Teorija programskih jezikov 2017–18

Note 5: Type Inference
Principal Type Schemes for LMH

Version of March 27, 2018

The MH fragment of Haskell requires every term declarations to be accompanied by an associated type declaration. This places the obligation on the programmer to determine the types required in declarations.

In fact the process of inferring such types can be fully automated. Thus it is unnecessary to impose this burden on the programmer. Indeed, full-scale functional programming languages, such as Haskell and OCaml, do automate this process of *type inference*, and the programmer is *not* required to specify types. (The programmer is nonetheless permitted to specify types, as this can be useful for many purposes, including improving the clarity of code.)

Today, we look at the type inference process. To do so, we consider a variant LMH of MH, in which programs are given by a sequence of term declarations *without* associated type declarations. The interpreter will automatically infer the type declarations as part of the process of static analysis. It will find such type declarations whenever they exist. Moreover, it will always find, in a precise sense possible, the *most general* type declarations.

1 LMH and type schemes

The following is an example of an LMH program.

```
z = 10 ;
neg b = if b then False else True ;
mnsdbl m n = m - n - n ;
once f x = f x ;
twice f x = f (f x) ;
```

This is just like an MH program, but without the type declarations.

Just like the MH typechecking interpreter, the LMH interpreter performs static analysis prior to execution in which it checks that the program is well-typed. In doing so it needs to find type declarations for the variables declared in the program. In other words, the interpreter needs to *infer types* for the variables.

In the case of the program above, the type inference process results in the following typings.

```
z :: Integer
neg :: Bool -> Bool
mnsdbl :: Integer -> Integer -> Integer
once :: (a1 -> a2) -> a1 -> a2
twice :: (a3 -> a3) -> a3 -> a3
```

The first three types here are as expected. The last two, however, are not MH types. They introduce a new phenomenon, the *type scheme* involving *type variables*.

In an MH program, the term declaration for `once` could be preceded by any of the following type declarations.

```
once :: (Integer -> Integer) -> Integer -> Integer
once :: (Bool -> Bool) -> Bool -> Bool
once :: (Integer -> Bool) -> Integer -> Bool
once :: ((Integer -> Integer) -> Integer) -> (Integer -> Integer) -> Integer
once :: ((Bool -> Bool) -> Bool -> Bool) -> (Bool -> Bool) -> Bool -> Bool
```

And so on. Observe that each of the types above can be obtained from the expression `(a1 -> a2) -> a1 -> a2` by substituting one MH type τ_1 for the *type variable* `a1`, and another τ_2 for `a2`. Indeed, every valid type for the function `once` can be obtained by such a substitution. Furthermore, every such substitution produces a valid type for `once`.

We call (the abstract syntax tree of) a type expression involving type variables a *type scheme*. The observations above assert that $(\mathtt{a1} \to \mathtt{a2}) \to \mathtt{a1} \to \mathtt{a2}$ is the *principal* (intuitively, most general) type scheme for `once`.

Consider now the case of `twice`. In this case the function `f` is applied both to `x` and to `f x`. So `twice` can be given the following types from the above list, but not the others.

```
twice :: (Integer -> Integer) -> Integer -> Integer
twice :: (Bool -> Bool) -> Bool -> Bool
twice :: ((Bool -> Bool) -> Bool -> Bool) -> (Bool -> Bool) -> Bool -> Bool
```

In this case, each type above can be obtained by substituting an MH type τ for the type variable `a3` in the type scheme (`a3` $\to$ `a3`) $\to$ `a3` $\to$ `a3`, and indeed every such substitution is a valid type for `twice`. Furthermore, every valid type for `twice` is obtained by such a substitution. So (`a3` $\to$ `a3`) $\to$ `a3` $\to$ `a3` is the principal type scheme for `twice`.

The use of type variables and type schemes also solves a problem that we encountered in Note 4. Whereas the (abstract syntax) expression

$$\lambda x\,.\,x$$

does not have a *unique type* it does have a *principal type scheme*, namely `a1` $\to$ `a1`. The LMH type inference mechanism will assign principal type schemes to expressions. Lambda abstractions cause no problems in this process. Accordingly we include lambda abstraction as an explicit LMH construct. Similarly, we also extend the LMH expression syntax with a `let` construct, whose type inference process also requires the use of type schemes.

The expression syntax of LMH is summarised below (omitting implementation details of concrete syntax, such as parentheses).

$$\begin{array}{rcl} \mathit{Exp} & ::= & \texttt{VAR} \mid \texttt{NUM} \mid \texttt{BOOLEAN} \\ & \mid & \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Exp}\ \texttt{else}\ \mathit{Exp} \\ & \mid & \mathit{Exp}\ \texttt{==}\ \mathit{Exp} \mid \mathit{Exp}\ \texttt{<}\ \mathit{Exp} \mid \mathit{Exp}\ \texttt{+}\ \mathit{Exp} \mid \mathit{Exp}\ \texttt{-}\ \mathit{Exp} \\ & \mid & \mathit{Exp}\ \mathit{Exp} \mid \backslash\texttt{VAR}\ \texttt{->}\ \mathit{Exp} \mid \texttt{let}\ \texttt{VAR}\ \texttt{=}\ \mathit{Exp}\ \texttt{in}\ \mathit{Exp} \end{array}$$

We correspondingly extend our existing mathematical notation for abstract syntax trees with let expressions as follows.

$$\begin{array}{rcl} \mathit{Exp} & ::= & x \mid n \mid b \\ & \mid & \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Exp}\ \texttt{else}\ \mathit{Exp} \\ & \mid & \mathit{Exp}\ \texttt{==}\ \mathit{Exp} \mid \mathit{Exp}\ \texttt{<}\ \mathit{Exp} \mid \mathit{Exp}\ \texttt{+}\mathit{Exp} \mid \mathit{Exp}\ \texttt{-}\ \mathit{Exp} \\ & \mid & \mathit{Exp}\ \mathit{Exp} \mid \lambda x\,.\,\mathit{Exp} \mid \texttt{let}\ x\ \texttt{=}\ \mathit{Exp}\ \texttt{in}\ \mathit{Exp} \end{array}$$

We need also to extend the operational semantics of MH to the full expression syntax of LMH. Lambda-expressions $\lambda x\,.\,e$ are already catered for in the operational semantics of MH. For `let` expressions, we implement an operational semantics that faithfully reflects the behaviour of Haskell, in which such expressions are potentially allowed to define the `let`-bound variable by recursion, as in the example below.

$$\begin{array}{l} \texttt{let fib = } \lambda x\,.\,\texttt{if}\ x\ \texttt{<}\ 2\ \texttt{then}\ 1\ \texttt{else fib}\ (x\ \texttt{-}1) + \texttt{fib}\ (x\ \texttt{-}2) \\ \quad \texttt{in fib}\ 10 \end{array}$$

The (big-step) operational semantics is extended with the following rule for evaluating such recursive let expressions.

$$\frac{e_2\,[\,x := (\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_1)\,]\ \Rightarrow\ v}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \Rightarrow\ v}$$

Note, in particular, the laziness in the above rule. The expression e_1 is evaluated only when x is required in the evaluation of e_2. Also, note that the recursive definition of x is dealt with in an indirect way, via use of the auxiliary expression $\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_1$.

In order to make full sense of the above extension of the operational semantics, it is necessary to define the set of free variables and the substitution operation on let expressions; i.e., we need to extend the existing definitions from Note 2 with clauses for $\mathsf{FV}(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2)$ and $(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2)\,[y := e]$. This is left as an **exercise**.

2 Principal type schemes

In Figure 1 we give an inference system for deriving *typing judgements* of the form

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

This is similar to Figure 1 in Note 4. The differences are that e is now an LMH expression, and $\sigma, \tau \ldots$ now range over type schemes, potentially containing type variables:

$$\mathit{TypeSch}\ ::=\ a \mid \texttt{Integer} \mid \texttt{Bool} \mid \mathit{TypeSch}\ \rightarrow\ \mathit{TypeSch}\ ,$$

where we use a as a meta-variable ranging over type variables. (We shall use the strings $\texttt{a0}$, $\texttt{a1}$, $\texttt{a2}$, ... as concrete type variables.) Similarly, Γ is now a *type-scheme environment*: a finite partial function mapping term variables to type schemes. Henceforth, for brevity, we shall normally talk about *types* and *type environments*, even when referring to type schemes. And we shall explicitly refer to *type schemes* only when we wish to especially emphasise the schematic nature.

Lemma 2.1 of Note 4 transfers verbatim to the present context.

Lemma 2.1 *If* $\Gamma \vdash e : \tau$ *and* $\Gamma, x\colon \tau \vdash e' : \tau'$ *then* $\Gamma \vdash e'[x := e] : \tau'$.

The proof is again by induction on the derivation of $\Gamma,\ x\colon \tau \vdash e' : \tau'$.

Our main concern today is with *type substitutions* rather than expression substitutions. A *(parallel) type substitution* is a finite partial function from type variables to type schemes, for example:

$$[\,\texttt{a1} := (\texttt{Integer} \rightarrow \texttt{a3})\,,\ \texttt{a3} := \texttt{Bool}\,]$$

$$\frac{}{\Gamma \vdash x : \tau}\ \Gamma(x) = \tau \qquad \frac{}{\Gamma \vdash n : \texttt{Integer}} \qquad \frac{}{\Gamma \vdash b : \texttt{Bool}}$$

$$\frac{\Gamma \vdash e_0 : \texttt{Bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Integer} \quad \Gamma \vdash e_2 : \texttt{Integer}}{\Gamma \vdash e_1 \star e_2 : \texttt{Bool}}\ \star \in \{\texttt{==}, \texttt{<}\} \qquad \frac{\Gamma \vdash e_1 : \texttt{Integer} \quad \Gamma \vdash e_2 : \texttt{Integer}}{\Gamma \vdash e_1 \star e_2 : \texttt{Integer}}\ \star \in \{\texttt{+}, \texttt{-}\}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \frac{\Gamma, x\colon \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda\, x\, .\ e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma, x\colon \tau_1 \vdash e_1 : \tau_1 \quad \Gamma, x\colon \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau_2}$$

Figure 1: Typing rules for LMH expressions

Every such substitution θ can be applied to an arbitrary type scheme τ to obtain another type scheme $\tau\theta$. For example, if θ is the substitution displayed above then

$$((\texttt{a1} \to \texttt{a2}) \to \texttt{a3})\,\theta \;=\; ((\texttt{Integer} \to \texttt{a3}) \to \texttt{a2}) \to \texttt{Bool})$$

Similarly, every such substitution can be applied to any type environment Γ to obtain a type environment $\Gamma\,\theta$. For example, if θ is again substitution displayed above then

$$\begin{aligned} &(\texttt{once} : (\texttt{a1} \to \texttt{a2}) \to \texttt{a1} \to \texttt{a2}\,,\, \texttt{twice} : (\texttt{a3} \to \texttt{a3}) \to \texttt{a3} \to \texttt{a3})\,\theta \\ &= \; \texttt{once} : ((\texttt{Integer} \to \texttt{a3}) \to \texttt{a2}) \to (\texttt{Integer} \to \texttt{a3}) \to \texttt{a2}\,, \\ &\quad\; \texttt{twice} : (\texttt{Bool} \to \texttt{Bool}) \to \texttt{Bool} \to \texttt{Bool} \end{aligned}$$

It is left as an **exercise** to give formal mathematical definitions of $\tau\theta$ and $\Gamma\theta$, for all types τ, type environments Γ, and substitutions θ.

Lemma 2.2 *If $\Gamma \vdash e : \tau$ then also $\Gamma\theta \vdash e : \tau\theta$, for all type substitutions θ.*

The proof is a straightforward induction on the derivation of $\Gamma \vdash e : \tau$.

Given two type substitutions θ_1 and θ_2, we write $\theta_1\,\theta_2$ for the *composite* substitution that first applies substitution θ_1 to a type, and then applies substitution τ_2 to the result of the first substitution. Thus $\tau\,\theta_1\,\theta_2$ can be understood either as $(\tau\,\theta_1)\,\theta_2$, or equivalently as $\tau\,(\theta_1\,\theta_2)$. It is left as an **exercise** to define formally the composite substitution $\theta_1\,\theta_2$.

Theorem 2.3 (Principal type scheme) *Let Γ be a type environment and e an LMH expression such that there exist a type substitution θ and type τ with $\Gamma\,\theta \vdash e : \tau$. Then there exist a type substitution θ_0 and type τ_0 enjoying the two properties below.*

1. $\Gamma\,\theta_0 \vdash e : \tau_0$.

2. *For any type substitution θ and type τ such that $\Gamma\,\theta \vdash e : \tau$, there exists a type substitution θ' such that $\theta = \theta_0\,\theta'$ and $\tau = \tau_0\,\theta'$.*

The type τ_0 is said to be the *principal type scheme* for e relative to Γ.

The main task in type inference for LMH is the computation of principal type schemes for expressions. In the next section, we outline the algorithm that does this.

3 Computing principal type schemes

Definition 3.1 Let τ_1, τ_2 be type schemes.

1. A substitution θ is said to *unify* τ_1 and τ_2 if $\tau_1\,\theta = \tau_2\,\theta$.

2. We say that τ_1 and τ_2 *unify* if there exists θ that unifies them.

3. We say that θ_0 is a *most general unifier* of τ_1 and τ_2 if:

   (a) θ_0 unifies τ_1 and τ_2 , and

   (b) for every unifier θ of τ_1 and τ_2, there exists a substitution θ' such that $\theta = \theta_0\,\theta'$.

Theorem 3.2 (Unification Theorem) *If τ_1 and τ_2 unify then they have a most general unifier θ. Moreover, θ can be chosen so that all type variables in its domain and range occur in τ_1 or τ_2.*

The Unification Theorem is due to Robinson who needed it for the correctness of his *resolution* theorem-proving method. Resolution makes use of an algorithm for computing most general unifiers. The type inference algorithm below exploits the same *unification algorithm*. In these notes, we shall not describe the unification algorithm in detail, but an implementation of it can be found in `LMH_TypeInference.hs`.

Type inference algorithm

The algorithm takes (Γ, e) as input, where Γ is a type environment and e an LMH expression. It either returns (θ_0, τ_0) as output, as in Theorem 2.3, or it fails if no such (θ_0, τ_0) exists.

Given (Γ, e) as input, the algorithm proceeds recursively on the structure of e. The underlined cases below, illustrate the algorithmic steps in a selection of cases for the structure of e.

$\underline{x}$**:** Return $([], \Gamma(x))$. (Here $[]$ is the empty substitution, which satisfies $\tau\,[] = \tau$ for every type scheme τ.)

$\underline{e_1\, e_2}$**:** Compute (θ_1, τ_1) for (Γ, e_1). Next compute (θ_2, τ_2) for $(\Gamma\,\theta_1, e_2)$. Let a be a fresh type variable. Let θ_3 be the most general unifier of $\tau_1\,\theta_2$ and $\tau_2 \to a$. Return $(\theta_1\,\theta_2\,\theta_3 \restriction_\Gamma,\; a\,\theta_3)$. (Here $\theta \restriction_\Gamma$ means the restriction of the domain of the substitution to type variables that appear in Γ.)

$\underline{\lambda\, x\, .\, e_0}$**:** Let a be a fresh type variable. Compute (θ, τ) for $((\Gamma,\; x\colon a),\; e_0)$. Return $(\theta \restriction_\Gamma,\; (a\,\theta) \to \tau)$.

If any of the individual steps in the above process fails, then the algorithm as a whole aborts and does not return an output. This amounts to the failure of type inference for e (relative to Γ).

4 Type inference in the LMH interpreter

The LMH interpreter `LMH_Interpreter` first performs type inference for the program loaded, in order to obtain a type environment T of principle type schemes for all variables declared in the program. It then again performs type inference on every expression e that the user enters for evaluation.

Type inference for an LMH program

An LMH program gives an environment E mapping the set *DeclVars* to expressions. Suppose that $\mathit{DeclVars} = \{x_1, \ldots, x_n\}$. The type environment T is calculated as follows.

- Start with the generic type environment T_0, which is defined to be

  $$x_1 : a_1 \,,\; \ldots \;,\; x_n : a_n$$

  where $a_1, \ldots, a_n$ are distinct type variables.

- For each i from 1 to n in turn, do the following
  - Find the principal (θ, τ) for (T_{i-1}, e_i).
  - Let θ' be the most general unifier of $T_{i-1}(x_i)\,\theta$ and τ.
  - Define T_i to be $T_{i-1}\,\theta\,\theta'$.
- If this succeeds then the program type-checks, the resulting type environment T of principal type schemes is defined to be T_n.

Type inference in the interpreter loop

The interpreter loop is entered after the program has been type-checked with principal type environment T. The loop proceeds as follows.

- The user inputs LMH expression e.
- LMH finds the principal (θ, τ) for (T, e) and prints τ.
- LMH evaluates e and prints the result.
- Return to start of loop.