4.2 The Type Inference Algorithm

Dienstag, 21. Juni 2016 15:30

We now present an algorithm $\mathcal{W}$ for type checking under a certain type assumption.

$\mathcal{W}(A, t)$ checks whether t is well typed under the type assumption A.

A: type assumption

t: λ-term

If yes, then $\mathcal{W}$ returns the result

$$(\Theta, \tau)$$

Θ: substitution of the free variables in A by types. This substitution is needed to make t well typed

τ: most general type of t under the type assumption $\underbrace{\Theta(A)}$

we also write $A\Theta$

We now introduce the algorithm $\mathcal{W}$ for the different forms of λ-terms t. (Slide 60)

4.2.1. Type Inference for Variables and Constants

Now t is $c \in \mathcal{C} \cup \mathcal{V}$

If the type assumption contains $c :: \forall a_1, \ldots, a_n . \tau$ where τ is a type (without "$\forall$"), then the most general type of c is τ. Here, we rename the type variables $a_1, \dots, a_n$ to fresh variables $b_1, \dots, b_n$ that do not occur free in A or τ.

To make c well typed, the current type assumption A does not need to be refined, i.e., Θ is the identity id.

$$\mathcal{W}(A_0, x) = (id, b) \text{ for } x \in \mathcal{V} \qquad A_0(x) = \forall a. a$$

$$\mathcal{W}(A_0, not) = (id, Bool \to Bool)$$

$$\mathcal{W}(A_0, Cons) = (id, c \to (List\ c) \to (List\ c)) \qquad A_0(Cons) = \forall a. a \to (List\ a) \to (List\ a)$$

General Rule for $c \in \mathcal{C} \cup \mathcal{V}$:

$$\mathcal{W}(A + \{c :: \forall a_1, \dots, a_n. \tau\}, c) = (id, \tau[a_1/b_1, \dots, a_n/b_n]),$$

where $b_1, \dots, b_n$ are fresh variables

4.2.2 Type Inference for Lambda Abstractions

Idea: To determine the type of $\lambda x. t$:

- assume that we know the type of x (i.e., x has some type b) $\leftarrow$ extend the previous type assumption by $\{x :: b\}$
- under this assumption, compute the type τ of t
  Moreover, we compute a subst. Θ which states how our type assumption

• Then the whole term has type $b\theta \to \tau$. must be refined.

General Rule for $\lambda x.t$:

$\mathcal{W}(A, \lambda x.t) = (\theta, b\theta \to \tau)$,

where $\mathcal{W}(A + \{x :: b\}, t) = (\theta, \tau)$,

b is a fresh type variable

We now illustrate this rule (and type checking for λ-abstractions) with several examples.

Ex: $\lambda f.\ \text{plus}\ (f\ \text{True})\ (f\ 3)$

plus: type $\text{Int} \to \text{Int} \to \text{Int}$; True: type Bool; 3: type Int

So f must have the type schema $\forall a.\ a \to \text{Int}$

Thus, the whole term has the type schema:

$(\forall a.\ a \to \text{Int}) \to \text{Int}$ ← this is the type of functions to Int where the argument is a fct. of type $a \to \text{Int}$

Which functions have the type schema $\forall a.\ a \to \text{Int}$?

$\lambda x. 1$, but not $\lambda x.x$, $\lambda x.\ x+1$, ...

So there is an important difference between the following type schemas:

$(\forall a.\ a \to Int) \to Int$

$\forall a.\ (a \to Int) \to Int$ ← If a fct. has this type, then it also has the types

$(Int \to Int) \to Int,\ (Bool \to Int) \to Int, \ldots$

E.g: $\lambda f.\ plus\ (f\ bot)\ (f\ bot)$

Type a

In Haskell, one is restricted to <u>shallow</u> type schemas.

<u>Def 4.2.1</u> (Shallow Type Schemas)

A type schema is <u>shallow</u> iff it has the form

$\forall a_1, \ldots, a_n.\ \tau$ where τ is a type (i.e., τ does not contain quantifiers).

Most prog. languages are restricted to shallow type schemes to make type checking decidable.

$\Rightarrow\ \lambda f.\ plus\ (f\ True)\ (f\ 3)$

is not well typed in Haskell.

(i.e., $\lambda x.t$)

Consequence: A variable bound by λ must have the same type for all (free) occurrences of x in t.

$\lambda f.\ plus\ (f\ True)\ (f\ 3)$ not well typed

let $f = \backslash x \to 1$ in $plus\ (f\ True)\ (f\ 3)$ is well typed

We now execute $\mathcal{W}$ for some examples:

Ex $\lambda x.\ Cons\ O\ (Cons\ x\ x)$

type $e \to List\ e \to List\ e$

is not well typed: x must get the same type everywhere in $Cons\ O\ (Cons\ x\ x)$

needs type Int

needs type List Int

To compute

$\mathcal{W}(A_0, \lambda x.\ Cons\ O\ (Cons\ x\ x))$, we need to compute

$\mathcal{W}(A_0 + \{x :: b\}, Cons\ O\ (Cons\ x\ x))$ which fails.

Note that we have $x :: b$, not $x :: \forall b.b$

Indeed, $\lambda x.\ Cons\ O\ (Cons\ x\ x)$ has the (non-shallow) type schema: $(\forall b.b) \to (List\ Int)$

Ex: $\mathcal{W}(A_0, \lambda x.\ tuple_2\ x\ x) = (id,\ b \to (b,b))$, since

$\mathcal{W}(A_0 + \{x :: b\}, \text{tuple}_2\ x\ x) = (id, (b, b))$

<u>Ex</u>: $\mathcal{W}(A_0, \lambda x.\,\text{plus}\ x\ x) = ([b/Int], \underbrace{b[b/Int]}_{Int} \to Int)$

$\mathcal{W}(A_0 + \{x :: b\}, \text{plus}\ x\ x) = ([b/Int], Int)$

$A_0(\text{plus}) = \underline{Int} \to \underline{Int} \to \underline{Int}$

The current type assumption must be refined (by instantiating b with Int) to make plus x x well-typed

4.2.3 Type Inference for Applications

$(t_1\ t_2)$: If t_1 has type τ_1

and t_2 has type τ_2,

then check whether τ_1 corresponds to $\tau_2 \to \tau_3$.

Then the result has type τ_3.

Ex: (not True): Here, "not" has type $\underbrace{Bool \to Bool}_{\tau_1}$

and "True" has type $\underbrace{Bool}_{\tau_2}$.

Thus, τ_1 corresponds to $\tau_2 \to \tau_3$ for $\tau_3 = Bool$. So term is well-typed and has type $\tau_3 = Bool$.

In general, "correspondence" of τ_1 and $\tau_2 \to \tau_3$ might involve instantiation of type variables.

Ex: Cons 0

Cons has type $\tau_1 = e \to List\ e \to List\ e$

0 has type $\tau_2 = Int$

Does τ_1 "correspond" to $\tau_2 \to \tau_3$ for some type τ_3?

This means:

Is there a substitution θ of type variables by types such that $\tau_1 \theta = (\tau_2 \to b)\theta$ holds for a fresh type variable b?

This means: We search for a <u>unifier</u> θ of

τ_1 and $\tau_2 \to b$.

In the example Cons 0:

τ_1 : $e \to List\ e \to List\ e$

τ_2 : Int

Unifier of τ_1 and $\tau_2 \to b$ is $\underbrace{[e/Int,\ b/List\ Int \to List\ Int]}_{\theta}$

Resulting type of Cons 0 is $b\theta = List\ Int \to List\ Int$.

<u>Def 4.22</u> (Unification)

Let θ be a substitution of type variables by types.

The subst. θ is a <u>unifier</u> of two types τ_1 and τ_2 iff $\tau_1\theta = \tau_2\theta$.

A subst. θ' is <u>most general unifier</u> (mgu) of τ_1 and τ_2 iff

- θ' is a unifier of τ_1 and τ_2
- for all unifiers θ of τ_1 and τ_2, there exists a subst. δ with $\theta = \theta'\delta$.

first apply the mgu θ', then apply a more special subst. δ.

We write $\theta' = mgu(\tau_1, \tau_2)$.

Ex: Cons bot

We have to unify $\tau_1 = e \to liste \to liste$
with $\tau_2 \to b = a \to b$

Possible unifiers: $\theta_1 = [a/e, b/liste \to liste]$. Resulting type of Cons bot is $b\,\theta_1 = liste \to liste$

mgu which leads to the most general type

$\theta_2 = [a/Int, e/Int, b/List\ Int \to List\ Int]$.
Resulting type of Cons bot is
$b\,\theta_2 = List\ Int \to List\ Int$

General Rule for type-checking applications $(t_1\ t_2)$:

$$\mathcal{W}(A, (t_1\ t_2)) = (\theta_1 \theta_2 \theta_3,\ b\theta_3),$$

where $\mathcal{W}(A, t_1) = (\theta_1, \tau_1)$ ← to make t_1 well typed, one has to refine A to $A\theta_1$. This should be taken into account when type-checking t_2

$\mathcal{W}(A\theta_1, t_2) = (\theta_2, \tau_2)$ ← to make t_2 well-typed, the type assumption must be refined further by θ_2. Then t_1 has the type $\tau_1\theta_2$.

$\theta_3 = mgu(\tau_1 \theta_2,\ \tau_2 \to b)$,

b is a fresh variable

Ex: $\lambda x.\ Cons\ x\ x$

$\mathcal{W}(A_0, \lambda x.\ Cons\ x\ x) =$

$\mathcal{W}(A_0 + \{x :: b\}, (Cons\ x)\ x) =$

$\mathcal{W}(A_0 + \{x :: b\}, Cons\ x) = ([b/e, b'/Liste \to Liste], Liste \to Liste)$

$\mathcal{W}(A_0 + \{x :: b\}, Cons) = (id, e \to Liste \to Liste)$

$\mathcal{W}(A_0 + \{x :: b\}, x) = (id, b)$

$mgu(e \to (Liste \to Liste), b \to b') = [b/e, b'/Liste \to Liste]$

$\mathcal{W}(\underbrace{(A_0 + \{x :: b\})[b/e, b'/\ldots]}_{A_0 + \{x :: e\}}, x) = (id, e)$

$mgu(Liste \to Liste,\ e \to b'')$ fails ("occur failure")

$\Rightarrow$ term is not well typed!

4.2.4 The Full Type Inference Algorithm

The algorithm $\mathcal{W}$ is a modified version of the $\mathcal{W}$-algorithm by R. Milner (1978).

<u>Thm 4.2.3</u> (Correctness of $\mathcal{W}$)

Let t be a λ-term over the constants $\mathcal{C}$ from Def 3.3.4 where $\mathcal{W}(A_0, t) = (\Theta, \tau)$.
Let $t \rightarrow_{\beta\delta}^{*} t'$ where δ is the Delta-Rule-Set for Haskell from Def 3.3.5. Then $\mathcal{W}(A_0, t') = (\Theta', \tau)$ for some Θ'.

This means: Static typechecking ensures that no type errors are introduced at runtime.