Really Gentle Introduction into Haskell's Type System

Hopefully, the first of many

Jan Sliacky

Qualified Types

So What are Qualified Types Anyway?

Asking the real questions.

So What are Types Anyway?

We have to go deeper.

Let’s Have a Small Language

Building from the Ground Up

- A small functional language
- Simple syntax

Lamb

A Sibling of Lambda Calculus or Something

```
Lamb, M, N, O ≔ α [ x, y, z ]           (variables)
              |  ( M N )                (application)
              |  (\ α → M )             (abstraction)
              |  if O then M else N  (if)
              |  let α = M in N         (let)
```

Types

Finally!

```
τ := Int                  (primitive)
  |  Bool                 (primitive)
  |  α  [ x, y, z ]       (variables)
  |  τ → τ                (function types)
```

Examples

Examples! Examples! Examples!

```
23 :: Int

True :: Bool

False :: Bool


(\ x → x + x) :: Int → Int (assuming + :: Int → Int → Int)


(\ a → a) :: ?
```

Polymorphism?

```
let id = (\ x → x)
in  let a = id 23
    in  let b = id True
        in …
```

Polymorphism

Type Schemes

Type Scheme

$$\sigma := \forall\ \alpha_1 \ldots \alpha_n\ .\ \tau$$

```
(\ x → x) :: ∀ a . a → a
```

More Examples

```
(\ x y → x) :: ∀ a b . a → b → a
```

```
(\ x y → if x then y else y) :: ∀ b . Bool → b → b
```

Type Inference / Type Synthesis

Making Stuff Up

```
Type Contexts

T :=  []

  |   (α :: τ) , T
```

Demonstration with More Examples

“Blackboard Inferring”

Limiting the Polymorphism

Less is more, sometimes.

- Types like `∀ x . x → x` are cool, but what can I ever do to the argument?
- Maybe I want to be polymorphic, but not as much.
- I want some way to restrict the set of types without enumerating on them.

Restricted Polymorphism

Taming of the Beast

- I want something like "for all types such that they have a quality X"

Examples

Examples! Examples! Examples!

- Suppose a function `foo`.
- Its type is `∀ x . x → x` but only for those x'es that have a quality Q.
- Suppose that Int has that quality but Bool does not.

```
> :type foo 23
> foo 23 :: Int

> :type foo True
> error
```

How to Represent That?

Types with Qualities

- The type schemes now carry the quality of all restricted type variables.

- `∀ x . (x has a quality Q) ⇒ x → y → …`

- Or a shorter version:

- `∀ x . (x of Q) ⇒ x → y → …`

How to Check That?

- When doing a type inference (or analysis in general) we make sure that qualified variables unify only with types having that quality.
- We simply observe that since those quality informations are within a type scheme, it will always be about instantiation.

Checking - Part 2

- What if we have the following:

```
foo :: ∀ x . (x of Q) ⇒ x → x

bar a = foo a


> :type bar

> ???
```

Checking - Part 2

- What if we have the following:

```
foo :: ∀ x . (x of Q) ⇒ x → x

bar a = foo a


> :type bar

> bar :: ∀ y . (y of Q) ⇒ y → y
```

How To Translate That?

- Any function with a type like:
  `∀ x . (x of Q) ⇒ x → x`
- Can be understood as a function taking a value of type x that has the quality Q.
- But what does it mean? How can the function bee sure that its argument has that quality?
- The function needs an **evidence**!
- So the type can be re-interpreted as:
  `∀ x . Evidence of (x of Q) → x → x`

More Intuition about the Evidence

- It can be any value that serves as an evidence that the type of the argument has that quality.
- If the quality would be something like: “is a collection containing a maximum element” - the evidence could be a function that obtains that maximum from the collection.
- If the quality would be something like: “is a record with a field ‘foo’” - the evidence could be a position of that field within the representation of the record.

Exercise

Questions with a Catch!

```
baz :: ∀ x . (x of Q) ⇒ x → x
```

Exercise

Questions with a Catch!

```
baz :: ∀ x . (x of Q) ⇒ Int → x
```

Exercise

Questions with a Catch!

```
data Phantom a = P Int
```

```
baz :: ∀ x . (x of Q) ⇒ Int → Phantom x
```

Exercise

Questions with a Catch!

```
data Container a = C a
```

```
baz :: ∀ x . (x of Q) ⇒ x → Container x
```

So Now for Real!

Qualified Types

Types with Contexts

```
Context := [Predicate]
```

Class Predicates

Type Classes

```
Predicate := C α
```

```
C is a name of a known class
```

Examples

Examples! Examples! Examples!

```
∀ a . (Add a) ⇒ a → a
```

```
∀ a b . (Add a, Add b) ⇒ a → b → Bool
```

Type Classes

```
class Add a where
  add :: a → a → a
```

```
> :type add
> add :: ∀ a . Add a ⇒ a → a → a
```

Type Classes

What Are They Anyway?

- Way to do **parametric overloading.**
- Way to implement a method on values of many types.
- Way of a new kind of polymorphism.

Way to Implement a Method on Values of Many Types

```
class Add a where
  add :: a → a → a

instance Add Int where
  add = add#int

instance Add Double where
  add = add#double
```