

Programming Language Implementation IX

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

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

Related Material, on which this presentation is loosely based, can be found in Wilhelm and Maurer, Chapter 9 (particularly Section 9.1) and in Carl A. Gunter, “Semantics of Programming Languages”, MIT Press, Cambridge/MA 1992, Chapter 7 (particularly Section 7.5) as well as in the material for the current version of the “Programming Languages” subject at Harvard University.

Reminder: Semantic Analysis

The semantic analysis determines non-syntactic properties of the program as well as properties that cannot (easily) be determined by a context-free grammar.

Type checking and **type inference** are among the most important tasks of the semantic analysis.

For example, recall that $L = \{a^n b^m c^n d^m \mid n, m \geq 1\}$ is not context-free.

Therefore it is not a context-free property whether or not the number of parameters in function declarations and in their calls agree.

Checking whether even the types agree is a much harder problem!

Reminder: Type Checking with Attribute Grammars

In some cases the types of expressions can be determined by a bottom up (post-order) tree traversal of the syntax tree.

As attribute grammars are more powerful than “pure” context-free grammars, this can in principle be embedded in an attribute grammar.

Recall the example grammar for simple assignment expressions:

Grammar rule: $assgn \rightarrow ident := exp$

Attribution Rules:

```
assgn.operation :=
  if ident.type = int then int_asg else real_asg
ident.env := assgn.env
exp.env := assgn.env
```

Condition:

```
coercible(exp.type, ident.type) and
ident.kind = var
```

Grammar rule: $exp1 \rightarrow exp2 + exp3$

Attribution Rules:

```
exp1.type :=
  if exp2.type = int and exp3.type = int
  then int else real
exp1.operation :=
  if exp1.type = int then int_add else real_add
exp2.env := exp1.env
exp3.env := exp1.env
```

Condition:

```
coercible(exp2.type, exp1.type) and
coercible(exp3.type, exp1.type)
```

Grammar rule: $exp \rightarrow ident$

Attribution Rules:

```
exp.type :=
  lookup(ident.symbol, exp.env)
```

Type Checking versus Type Inference

Note how in the above grammar the type of the operations is decided from the types of the operands: for a simple expression the information on the types flows in principle only in one direction (as long as the environment is only used for lookup).

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 'a 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).

- Type abstraction happens at binding time
- Type application happens when a function is applied

We introduce a language ML_0 to discuss type inference. It represents the core of ML-like languages.

$$\begin{aligned} expr &\rightarrow var \\ expr &\rightarrow fn\ var_1 \dots var_n \Rightarrow expr \\ expr &\rightarrow expr(expr_1 \dots expr_n) \\ expr &\rightarrow if\ expr\ then\ expr\ else\ expr \\ expr &\rightarrow let\ var = expr\ in\ expr \\ var &\rightarrow x \mid \dots \mid z \mid \dots \end{aligned}$$

Note that all expressions are implicitly typed.

Type inference in implicitly typed languages is also called *type reconstruction*.

Type Schemas

To represent (polymorphic) types we use type schemas. We have

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

A grammar for type schemas:

$$\begin{aligned} type &\rightarrow typevar \mid bool \mid \dots \mid int \\ type &\rightarrow typeconstructor(type_1, \dots, type_n) \\ schema &\rightarrow type \mid \forall x.type \end{aligned}$$

These are the normal ML types, for example:

$$int, bool, real, 'a, 'b, \mapsto (int, \mapsto (int, real)), \mapsto (list('a), list('a)).$$

Note

- for technical convenience we use an alternative notation for type constructors, i.e. we write them like function applications. Example: $list(int, real) = (int * real)\ list$
- types like $list('a)$ are implicitly quantified ($'a$ is quantified over all possible 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 $int\ list <: \forall 'a.\ 'a\ list$.

Judgement

We also need the idea of type environments and judgements.

- A type environment Γ contains all the type bindings
- A judgement on an expression e , written $\Gamma \vdash e : t$ gives the expression e the type t in environment Γ .

Γ is a collection of type bindings $x : t$.

The judgement is the most fundamental type inference justified by the rule:

$$H, x : t, H' \vdash x : t$$

This means that we can make the judgement $\Gamma \vdash e : t$ if the environment contains the binding t for the type variable x .

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:

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

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

$$\frac{\Gamma = H, x : T, H' \quad 'a <: T}{\Gamma \vdash x : 'a} \text{ (Var)}$$

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

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

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

$$\frac{\Gamma \vdash \hat{e} : '\hat{a} \quad \Gamma, (x : \forall 'a_1, \dots, 'a_n. '\hat{a}) \vdash e : 'a \quad \{\ 'a_1, \dots, 'a_n \} = \text{fv}(\hat{e}) - \text{fv}(\Gamma)}{\Gamma \vdash (\text{let } x = \hat{e} \text{ in } e) : 'a} \text{ (Let)}$$

Note:

- The name “Lambda” for function abstraction stems from the roots of functional languages in the Λ -calculus.
- $x_i \mapsto 'a_i$ in (Lambda) is implicitly assumed to be a universal closure: $x_i \mapsto \forall 'a_i$.

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 here is the *substitution*. It means that we replace a type variable with a type schema (i.e. a concrete type, a type variable or a polymorphic type).

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:

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

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

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

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

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

Finding the right substitutions

When applying a type rule perform the following steps.

Example: Infer the type of function application $e_1(e_2)$

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

- denote the types of all expressions by fresh type variables $\Gamma = e_1 : 'b, e_2 : 'c$.
- the rule application dictates that we must have $\Theta 'b = 'b_1 \mapsto 'b_2$, $\Theta 'a_1 = \Theta 'b_1$, $\Theta 'a = \Theta 'b_2$.

Substitution Example

```
val f = fn (x, y) => y+1
val g = fn y => f (0, y)
```

```
f (0, y)
```

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

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 ('a, 'b) f = fn : 'a * 'b -> ('a * 'b) list
- val g = fn x => f(1,x);
> val 'a 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).

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 .

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 most general unifier for $'a$ and $'b$ is a substitution Θ such

- $\Theta('a) = \Theta('b)$
- $\neg \exists \Theta'. \Theta'('a) = \Theta'('b)$ and Θ' is more general than Θ .

Unification Algorithm

The following algorithm constructs the most general unifier.

Unify(C): C is a set of type equations, S a substitution

Initialize S to empty

While C is not empty

 select type equation c from C

 if c is of the form $X=X$ then remove c from C

 (* identical basic types *)

 elseif c is of the form (* type constructor *)

$f(X_1, \dots, X_n) = g(Y_1, \dots, Y_m)$

 then return false if not($f=g$) or not($m=n$)

 elseif c is of the (* type constructor *)

 form $f(X_1, \dots, X_n) = g(Y_1, \dots, Y_n)$

 then replace c in C with $X_1=Y_1, \dots, X_n=Y_n$

 elseif c is of the form expression=X

 and X is a type variable then

 replace c in C with $X=expression$

 elseif c is of the form $X=expression$ and

 X is a type variable then

 if term contains X then return false

 else (1) remove c from C,

 (2) replace X with expression
 throughout C and S

 (3) add (expression/X) to S

 endif

 endif

endwhile

return S

Note that a recursive definition is not allowed. For example, $'a = 'a \text{ list}$ cannot be unified.

Unification Examples

- $int = int$?
- $real = 'a$?
- $real = list(real)$?
- $list(real) = list('a)$?
- $list(real) = list(int)$?
- $\mapsto (int, list(real)) = \mapsto (int, real)$?
- $type1(int, real, 'a) = type2('b, 'c, 'b)$?
- $type1(int, real, 'a) = type1('b, 'b, 'b)$?
- $type1(f(type2), h(type3)) = type1(f('a), 'b))$?
- $'a = g('a)$?
- $g('a, 'b) = g(g('a), int)$?

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

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

we obtain

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

Extending Type Rules with Unification (Lambda)

From

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

we obtain

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

Extending Type Rules with Unification (Apply)

From

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

we obtain

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

Extending Type Rules with Unification (Let)

From

$$\frac{\begin{array}{l} \Gamma \vdash \hat{e} : 'a \\ \Gamma, x : \forall 'a_1, \dots, 'a_n. 'a \vdash e : 'a \\ \{ 'a_1, \dots, 'a_n \} = fv('a) - fv(\Gamma) \end{array}}{\Gamma \vdash (let\ x = \hat{e}\ in\ e) : 'a} \text{ (Let)}$$

we obtain

$$\frac{\begin{array}{l} \Theta' \Gamma \vdash \hat{e} : 'a \\ \Theta((\Theta' \Gamma), x : \forall 'a_1, \dots, 'a_n. 'a') \vdash e : 'a \\ \{ 'a_1, \dots, 'a_n \} = fv('a) - fv(\Theta' \Gamma) \end{array}}{(\Theta \circ \Theta') \Gamma \vdash (let\ x = \hat{e}\ in\ e) : 'a}$$

Note: $fv(e)$ are the free variables in expression e .

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 $\Theta\Gamma$

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 'a 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 $\Theta = \{ 'e \mapsto 'f / 'a \}$

$([], y)$ yields $\Theta = \{ 'g \text{ list} * 'd / 'e \}$

$\Rightarrow y$ yields $\Theta = \{ 'd / 'f \}$

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

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

Composing all the above unifiers we obtain the parametric type

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

ML₀ versus ML

We have used ML₀ to illustrate the core concepts of ML

- variables
- function definition
- function application
- local variables with let polymorphism

Like for the “if-then-else” construction, Hindley-Milner type inference can easily be extended to many constructions of full ML. However, some extensions of real ML pose severe problems.

Consider, for example, the **ref** values. Consider the program

```
(1)  val r = ref (fn x => x)
(2)  r := (fn x => 1 + x)
(3)  val b = !r false
```

According to the discussion above r would have the polymorphic type $\forall 'a . \text{Ref}('a \rightarrow 'a)$ in line (1), then $\text{int} \mapsto \text{int}$ in line (2), and finally we would apply a function of type $\text{int} \mapsto \text{int}$ to an argument of type bool .

This problem stems essentially from having a quantified type expression at the top level. SML prohibits this (and related) problematic uses of polymorphism. This is called the *value restriction*. Observe the following behaviour:

```

- val x = ref (fn x => x);
stdIn:17.1-17.24 Warning: type vars not generalized
  because of value restriction are instantiated to
  dummy types (X1,X2,...)
val x = ref fn : (?X1 -> ?X1) ref

```

Other Implementations may exhibit slightly different behaviour:

```

      Moscow ML version 2.00 (June 2000)
- val x = ref (fn x => x);
! Warning: Value polymorphism:
! Free type variable(s) at top level in
  value identifier x
> val x = ref fn : ('a -> 'a) ref
- x := (fn x => 1 + x);
! Warning: the free type variable 'a has been
  instantiated to int
> val it = () : unit
- !x false;
! Toplevel input:
! !x false;
!      ^^^^^
! Type clash: expression of type
!   bool
! cannot have type
!   int
- !x 5;
> val it = 6 : int

```

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;
```