == t2r
    _          == _          = False
\end{verbatim}
Derived Instances

Instance declarations are often obvious and mechanical. Thus, for certain *built-in* classes (notably `Eq`, `Ord`, `Show`), Haskell provides a way to *automatically derive* instances, as long as

- the data type is sufficiently simple
- we are happy with the standard definitions

Thus, we can do:

```haskell
data Tree a = Leaf a
    | Node (Tree a) (Tree a)
deriving Eq
```
Class Hierarchy

Type classes form a hierarchy. E.g.:

```haskell
class Eq a => Ord a where
  (<=) :: a -> a -> Bool
...
```

Eq is a superclass of Ord; i.e., any type in Ord must also be in Eq.
Haskell vs. OO Overloading (1)

A method, or overloaded function, may thus be understood as a family of functions where the right one is chosen depending on the types.

A bit like OO languages like Java. But the underlying mechanism is quite different and much more general. Consider \texttt{read}:

\[
\text{read} :: (\text{Read a}) \Rightarrow \text{String} \to a
\]

Note: overloaded on the \texttt{result} type! A method that converts from a string to \texttt{any} other type in class \texttt{Read}!
Haskell vs. OO Overloading (2)

```haskell
> let xs = [1,2,3] :: [Int]
> let ys = [1,2,3] :: [Double]
> xs
[1,2,3]
> ys
[1.0,2.0,3.0]
> (read "42" : xs)
[42,1,2,3]
> (read "42" : ys)
[42.0,1.0,2.0,3.0]
> read "'a'
'a'
```
The class constraints represent extra implicit arguments that are filled in by the compiler. These arguments are (roughly) the functions to use.

Thus, internally \( (==) \) is a *higher order function* with *three* arguments:

\[
(==) \quad \text{eqF} \quad x \quad y = \quad \text{eqF} \quad x \quad y
\]
Implementation (2)

An expression like

1 == 2

is essentially translated into

(==) primEqInt 1 2
Implementation (3)

So one way of understanding a type like

\[
(==) :: \text{Eq} \ a \Rightarrow a \rightarrow a \rightarrow \text{Bool}
\]

is that \(\text{Eq} \ a\) corresponds to an extra implicit argument.

The implicit argument corresponds to a so-called directory, or tuple/record of functions, one for each method of the type class in question.
Some Standard Haskell Classes (1)

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

class (Eq a) => Ord a where
    compare :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min :: a -> a -> a

class Show a where
    show :: a -> String
class (Eq a, Show a) => Num a where
    (+), (-), (*) :: a -> a -> a
    negate     :: a -> a
    abs, signum :: a -> a
    fromInteger :: Integer -> a

Quiz: What is the type of a numeric literal like 42?
42 :: Int? Why?
Application: Automatic Differentiation

- **Automatic Differentiation**: method for augmenting code so that derivative(s) computed along with main result.
- Purely algebraic method: arbitrary code can be handled
- Exact results
- But no separate, self-contained representation of the derivative.
Consider a code fragment:

\[
\begin{align*}
  z_1 &= x + y \\
  z_2 &= x \times z_1
\end{align*}
\]

Suppose the derivatives of \(x\) and \(y\) w.r.t. common variable is available in the variables \(x'\) and \(y'\).

Then code can be augmented to compute derivatives of \(z_1\) and \(z_2\):

\[
\begin{align*}
  z_1 &= x + y \\
  z_1' &= x' + y' \\
  z_2 &= x \times z_1 \\
  z_2' &= x' \times z_1 + x \times z_1'
\end{align*}
\]
Approaches

- Source-to-source translation
- Overloading of arithmetic operators and mathematical functions

The following variation is due to Jerzy Karczmarczuk. Infinite list of derivatives allows derivatives of \textit{arbitrary} order to be computed.
Introduce a new numeric type \( C \): value of a continuously differentiable function at a point along with \( \text{all} \) derivatives at that point:

\[
\text{data } C = C \ \text{Double} \ \ C
\]

\[
\begin{align*}
\text{valC} \ (C \ a \ _) &= a \\
\text{derC} \ (C \ _ \ x') &= x'
\end{align*}
\]
Constants and the variable of differentiation:

zeroC :: C
zeroC = C 0.0 zeroC

constC :: Double -> C
constC a = C a zeroC

dVarC :: Double -> C
dVarC a = C a (constC 1.0)
Part of numerical instance:

```haskell
instance Num C where
    (C a x') + (C b y') =
        C (a + b) (x' + y')

    (C a x') - (C b y') =
        C (a - b) (x' - y')

    x@(C a x') * y@(C b y') =
        C (a * b) (x' * y + x * y')

    fromInteger n =
        constC (fromInteger n)
```

LiU TF2010 Part II: Lecture 5 – p.25/27
Functional Automatic Differentiation (4)

Computation of $y = 3t^2 + 7$ at $t = 2$:

\[
t = \text{dVarC} 2 \\
y = 3 \times t \times t + 7 \\
\text{valC } y \Rightarrow 19.0 \\
\text{valC } (\text{derC } y) \Rightarrow 12.0 \\
\text{valC } (\text{derC } (\text{derC } y)) \Rightarrow 6.0 \\
\text{valC } (\text{derC } (\text{derC } (\text{derC } y))) \Rightarrow 0.0
\]
Reading