System FC and type inference

AFP Summer School

Wouter Swierstra and Alejandro Serrano

System FC or GHC Core

System F_ω plus

- Algebraic data types and pattern matching
- `let` bindings as a primitive
- Coercions (build-in equality proofs)
- Promoted data types
- Roles (not discussed here)

System F_ω

Your usual λ-calculus where

- Abstractions are annotated with the type
- Type abstraction and application is explicit

```
e  ::=  x                (variables)
     |  e e              (application)
     |  λ (x : τ) . e    (abstraction)
     |  e @τ             (type application)
     |  Λ (α : κ) . e    (type abstraction)
```

For example, here is how we define and use `id`

```
id = Λ (α : *). λ (x : α). x
> id @Bool True
```

ADTs and pattern matching

```
e  ::=  ...
    |   case e of
          K x1 ... xn -> e
          ...
          L y1 ... ym -> e
```

- Pattern matching is only allowed to look at one layer
  - Complex pattern have to be turned into a `case`-tree
- Operationally, `case` drives evaluation

Coercions – the C in FC

Built-in version of `Equal` data type

```
e  ::=  ...
    |   e |> γ          (coercion application - cast)

γ  ::=  refl            (reflexivity)
    |   sym γ           (symmetry)
    |   γ1 ; γ2         (transitivity)
    |   K γ1 .. γn      (same constructor)
    |   ...

τ  ::=  ...
    |   τ ~ τ           (type equality)
```

Coercions – the C in FC

Take the following Haskell term

```
coerce :: a ~ b => a -> b
coerce x = x
```

How does it look like in System FC?

Coercions – the C in FC

Take the following Haskell term

```
coerce :: a ~ b => a -> b
coerce x = x
```

How does it look like in System FC?

```
coerce = Λ (a : *). Λ (b : *).
         λ (γ : a ~ b).
         λ (x : a). x |> γ
```

Where are type classes?

Type classes are translated to *records*

- This is called the *dictionary translation*

Where are type classes?

Type classes are translated to *records*

- This is called the *dictionary translation*

```
class Show a where
  show :: a -> String
===>
data ShowDict a = ShowDict { show :: a -> String }

shout :: Show a => a -> String
shout x = show x ++ "!"
===>
shout :: ShowDict a -> a -> String
shout sd a = show sd x ++ "!"
```

Dictionary translation, arguments

```
shout :: ShowDict a -> a -> String
shout sd a = show sd x ++ "!"
```

Try to write it as a System FC term!

Dictionary translation, arguments

```
shout :: ShowDict a -> a -> String
shout sd a = show sd x ++ "!"
```

Try to write it as a System FC term!

```
shout = Λ (a : *). λ (sd : ShowDict a). λ (x : a).
        (++) @Char (show @a sd x)
             ((:) @Char '!' ([] @Char))
```

Dictionary translation, instances

- ▶ Simple instances are plain values

```
boolShow :: ShowDict Bool
boolShow = ShowDict $ \x ->
             case x of
               True  -> "True"
               False -> "False"
```

- ▶ Recursive instances are functions

```
maybeShow :: ShowDict a -> ShowDict (Maybe a)
maybeShow sd = ShowDict $ \x ->
                 case x of
                   Nothing -> "Nothing"
                   Just y  -> "Just " ++ show sd y
```

Dictionary translation, superclasses

Superclasses appear as fields of the child class

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
===>
data OrdDict a = OrdDict {
                   eqDict  :: EqDict a,
                   compare :: a -> a -> Ordering
                 }
```

The need for inference

In surface Haskell many information is implicit

- Type abstraction and application
- Dictionaries for type class instances

On the other hand they are explicit in System FC

Inference is the process of obtaining that information

Types and free variables

Question:

How do we assign a type to a term with free variables?

```
plus x one
```

Types and free variables

Question:

How do we assign a type to a term with free variables?

```
plus x one
```

Answer

We cannot unless we know the types of the free variables.

Environments

We therefore do not assign types to terms, but types to terms in a certain *environment* (also called *context*).

Environments

```
Γ ::= ε           -- empty environment
   |  Γ, x : τ    -- binding
```

Later bindings for a variable always shadow earlier bindings.

The typing relation

A statement of the form $\Gamma \vdash \mathtt{e} : \tau$ means “in environment Γ, term e has type τ”.

This defines a ternary *relation* between an environment, a term and a type.

The $\vdash$ (called turnstile) and the colon are just notation for making the relation look nice but carry no meaning. We could have chosen the notation $\mathtt{T}(\Gamma,\mathtt{e},\tau)$ for the relation as well, but $\Gamma \vdash \mathtt{e} : \tau$ is commonly used.

Type rules

The relation is defined inductively, using *inference rules*.

Variables

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

- Above the bar are the *premises*.
- Below the bar is the *conclusion*.
- If the premises hold, we can infer the conclusion.

(Hindley-)Damas-Milner type inference

Mainly based on a paper by Milner (1978).

This algorithm is:

- the basis of the algorithm used for the ML family of languages as well as Haskell;
- allows type inference essentially for the simply-typed lambda calculus extended with a limited form of polymorphism (sometimes called `let`-polymorphism);
- is a “sweet spot” in the design space: some simple extensions are possible (and performed), but fundamental extensions are typically significantly more difficult.

Monotypes and type schemes

Damas-Milner types are all quantified at the outermost level.

That is why Haskell typically does not use an explicit universal quantifier.

Monotypes

Monotypes τ are types built from variables and type constructors.

Type schemes (or polytypes)

```
σ ::= τ          -- monotypes
   |  ∀ α . s    -- quantified type
```

The key idea

The Damas-Milner algorithm distinguishes lambda-bound and let-bound (term) variables:

- lambda-bound variables are always assumed to have a monotype;
- let-bound variables, we know what they are bound to, therefore they can have polymorphic type.

Inference variables

Whenever a lambda-bound variable is encountered, a *fresh* inference variable is introduced. The variable represents a monotype.

When we learn more about the types, inference variables can be substituted by types.

Inference variables are different from universally quantified variables that express polymorphism.

Term language

```
e ::= x                   -- variables
    | e e                 -- application
    | \x -> e             -- abstraction
    | let x = e in e      -- let binding
```

Only a simple language to start with, but we include `let` compared to plain lambda calculus.

Example

Assume an environment Γ `= neg : Nat -> Nat`.

Consider inferring the type of the expression `\x -> neg x`.

For x, we introduce an type variable v and assume `x : v`.

Example

Assume an environment Γ `= neg : Nat -> Nat`.

Consider inferring the type of the expression `\x -> neg x`.

For `x`, we introduce an type variable `v` and assume `x : v`.

To typecheck `neg x`, we first determine the types of the components.

Example

Assume an environment Γ `= neg : Nat -> Nat`.

Consider inferring the type of the expression `\x -> neg x`.

For `x`, we introduce an type variable `v` and assume `x : v`.

To typecheck `neg x`, we first determine the types of the components.

In the environment we can find the types of the variables:

`neg : Nat -> Nat` and `x : v`.

Example

Assume an environment Γ `= neg : Nat -> Nat`.

Consider inferring the type of the expression `\x -> neg x`.

For `x`, we introduce an type variable `v` and assume `x : v`.

To typecheck `neg x`, we first determine the types of the components.

In the environment we can find the types of the variables:

`neg : Nat -> Nat` and `x : v`.

We now unify `Nat` and `v`, introducing the substitution:

`v` $\mapsto$ `Nat`.

Generalization and instantiation

```
let id = \x -> x in (id False, id ’x’)
```

Inference for `\x -> x` gives us the type `v -> v` for some inference variable v, and there are no further assumptions about v.

Generalization and instantiation

```
let id = \x -> x in (id False, id 'x')
```

Inference for `\x -> x` gives us the type `v -> v` for some inference variable v, and there are no further assumptions about v.

On a let-binding, the algorithm generalizes the inferred type as much as possible, in this case to `id : ∀ a. a -> a`.

Generalization and instantiation

```
let id = \x -> x in (id False, id 'x')
```

Inference for `\x -> x` gives us the type `v -> v` for some inference variable v, and there are no further assumptions about v.

On a let-binding, the algorithm generalizes the inferred type as much as possible, in this case to `id : ∀ a. a -> a`.

For every use, a polymorphic type is instantiated with fresh inference variables. For example, we get `w -> w` for the first call, `u -> u` for the second.

Generalization and instantiation

```
let id = \x -> x in (id False, id 'x')
```

Inference for `\x -> x` gives us the type `v -> v` for some inference variable v, and there are no further assumptions about v.

On a let-binding, the algorithm generalizes the inferred type as much as possible, in this case to `id : ∀ a. a -> a`.

For every use, a polymorphic type is instantiated with fresh inference variables. For example, we get `w -> w` for the first call, `u -> u` for the second.

The w gets unified with `Bool`, and u with `Char`.

Generalization again

Assume: `singleton : ∀ a . a -> [a]`

```
\ x -> (let y = singleton x in head y)
```

Generalization again

Assume: `singleton : ∀ a . a -> [a]`

```
\ x -> (let y = singleton x in head y)
```

For x, an inference variable v is introdued.

Generalization again

Assume: `singleton : ∀ a . a -> [a]`

```
\ x -> (let y = singleton x in head y)
```

For x, an inference variable v is introdued.

Consequently, we infer the type `[v]` for `singleton x`.

Generalization again

Assume: `singleton : ∀ a . a -> [a]`

```
\ x -> (let y = singleton x in head y)
```

For x, an inference variable v is introded.

Consequently, we infer the type `[v]` for `singleton x`.

But we must not generalize the type of y to `∀ a . [a]`.

We can only generalize if a variable is not mentioned in the environment.

Motivation: unification

Question: What is the type of the following expressions?

```
[  \x y -> 'a',  \x y -> if x then y else y ]
```

Motivation: unification

Question: What is the type of the following expressions?

```
[  \x y -> 'a',  \x y -> if x then y else y ]
```

We have to unify the two types

```
v -> w -> Char              Bool -> u -> u
```

`u ↦ Char, w ↦ Char, v ↦ Bool`

Motivation: unification

Question: What is the type of the following expressions?

```
[  \x y -> 'a',  \x y -> if x then y else y ]
```

We have to unify the two types

```
v -> w -> Char                 Bool -> u -> u
```

```
u ↦ Char, w ↦ Char, v ↦ Bool
```

We are interested in the *minimal* substitution.

```
v ↦ w, u ↦ Char
```

Preventing infinite types

What if we want to unify the types:

```
u      u -> u
```

A substitution u ↦ u -> u would result in an infinite type. Most systems (including Haskell) reject infinite types, and make this a type error.

Idea of the unification algorithm

We distinguish the following cases:

- if we have two equal variables, there is nothing to do;
- if we have an inference variable and another type that does not contain the inference variable (*occurs check* to prevent infinite types), we substitute the variable by the other type;
- if we have two function types, we recursively unify the domains and codomains;
- if we have any other situation, unification fails.

Principal types

There is a similar notion for types as we had for unifications. One type can be more general than another:

```
a         -> b
(a,  b)   -> (b,  a)
(a,  a)   -> (a,  a)
(Int,Int) -> (Int,Int)
```

Damas-Milner type inference always infers the most general type (called the *principal type*).

Everything is a lie

This is not how *modern* GHC does type inference

Constraint-based type inference

Type checking and inference is a two-step process

1. *Constraint generation* or *gathering*
   - Obtains a set of constraints which describe the relations between types in the program
   - Can**not** fail, except for ill-scoped variables
2. *Constraint solving*
   - Finds a solution for the set of constraints
   - Works by rewriting the constraints into simpler forms

Constraint-based type inference, example

Take Γ `= neg : Nat -> Nat` and infer `\x -> neg x`.

1. We first assign a new fresh variable α to `x`.
2. The type of `neg` is `Nat -> Nat` from the environment.
3. The type of `x` in the body is α as introduced.
4. Since we have an application, we know that:
   - The type of the function must be β `->` γ;
   - The argument type α has to coincide with β;
   - The result type is γ.
5. The whole is an abstraction with a body of type γ.

In summary, we have the following two constraints,

`Nat -> Nat ~` β `->` γ $\qquad$ α `~` β

and the inferred type of the expression is α `->` γ.

Constraint gathering rules

$\Gamma \vdash e : \tau \rightsquigarrow C$ means "in the environment Γ, the expression e has type τ whenever the constraints C are satisfied".

$$\frac{x : \forall \overline{a}.\, \tau \in \Gamma \qquad \overline{\alpha} \text{ fresh}}{\Gamma \vdash x : [\overline{a \mapsto \alpha}]\tau \rightsquigarrow \top}$$

Constraint gathering rules

$\Gamma \vdash e : \tau \rightsquigarrow C$ means "in the environment Γ, the expression e has type τ whenever the constraints C are satisfied".

$$\frac{x : \forall \overline{a}.\, \tau \in \Gamma \qquad \overline{\alpha} \text{ fresh}}{\Gamma \vdash x : [\overline{a \mapsto \alpha}]\tau \rightsquigarrow \top}$$

$$\frac{\alpha \text{ fresh} \qquad \Gamma, x : \alpha \vdash e : \tau \rightsquigarrow C}{\Gamma \vdash \lambda x.\, e : \alpha \to \tau \rightsquigarrow C}$$

$$\frac{\Gamma \vdash f : \tau_1 \rightsquigarrow C_1 \qquad \Gamma \vdash e : \tau_2 \rightsquigarrow C_2 \qquad \beta \text{ fresh}}{\Gamma \vdash f\, e : \beta \rightsquigarrow C_1 \wedge C_2 \wedge \tau_1 \sim \tau_2 \to \beta}$$

Solving rules for equalities

Solving is done by rewriting constraints into simpler forms:

```
τ ~ τ    ===>  -- remove it

T τ1 ... τn ~ T σ1 ... σn
         ===>  τ1 ~ σ1, ..., τn ~ σn
T τ1 ... τn ~ S σ1 ... σn
         ===>  error  -- if T /= S

α ~ τ    ===>  error  -- if α is free in τ
α ~ τ, C ===>  [α ↦ τ]C
```

We need some care to not end up in an infinite loop.

Rigid variables a.k.a. Skolems

A *rigid variable* a represents an type which exists but we cannot touch, assign or inspect.

```
a ~ τ    ===>   error -- if τ /= a
```

Type class constraints

Type signatures may have constraints, $\forall\overline{a}.\,C \Rightarrow \tau$.

- For example, `show :: Show a => a -> String`.

Constraint generation

$$\frac{x : \forall\overline{a}.\,C \Rightarrow \tau \in \Gamma \qquad \overline{\alpha}\ \text{fresh}}{\Gamma \vdash x : [\overline{a \mapsto \alpha}]\tau \leadsto [\overline{a \mapsto \alpha}]C}$$

Type class constraints

Constraint solving

What are the rewriting rules corresponding to these definitions?

```
1. instance Eq Int
2. instance Eq a => Eq [a]
3. class Eq a => Ord a
```

Type class constraints

Constraint solving

What are the rewriting rules corresponding to these definitions?

```
1. instance Eq Int
2. instance Eq a => Eq [a]
3. class Eq a => Ord a
```

```
Eq Int    ===>   -- remove it
Eq [τ]    ===>   Eq τ
Ord τ     ===>   Eq τ, Ord τ  -- keep Ord
```

See *Understanding Functional Dependencies via Constraint Handling Rules*

Type class constraints and termination

```
instance C [a] => C a

C Int ===> C [Int] ===> C [[Int]] ===> ...
```

Type class constraints and termination

```
instance C [a] => C a

C Int ===> C [Int] ===> C [[Int]] ===> ...
```

The compiler has an infinite loop!

We need conditions to prevent this situation

Caveat: this *is* the halting problem.

- Turing taught us that it is undecidable.
- Every heuristic is just an approximation.

Classical termination conditions

1. Every instance must be of the form

   ```
   instance (C1, ..., Cn) => C (T a1 ... am)
   ```

   for a constructor `T` and distinct type variables `a1` to `am`. If the class has multiple parameters, the constructor condition only has to apply to one of them.

2. The context in any type, class or instance declaration

   ```
   fn    ::  (C1, ..., Cn) => ...
   class     (C1, ..., Cn) => ...
   instance (C1, ..., Cn) => ...
   ```

   must consist only of type classes applied to type variables (we say such contexts are *simple*).

Patterson termination conditions

`FlexibleContexts` and `FlexibleInstances`

For each class constraint `C t1 ... tn` in the context:

- No type variable has more occurrences in the constraint than in the head.
- The constraint has fewer constructors and variables (taken together and counting repetitions) than the head, in class and instance declarations.
- The constraint mentions no type families.

Intuitively, constraints *shrink* at every step of rewriting.

Undecidable instances

Lifts the restrictions over termination of instances.

- *You* are responsible for checking termination.

Higher-rank types

Question

What is the type of this expression?

```
\f -> (f 'a', f Bool)
```

Higher-rank types

Question

What is the type of this expression?

```
\f -> (f 'a', f Bool)
```

A couple of non-comparable solutions

```
      (∀ a . a -> a) -> (Int, Bool)
∀ r . (∀ a . a -> r) -> (r,   r)
```

Higher-rank types

`(∀ a . a -> a) -> (Int, Bool)` is a *rank-2* type

- The ∀ is buried one level deep at the *left* of an arrow.

Question

Why do we insist on *left* on an arrow?
Why is `(Int, Bool) -> (∀ a . a -> a)` rank-1?

Uses of higher-rank types

- Encapsulation of side effects:

  ```
  runST :: ∀ v . (∀ s . ST s v) -> v
  ```

- Dynamic types / Leibniz equality:

  ```
  data Equal a b = Equal (∀ f . f a -> f b)
  ```

  Roughly, a is equal to b if you can susbtitute one for the other in all contexts.

- Generic programming à la Scrap Your Boilerplate:

  ```
  everywhere :: (∀ b. Data b => b -> b)
             ->  ∀ a. Data a => a -> a
  ```

Uses of higher-rank types

- van Laarhoven lenses:

```
type Lens  s a = ∀ f. Functor f
                => (a -> f a) -> s -> f s
type Prism s a = ∀ f. Applicative f
                => (a -> f a) -> s -> f s
```

- Dictionaries for higher-kinded types:

```
data MonadDict m where
  MonadDict :: {
      return :: ∀ a . a -> m a
    , (>>=)  :: ∀ a b . m a -> (a -> m b) -> m b
    } -> MonadDict m
```

Impredicative types

What if now we have a list with `runST`?

```
[runST] :: ∀ v . [(∀ s . ST s v) -> v]
        :: [∀ v . (∀ s . ST s v) -> v]
     -- the types are non-comparable
```

Impredicativity means that a type variable is instantiated with a polymorphic type.

- Damas-Milner restricts instantiation to monotypes.

Impredicativity in the compiler

System FC is *fully impredicative*.

Type inference for System FC is *undecidable*.

- There is a good story for higher-rank types.
- We do not know yet how to do inference for impredicativity.

As a result, in current GHC:

- Higher-rank types are available with `RankNTypes`.
- `ImpredicativeTypes` is deprecated.
- If you really need impredicativity, you need to annotate *every* instantiation using `TypeApplications`.

Summary

- GHC uses System FC as a target for compilation.
  - Typed lambda-calculus + data types + coercions
  - Type classes are translated to *dictionaries*.
  - Types are fully explicit.
- *Inference* obtains the explicit information from the source code.
  - Hindley-Damas-Milner is the classic approach.
  - Nowadays, inference uses constraints.