Some FP lecture notes (v0.6)

Tomáš Křen

January 11, 2018

Contents

Chapter 1

Hindley-Milner Type System

The text is not completely finished, but hopefully the sections dealing with the Hindley-Milner algorithm **W** are sufficiently understandable.

TODO list

- Define rest of the context stuff
- Textually describe unification algorithm
- In W comment for each case what is going on.
- write Intro

1.1 Introduction

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

1.2 Type Language

TODO probably replace simple-type with mono-type

A type system connects a type language with a program language. In this section we present the type language (i.e. what are legal type expressions and what are they supposed to mean) used in Hindley-Milner type system.

We start with a subset of the language, *simple-types*. The whole language is obtained by enriching the simple-types with polymorphism.

Language of simple-types consists of tree type constructs, a simple-type is either: a type *symbol*, a type *term*, or a type *variable*. Let us have a closer look on each of the constructs.

A *type symbol* is a symbol of a specific basic type, such as Int, $Bool$, $String$, or $\mathbb{R}$ (which are all inhabited by values); or a symbol of a specific parametric type such as $List$ or $\rightarrow$ which are not inhabited. These uninhabited symbols serve as "functions over types" and need one or more types as type parameters. For example, in $(List\ Int)$ or $(Int \rightarrow Bool)$, we need to know what elements

does the list have, or what is the function's domain and co-domain. Type symbol is written as a single capitalized word.

A *type term* is a construct for creating compound types given by a sequence of types, e.g., $(List\ Int)$, or $(Int \to Bool)$. Type term construct is a basis for construction of *parametric* types, which are generally written as a parenthesized sequence of type expressions, i.e. $(\tau_1\ \tau_2\ \ldots\ \tau_n)$, where τ_i are arbitrary simple-type expressions.

A *type variable* is a construct representing a type which has not been fully specified yet. For instance, α, β_1 and β_2 are type variables. Type variables form *polymorphic* types, e.g. $(List\ \alpha)$, standing for a list of every (not yet specified) type. For instance, this list is inhabited by an empty list, $[\ ] : (List\ \alpha)$. Other example of polymorphic type includes $(\alpha \to \alpha)$ which stands for an unary operation over every type, for instance inhabited by an identity function $id : \alpha \to \alpha$. But not all polymorphic types are inhabited, e.g., there is no reasonable program having type $(\alpha \to \beta)$ since there is no function such that it is from any type to any other type. In this text, type variables are written as Greek letters from the beginning of the alphabet (α and β with possible index, e.g. β_{42}); in Haskell a type variable is written as a single lowercase word, usually one character long.

We get the full type language by adding *poly-types* (from *polymorphic*). Poly-types are created by use of the fourth type construct: the universal quantification using $\forall$.

More specifically we can take any type expression σ (simple-type or poly-type) and make it a poly-type by prefixing it with a quantification of one type variable (which usually appears inside σ). If the quantified type variable is α, then the newly created poly-type expression is $\forall\alpha.\sigma$. All the occurrences of α inside $\forall\alpha.\sigma$ are said to be *bound*, as opposed to *free*.

Here is a subtle and confusing yet very important difference between the way how polymorphic types are written in Haskell type language and in Hindley-Milner type language, simply said:

In Haskell all the type variables in polymorphic types are implicitly quantified.

For example the `map` function has Haskell type `(a -> b) -> [a] -> [b]`, or we can pretend that it is `(a -> b) -> (List a) -> (List b)` to be compatible with our type term notation. And `map` in Haskell with this type is polymorphic in both its type variables `a` and `b`. So if we need to translate this Haskell type to Hindley-Milner type, we must prepend a quantification for each type variable occurring in the type:

$\forall a.(\forall b.(a \to b) \to (\text{List}\ a) \to (\text{List}\ b))$

Which is abbreviated (similarly as with lambda abstractions) as:

$\forall a\, b\,.\,(a \to b) \to (\text{List}\ a) \to (\text{List}\ b)$

Or even closer to our conventions as (because when something is bounded we can rename in the similar fashion as with lambda abstractions bounding program variables):

$\forall \alpha\, \beta\,.\,(\alpha \to \beta) \to (\text{List}\ \alpha) \to (\text{List}\ \beta)$

So what is the purpose of free type variables?

We may say that in Hindley-Milner algorithm the type variables are used for two different purposes, depending on whether they are free or bound by $\forall$ quantifier.

The bound variables act as we know them from Haskell, the tricky part are the free variables.

TODO specifiy the convetions regarding σ for poly-types (well more precisely any-type) and τ for mono-types

We demonstrate the difference on an example type $\tau = (\alpha \to \beta)$ and $\sigma = (\forall\alpha\beta.\alpha \to \beta)$:

- There is no program expression e such that $e : \forall\alpha\beta.\alpha \to \beta$, because it would be a polymorphic function magically converting values between every pair of types.
- On the other hand statement $e : \alpha \to \beta$ tells us only that e is some function, we don't know yet, which specific types will stand for α and β. It corresponds to a more intermediate result.

  TODO: formulate more precisely...

1.3 Type Substitutions

Generally speaking, a substitution is a function used for replacement of variables in an expression by some other expression. Substitution is used both in program expressions (e.g. it is the core of β-reduction) and type expressions. Here we will be dealing with substitutions on the type level.

A type substitution is a finite mapping from type variables to types. It is usually denoted as a collection of $key \mapsto value$ pairs. The general form is:

$$\{\alpha_1 \mapsto \tau_1, \alpha_2 \mapsto \tau_2, \ldots, \alpha_n \mapsto \tau_n\}$$

For example, let
$S = \{\alpha_3 \mapsto \text{Int}, \beta_3 \mapsto (\text{List } \alpha_6), \alpha_5 \mapsto \beta_1, \beta_{23} \mapsto (\alpha_3 \to \beta_1)\}$ and
$\tau = ((\alpha_3 \to \alpha_5) \to (\beta_1 \to (\text{List } \alpha_6)))$, then
$\tau' = S(\tau) = ((\text{Int} \to \beta_1) \to (\beta_1 \to (\text{List } \alpha_6)))$.

Generally, by applying a substitution S to type τ, we get a *more specific* type $\tau' = S(\tau)$.

A special (but often seen, e.g. in Hindley-Milner algorithm) case of substitution is an empty substitution, denoted as $\{\}$, having no effect when applied; i.e. $\{\}(\tau) = \tau$.

Because substitutions can be dealt with as with functions, we can compose them using composition operator $\circ$. Let $R = S_2 \circ S_1$, then $R(\tau) = (S_2 \circ S_1)(\tau) = S_2(S_1(\tau))$.

1.4 Typing 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]

[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

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

1.5 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 $1 : \text{Int}$, 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 $\Gamma = \{1 : \text{Int}, + : \alpha\}$, 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:

---

substitution "keys" are type variables. Maybe this fact could be utilized in an interesting way...

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.

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

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*).

Whenever there is a type variable β (or β_i) it is meant to be a *new fresh type variable*, that has not occurred anywhere in Γ or during the computation of some other sub-expression. Namely, fresh type variables β are introduced in cases **(1)**, **(2)** and **(3)**.

During the computation of cases **(2)**, **(3)** and **(4)** there is a recursive call to **W** which can possibly fail; whenever a recursive call fails, 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 **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:

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

1.6 Inference Rules

`TAUT` rule:

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

`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:

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

`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:

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

The same $\sqsupseteq$ rule again, hopefully more readable:

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

Correctness and Completeness of W

Correctness of W

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

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 ⊥
```

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 $R(\overline{S(\Gamma)})(\tau) \sqsupseteq \tau'$.

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 ($S' = R \circ S$).

Before it is possible to explain the second part of point two, it is necessary to introduce the "closure overline" somewhere above.

1.7 Unification Algorithm

todo: Add some explanation here!

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
```