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) (MN)

(application) $(\land \alpha \rightarrow M)$ (abstraction) if O then M else N (if) $let \alpha = M in N$ (let)

Types Finally!

 $\tau := Int \qquad (primitive)$ $\mid Bool \qquad (primitive)$ $\mid \alpha [x, y, z] \qquad (variables)$ $\mid \tau \rightarrow \tau \qquad (function types)$

Examples **Examples! Examples! Examples!**

- 23 :: Int
- True :: Bool
- False :: Bool

 $(X \rightarrow X + X) :: Int \rightarrow Int (assuming + :: Int \rightarrow Int \rightarrow Int)$

 $(\land a \rightarrow a) :: ?$

Polymorphism?

let id = $(\setminus x \rightarrow x)$ in let a = id 23
in let b = id True
in ...

Polymorphism Type Schemes

Type Scheme

 $\sigma := \forall \alpha_1 \dots \alpha_n \cdot \tau$

 $(\setminus x \rightarrow x) :: \forall a \cdot a \rightarrow a$

Nore Examples

$(\land x \land y \rightarrow x) :: \forall a b \cdot a \rightarrow b \rightarrow a$

 $(\land x \land y \rightarrow if \land then \land y else \land y) :: \forall b . Bool \rightarrow b \rightarrow b$

Type Inference / Type Synthesis Making Stuff Up

Type Contexts T := [] $| (\alpha :: \tau), T$

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 $\forall x$. $x \rightarrow x$ 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.

• $\forall x . (x has a quality Q) \Rightarrow x \rightarrow y \rightarrow ...$

- Or a shorter version:
- $\forall x . (x of Q) \Rightarrow x \rightarrow y \rightarrow ...$





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 :: $\forall x$. (x of Q) $\Rightarrow x \rightarrow x$
- bar a = foo a

- > :type bar
- > ???

X

Checking - Part 2

- What if we have the following:
- foo :: $\forall x$. (x of Q) $\Rightarrow x \rightarrow x$
- bar a = foo a

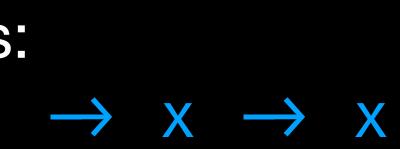
- > :type bar
- > bar :: \forall y . (y of Q) \Rightarrow y \rightarrow y

How lo ranslate That?

- Any function with a type like: $\forall x . (x of Q) \Rightarrow x \rightarrow x$
- that quality?
- The function needs an evidence!
- So the type can be re-interpreted as: $\forall x \cdot Evidence of (x of Q) \rightarrow x \rightarrow x$



Can be understood as a function taking a value of type x that has the quality \mathbf{Q} . But what does it mean? How can the function bee sure that its argument has



Nore Intuition about the Evidence

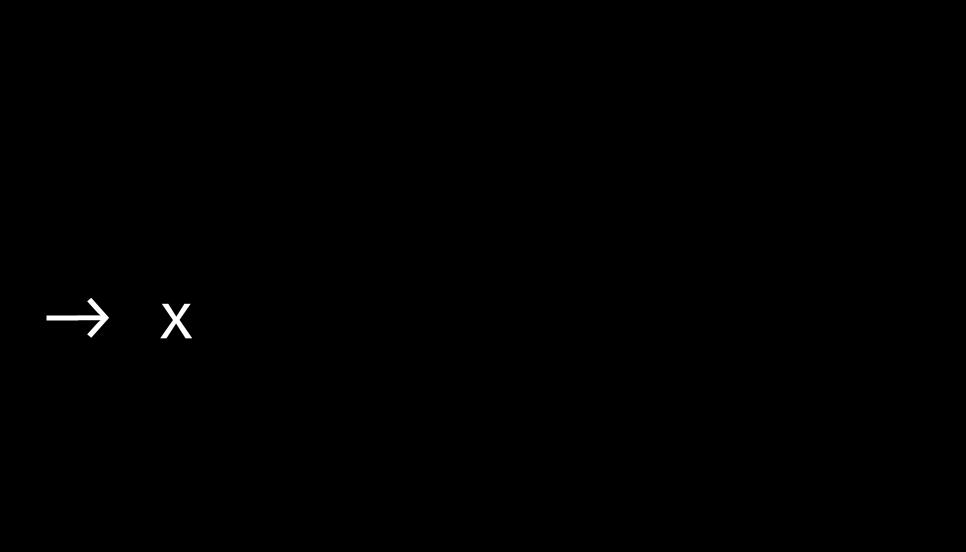
- has that quality.
- the collection.
- record.

• It can be any value that serves as an evidence that the type of the argument

• 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

• 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

baz :: $\forall x . (x \text{ of } Q) \Rightarrow x \rightarrow x$



baz :: $\forall x . (x \text{ of } Q) \Rightarrow \text{Int} \rightarrow x$

data Phantom a = P Int

baz :: $\forall x$. (x of Q) \Rightarrow Int \rightarrow Phantom x

data Container a = C a

baz :: $\forall x \cdot (x \text{ of } Q) \Rightarrow x \rightarrow \text{Container } x$

So Now for Real!

Qualified Types Types with Contexts

Context := [Predicate]

Class Predicates Type Classes

Predicate := $C \alpha$

C is a name of a known class

Examples **Examples! Examples! Examples!**

 \forall a . (Add a) \Rightarrow a \rightarrow a

 $\forall a b \cdot (Add a, Add b) \Rightarrow a \rightarrow b \rightarrow Bool$

Type Classes

class Add a where add :: $a \rightarrow a \rightarrow a$

> :type add > add :: \forall a . Add a \Rightarrow a \rightarrow a \rightarrow 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 :: $a \rightarrow a \rightarrow a$

instance Add Int where add = add#int

instance Add Double where add = add#double

