4. Types and Polymorphism

Oscar Nierstrasz
Roadmap

> Static and Dynamic Types
> Type Completeness
> Types in Haskell
> Monomorphic and Polymorphic types
> Hindley-Milner Type Inference
> Overloading
References


Roadmap

- Static and Dynamic Types
- Type Completeness
- Types in Haskell
- Monomorphic and Polymorphic types
- Hindley-Milner Type Inference
- Overloading
What is a Type?

Type errors:

```
? 5 + [ ]
ERROR: Type error in application
*** expression : 5 + [ ]
*** term : 5
*** type : Int
*** does not match : [a]
```

A type is a set of values?

> `int = { ... -2, -1, 0, 1, 2, 3, ... }`
> `bool = { True, False }`
> `Point = { [x=0,y=0], [x=1,y=0], [x=0,y=1] ... }`
What is a Type?

A type is a partial specification of behaviour?

> n, m: int ⇒ n+m is valid, but not(n) is an error

> n: int ⇒ n := 1 is valid, but n := "hello world" is an error

What kinds of specifications are interesting? Useful?
Values have static types defined by the programming language. A variable may have a declared, static type. Variables and expressions have dynamic types determined by the values they assume at run-time.

`Applet myApplet = new GameApplet();`

- declared, static type is `Applet`
- static type of value is `GameApplet`
- actual dynamic type is `GameApplet`
Static types restrict the programs you may write!

Object wyatt = new Cowboy();
yatt.draw();
Static and Dynamic Typing

A language is **statically typed** if it is always possible to *determine the (static) type* of an expression *based on the program text alone*.

A language is **dynamically typed** if *only values have fixed type*. Variables and parameters may take on different types at run-time, and must be checked immediately before they are used.

A language is “strongly typed” if it is impossible to perform an operation on the wrong kind of object.

Type consistency may be assured by

I. compile-time type-checking,
II. type inference, or
III. dynamic type-checking.
## Strong, weak, static, dynamic

<table>
<thead>
<tr>
<th></th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Strong&quot;</td>
<td>Java, Pascal</td>
<td>Smalltalk, Ruby</td>
</tr>
<tr>
<td>&quot;Weak&quot;</td>
<td>C</td>
<td>Assembler</td>
</tr>
</tbody>
</table>
Kinds of Types

All programming languages provide some set of built-in types.
> **Primitive types**: booleans, integers, floats, chars ...
> **Composite types**: functions, lists, tuples ...

Most strongly-typed modern languages provide for additional user-defined types.
> **User-defined types**: enumerations, recursive types, generic types, objects ...
Roadmap

> Static and Dynamic Types
> **Type Completeness**
> Types in Haskell
> Monomorphic and Polymorphic types
> Hindley-Milner Type Inference
> Overloading
The Type Completeness Principle:

No operation should be arbitrarily restricted in the types of values involved.

— Watt

First-class values can be evaluated, passed as arguments and used as components of composite values.

Functional languages attempt to make no class distinctions, whereas imperative languages typically treat functions (at best) as second-class values.
Roadmap

> Static and Dynamic Types
> Type Completeness
> **Types in Haskell**
> Monomorphic and Polymorphic types
> Hindley-Milner Type Inference
> Overloading
Function Types

Function types allow one to deduce the types of expressions without the need to evaluate them:

\[ \text{fact} :: \text{Int} \rightarrow \text{Int} \]
\[ 42 :: \text{Int} \quad \Rightarrow \quad \text{fact} \ 42 :: \text{Int} \]

**Curried types:**

\[ \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \quad = \quad \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \]

and

\[ \text{plus} \ 5 \ 6 \quad = \quad ((\text{plus} \ 5) \ 6) \]

**so:**

\[ \text{plus} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \quad \Rightarrow \quad \text{plus} \ 5 :: \text{Int} \rightarrow \text{Int} \]
List Types

A list of values of type \( a \) has the type \([ a ]\):

\[
[ 1 ] :: [ \text{Int} ]
\]

NB: All of the elements in a list must be of the same type!

\[ [\text{'a'}, 2, \text{False}] \quad -- \text{illegal! can’t be typed!} \]
Tuple Types

If the expressions $x_1, x_2, \ldots, x_n$ have types $t_1, t_2, \ldots, t_n$ respectively, then the tuple $(x_1, x_2, \ldots, x_n)$ has the type $(t_1, t_2, \ldots, t_n)$:

$$(1, [2], 3) :: (\text{Int}, [\text{Int}], \text{Int})$$

$$(\text{'a'}, \text{False}) :: (\text{Char}, \text{Bool})$$

$$((1,2),(3,4)) :: ((\text{Int, Int}), (\text{Int, Int}))$$

The unit type is written $(\ )$ and has a single element which is also written as $(\ )$. 
New data types can be introduced by specifying
   I. a datatype name,
   II. a set of parameter types, and
   III. a set of constructors for elements of the type:

\[
data \text{DatatypeName a1 ... an} = \text{constr1} | ... | \text{constrm}
\]

where the constructors may be either:
1. Named constructors:

\[
\text{Name type1 ... typek}
\]

2. Binary constructors (i.e., anything starting with “:”):

\[
\text{type1 BINOP type2}
\]
User data types that do not hold any data can model enumerations:

```haskell
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

Functions over user data types must *deconstruct* the arguments, with one case for each constructor:

```haskell
whatShallIDo Sun     = "relax"
whatShallIDo Sat     = "go shopping"
whatShallIDo _       = "guess I'll have to go to work"
```
Union types

\[
data \text{Temp} = \text{Centigrade \ Float} \mid \text{Fahrenheit \ Float}
\]

\[
\text{freezing :: Temp} \rightarrow \text{Bool}
\]

\[
\text{freezing (Centigrade temp)} = \text{temp} \leq 0.0
\]

\[
\text{freezing (Fahrenheit temp)} = \text{temp} \leq 32.0
\]
Recursive Data Types

A recursive data type provides constructors over the type itself:

```haskell
data Tree a = Lf a | Tree a :^: Tree a
mytree = (Lf 12 :^: (Lf 23 :^: Lf 13)) :^: Lf 10
```

? :t mytree
⇒ mytree :: Tree Int
Using recursive data types

```haskell
leaves, leaves' :: Tree a -> [a]
leaves (Lf l)  = [l]
leaves (l :^: r) = leaves l ++ leaves r

leaves' t = leavesAcc t []
    where leavesAcc (Lf l) = (l:)
          leavesAcc (l :^: r) = leavesAcc l . leavesAcc r
```

**NB:** \((f \cdot g) x = f (g x)\)

✍️ What do these functions do?
✍️ Which function should be more efficient? Why?
✍️ What is \((l :^: )\) and what does it do?
Roadmap

> Static and Dynamic Types
> Type Completeness
> Types in Haskell
> **Monomorphic and Polymorphic types**
> Hindley-Milner Type Inference
> Overloading
Languages like Pascal and C have monomorphic type systems: every constant, variable, parameter and function result has a unique type.

- good for type-checking
- bad for writing generic code
  — it is impossible in Pascal to write a generic sort procedure
Polymorphism

A polymorphic function accepts arguments of different types:

\[
\text{length} \quad :: \quad [a] \to \text{Int}
\]
\[
\text{length} \; [\;] \quad = \quad 0
\]
\[
\text{length} \; (x:xs) \quad = \quad 1 \; + \; \text{length} \; xs
\]

\[
\text{map} \quad :: \quad (a \to b) \to [a] \to [b]
\]
\[
\text{map} \; f \; [\;] \quad = \quad [\;]
\]
\[
\text{map} \; f \; (x:xs) \quad = \quad f \; x \; : \; \text{map} \; f \; xs
\]

\[
(\cdot) \quad :: \quad (b \to c) \to (a \to b) \to (a \to c)
\]
\[
(f \; \cdot \; g) \; x \quad = \quad f \; (g \; x)
\]
Roadmap

- Static and Dynamic Types
- Type Completeness
- Types in Haskell
- Monomorphic and Polymorphic types
- Hindley-Milner Type Inference
- Overloading
We can *infer* the type of many expressions by simply examining their structure. Consider:

\[
\begin{align*}
\text{length } [ ] &= 0 \\
\text{length } (x:xs) &= 1 + \text{length } xs
\end{align*}
\]

Clearly:

\[
\text{length } :: \text{a } \rightarrow \text{b}
\]

Furthermore, \text{b} is obvious \text{int}, and \text{a} is a list, so:

\[
\text{length } :: [c] \rightarrow \text{int}
\]

We cannot further refine the type, so we are done.
Composing polymorphic types

We can deduce the types of expressions using polymorphic functions by simply *binding type variables to concrete types*.

Consider:

\[
\text{length} :: [a] \rightarrow \text{Int} \\
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

Then:

\[
\text{map length} :: [[a]] \rightarrow [\text{Int}] \\
[ \text{“Hello”, “World”} ] :: [[\text{Char}]] \\
\text{map length} [ \text{“Hello”, “World”} ] :: [\text{Int}]
\]
Polymorphic Type Inference

Hindley-Milner Type Inference automatically determines the types of many polymorphic functions.

```
map :: X -> Y -> Z
map f (x:xs) = f x : map f xs
```

The corresponding type system is used in many modern functional languages, including ML and Haskell.
A polymorphic function may be explicitly assigned a *more specific type*:

```
idInt :: Int -> Int
idInt x = x
```

Note that the `:` command can be used to find the type of a particular expression that is inferred by Haskell:

```
? :t \x -> [x]
\x -> [x] :: a -> [a]

? :t (\x -> [x]) :: Char -> String
\x -> [x] :: Char -> String
```
Kinds of Polymorphism

> Universal polymorphism:
  — Parametric: polymorphic map function in Haskell; nil/void pointer type in Pascal/C
  — Inclusion: subtyping — graphic objects

> Ad Hoc polymorphism:
  — Overloading: + applies to both integers and reals
  — Coercion: integer values can be used where reals are expected and v.v.
Roadmap

- Static and Dynamic Types
- Type Completeness
- Types in Haskell
- Monomorphic and Polymorphic types
- Hindley-Milner Type Inference
- Overloading
Coercion vs overloading

Coercion or overloading — how do you distinguish?

Are there several overloaded + functions, or just one, with values automatically coerced?
Overloading

Overloaded operators are introduced by means of **type classes**: 

```haskell
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
  -- NB: defined in standard prelude
```

A type class must be *instantiated* to be used:

```haskell
instance Eq Bool where
  True == True = True
  False == False = True
  _ == _ = False
```
Instantiating overloaded operators

For each overloaded instance a separate definition must be given

```haskell
instance Eq Int where (==) = primEqInt

instance Eq Char where c == d = ord c == ord d

instance (Eq a, Eq b) => Eq (a,b) where
  (x,y) == (u,v) = x==u && y==v

instance Eq a => Eq [a] where
  [ ] == [ ] = True
  [ ] == (y:ys) = False
  (x:xs) == [ ] = False
  (x:xs) == (y:ys) = x==y && xs==ys
```
Equality for Data Types

Why not automatically provide equality for all types of values?

User data types:

```haskell
data Set a = Set [a]
instance Eq a => Eq (Set a) where
    Set xs == Set ys = xs `subset` ys && ys `subset` xs
    where xs `subset` ys = all (`elem` ys) xs
```

How would you define equality for the Tree data type?

NB: all (`elem` ys) xs tests that every x in xs is an element of ys
Equality for Functions

**Functions:**

\[(1==) \equiv (\lambda x. 1==x)\]

**ERROR:** Cannot derive instance in expression

*** Expression : \( (==) \ d148 \ ((==) \ {\text{dict}} \ 1) \ (\lambda x. (==) \ {\text{dict}} \ 1 \ x) \)

*** Required instance : Eq (Int -> Bool)

**Determining equality of functions is undecidable in general!**
What you should know!

- How are the types of functions, lists and tuples specified?
- How can the type of an expression be inferred without evaluating it?
- What is a polymorphic function?
- How can the type of a polymorphic function be inferred?
- How does overloading differ from parametric polymorphism?
- How would you define == for tuples of length 3?
- How can you define your own data types?
- Why isn’t == pre-defined for all types?
Can you answer these questions?

- Can any set of values be considered a type?
- Why does Haskell sometimes fail to infer the type of an expression?
- What is the type of the predefined function `all`? How would you implement it?
- How would you define equality for the Tree data type?
Attribution-ShareAlike 3.0

You are free:
- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

**BY:** Attribution. You must attribute the work in the manner specified by the author or licensor.

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

[http://creativecommons.org/licenses/by-sa/3.0/]