

Type Inference for ML-like Languages

In this lecture we will look at type handling in semantic analysis

- **Type inference / reconstruction for ML-like languages**
- **Hindley-Milner type system** (in a toy version)

Type Checking versus Type Inference

In an attribute grammar for type checking the type of the operations is typically decided from the types of the operands: for a simple expression the information on the types flows in principle only in one direction.

Such a computation is only possible if the types of all identifiers are declared explicitly and all type coercions are explicit.

Consider the ML example:

```
fun q (x::xs) y = 1::(q xs y) |  
  q [] y = y ;
```

```
> val q = fn : 'a list -> int list -> int list
```

To determine (implicit) types information needs to be propagated in a more complex way.

Hindley-Milner Type Inference

The Hindley-Milner type system is the basis for languages like ML, Objective Caml, Haskell etc.

It is a formal method for type inference in ML-like languages and offers a restricted form of parametric polymorphism (let-polymorphism).

We introduce a simplified version of this type system for the core of ML.

This core is encapsulated in the toy language ML_0 .

$$\begin{aligned} \text{expr} &\rightarrow \text{var} \\ \text{expr} &\rightarrow \text{fn } \text{var}_1 \dots \text{var}_n \Rightarrow \text{expr} \\ \text{expr} &\rightarrow \text{expr}(\text{expr}_1 \dots \text{expr}_n) \\ \text{expr} &\rightarrow \text{if } \text{expr} \text{ then } \text{expr} \text{ else } \text{expr} \\ \text{expr} &\rightarrow \text{let } \text{var} = \text{expr} \text{ in } \text{expr} \\ \text{var} &\rightarrow x \mid \dots \mid z \mid \dots \end{aligned}$$

Note that all expressions are implicitly typed.

Type Environments

We need the idea of type environments and judgements.

- A type environment Γ contains all the type bindings
- Γ is a collection of type bindings $x : t$.

We can make the judgement $\Gamma \vdash e : t$ (“ e has type t ”) if the environment contains the binding t for the type variable x .

Note how a type environment is somewhat similar to a normal variable environment that collects the variables and their value bindings.

Type Instantiations

Polymorphic types are represented by type schemas. We have

- type variables $'a, \dots$
- simple types $bool, int, \dots$
- polymorphic types $'a\ list, \dots$
(by application of type constructors)

These are the normal ML types.

We also need to introduce the idea of an instantiation

$$'a <: 'b$$

means that $'a$ is an instantiation of (more specific than) the type schema $'b$, for example

$$int\ list <: 'a\ list.$$

Substitutions

By substituting a type variable we can make a type schema only more specific. A type schema in which a type variable has been substituted is an instance of the schema without substitution, for example:

$$\begin{aligned} \text{int} &<: 'a \\ \text{int list} &<: 'a \\ \text{int list} &<: 'a \text{ list} \end{aligned}$$

We write $\{T/'a\}$ to indicate that we substitute $'a$ with T . To apply a substitution Θ to an expression e we write Θe , for example

$$\begin{aligned} \Theta &= \{\text{int}/'a, \text{int list}/'b\} \\ e &= 'a \times 'a \mapsto 'b \\ \Theta e &= \text{int} \times \text{int} \mapsto \text{int list}. \end{aligned}$$

We can express the instance relation $'a <: 'b$ by substitution:

$'a <: 'b$ if and only if $\exists \Theta. 'a = \Theta 'b$

Type Rules

Next we need to introduce the notion of how to build up more and more complex type judgements. For this we use type rules, i.e. inference rules for types.

We write type rules (inference rules for types) in the following form:

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : 'a \quad \Gamma \vdash e_3 : 'a}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : 'a} \text{ (If)}$$

This can be read in either of two ways:

- **bottom-up:** To show that “if e_1 then e_2 else e_3 ” is an expression of type $'a$ we need to show that e_1 is of type bool and e_2, e_3 are both of type $'a$.
- **top-down:** If we know that e_1 is of type bool and e_2, e_3 are both of type $'a$ we may conclude that “if e_1 then e_2 else e_3 ” is an expression of type $'a$.

ML₀ Type Rules

$$\frac{\Gamma \vdash x : T \quad 'a \leq : T}{\Gamma \vdash x : 'a} \text{ (Var)}$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : 'a \quad \Gamma \vdash e_3 : 'a}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : 'a} \text{ (If)}$$

$$\frac{\Gamma \vdash e_i : 'a_i, i = 1 \dots n \quad \Gamma \vdash f : 'a_1 \rightarrow \dots \rightarrow 'a_n \rightarrow 'a}{\Gamma \vdash (f \ e_1 \dots e_n) : 'a} \text{ (Apply)}$$

$$\frac{\Gamma, x_1 : 'a_1, \dots, x_n : 'a_n \vdash e : 'a}{\Gamma \vdash (\text{fn } x_1 \dots x_n \Rightarrow e) : ('a_1 \rightarrow \dots \rightarrow 'a_n \rightarrow 'a)} \text{ (Lambda)}$$

Note:

- The name “Lambda” for function abstraction stems from the roots of functional languages in the Λ -calculus.
- In the interest of a simplified treatment we skip the rule for *let*

Type Inference

To perform type inference we have to do the following:

- Represent the type of each expression with unknown type by a fresh variable
- Collect these types in a type environment
- using the type rules, decompose the expression stepwise
 - apply a type rule
 - substitute type variables in the environment according to the type information in the rule

until the expression is completely decomposed and all types are known.

The key notion is the use of type substitution.

Informal Inference Example

```
val f = fn (x, y) => y+1
```

What is the type of $f(0, y)$?

Clearly we have: $f: 'a * int \rightarrow int$

With $(0, y): int * 'b$ we obtain

$\Theta = \{int/'a, int/'b\}$.

When deriving the type of an expression we are only allowed to make substitutions that are as general as possible, example:

```
- val f = fn (x,y) => (x,y)::[];  
> val f = fn : 'a * 'b -> ('a * 'b) list  
- val g = fn x => f(1,x);  
> val g = fn : 'a -> (int * 'a) list
```

Note that $'a$ remains unsubstituted.

Unification

Finding the most general instantiation is done via unification of type schemata.

Two type schemata are unified by a substitution Θ if

$$\Theta'a = \Theta'b .$$

Θ is called the unifier

We need to find the most general unifier (MGU)

Example: $\Theta_1 = \{ 'b \text{ list} / 'a \}$, $\Theta_2 = \{ \text{int list} / 'a \}$, $\Theta = \{ \text{int} / 'b \}$

Θ_2 is more general than Θ_1 .

A unifier Θ_1 is more general than another unifier Θ_2 if,

$$\exists \Theta. \Theta \circ \Theta_1 = \Theta_2$$

i.e. there is another substitution Θ that makes Θ_1 equal to Θ_2 .

A simple unification algorithm that computes the MGU is due to A. Robinson (1965)

Unification Examples

- $int = int$?
- $real = 'a$?
- $real = real\ list$?
- $real\ list = 'a\ list$?
- $real\ list = int\ list$?
- $int \rightarrow real\ list = int \rightarrow real$?
- $int * real * 'a = 'b * 'c * 'b$?
- $int * real * 'a = 'b * 'b * 'b$?
- $int * (real \rightarrow int) = 'a * 'b$?
- $'a\ list = 'b\ list$?

Extending Type Rules with Unification (If)

We now rewrite the type rules from above to do the following:

From a given type environment Γ and expression e we compute the most general substitution Θ such that e is well typed in $\Theta\Gamma$.

From

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : 'a \quad \Gamma \vdash e_3 : 'a}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : 'a} \text{ (If)}$$

we obtain

$$\frac{\Theta\Gamma \vdash e_1, e_2, e_3 : 'a_1, 'a_2, 'a_3 \quad \Theta('a_1) = \text{bool} \quad \Theta('a_2) = \Theta('a_3)}{(\Theta' \circ \Theta)\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \Theta('a_3)}$$

Extending Type Rules with Unification (Lambda)

From

$$\frac{\Gamma, x_1 : 'a_1, \dots, x_n : 'a_n \vdash e : 'a}{\Gamma \vdash (\text{fn } x_1 \dots x_n \Rightarrow e) : ('a_1 \rightarrow \dots \rightarrow 'a_n \rightarrow 'a)} \text{ (Lambda)}$$

we obtain

$$\frac{\Theta(\Gamma, x_1 : 'a_1, \dots, x_n : 'a_n) \vdash e : 'a}{\Theta\Gamma \vdash (\text{fn } x_1 \dots x_n \Rightarrow e) : (\Theta('a_1) \rightarrow \dots \rightarrow \Theta('a_n) \rightarrow 'a)}$$

Extending Type Rules with Unification (Apply)

From

$$\frac{\Gamma \vdash e_i : 'a_i, i = 1 \dots n \quad \rightarrow \vdash f : 'a_1 \rightarrow \dots \rightarrow 'a_n \mapsto 'a}{\Gamma \vdash (f e_1 \dots e_n) : 'a} \text{ (Apply)}$$

we obtain

$$\frac{\Theta \Gamma \vdash f, e_1, \dots, e_n : 'a, 'a_1 \dots 'a_n \quad \Theta('a) = \Theta('a_1 \rightarrow \dots \rightarrow 'a_n \rightarrow 'b) \text{ where 'b is a fresh type variable}}{(\Theta \circ \Theta) \Gamma \vdash (f e_1 \dots e_n) : \Theta('b)}$$

How to Perform Type Reconstruction

On the basis of these type rules and unification we can perform type inference (reconstruction) for ML like languages in much the same way as for fully (explicitly typed) expressions.

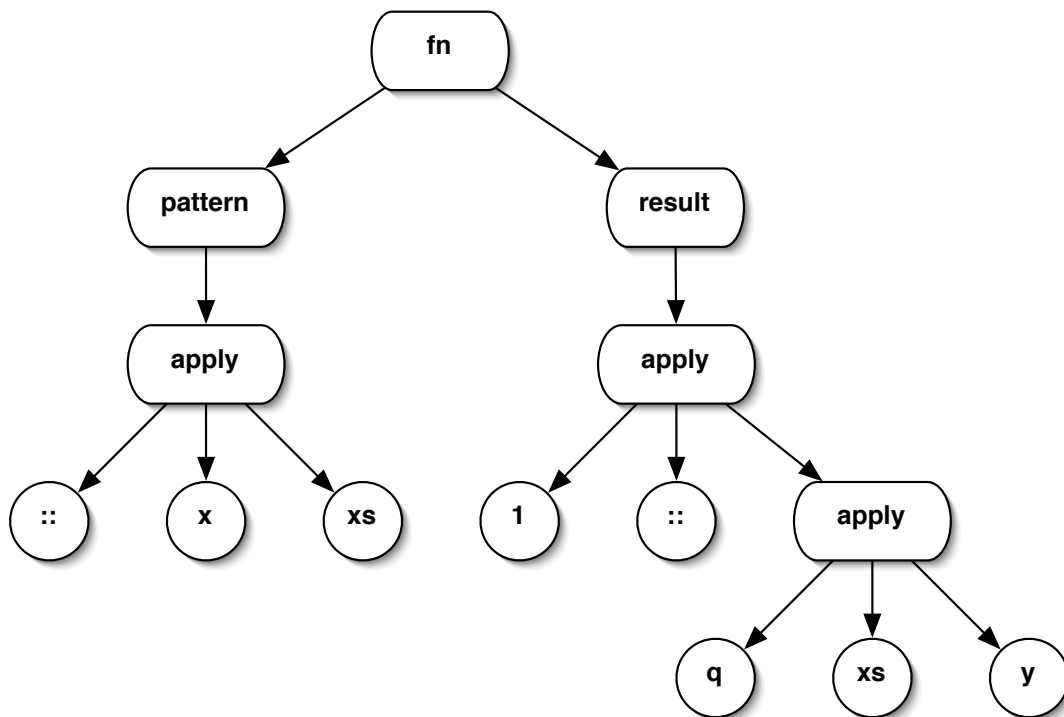
Perform a pass over the syntax tree

- type each unknown expression with a fresh type variable
- add the fresh variable to the type environment
- get the current substitution Γ
- find the MGU Θ on the basis of the type rule to be applied
- modify the environment to be $\Theta\Gamma$

Example Revisited

Exercise: Use the following (partial) syntax tree for our introductory example to perform type inference for

```
fun q (x::xs) y = 1::(q xs y)
```



Initial Type Environment:

X - 'a

Y - 'b

Xs - 'c

q - 'd

x::xs - 'e

(q xs y) - 'f

(1::(q xs y)) - 'g

Type Reconstruction Example

To reconstruct the type of

```
val f = fn x => x + 1
```

- start with head `fn x`
- add new type binding $x : 'a$ to Γ , nothing is yet known about this type
- continue with the body `x + 1`
- lookup definition of `+`.
 $\Gamma \vdash + : int \times int \mapsto int.$
- Unify $(x,1) : ('a * int)$ with $int * int$. $\Theta = \{int/'a\}$.
- modify the environment Γ by applying Θ . The new current environment is $\Theta\Gamma$.

Another Example

Reconsider the introductory example (modified to use `fn` instead of `fun`).

```
val rec q = fn ([],y)      => y
             | (x::xs, y) => (1::(q (xs,y))) ;
```

```
> val q = fn : 'a list * int list -> int list
```

initialize the type environment:

$l : int, q : 'a, x : 'b, xs : 'c, y : 'd$

$q = \text{fn} \dots$ **yields** $\Theta = \{ 'e \mapsto 'f / 'a \}$ because q is a function;

$([], y)$ **yields** $\Theta = \{ 'g \text{ list} * 'd / 'e \}$ because $([], y)$ is the argument of q ;

$\Rightarrow y$ **yields** $\Theta = \{ 'd / 'f \}$ because y is the result of q ;

$x :: xs$ **yields** $\Theta = \{ 'b \text{ list} / 'c \}$ **and** $\Theta = \{ 'g / 'b \}$

because $x :: xs$ must be a list and because the types of the two argument patterns for q must unify;

$\Rightarrow (1 :: (q \dots))$; **yields** $\Theta = \{ int \text{ list} / 'd \}$

because lists must be homogeneous and because the type must unify with the return type of q .

Composing all the above unifiers we obtain the parametric type

$q : 'a = 'g \text{ list} * int \text{ list} \mapsto int \text{ list}.$

Note how using unification has removed the need to find an ordering in which to analyze the types.

Summary

We have looked at Type handling in semantic analysis

- Type inference / reconstruction for ML-like languages
- Hindley-Milner type system

Homework

- Perform a full, detailed type reconstruction for

```
fun q [] y = y
  | q (x::xs) y = x::(q xs y);
fun r x = q [1] x;
```