CSE 116: Fall 2019

Introduction to Functional Programming

Polymorphism and Type Inference

Owen Arden
UC Santa Cruz

Based on course materials developed by Nadia Polikarpova

Roadmap

Past two weeks:

How do we *implement* a tiny functional language?

1. *Interpreter:* how do we *evaluate* a program given its AST?
2. *Parser:* how do we convert strings to ASTs?

This week: adding types

How do we check statically if our programs "make sense"?

1. *Type system:* formalizing the intuition about which expressions have which types
2. *Type inference:* computing the type of an expression

Reminder: Nano2

```
e ::= n | x             -- numbers, vars
    | e1 + e2           -- arithmetic
    | \x -> e           -- abstraction
    | e1 e2             -- application
    | let x = e1 in e2 -- let binding
```

QUIZ

Answer: **D**.

A adds a function;

B applies a number;

C defines `f` to take an `Int` and then passes in a function;

E requires a type `T` that is equal to `T -> T`, which doesn't exit.

Type system for Nano2

A **type system** defines what types an expression can have

To define a type system we need to define:

- the *syntax* of types: what do types look like?
- the *static semantics* of our language (i.e. the typing rules): assign types to expressions

Type system: take 1

Syntax of types:

```
T ::= Int       -- integers
    | T1 -> T2  -- function types
```

Now we want to define a *typing relation* `e :: T` (`e` has type `T`)

We define this relation *inductively* through a set of *typing rules*:

```
[T-Num]  n :: Int

         e1 :: Int    e2 :: Int   -- premises
[T-Add]  ----------------------
            e1 + e2 :: Int        -- conclusion

[T-Var]   x :: ???
```

What is the type of a variable?

We have to remember what type of expression it was bound to!

Type Environment

An expression has a type in a given **type environment** (also called **context**), which maps all its *free variables* to their *types*

```
G = x1:T1, x2:T2, ..., xn:Tn
```

Our *typing relation* should include the context G:

`G |- e :: T` (e has type T in context G)

Typing rules: take 2

```
[T-Num]  G |- n :: Int

         G |- e1 :: Int   G |- e2 :: Int
[T-Add]  ------------------------------
               G |- e1 + e2 :: Int

[T-Var]  G |- x :: T          if x:T in G

          G,x:T1 |- e :: T2
[T-Abs] -----------------------
        G |- \x -> e :: T1 -> T2

        G |- e1 :: T1 -> T2   G |- e2 :: T1
[T-App] ----------------------------------
                G |- e1 e2 :: T2

        G |- e1 :: T1   G,x:T1 |- e2 :: T2
[T-Let] -----------------------------------
           G |- let x = e1 in e2 :: T2
```


Typing rules

`G |- e :: T`

An expression e **has type** T in G if we can derive `G |- e :: T` using these rules

An expression e is **well-typed** in G if we can derive `G |- e :: T` for some type T

- and **ill-typed** otherwise

Examples

Example 1:

Let's derive: `[] |- (\x -> x) 2 :: Int`

```
[T-Var] -------------------
         [x:Int] |- x :: Int
[T-Abs] -------------------             -------------- [T-Num]
         [] |- \x -> x :: Int -> Int    [] |- 2 :: Int
[T-App] ------------------------------------------------
         [] |- (\x -> x) 2 :: Int
```

But we *cannot* derive: `[] |- 1 2 :: T` for any type `T`

- Why?
- `T-App` only applies when LHS has a function type, but there's no rule to derive a function type for `1`

Examples

Example 2:

Let's derive: `[] |- let x = 1 in x + 2 :: Int`

```
                [T-Var]-----------------   -----------------[T-Num]
                       x:Int |- x :: Int   x:Int |- 2 :: Int
[T-Num] -------------   ----------------------------------[T-Add]
        [] |- 1 :: Int   x:Int |- x + 2 :: Int
[T-Let] ----------------------------------
        [] |- let x = 1 in x + 2 :: Int
```

But we *cannot* derive: `[] |- let x = \y -> y in x + 2 :: T` for any type `T`

The `[T-Var]` rule above will fail to derive `x :: Int`

Examples

Example 3:

We cannot derive: `[] |- (\x -> x x) :: T` for any type `T`

We cannot find any type `T` to fill in for `x`, because it has to be equal to `T -> T`

A note about typing rules

According to these rules, an expression can have *zero*, *one*, or *many* types

- examples?

`1 2` has no types; `1` has one type (`Int`)

`\x -> x` has many types:

- we can derive `[] |- \x -> x :: Int -> Int`
- or `[] |- \x -> x :: (Int -> Int) -> (Int -> Int)`
- or `T -> T` for any concrete `T`

We would like every well-typed expression to have a single **most general** type!

- most general type = allows most uses
- infer type once and reuse later

QUIZ

Answer: **B**.

Double identity

```
let id = \x -> x in
  let y = id 5 in
    id (\z -> z + y)
```

Intuitively this program looks okay, but our type system *rejects* it:

- in the first application, `id` needs to have type `Int -> Int`
- in the second application, `id` needs to have type `(Int -> Int) -> (Int -> Int)`
- the type system forces us to pick *just one type* for each variable, such as `id` :(

What can we do?

Polymorphic types

Intuitively, we can describe the type of `id` like this:

- it's a function type where
- the argument type can be any type `T`
- the return type is then also `T`

Polymorphic types

We formalize this intuition as a **polymorphic type**: `forall a . a -> a`

- where `a` is a (bound) type variable
- also called a **type scheme**
- Haskell also has polymorphic types, but you don't usually write `forall a.`

We can **instantiate** this scheme into different types by replacing `a` in the body with some type, e.g.

- instantiating with `Int` yields `Int -> Int`
- instantiating with `Int -> Int` yields `(Int -> Int) -> Int -> Int`
- etc.

Inference with polymorphic types

With polymorphic types, we can derive `e :: Int -> Int` where `e` is

```
let id = \x -> x in
  let y = id 5 in
    id (\z -> z + y)
```

At a high level, inference works as follows:

1. When we have to pick a type `T` for `x`, we pick a **fresh type variable** `a`
2. So the type of `\x -> x` comes out as `a -> a`
3. We can **generalize** this type to `forall a . a -> a`
4. When we apply `id` the first time, we **instantiate** this polymorphic type with `Int`
5. When we apply `id` the second time, we **instantiate** this polymorphic type with `Int ->Int`

Let's formalize this intuition as a type system!

Type system: take 3

Syntax of types

```
-- Mono-types
T ::= Int       -- integers
    | T1 -> T2  -- function types
    | a         -- NEW: type variable

-- NEW: Poly-types (type schemes)
S ::= T            -- mono-type
    | forall a . S -- polymorphic type
```

where *a* ∈ *TVar*, *T* ∈ *Type*, *S* ∈ *Poly*

Type Environment

The type environment now maps variables to poly-types: `G : Var -> Poly`

- example, `G = [z: Int, id: forall a . a -> a]`

Type system: take 3

Type Substitutions

We need a mechanism for replacing all type variables in a type with another type

A **type substitution** is a finite map from type variables to types: `U : TVar -> Type`

- example: `U1 = [a / Int, b / (c -> c)]`

To **apply** a substitution `U` to a type `T` means replace all type vars in `T` with whatever they are mapped to in `U`

- example 1: `U1 (a -> a) = Int -> Int`
- example 2: `U1 Int = Int`

QUIZ

(B) `(c -> c) -> d -> (c -> c)`

Answer: B

Typing rules

We need to change the typing rules so that:

1. Variables (and their definitions) can have polymorphic types

```
[T-Var]  G |- x :: S          if x:S in G

        G |- e1 :: S   G, x:S |- e2 :: T
[T-Let] ------------------------------------
           G |- let x = e1 in e2 :: T
```


Typing rules

2. We can *instantiate* a type scheme into a type

```
         G |- e :: forall a . S
[T-Inst] ----------------------
          G |- e :: [a / T] S
```

3. We can *generalize* a type with free type variables into a type scheme

```
           G |- e :: S
[T-Gen] ---------------------- if not (a in FTV(G))
        G |- e :: forall a . S
```


Typing rules

The rest of the rules are the same:

```
[T-Num]  G |- n :: Int

         G |- e1 :: Int   G |- e2 :: Int
[T-Add]  -------------------------------
               G |- e1 + e2 :: Int

         G, x:T1 |- e :: T2
[T-Abs] ------------------------
        G |- \x -> e :: T1 -> T2

        G |- e1 :: T1 -> T2   G |- e2 :: T1
[T-App] -----------------------------------
                 G |- e1 e2 :: T2
```

Examples

Example 1

Let's derive: `[] |- \x -> x :: forall a . a -> a`

```
[T-Var] ---------------
        [x:a] |- x :: a
[T-Abs] -----------------------
        [] |- \x -> x :: a -> a
[T-Gen] ----------------------------------- not (a in FTV([]))
        [] |- \x -> x :: forall a . a -> a
```

Can we derive: `[x:a] |- x :: forall a . a`?

No! The side condition of `[T-Gen]` is violated because `a` **is present** in the context

Examples

Example 2

Let's derive: `G1 |- id 5 :: Int` where `G1 = [id : (forall a . a -> a)]`:

```
[T-Var]-------------------------
     G1 |- id :: forall a.a -> a
[T-Inst]------------------------   --------------[T-Num]
        G1 |- id :: Int -> Int     G1 |- 5 :: Int
[T-App] ---------------------------------------
        G1 |- id 5 :: Int
```


Examples

Example 3

Finally, we can derive:

```
(let id = \x -> x in
  let y = id 5 in
    id (\z -> z + y)) :: Int -> Int
```

Examples

```
                                                easy
    [T-Var]-----------------------------     ------------------ [Add]
          G2 |- id::forall a.a -> a          G3 |- z + y :: Int
[T-Inst]------------------------------   -------------------------[T-Abs]
        G2 |- id::(Int->Int)->Int->Int  G2 |- \z -> z+y :: Int->Int
                                |                          |
               example 2        |                          |
           -----------------   ----------------------------------[T-App]
           G1 |- id 5 :: Int   G2 |- id (\z -> z+y) :: Int -> Int
   [T-Let] -----------------------------------------------------
                              G1 |- let y = id 5 in ... :: Int -> Int
                                                         |
                  example 1                              |
[T-Abs] --------------------------------                 |
        [] |- \x -> x :: forall a.a -> a                 |
[T-Let] -------------------------------------------------------------
        [] |- let id = \x -> x in ... :: Int -> Int
```

- `G1 = [id : (forall a . a -> a)]`
- `G2 = [y : Int, id : (forall a . a -> a)]`
- `G3 = [z : Int, y : Int, id : (forall a . a -> a)]`

Type inference algorithm

Our ultimate goal is to implement a Haskell function `infer` which

- given a context `G` and an expression `e`
- returns a type `T` such that `G |- e :: T`
- *or* reports a type error if `e` is ill-typed in `G`

Representing types

First, let's define a Haskell datatype to represent Nano2 types:

```
data Type
  = TInt             -- Int
  | Type :=> Type    -- T1 -> T2
  | TVar String        -- a, b, c

data Poly = Mono Type
          | Forall TVar Poly

type TVar = String
type TEnv = [(Id, Poly)]    -- type environment
type Subst = [(String, Type)] -- type substitution
```

Inference: main idea

Let's implement `infer` like this:

1. Depending on what kind of expression `e` is, find a typing rule that applies to it
2. If the rule has premises, recursively call `infer` to obtain the types of sub-expressions
3. Combine the types of sub-expression according to the conclusion of the rule
4. If no rule applies, report a type error

Inference: main idea

```
-- | This is not the final version!!!
infer :: TypeEnv -> Expr -> Type
infer _    (ENum _)     = TInt
infer tEnv (EVar var)   = lookup var tEnv
infer tEnv (EAdd e1 e2) =
  if t1 == TInt && t2 == TInt
    then return TInt
    else throw "type error: + expects Int operands"
  where
    t1 = infer tEnv e1
    t2 = infer tEnv e2
...
```

This doesn't quite work (for other cases). Why?

Inference: tricky bits

The trouble is that our typing rules are *nondeterministic*!

- When building derivations, sometimes we had to *guess* how to proceed

Problem 1: Guessing a type

```
       -- oh, now we know!
[T-Var]----------------
       [x:?] |- x: Int    [x:?] |- 1 :: Int
[T-Add]---------------------------
       [x:?] |- x + 1 :: ?? -- what should "?" be?
[T-Abs]------------------------------
       [] |- (\x -> x + 1) :: ? -> ??
```

Inference: tricky bits

Problem 1: Guessing a type

So, if we want to implement

```
infer tEnv (ELam x e) = tX :=> tBody
  where
    tEnv'  = extendTEnv x tX tEnv
    tX     = ??? -- what do we put here?
    tBody  = infer tEnv' e
...
```


Inference: tricky bits

Problem 2: Guessing when to generalize

In the derivation for

```
(let id = \x -> x in
  let y = id 5 in
    id (\z -> z + y)) :: Int -> Int
```

we had to *guess* that the type of `id` should be generalized into

```
forall a . a -> a
```

Let's deal with problem 1 first

Constraint-based type inference

```
       -- oh, now we know!
[T-Var]----------------
       [x:?] |- x: Int    [x:?] |- 1 :: Int
[T-Add]----------------------------
       [x:?] |- x + 1 :: ?? -- what should "?" be?
[T-Abs]------------------------------
       [] |- (\x -> x + 1) :: ? -> ??
```

Main idea:

1. Whenever you need to "guess" a type, don't.
   - just return a **fresh** type variable
   - *fresh* = not used anywhere else in the program
2. Whenever a rule *imposes a constraint* on a type (i.e. says it should have certain form):
   - try to find the right *substitution* for the free type vars to satisfy the constraint
   - this step is called **unification**

Example

Let's infer the type of `\x -> x + 1`:

```
-- TEnv     Expression     Step            Subst      Inferred type
1  []       \x -> x + 1    [T-Abs]         []
2  [x:a0]         x + 1    [T-Add]
3                 x        [T-Var]                    a0
4                 x + 1    unify a0 Int    [a0/Int]
5  [x:Int]            1    [T-Num]                    Int
6                 x + 1    unify Int Int
7                 x + 1                               Int
8  []       \x -> x + 1                               Int -> Int
```


Example

1. Infer the type of (`\x -> x + 1`) in `[]` (apply `[T-Abs]`)
2. For the type of `x`, pick *fresh type variable* (say, `a0`); infer the type of `x + 1` in `[x:a0]`(apply `[T-Add]`)
3. Infer the type of `x` in `[x:a0]` (apply `[T-Var]`); result: `a0`
4. `[T-Add]` *imposes a constraint*: its LHS must be of type `Int`, so *unify* `a0` and `Int` and update the *current substitution* to `[a0 / Int]`
5. *Apply* the current substitution `[a0/Int]` to the type environment `[x:a0]` to get `[x:Int]`. Infer the type of `1` in `[x:Int]` (apply `[T-Num]`); result: `Int`
6. `[T-Add]` *imposes a constraint*: its RHS must be of type `Int`, so *unify* `Int` and `Int`; current substitution doesn't change\
7. By conclusion of `[T-Add]`: return `Int` as the inferred type\
8. By conclusion of `[T-Lam]`: return `Int -> Int` as the inferred type

Unification

The **unification** problem: given two types `T1` and `T2`, find a type substitution `U` such that `U T1 =U T2`.

Such a substitution is called a *unifier* of `T1` and `T2`

Examples:

The unifier of:

```
a          and  Int         is [a / Int]
a -> a     and  Int -> Int  is [a / Int]
a -> Int   and  Int -> b    is [a / Int, b / Int]
Int        and  Int         is []
a          and  a           is []
Int        and Int -> Int   cannot unify!
Int        and a -> a       cannot unify!
a          and a -> a       cannot unify!
```

QUIZ

(C), (D) and (E) are all unifiers!

But somehow (D) and (E) are *better* than (C)

- they make the *least commitment* required to make these types equal
- this is called **the most general unifier**

Infer: take 2

Let's add constraint-based typing to `infer`!

```
-- | Now has to keep track of current substitution!
infer :: Subst -> TypeEnv -> Expr -> (Subst, Type)
infer sub _    (ENum _)     = (sub, TInt)
infer sub tEnv (EVar var)   = (sub, lookup var tEnv)

-- Lambda case: simply generate fresh type variable!
infer sub tEnv (ELam x e) = (sub1, tX' :=> tBody)
  where
    tEnv'          = extendTEnv x tX tEnv
    tX             = freshTV -- we'll get to this
    (sub1, tBody)  = infer sub tEnv' e
    tX'            = apply sub1 tX
```


Infer: take 2

```
-- Add case: recursively infer types of operands
-- and enforce constraint that they are both Int
infer sub tEnv (EAdd e1 e2) = (sub4, TInt)
  where
    (sub1, t1) = infer sub tEnv e1   -- 1. infer type of e1
    sub2       = unify sub1 t1 Int   -- 2. constraint: t1 is Int
    tEnv'      = apply sub2 tEnv     -- 3. apply subst to context
    (sub3, t2) = infer sub2 tEnv' e2 -- 4. infer e2 type in new ctx
    sub4       = unify sub3 t2 Int   -- 5. constraint: t2 is Int
```

Why are all these steps necessary? Can't we just return `(sub, TInt)`?

QUIZ

Answer: E.

A fails in step 1 (LHS is ill-typed);

B fails in step 4 (RHS is ill-typed);

C fails in step 2 (LHS is not `Int`);

D fails in step 5 (RHS is not `Int`);

finally, E should fails because LHS and RHS by themselves are fine, but not together!

Fresh type variables

```
-- | Now has to keep track of current substitution!
infer :: Subst -> TypeEnv -> Expr -> (Subst, Type)

-- Lambda case: simply generate fresh type variable!
infer tEnv (ELam x e) = tX :=> tBody
  where
    tEnv'  = extendTEnv x tX tEnv
    tX     = freshTV -- how do we do this?
    tBody  = infer tEnv' e
```

Intended behavior:

- First time we call `freshTV` it returns `a0`
- Second time it returns `a1`
- .. and so on

Can we do that in Haskell?

No, Haskell is pure. Have to thread the counter through :(

Polymorphism: the final frontier

Back to double identity:

```
let id = \x -> x in   -- Must generalize the type of id
  let y = id 5 in     -- Instantiate with Int
    id (\z -> z + y)  -- Instantiate with (Int -> Int)
```

- When should we to generalize a type like `a -> a` into a polymorphic type like `forall a .a -> a`?
- When should we instantiate a polymorphic type like `forall a . a -> a` and with what?

Polymorphism: the final frontier

Generalization and instantiation:

- Whenever we infer a type for a let-defined variable, generalize it!
  - it's safe to do so, even when not strictly necessary
- Whenever we see a variable with a polymorphic type, instantiate it
  - with what type?
  - well, what do we use when we don't know what type to use?
  - *fresh type variables!*

Example

Let's infer the type of `let id = \x -> x in id 5`:

```
-- TEnv   Expression             Step           Subst           Type

1  []     let id=\x->x in id 5   [T-Let]        []
2                \x->x           [T-Abs]
3  [x:a0]           x            [T-Var]                        a0
4                \x->x                                          a0 -> a0
5  []     let id=\x->x in id 5   generalize a0
6  tEnv                   id 5   [T-App]
7                         id     [T-Var]
8                         id     instantiate                    a1 -> a1
9                            5   [T-Num]                        Int
10                        id 5   unify (a1->a1)
                                      (Int->a2) [a1/Int,a2/Int]
10                        id 5                                  Int
11 []     let id=\x->x in id 5                                  Int
```

Here `tEnv = [id : forall a0.a0->a0]`

What we learned this week

Type system: a set of rules about which expressions have which types

Type environment (or context): a mapping of variables to their types

Polymorphic type: a type parameterized with type variables that can be instantiated with any concrete type

Type substitution: a mapping of type variables to types; you can **apply** a substitution to a type by replacing all its variables with their values in the substitution

Unifier of two types: a substitution that makes them equal; **unification** is the process of finding a unifier

What we learned this week

Type inference: an algorithm to determine the type of an expression

Constraint-based type inference: a type inference technique that uses fresh type variables and unification

Generalization: turning a mono-type with free type variables into a polymorphic type (by binding its variables with a `forall`)

Instantiation: turning a polymorphic type into a mono-type by substituting type variables in its body with some types