COMP3161/COMP9164 Supplementary Lecture Notes

Type Inference

Liam O'Connor

December 10, 2019

Explicitly typed polymorphic languages, where the user must make explicit type abstractions and applications, such as the version of MinHS introduced with parametric polymorphism, are very awkward to use in practice. Ideally, we would like to leave these type annotations *implicit*, and have the compiler infer types for us.

Considering the following expression,

$$\mathbf{recfun}\ f\ x = \mathsf{fst}\ x + 1$$

We could give a number of possible types to this expression:

- $(\texttt{Int} \times \texttt{Int}) \to \texttt{Int}$
- $(\texttt{Int} \times \texttt{Bool}) \to \texttt{Int}$
- $(\texttt{Int} \times \mathbf{0}) \to \texttt{Int}$

The exact type inferred must depend on the surrounding context, that is, the argument to which this function is applied. If the function is applied to many different arguments, then we would need to *generalise* the type to $\forall a.\ (\texttt{Int} \times a) \to \texttt{Int}$.

In order to make type annotations implicit, we will start with polymorphic MinHS but remove the following features:

- type signatures from **recfun**, **let**, etc.
- explicit **type** abstractions, and type applications (the @ operator).
- recursive types, because there is no unique most general type (*principal type*) for a given term if we have general recursive types.

Our types may still contain type variables quantified by the $\forall$ operator, however now the compiler, not the user, determines when to generalise and specialise types.

1 Implicitly-typed MinHS

The basic constructs of implicitly-typed MinHS are identical to explicitly-typed MinHS:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\textsc{Var} \qquad\qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\textsc{App}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e_3) : \tau}\textsc{If}$$

For simplicity, however, we will treat constructors and primitive operations as functions, whose types are available in the environment. Uses of these operations and constructors are then just function applications:

$$(+) : \texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int}, \Gamma \vdash (\texttt{App}\ (\texttt{App}\ (+)\ (\texttt{Num}\ 2))\ (\texttt{Num}\ 1)) : \texttt{Int}$$

Other functions are defined as usual with **recfun**, but now types are not mentioned in the term:

$$\frac{x : \tau_1, f : \tau_1 \rightarrow \tau_2, \Gamma \vdash e : \tau_2}{\Gamma \vdash (\texttt{Recfun}\ (f.x.\ e)) : \tau_1 \rightarrow \tau_2}\textsc{Func}$$

The two constructs for polymorphism, type abstraction (**type**) and application (the `@` operator), have now been removed. But, we still have the typing rules that allow us to specialise a polymorphic type (replacing `@`):

$$\frac{\Gamma \vdash e : \forall a. \tau}{\Gamma \vdash e : \tau[a := \rho]}\textsc{All}_\textsc{E}$$

And to quantify over any variable that has not already been used (replacing **type**)[1]:

$$\frac{\Gamma \vdash e : \tau \quad a \notin TV(\Gamma)}{\Gamma \vdash e : \forall a.\ \tau}\textsc{All}_\textsc{I}$$

2 An Algorithm

We want a fully deterministic algorithm for type inference, which has a clear input and output. We could imagine interpreting our existing rules as an algorithm, where the context and expression are the input and the type is the output:

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\textsc{App}$$

$$\texttt{infer} :: \texttt{Context} \rightarrow \texttt{Expr} \rightarrow \texttt{Type}$$

However this causes problems when we examine the rules for polymorphism ($\textsc{All}_E$ and $\textsc{All}_I$). Neither the rule to introduce nor the rule to eliminate $\forall$ quantifiers is syntax directed. They can be applied at any time. For example, our $\textsc{All}_I$ rule:

$$\frac{\Gamma \vdash e : \tau \quad a \notin TV(\Gamma)}{\Gamma \vdash e : \forall a.\ \tau}\textsc{All}_\textsc{I}$$

Because this rule works on any expression and context, we have an infinite number of possible types for every possible expression. `Num` 5 could be of type `Int` or $\forall a.$ `Int` or $\forall a.\ \forall b.$ `Int` etc.

In order to have an algorithmic set of rules, we need to fix not just *when* these rules are applied but also how they are applied. For example, the rule to specialise a polymorphic type replaces a quantified type variable with any type ρ, where this type is not able to be determined from the input context and expression:

$$\frac{\Gamma \vdash e : \forall a. \tau}{\Gamma \vdash e : \tau[a := \rho]}\textsc{All}_\textsc{E}$$

If the compiler makes the wrong decision when applying this rule, it can lead to typing errors even for well-typed programs:

$$\dfrac{\dfrac{\Gamma \vdash \mathsf{fst} : \forall a.\ \forall b.\ (a \times b) \rightarrow a}{\Gamma \vdash \mathsf{fst} : (\texttt{Bool} \times \texttt{Bool}) \rightarrow \texttt{Bool}} \quad \dfrac{\cdots}{\Gamma \vdash (\texttt{Pair}\ 1\ \mathsf{True}) : (\texttt{Int} \times \texttt{Bool})}}{\Gamma \vdash (\texttt{Apply}\ \mathsf{fst}\ (\texttt{Pair}\ 1\ \mathsf{True})) :\ ???}$$

In the above example, we instantiated the type variable a to `Bool`, even though the provided pair is actually `Int` $\times$ `Bool`.

[1]Where $TV(\Gamma)$ here is all type variables occurring free in the types of variables in Γ

The Solution

To start with, we will make two decisions:

1. $\forall$ quantified type variables will be instantiated to particular types as soon as a polymorphic type is found in the context for a particular term variable. That is, we shall merge the $\textsc{All}_E$ and $\textsc{Var}$ rules, and not have a separate $\textsc{All}_E$ rule.

2. $\forall$ quantifiers will only be introduced for the types of variables bound in **let** expressions. So, we will not have a separate $\textsc{All}_I$ rule either.

Leaving the second decision aside for a moment, we still have a problem with the first. We have fixed *when* the rule is applied but not *how*: If we instantiate each $\forall$-quantified variable to a particular type as soon as possible, we will not (yet) know what type to instantiate it to. For example, looking up the type of fst in the context gives us a type $\forall a.\ \forall b.\ (a \times b) \to a$, but we do not know at that point what a and b should be replaced with.

To resolve this, we allow types to include *unknowns*, also known as *unification variables* or *schematic variables*. These are placeholders for types that we haven't worked out yet. We shall use α, β etc. for these variables. For example, $(\mathtt{Int} \times \alpha) \to \beta$ is the type of a function from tuples where the left side of the tuple is $\mathtt{Int}$, but no other details of the type have been determined yet.

As we encounter situations where two types should be equal, we *unify* the two types to determine what the unknown variables should be, producing a substitution to these unknowns.

$$\dfrac{\dfrac{\Gamma \vdash \mathsf{fst} : \forall a.\ \forall b.\ (a \times b) \to a}{\Gamma \vdash \mathsf{fst} : (\alpha \times \beta) \to \alpha} \quad \dfrac{\vdots}{\Gamma \vdash (\mathtt{Pair}\ 1\ \mathsf{True}) : (\mathtt{Int} \times \mathtt{Bool})}}{\Gamma \vdash (\mathtt{Apply}\ \mathsf{fst}\ (\mathtt{Pair}\ 1\ \mathsf{True})) : \gamma}$$

In the above example, we instantiated the quantified variables a and b in the type of fst to α and β, and used a placeholder γ to refer to the return type of the overall function application. Once we inferred the type of the argument as $(\mathtt{Int} \times \mathtt{Bool})$, we must now unify the type of the function we inferred ($(\alpha \times \beta) \to \alpha$) and the type of the function we expect based on the type of the argument we inferred $(\mathtt{Int} \times \mathtt{Bool}) \to \gamma$:

$$(\alpha \times \beta) \to \alpha \quad \sim \quad (\mathtt{Int} \times \mathtt{Bool}) \to \gamma$$

Once we unify these two types, we get the *unifier* substitution:

$$[\alpha := \mathtt{Int}, \beta := \mathtt{Bool}, \gamma := \mathtt{Int}]$$

Observe that if this substitution is applied to the two types above, they become the same.

Unifiers

A substitution S to unification variables is a *unifier* of two types τ and ρ iff $S\tau = S\rho$.

Furthermore, it is the *most general unifier*, or *mgu*, of τ and ρ if there is no other unifier S' where $S\tau \sqsubseteq S'\tau$.

We write $\tau \overset{U}{\sim} \rho$ if U is the mgu of τ and ρ.

Sometimes two types do not have a unifier. A clear example is $\mathtt{Int}$ and $\mathtt{String}$ — both types are concrete, and no amount of substitution to unknown variables will make them the same.

We can compute unifiers by structurally matching them. Our unify function would have a type like below, where the $\mathtt{Type}$ arguments do not include any $\forall$ quantifiers and the $\mathtt{Unifier}$ returned is the mgu:

$$\mathtt{unify} :: \mathtt{Type} \to \mathtt{Type} \to \mathtt{Maybe\ Unifier}$$

We shall discuss cases for `unify` τ_1 τ_2:

1. Both are type variables: $\tau_1 = v_1$ and $\tau_2 = v_2$:
   - $v_1 = v_2 \Rightarrow$ empty unifier
   - $v_1 \neq v_2 \Rightarrow [v_1 := v_2]$
2. Both are primitive type constructors: $\tau_1 = C_1$ and $\tau_2 = C_2$:
   - $C_1 = C_2 \Rightarrow$ empty unifier
   - $C_1 \neq C_2 \Rightarrow$ **no unifier**
3. Both are product types $\tau_1 = \tau_{11} \times \tau_{12}$ and $\tau_2 = \tau_{21} \times \tau_{22}$.
   (a) Compute the mgu S of τ_{11} and τ_{21}.
   (b) Compute the mgu S' of $S\tau_{12}$ and $S\tau_{22}$.
   (c) Return $S \cup S'$

   (same for sum, function types)
4. One is a type variable v, the other is just any term t.
   - v occurs in $t \Rightarrow$ no unifier
   - otherwise $\Rightarrow [v := t]$
5. Any other case $\Rightarrow$ no unifier.

Try the algorithm out on the following examples:

1. $\alpha \times (\alpha \times \alpha) \quad \sim \quad \beta \times \gamma$
$$[\beta := \alpha, \gamma := (\alpha \times \alpha)]$$
2. $(\alpha \times \alpha) \times \beta \quad \sim \quad \beta \times \gamma$
$$[\gamma := (\alpha \times \alpha), \beta := (\alpha \times \alpha)]$$
3. $\texttt{Int} + \alpha \quad \sim \quad \alpha + \texttt{Bool}$

   (no unifier)
4. $(\alpha \times \alpha) \times \alpha \quad \sim \quad \alpha \times (\alpha \times \alpha)$

   (no unifier)

The last example is particularly interesting because if we ignore the "occurs check" in case 4 of the algorithm, and naively try to structurally match, we end up with a substitution:

$$[\alpha := (\alpha \times \alpha)]$$

But, applying this substitution to both sides of the original problem yields:

$$((\alpha \times \alpha) \times (\alpha \times \alpha)) \times (\alpha \times \alpha) \quad \sim \quad (\alpha \times \alpha) \times ((\alpha \times \alpha) \times (\alpha \times \alpha))$$

And both type terms are still not the same. Even worse, trying again yields the exact same substitution we started with. This is called an *infinite type* error.

Type Inference Rules

We will *decompose* the typing judgement to allow for an additional output — a substitution that contains all the unifiers we have found about unknowns so far.

Inputs Expression, Context

Outputs Type, Substitution

We will write this as $S\Gamma \vdash e : \tau$, to make clear how the original typing judgement may be reconstructed.

Our new, combined variable and instantiation rule replaces all quantified variables with fresh unknown variables. Here "fresh" just indicates that the variable name has never been used before:

$$\frac{(x : \forall a_1.\ \forall a_2.\ \ldots \forall a_n.\ \tau) \in \Gamma}{\Gamma \vdash x : \tau[a_1 := \alpha_1, a_2 := \alpha_2, \ldots, a_n = \alpha_n]}\ (\alpha_1 \ldots \alpha_n \text{ fresh})$$

Observe that when the variable's type is not polymorphic (i.e. no quantifiers), then the above rule simplifies to our previous Var rule.

$$\frac{S_1\Gamma \vdash e_1 : \tau_1 \qquad S_2 S_1 \Gamma \vdash e_2 : \tau_2 \qquad S_2\tau_1 \overset{U}{\sim} (\tau_2 \to \alpha)}{U S_2 S_1 \Gamma \vdash (\texttt{Apply}\ e_1\ e_2) : U\alpha}\ (\alpha \text{ fresh})$$

Our rule for function application above mirrors the process we used informally in the previous example. A type is inferred for the function (τ_1) and a type for its argument τ_2. We generate a new placeholder α for the overall type of the application, and unify the type of the function with the type we expect given the type of the argument. We also apply the substitution S_2 we get from inferring τ_2 to the type τ_1 here, so that any information we learn about unification variables during the inference of τ_2 is applied before we attempt to unify the two types. Ultimately, we return the unifier applied to the α placeholder as our type, and the union of all of the substitutions computed so far as our returned substitution.

$$\frac{S(\Gamma, x : \alpha_1, f : \alpha_2) \vdash e : \tau \qquad S\alpha_2 \overset{U}{\sim} (S\alpha_1 \to \tau)}{US\Gamma \vdash (\texttt{Recfun}\ (f.x.\ e)) : U(S\alpha_1 \to \tau)}\ (\alpha_1, \alpha_2 \text{ fresh})$$

For functions, we generate two placeholders for the type of the function and its argument respectively, and then unify the function's type with the expected one based on the inferred return type τ.

Let Generalisation

Earlier we decided to use **let** expressions as the syntactic point for $\forall$-generalisation. If we consider this example:

$$\textbf{let}\ f = (\textbf{recfun}\ f\ x = (x, x))\ \textbf{in}\ (\mathsf{fst}\ (f\ 4), \mathsf{fst}\ (f\ \texttt{True}))$$

Just examining the inner **recfun**, we would compute a type like $\alpha \to (\alpha \times \alpha)$. The placeholder α would not be in use anywhere else — it would not be mentioned in the context outside of the **recfun**. We would expect the function f in the context to have a type like $\forall a.\ a \to (a \times a)$. Thus, we can define our *generalisation* operation to take all free placeholder variables in the type that are not still in use in our context, and $\forall$ quantify them. More formally, we define $Gen(\Gamma, \tau) = \forall(TV(\tau) \setminus TV(\Gamma)).\ \tau$

Then our rule for **let** expressions generalises the type before adding it to the context:

$$\frac{S_1\Gamma \vdash e_1 : \tau \qquad S_2(S_1\Gamma, x : Gen(S_1\Gamma, \tau)) \vdash e_2 : \tau'}{S_2 S_1 \Gamma \vdash (\texttt{Let}\ e_1 (x.\ e_2)) : \tau'}$$

This means that **let** expressions are now not just sugar for a function application, they actually play a vital role in the language's syntax, as a place for generalisation to occur.

Overall

We've specified Robin Milner's algorithm $\mathcal{W}$ for type inference, also called Damas-Milner type inference. Many other algorithms exist, for other kinds of type systems, including explicit *constraint-based* systems. This algorithm is restricted to the Hindley-Milner subset of decidable polymorphic instantiations, and requires that polymorphism is top-level — polymorphic functions are not first class.

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder