Some FP lecture notes (v0.3)

Tomáš Křen

January 8, 2018

Contents

Chapter 1

Hindley-Milner type system

Still work in progress... but the sections about Hindley-Milner algorithm W are almost OK now.

1.1 Introduction

Why Hindley-Milner (= simplified system F capable of type inference in curry style)

1.2 Language of type expressions

Explain simple types and type schemes.

1.3 Type substitutions

definition ; informally explain: makes a type more specific ; as function ; can be composed by $\circ$

1.4 Contexts

Definition. A *term* : *type* statement $M : \tau$ states that (program) term M has type τ. A *declaration* is a statement $s : \tau$ where s is a term symbol and τ is a type. A *context* is set of declarations with distinct term symbols.[1]

building symbols
def Γ_x ; def $\overline{\Gamma}(\tau)$

1.5 Inference rules

`TAUT` rule:

[1] Interestingly, the definition of a *context* and definition of a *substitution* are almost the same. The difference is that "keys" in a context are term symbols/variables, whereas substitution "keys" are type variables. Maybe this fact could be utilized in an interesting way...

$$\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma}$$

`COMB` rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2}$$

`ABS` rule:

$$\frac{\Gamma_x, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x.e) : \tau_1 \to \tau_2}$$

`LET` rule:

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma_x, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash (\mathtt{let}\, x = e_1\, \mathtt{in}\, e_2) : \tau}$$

`INST` rule:

$$\frac{\Gamma \vdash e : \sigma \qquad \sigma \sqsupseteq \sigma'}{\Gamma \vdash e : \sigma'}$$

`GEN` rule:

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \mathrm{FTV}(\Gamma)}{\Gamma \vdash e : \forall \alpha.\sigma}$$

$\sqsupseteq$ rule:

$$\frac{\beta_i \notin \mathrm{FTV}(\forall \overline{\alpha}.\tau) \qquad \tau' = \{\overline{\alpha} \mapsto \overline{\tau}\}(\tau)}{\forall \overline{\alpha}.\tau \;\sqsupseteq\; \forall \overline{\beta}.\tau'}$$

$\sqsupseteq$ rule, hopefully more readable:

$$\frac{\beta_i \notin \mathrm{FTV}(\forall \alpha_1 \ldots \alpha_n.\tau) \text{ for } i \in \{1, \ldots, k\} \qquad \tau' = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}(\tau) \qquad n, k \geq 0}{\forall \alpha_1 \ldots \alpha_n.\tau \;\sqsupseteq\; \forall \beta_1 \ldots \beta_k.\tau'}$$

1.6 Hindley-Milner algorithm W

The Hindley-Milner algorithm **W** is used for type inference. Loosely speaking, we give to **W** as an input a *program expression* e without type information and it returns a *type* τ of that expression as a result, or it tells us that the expression cannot be typed correctly.

From this simplified point of view we may see the algorithm usage as:
(1) We have an expression e, for which we would like to know the type.
So we run **W** on e and we may either get as a result:
(2a) a type τ, so we know that e has type τ,
(2b) or the *fail result* $\perp$ (usually called *bottom*), so we know there is a type error inside e.

The first simplification of this description lies in that we have omitted the typing contexts (the "Gammas"). All the inference rules deal with *judgments* of the form:

$$\Gamma \vdash e : \tau$$

And so does the **W** algorithm. If e is the top-level program expression, we can think of a context Γ as a collection of type information about the "library" in which the program expression e is written. Or, if e is some local sub-expression, then its Γ contains also type information about all the local variables defined in its scope.

We can think of a judgment of the form $\Gamma \vdash e : \tau$ as: *From the building symbols described in the typing context Γ we can build a well-typed program expression e which has type τ.* Therefore it makes sense to provide a typing context Γ to the **W** algorithm as another argument: $\mathbf{W}(\Gamma, e)$.

But **W** algorithm is even stronger: We may use libraries for which we do not know the proper typing information yet.

For example consider the following expression e:

$$\lambda x.((+\ ((+\ x)\ 1))\ x)$$

Or, in a more readable fashion, $e = \lambda x.(x + 1) + x$.

And let's pretend that the only thing we know is that $1 : \mathrm{Int}$, but we don't know the type of $+$. **W** can deal with this situation and infer that e has type $Int \to Int$ and that $+$ has type $Int \to Int \to Int$. This can be achieved by calling **W** with typing context $\Gamma = \{1 : \mathrm{Int}, + : \alpha\}$, where α is a *type variable.*

But if the only result of the $\mathbf{W}(\Gamma, e)$ is the type τ of e, how we get the information about the inferred type of $+$? Well, **W** actually returns a pair (S, τ), where S is a substitution containing the rest of the inferred type information. More specifically, $S(\alpha) = Int \to Int \to Int$.

Now we can state the behavior of the **W** algorithm more formally:

Given context Γ and expression e the Hindley-Milner algorithm **W** is looking for the substitution S and type τ such that:

$$S(\Gamma) \vdash e : \tau$$

If there are no such S and τ, then the **W** algorithm fails. But if there are any, **W** finds the most general S a τ.

$$\mathbf{W}(\Gamma, e) = \begin{cases} (S, \tau) & \textbf{if} \text{ there is any } S' \text{ and } \tau' \text{ such that } S'(\Gamma) \vdash e : \tau' \\ \bot & \textbf{otherwise} \end{cases}$$

Definition of W algorithm

Here we present a recursive definition of **W** algorithm based on case analysis of all possible patterns that program expression e may have (i.e. e may be a *variable*, an *application*, an *abstraction*, or a *let-expression*).

need to comment the need for fresh type variables βs

need to comment that when sub call fail, then also the calling computation fails

(1) Expression e is a *variable*; $e = x$:

$$\mathbf{W}(\Gamma, x) \coloneqq \begin{cases} (\{\}, R(\tau')) & \textbf{if } (x : \forall \alpha_1 \ldots \alpha_n . \tau') \in \Gamma \\ \bot & \textbf{otherwise} \end{cases}$$
$$\textbf{where } R = \{\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n\}$$

(2) Expression e is an *application*; $e = (e_1 \ e_2)$:

$$\mathbf{W}(\Gamma, (e_1 \ e_2)) \coloneqq \begin{cases} (R \circ S_2 \circ S_1, R(\beta)) & \textbf{if } R \neq \bot \\ \bot & \textbf{if } R = \bot \end{cases}$$
$$\begin{aligned} \textbf{where } & (S_1, \tau_1) = \mathbf{W}(\Gamma, e_1), \\ & (S_2, \tau_2) = \mathbf{W}(S_1(\Gamma), e_2), \\ & R = \mathbf{MGU}(S_2(\tau_1), \tau_2 \to \beta). \end{aligned}$$

(3) Expression e is an *abstraction*; $e = \lambda x.e_1$:

$$\begin{aligned} \mathbf{W}(\Gamma, \lambda x.e_1) \coloneqq \ & (S_1, S_1(\beta) \to \tau_1) \\ & \textbf{where } (S_1, \tau_1) = \mathbf{W}(\Gamma_x, x : \beta \ ; \ e_1). \end{aligned}$$

(4) Expression e is a *let-expression*; $e = (\texttt{let}\, x = e_1 \,\texttt{in}\, e_2)$:

$$\begin{aligned} \mathbf{W}(\Gamma, \texttt{let}\, x = e_1 \,\texttt{in}\, e_2) \coloneqq \ & (S_2 \circ S_1, \tau_2) \\ \textbf{where } & (S_1, \tau_1) = \mathbf{W}(\Gamma, e_1), \\ & (S_2, \tau_2) = \mathbf{W}(S_1(\Gamma_x), x : \overline{(S_1(\Gamma))}(\tau_1); e_2). \end{aligned}$$

Example run of W algorithm

We demonstrate **W** algorithm on simple example which contains all four possible forms of expressions as sub-expressions:

$$\texttt{let}\, x = \lambda x.x \,\texttt{in}\, f \ f$$

All the contained program variables (f and x) are locally defined variables, so we don't need to provide any further type information, therefore we call **W** with an empty typing context $\Gamma = \emptyset$.

$$\mathbf{W}(\emptyset, \texttt{let}\, f = \lambda x.x \,\texttt{in}\, f \ f)$$

The expression matches the case **(4)**:

$$\begin{aligned}\mathbf{W}(\emptyset, \texttt{let}\, f = \lambda x.x \,\texttt{in}\, f\ f) \coloneqq\ & (S_2 \circ S_1, \tau_2) \\ \textbf{where}\ & (S_1, \tau_1) = \mathbf{W}(\emptyset, \lambda x.x), \\ & (S_2, \tau_2) = \mathbf{W}(S_1(\emptyset_f), f : \overline{(S_1(\emptyset))}(\tau_1); f\ f).\end{aligned}$$

So we need to first compute the type (and substitution) of the $e_1 = \lambda x.x$, matching the case **(3)**:

$$\begin{aligned}\mathbf{W}(\emptyset, \lambda x.x) \coloneqq\ & (S_1, S_1(\beta_1) \to \tau_1) \\ \textbf{where}\ & (S_1, \tau_1) = \mathbf{W}(\{x : \beta_1\}\ ,\ x).\end{aligned}$$

Finally we get to the first variable (case **(1)**), thus we will get our first result.

$$\begin{aligned}\mathbf{W}(\{x : \beta_1\}\ ,\ x) \coloneqq\ & (\{\}, R(\beta_1))\,\textbf{where}\ \ R = \{\} \\ =\ & (\{\}, \beta_1)\end{aligned}$$

Because x has type β_1 in the context $\{x : \beta_1\}$, and β_1 has no universally quantified prefix head, the substitution R is empty, therefore identity. With this information we can get back to computation of $\mathbf{W}(\emptyset, \lambda x.x)$ and finish it.

$$\begin{aligned}\mathbf{W}(\emptyset, \lambda x.x) \coloneqq\ & (S_1, S_1(\beta_1) \to \tau_1)\,\textbf{where}\ (S_1, \tau_1) = (\{\}, \beta_1) \\ =\ & (\{\}, \{\}(\beta_1) \to \beta_1) = (\{\}, \beta_1 \to \beta_1)\end{aligned}$$

By this we have finished the first recursive call in computation of $\mathbf{W}(\emptyset, \texttt{let}\, f = \lambda x.x \,\texttt{in}\, f\ f)$. So we can compute the second recursive call:

$$\begin{aligned}&\mathbf{W}(S_1(\emptyset_f), f : \overline{(S_1(\emptyset))}(\tau_1); f\ f)\,\textbf{where}\ (S_1, \tau_1) = (\{\}, \beta_1 \to \beta_1) \\ =\,&\mathbf{W}(\{\}(\emptyset_f), f : \overline{(\{\}(\emptyset))}(\beta_1 \to \beta_1); f\ f) \\ =\,&\mathbf{W}(\{f : \forall\beta_1.\beta_1 \to \beta_1\}; f\ f)\end{aligned}$$

Now wee need to compute the type of expression $(f\ f)$ which is an application (case **(2)**) from typing context $\{f : \forall\beta_1.\beta_1 \to \beta_1\}$, specifying that f has type of the *polymorphic* identity.

$$\begin{aligned}\mathbf{W}(\{f : \forall\beta_1.\beta_1 \to \beta_1\}; f\ f) \coloneqq\ & (R \circ S_2 \circ S_1, R(\beta_?)) \\ \textbf{where}\ & (S_1, \tau_1) = \mathbf{W}(\{f : \forall\beta_1.\beta_1 \to \beta_1\}, f), \\ & (S_2, \tau_2) = \mathbf{W}(S_1(\{f : \forall\beta_1.\beta_1 \to \beta_1\}), f), \\ & R = \mathbf{MGU}(S_2(\tau_1), \tau_2 \to \beta_?).\end{aligned}$$

You can see $\beta_?$ which is used to signify that it is not obvious what index the new fresh variable will have, since the two recursive calls to $\mathbf{W}$ may produce some new fresh variables before $\beta_?$ is introduced. Actually both calls produce one new type variable, thus $\beta_?$ will be β_4, as we will see.

$$\begin{aligned}\mathbf{W}(\{f : \forall\beta_1.\beta_1 \to \beta_1\}, f) \coloneqq\ & (\{\}, R(\beta_1 \to \beta_1))\ \ \textbf{where}\ \ R = \{\beta_1 \mapsto \beta_2\} \\ =\ & (\{\}, \beta_2 \to \beta_2)\end{aligned}$$

Now we can continue with the second call:

$$\begin{aligned}\mathbf{W}(\{\}(\{f:\forall\beta_1.\beta_1\to\beta_1\}),f) \coloneqq\ & (\{\},R(\beta_1\to\beta_1))\ \textbf{where}\ \ R=\{\beta_1\mapsto\beta_3\}\\ =\ & (\{\},\beta_3\to\beta_3)\end{aligned}$$

And finally we compute the *most general unification* R:

$$\begin{aligned}R &= \mathbf{MGU}(\beta_2\to\beta_2,(\beta_3\to\beta_3)\to\beta_4)\\ &= \{\beta_2\mapsto(\beta_3\to\beta_3),\ \beta_4\mapsto(\beta_3\to\beta_3)\}\end{aligned}$$

One can see that R really unifies $\beta_2\to\beta_2$ and $(\beta_3\to\beta_3)\to\beta_4$, because $R(\beta_2\to\beta_2)=(\beta_3\to\beta_3)\to(\beta_3\to\beta_3)=R((\beta_3\to\beta_3)\to\beta_4)$. Now the computation of $\mathbf{W}(\{f:\forall\beta_1.\beta_1\to\beta_1\};f\ f)$ can be finished:

$$\begin{aligned}\mathbf{W}(\{f:\forall\beta_1.\beta_1\to\beta_1\};f\ f) \coloneqq\ & (R\circ S_2\circ S_1,R(\beta_4))\\ =\ & (\{\beta_2\mapsto(\beta_3\to\beta_3),\beta_4\mapsto(\beta_3\to\beta_3)\}\circ\{\}\circ\{\},\ \beta_3\to\beta_3)\\ =\ & (\{\beta_2\mapsto(\beta_3\to\beta_3),\beta_4\mapsto(\beta_3\to\beta_3)\},\ \beta_3\to\beta_3)\end{aligned}$$

Now we can compute the final result:

$$\begin{aligned}\mathbf{W}(\emptyset,\texttt{let}\,f=\lambda x.x\,\texttt{in}\,f\ f) =\ & (S_2\circ S_1,\tau_2)\\ \textbf{where}\ & (S_1,\tau_1)=(\{\},\beta_1\to\beta_1),\\ & (S_2,\tau_2)=(\{\beta_2\mapsto(\beta_3\to\beta_3),\beta_4\mapsto(\beta_3\to\beta_3)\},\ \beta_3\to\beta_3).\end{aligned}$$

And therefore:

$$\mathbf{W}(\emptyset,\texttt{let}\,f=\lambda x.x\,\texttt{in}\,f\ f) =\ (\{\beta_2\mapsto(\beta_3\to\beta_3),\beta_4\mapsto(\beta_3\to\beta_3)\},\ \beta_3\to\beta_3)$$

We get an unsurprising result:

$$\emptyset\vdash(\texttt{let}\,f=\lambda x.x\,\texttt{in}\,f\ f):\beta_3\to\beta_3$$

todo

Correctness and completeness of W

Correctness of W

If $\mathbf{W}(\Gamma,e)=(S,\tau)$, then exist derivation of the judgment $S(\Gamma)\vdash e:\tau$.

Completeness of W

Let Γ be a context and e a program expression, and let S' be a substitution and τ' a type such that: $S'(\Gamma)\vdash e:\tau'$, then:

(1) $\mathbf{W}(\Gamma,e)$ succeeds (i.e. $\mathbf{W}(\Gamma,e)\neq\bot$), let $\mathbf{W}(\Gamma,e)=(S,\tau)$,

(2) there is a substitution R such that $S'=R\circ S$ and $\overline{R(S(\Gamma))}(\tau)\sqsupseteq\tau'$.

Algorithm 1: Algorithm finding the most general unification.

```
function MGU(τ1, τ2)
    result = {}
    agenda ← [(τ1, τ2)]
    isOK ← True

    while agenda not empty ∧ isOK do
        (τa, τb) ← agenda.removeFirst()
        isOK = process(τa, τb, agenda, result)

    if isOK then
        return result
    else
        return ⊥
```

The correctness theorem states that if **W** finds a solution, then the solution is correct. The first part of the completeness theorem, states that if there is a solution, then **W** finds one. And the second part formally states that the found solution (S, τ) is the most general one, by comparing it with an arbitrary solution (S', τ'). The substitution R acts as a witness of the fact that we can obtain S' by making S more specific ($S' = R \circ S$).

Než bude možno vysvětlit druhou část dvojky, je třeba zavést ten closure overline někde vejš

1.7 Unification algorithm

todo: some comment

Algorithm 2: Processes one type pair.

```
function process(τ1, τ2, agenda, result)
    if τ1 and τ2 are the same TypeVar then
        return True

    if τ1 and τ2 are the same TypeSym then
        return True

    if τ1 and τ2 are both TypeTerm with the same length then
        agenda.addAll(zip(τ1.args(), τ2.args()))
        return True

    if τ1 is a TypeVar then
        return processTypeVar(τ1, τ2, agenda, result)

    if τ2 is a TypeVar then
        return processTypeVar(τ2, τ1, agenda, result)

    return False
```

Algorithm 3: Processes one $var \mapsto type$ binding.

```
function processTypeVar(var, type, agenda, result)
    if type contains var then
        return False

    S ← {var ↦ type}

    for entry in result do
        (v ↦ τ) ← entry
        entry.set_τ(S(τ))

    for entry in agenda do
        (τ1, τ2) ← entry
        entry.set_τ1(S(τ1))
        entry.set_τ2(S(τ2))

    result.add(var ↦ type)

    return True
```