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?

No!!! Cannot work uniformly for arbitrary types!

Haskell Overloading (2)

A function like the identity function

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

is \textit{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 \textit{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).

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
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 `Eq`

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

(==) is not a function, but a **method** of the **type class** `Eq`. It's type signature is:

(==) :: Eq a => a -> a -> Bool

`Eq a` is a **class constraint**. It says that the equality method works for any type belonging to the type class `Eq`.

Instances of `Eq` (1)

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

```haskell
instance Eq Int where
  x == y = primEqInt x y

instance Eq Char where
  x == y = primEqChar x y
```

Instances of `Eq` (2)

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 Eq (3)

Consider:

```
data Tree a = Leaf a
  | Node (Tree a) (Tree a)
```

Can `Tree` be made an instance of `Eq`?

Instances of Eq (4)

Yes, for any type `a` that is already an instance of `Eq`:

```
instance (Eq a) => Eq (Tree a) where
    Leaf a1    == Leaf a2    = a1 == a2
    Node t1l t1r == Node t2l t2r = t1l == t2l
                                  && t1r == t2r
    _         == _             = False
```

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:

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

Class Hierarchy

Type classes form a hierarchy. E.g.:

```
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 read:

```haskell
read :: (Read a) => String -> a
```

Note: overloaded on the *result* type! A method that converts from a string to any other type in class *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'" :: Char
'a'
```

Implementation (1)

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:

```haskell
(==) eqF x y = eqF x y
```

Implementation (2)

An expression like

```haskell
1 == 2
```

is essentially translated into

```haskell
(==) 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)

```haskell
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
```

Some Standard Haskell Classes (2)

```haskell
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.
**Automatic Differentiation: Key Idea**

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 *arbitrary* order to be computed.

**Functional Automatic Differentiation (1)**

Introduce a new numeric type \(C\): value of a continuously differentiable function at a point along with all derivatives at that point:

\[
\begin{align*}
    \text{data } C &= \text{C Double C} \\
    \text{valC (C a _)} &= a \\
    \text{derC (C _ x')} &= x'
\end{align*}
\]

**Functional Automatic Differentiation (2)**

Constants and the variable of differentiation:

\[
\begin{align*}
    \text{zeroC :: C} \\
    \text{zeroC = C 0.0 zeroC} \\
    \text{constC :: Double -> C} \\
    \text{constC a = C a zeroC} \\
    \text{dVarC :: Double -> C} \\
    \text{dVarC a = C a (constC 1.0)}
\end{align*}
\]
**Functional Automatic Differentiation (3)**

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)
```

**Reading**


**Functional Automatic Differentiation (4)**

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

```haskell
t = dVarC 2
y = 3 * t * t + 7
valC y ⇒ 19.0
valC (derC y) ⇒ 12.0
valC (derC (derC y)) ⇒ 6.0
valC (derC (derC (derC y))) ⇒ 0.0
```