COMP3161/COMP9164 Supplementary Lecture Notes

Type Inference

Liam O'Connor Johannes Åman Pohjola* Rob Sison†

November 5, 2025

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

Considering the following expression:

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

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:

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

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

*Minor edits.

†Selection of content adapted to new type inference approach.

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:

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

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

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

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:

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

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

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:

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

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, $(\texttt{Int} \times \alpha) \to \beta$ is the type of a function from tuples where the left side of the tuple is `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, using unification judgements of the following form:

$$\Gamma_1 \vdash \tau_1 \sim \tau_2 \Longrightarrow \Gamma_2$$

which are defined such that:

1. Γ_1 and Γ_2 contain the same type variables;

2. Γ_2 is *more informative* than Γ_1 in the sense that declared type variables have been given definitions in order for $\tau_1 \sim \tau_2$ to hold: $\Gamma_1 \sqsubseteq \Gamma_2$

3. The information increase is *minimal* (most general) in the sense that it makes the least commitment in order to solve the equation: any other solution $\Gamma_1 \sqsubseteq \Gamma'$ factors through $\Gamma_1 \sqsubseteq \Gamma_2$.

Type Inference Rules

To keep track of the solutions to unification problems in context, we will *decompose* the typing judgement to allow for an additional output — an updated typing context which represents the *minimal information increase* over the input context (obtained via unification rules!) in order to infer the type of the expression.

Inputs Expression, Context

Outputs Type, Context

We will write this as $\Gamma \vdash e \Longrightarrow \tau \dashv \Gamma$, to make clear what the inputs and outputs are.

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. This operation will be used only by the type inference rule for **let** expressions, as well as the one for the entire program at the top level.

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 started examining characteristics of a variant of Robin Milner's algorithm $\mathcal{W}$ (variant thanks to Adam Gundry) for type inference. Further details will be released with Assignment 2. 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.