Type inference

DENIS MIGINSKY

Questions of the day

- Why do we need types in programming languages (PL)?
- What is a type in PL?
- Can we have PLs without types?
- What are the differences and similarities between types in PL and types in λ-calculus?

Type’s purposes

1. How does a type relate to other types? What is the structures and representation of a type? How to construct instances of a type and extract their parts?
   Ex: Complex ::= Float × Float

2. How do instances of a type behave?
   Ex: sum : Complex->Complex->Complex

3. Check the overall correctness of the program (syntactic level only!) without running it
   Ex: sum 2 “word” – justify that this is incorrect program!

Static and dynamic typing

In languages with **static typing** types are used for checking/inference in compile time.

Pros: a compiler does a lot of work to verify the correctness of a program.

Cons: static typing usually limits the capabilities of a language (consider the array with arbitrary types of elements) and extend the amount of "redundant" code. The problems are less significant with more advanced type system.

In languages with **dynamic typing** types are used only in runtime to determine possible behavior.

Pros: no limitations from a type system, a compiler is fast

Cons: a lack of static type verification requires other techniques to manage quality. Dynamic type checking is usually much simpler than dynamic, but it also takes time. Dynamic languages are significantly slower.

Don't mess with strong and weak typing. Both of these are strong!

Type: λ vs programming languages

λ-calculus	C	Haskell	Java	λ in Clojure syntax
x: Int ⊢ x	int x; x;			{x :Int} x
inc: Int->Int ⊢ inc x		inc :: Int->Int inc x		{inc (-> :Int :Int)} (inc x)

x: List Int ⊢ head x			List<Integer> x;
			x.get(0);
				{x (:List :Int)} (head x)

⊢ λ x.x		\x -> x	<T> T f (T x) {return x;} x-> x	{} (lambda x x)
⊢(λ x.x) 10		(\x -> x) 10		{} ((lambda x x) 10)

add: Int->Int->Int⊢(λ xy.add x y)	int add (int x, int y){
	…
	}
	//what next?
				{add (-> :Int
				(-> :Int :Int)) }
				(lambda x
				(lambda y (add x y)))

add : Int × Int -> Int	int add (int x, int y){ …}	add :: (Int, Int) ->Int		{add (-> (cross :Int :Int) :Int) }

Str × Int × Int	struct {
	char* name;
	int x, y;}
		(String, Int, Int)		(cross :Str :Int :Int)

Fill the gaps

Type inference problem

Type inference (reconstruction) problem: for the given term and the context (Γ) for each subterm (including the term itself) the following should be done:

- If a subterm has known type: check, if it is correct (**type checking**)
- If a subterm has no type: infer this type or prove that it is impossible (**type inference**)

Inference:

$y:\ \text{Str} \vdash (\lambda x: ?.\ y) : ?$

$x:\ \text{Int},\ +: \text{Int->Int->Int} \vdash (\lambda y: ?\ .\ (x + y): ?) : ?$

$\vdash ((\lambda x: ?.x): ?\ 10): ?$

Checking:

$+: \text{Int->Int->Int} \vdash (\lambda x: \text{Int}.\ \lambda y: \text{Str}.\ x+y): \text{Int->Int->Str}$

Simply typed λ-calculus

$$\frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha}\ [Var]$$

When a ***variable*** is met, its type should be taken from the current ***lexical scope***

$$\frac{\alpha = typeOfConst(c)}{\vdash c : \alpha}\ [Const]$$

When a ***constant*** is met, its type should be known from the language syntax

$$\frac{\Gamma \vdash f : \alpha \to \beta \qquad \Gamma \vdash x : \alpha}{\Gamma \vdash f\ x : \beta}\ [App]$$

When an ***application*** is met:

1. Infer the type of the ***function*** to be in form $\alpha \to \beta$ (or fail otherwise)
2. Inter the type of the ***actual argument***, it must be α (or fail otherwise)

Now it can be deduced that the type of the whole **application** is β

$$\frac{\Gamma, x : \alpha \vdash y : \beta}{\Gamma \vdash \lambda\, x.y : \alpha \to \beta}\ [Abs]$$

When an ***abstraction (function definition)*** is met:

1. Make sure that type α of the ***formal argument*** is provided
2. Infer the ***body*** type β in the scope Γ enriched with x:α

Now it can be deduced that the type of the ***function*** is $\alpha \to \beta$

This is type checking mostly, not inference

Example: inference algorithm

$$\frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha}\ [Var]$$

$$\frac{\alpha = typeOfConst(c)}{\vdash c : \alpha}\ [Const]$$

$$\frac{\Gamma \vdash f : \alpha \rightarrow \beta \qquad \Gamma \vdash x : \alpha}{\Gamma \vdash f\ x : \beta}\ [App]$$

$$\frac{\Gamma, x : \alpha \vdash y : \beta}{\Gamma \vdash \lambda\, x . y : \alpha \rightarrow \beta}\ [Abs]$$

Problem:
Γ = {inc: Int->Int}
Γ ⊢ (λx:Int . inc x: ?) :?
Solution:
1. infer (Γ , (λx:Int . *inc* x: ?)) ***[Abs]*** Int -> infer ({Γ, x:Int}, inc x)
2. infer ({Γ, y:Int}, inc x) ***[App]*** check(typeOf(inc)= * -> *),
   check(infer({Γ,x:Int}, x)=Int),
   Int
3. infer({Γ, x:Int}, x) **[Var]** Int
4. in step 2: check is OK, Int
5. in step 1: Int->Int

Problem:
((λx.x) 10)
Solution:
???

Notes on practical application

- The type systems itself is very simple
- This is pretty close to C type system

Cons:

- There are no composite types like tuples, records (this is trivial extension)
- There are no named function definitions (not so hard)
- There is no recursion (a little bit harder)
- ***Lack of inference possibilities.*** That is why C (and many other static typing languages) requires the explicit typing of almost everything (unlike Haskell, for instance). ***This is more difficult.***

Tuple extension

$$\frac{\Gamma \vdash x_1 : \alpha_1 \ \dots \ \Gamma \vdash x_n : \alpha_n}{\Gamma \vdash (x_1 \dots x_n) : \alpha_1 \times \dots \times \alpha_n} \ [Tuple]$$

A **record** (or a structure in C) is pretty similar, only syntax is a little bit different

Let extension

Syntax: *let* **x** = **y** *in* **z**

Semantics: substitute **x** with **y** inside **z**: $\dfrac{let\ x = y\ in\ z}{z[x/y]}$

Types:

$$\frac{\Gamma \vdash y : \alpha \quad \Gamma, x{:}\alpha \vdash z{:}\,\beta}{\Gamma \vdash let\ x = y\ in\ z : \beta}\ [Let]$$

When a ***let clause*** is met:

1. Infer the type of the ***substitution*** (y) α, **variable** (x) is of the same type
2. Inter the type of the ***body*** (z) β with outer ***scope*** (Γ) enriched with x: α

Unlike the evaluation, there is no need to apply the substitution to infer the type of ***body***.

Letrec extension

Syntax: *letrec* **x** = **y** *in* **z**

Semantics (informally)**:** evaluate **y** with assumption that **x** is already defined and available (recursively). Next, substitute **x** inside **z** with the evaluation result of **y**

Types:

$$\frac{\Gamma, \boldsymbol{x}{:}\boldsymbol{\alpha} \vdash y{:}\,\alpha \quad \Gamma, x{:}\alpha \vdash z{:}\,\beta}{\Gamma \vdash let\ x = y\ in\ z : \beta}\ [LetRec]$$

When the ***letrec clause*** is met:

1. Infer the type of the ***substitution*** (y) with outer ***scope*** (Γ) enriched with **x**: α. The inferred type must be α.
2. Inter the type of the ***body*** (z) β with outer ***scope*** (Γ) enriched with **x:** α

Unlike the evaluation, there is no need to apply the substitution to infer the type of ***body***.

Unlike **let**, in **letrec** the type for **x** must be explicitly provided in simple calculus

Monomorphic and polymorphic types

⊢ 42: Int	monomorphic type (monotype)
inc: Int->Int ⊢ inc: Int->Int	monotype
inc: Int->Int ⊢ inc 42: Int	monotype
⊢ λ x.x : ???	cannot be typed in our current calculus

But, we can reason that the last term has:

1. some functional type, and
2. the argument and the result are of the same type.

So:

⊢ λ x.x : ω->ω

Where ω is a kind of variable. Because all the possible binding of ω are types, it is called **type variable**.

ω->ω or, more formally, ∀ω.ω->ω is a **polymorphic type (polytype)**, i.e. type with at least one unresolved type variable.

Examples of useful polytypes

list: List α

first: List α -> α

size: ???

map: List α -> (α->β) -> List β

filter: ???

genericMap: ς α -> (α->β) -> ς β

reduce/fold: List α -> β -> (α -> β -> β) -> β

If-then-else: ???

Generalization and instantiation

$\beta = inst(\alpha)$ means that there is a substitution **S** that $\beta = \alpha$**S**

α is **more general** type and β is **more specific** type in this case

Examples:

Int = $inst(\alpha)$, S=[α/Int]

α->Int->Int = $inst(\alpha$->$\beta)$, S=[β/Int->Int]

Int->Int = $inst(\alpha$->$\alpha)$, S=[α/Int]

α->α = $inst(\alpha$->$\beta)$, S=[β/α]

- $\alpha = inst(\alpha)$ for any α
- A **monotype** has no more specific types besides itself.
- A **polytype** of kind α (just a single type variable) is the most general polytype

Polytypes in type inference

Problem: ⊢ ((λx:?.x:?):? 10): ?

Solution:

1. infer ({}, ((λx:?.x:?):? 10)) ***[App]*** infer({}, λx?.x:?)= α->β,
   10:Int,
   unify(α, Int)=[α/Int]

2. infer ({}, (λx:?.x:?) :?) ***[Abs]*** infer({x: Int}, x/*body*/)=β,
   unify(Int, β) = [β/Int]

unify(α, β) finds the **most general unification** for α and β (with Robinson algorithm or any other) or fails.

Hindley-Milner type system: basic rules

$$\frac{\alpha = typeOfConst(c)}{\vdash c : \alpha}\ [Const] \qquad \frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha}\ [Var] \qquad \frac{\Gamma \vdash x_1 : \alpha_1 \ldots \Gamma \vdash x_n : \alpha_n}{\Gamma \vdash (x_1 \ldots x_n) : \alpha_1 \times \ldots \times \alpha_n}\ [Tuple]$$

$$\frac{x \notin free(\Gamma) \quad \omega = newvar}{\Gamma \vdash x : \omega}\ [AVar2]$$

When a **variable** is met that is not defined in the **scope**, assume that its type is of some **type variable**, not used before (ω=$newvar$).

$$\frac{\boldsymbol{\omega = newvar} \qquad \Gamma, x : \omega \vdash y : \beta}{\Gamma \vdash \lambda x. y : \omega \rightarrow \beta}\ [AAbs]$$

When an ***abstraction (function definition)*** is met:

1. Get new type variable for the formal argument
2. Infer the **body** type β in the scope Γ enriched with x: ω

Now it can be deduced that the type of the ***function*** is $\omega \rightarrow \beta$

$$\frac{\Gamma \vdash f : \alpha \rightarrow \beta \quad \Gamma \vdash x : \alpha' \quad \boldsymbol{unify(\alpha, \alpha')}}{\Gamma \vdash f\ x : \beta}\ [AApp]$$

When an ***application*** is met:

1. Infer the type of the ***function*** to be in form $\alpha \rightarrow \beta$ (or fail)
2. Inter the type α' of the ***actual argument***, and perform **unification** with α

Now it can be deduced that the type of the whole **application** is β

Let polymorphism

Problems:
⊢((λf x y.(f x, f y)) (λx.x):? 10 “abc”):?

⊢(let f=(λx.x):? in (f 10, f “abc”)) :?

f will have different monotypes in different situations.
So the unification will fail.
Extension: don’t ask about **f**’s type

⊢(let f=(λx.x) in (f 10, f “abc”)) :?
And replace the rule with the following:

```
//Java: the same as polymorphic let
public class SomeClass{
    public static <X> X f(X x){
        return x;
    }
    public static void main(){
        System.out.println(
            new Tuple2<Integer, String>(
                f(10), f(“abc”)));
    }
}
```

$$\frac{\Gamma\vdash[x/y]\ z:\beta}{\Gamma\vdash let\ x=y\ in\ z:\beta}\ [ALet]$$

For a ***let clause***:
1. Apply syntactic substitution $[\boldsymbol{x}/\boldsymbol{y}]\ \boldsymbol{z}$
2. Infer the type of z after the substitution. Type for $\boldsymbol{y}$ will be inferred separately for each occurrence inside $\boldsymbol{z}$

Letrec with polymorphism

letrec **x** = **y** *in* **z**
A type for this clause with polymorphic extension cannot be inferred by simple substitution
z [x/y] because **y** can also contain **x** for recursion purposes.

Strategy: evaluate **letrec** as there is separate **letrec** definition for each occurrence of **x** inside **z**

$$\frac{\forall i.\,\omega_i = newvar \quad \forall i.\,x_i = newvar \quad \forall i.\,y_i = y[x/x_i] \qquad \forall i \Gamma, x_i{:}\,\omega_i \vdash y_i{:}\,\alpha_i \quad \forall i.\,unify(\omega_i, \alpha_i) \qquad \Gamma, \forall i.\,x_i{:}\omega_i \vdash z[x\,/\,x_1 \ldots x_n] : \beta}{\Gamma \vdash letrec\ x = y\ in\ z : \beta}\ [ALetRec]$$

For a ***letrec clause***:
1. Find all the occurrences of **x** in **z**
2. For each occurrence **i**:
    a) Generate new variable **x_i** to replace **x**, type variable **ω_i** for it and apply the substitution **y_i=y[x/x_i]**
    b) Infer the type of **y_i** in the scope enriched with **x_i:ω_i** and unify it with **ω_i**.
    c) Substitute this occurrence of **x** in **z with x_i**
3. Now infer the type of **z** with all the substitutions in the scope enriched with **∀i.x_i:ω_i**

Hindley-Milner type system: inference algorithm

With given scopes of term variables **Γ** and empty scope of substitutions **Ω** for type variables:

1. Take the top-most syntactic structure
2. Find appropriate rule and apply it (there will be only one for each case)
3. All the unifications update **Ω**

- Some of the rules are recursive, i.e. they "ask" to infer types of sub-clauses of a given clause.
- Scopes of **term variables** are different for different nodes of syntax tree.
  In different branches there could be different term variables with the same name.
- Scope of **type variables Ω** is global.

Extended Hindley-Milner type system: rules for algorithm

$$\frac{x\colon \alpha \in \Gamma}{\Gamma \vdash x : \alpha}\ [Var]$$

$$\frac{\omega = newvar \qquad \Gamma, x : \omega \vdash y\colon \beta}{\Gamma \vdash \lambda\, x. y : \omega \to \beta}\ [AAbs]$$

$$\frac{x \notin free(\Gamma) \quad \omega = newvar}{\Gamma \vdash x : \omega}\ [AVar2]$$

$$\frac{\Gamma \vdash f : \alpha \to \beta \quad \Gamma \vdash x : \alpha' \quad unify(\alpha, \alpha')}{\Gamma \vdash f\ x : \beta}\ [AApp]$$

$$\frac{\alpha = typeOfConst(c)}{\vdash c\colon \alpha}\ [Const]$$

$$\frac{\begin{array}{c}\forall i. \omega_i = newvar \qquad \forall i. x_i = newvar \qquad \forall i. y_i = y[x/x_i] \\ \forall i \Gamma, x_i\colon \omega_i \vdash y_i\colon \alpha_i \qquad \forall i. unify(\omega_i, \alpha_i) \\ \Gamma, \forall i. x_i{:}\omega_i \vdash z[x\ /\ x_1 \ldots x_n] : \beta\end{array}}{\Gamma \vdash letrec\ x = y\ in\ z : \beta}\ [ALetRec]$$

$$\frac{\Gamma \vdash x_1\colon \alpha_1 \ldots \Gamma \vdash x_n\colon \alpha_n}{\Gamma \vdash (x_1 \ldots x_n)\colon \alpha_1 \times \ldots \times \alpha_n}\ [Tuple]$$

Syntax

Terms	λ	λ in Clojure
Term variable	x	x
Application	f x	(f x)
Abstraction	λ x.y	(lambda x y)
Let/Letrec	let x=y in z	(let [x y] z)
Tuple	(x, y, z)	[x y z]

Types	λ	λ in Clojure
Primitive type	Int List	:Int :List
Type variable	α β	A B
Scope	Γ={x:Int, y:Str}	{x :Int, y :Str}
Functional type	Int->α	(-> :Int A)
Type instantiation	List Int	(:List :Int)
Cross product	Int×Str	(cross :Int :Str)

Problem	λ	λ in Clojure
	Γ = {inc: Int->Int} Γ ⊢ (λx. inc x)	{inc (-> :Int :Int)} (lambda x (inc x))
	Γ = {*: Int->Int->Int, =: α->α->Bool} Γ ⊢ let fact=λn.if (n=1) 1 n*(fact (dec n)) in fact 10	{* (-> :Int (-> :Int :Int)), = (-> A (-> A :Bool))} (let [fact (lambda n (if (= n 1) 1 (* n (fact (dec n)))))] (fact 10))