Implicitly Typed MinHS

Damas-Milner Type Inference

Johannes Åman Pohjola
UNSW
Term 3 2022
Implicitly Typed MinHS

Explicitly typed languages are awkward to use$^1$. Ideally, we’d like the compiler to determine the types for us.

Example

What is the type of this function?

\[
\text{recfun } f \ x = \text{fst } x + 1
\]

We want the compiler to infer the most general type.

$^1$See Java
Implicitly Typed MinHS

Start with our polymorphic MinHS, then:

- **remove** type signatures from `recfun`, `let`, etc.
- **remove** explicit `type` abstractions, and type applications (the `@` operator).
- **keep** `∀`-quantified types.
Implicitly Typed MinHS

Start with our polymorphic MinHS, then:

- **remove** type signatures from `recfun, let`, etc.
- **remove** explicit `type` abstractions, and type applications (the `@` operator).
- **keep** $\forall$-quantified types.
- **remove** recursive types, as we can’t infer types for them.

see “whiteboard” for why.
Typing Rules

\[ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \text{VAR} \]
Typing Rules

\[
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \text{VAR}
\]

\[
\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \ e_2 : \tau_2} \quad \text{APP}
\]
Typing Rules

\[
\begin{align*}
\frac{\Gamma \vdash x : \tau}{\Gamma, x : \tau} & \quad \text{VAR} \\
\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \ e_2 : \tau_2} & \quad \text{APP} \\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{Pair} \ e_1 \ e_2 : \tau_1 \times \tau_2} & \quad \text{CONJ}_I
\end{align*}
\]
Typing Rules

\[
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \text{VAR}
\]

\[
\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \; e_2 : \tau_2} \quad \text{APP}
\]

\[
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\text{Pair} \; e_1 \; e_2) : \tau_1 \times \tau_2} \quad \text{CONJ}_I
\]

\[
\frac{\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\text{If} \; e_1 \; e_2 \; e_3) : \tau} \quad \text{IF}
\]
For convenience, we treat prim ops as functions, and place their types in the environment.

\[(+): \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}, \Gamma \vdash (\text{App} (\text{App} (+) (\text{Num} 2)) (\text{Num} 1)): \text{Int}\]
Functions

\[ x : \tau_1, f : \tau_1 \to \tau_2, \Gamma \vdash e : \tau_2 \overset{\text{FUNC}}{\Rightarrow} \Gamma \vdash (\text{Recfun } (f . x . e)) : \tau_1 \to \tau_2 \]
Sum Types

\[
\Gamma \vdash e : \tau_1 \\
\Gamma \vdash \text{InL } e : \tau_1 + \tau_2
\]

\[
\Gamma \vdash e : \tau_2 \\
\Gamma \vdash \text{InR } e : \tau_1 + \tau_2
\]

Note that we allow the other side of the sum to be \text{any} type.
Polymorphism

If we have a polymorphic type, we can instantiate it to any type:

\[
\frac{\Gamma \vdash e : \forall a. \tau}{\Gamma \vdash e : \tau[a := \rho]} \quad \text{ALLE}
\]
Polymorphism

If we have a polymorphic type, we can instantiate it to any type:

\[ \Gamma \vdash e : \forall a. \tau \]

\[ \Gamma \vdash e : \tau[a := \rho] \]

\[ \text{ALLE} \]

We can quantify over any variable that has not already been used.

\[ \Gamma \vdash e : \tau \quad a \notin TV(\Gamma) \]

\[ \Gamma \vdash e : \forall a. \tau \]

\[ \text{ALLI} \]

(Where \( TV(\Gamma) \) here is all type variables occurring free in the types of variables in \( \Gamma \))
The Goal

We want an algorithm for type inference:

- With a clear input and output.
- Which terminates.
- Which is fully deterministic.
Typing Rules

\[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \]
\[ \Gamma \vdash (\text{Pair } e_1 \; e_2) : \tau_1 \times \tau_2 \]

Can we use the existing typing rules as our algorithm?

\textit{infer} :: \textit{Context} \to \textit{Expr} \to \textit{Type}
Implicitly Typed MinHS
Inference Algorithm
Unification

Typing Rules

Γ ⊢ e₁ : τ₁  Γ ⊢ e₂ : τ₂
Γ ⊢ (Pair e₁ e₂) : τ₁ \times τ₂

Can we use the existing typing rules as our algorithm?

infer :: Context → Expr → Type

This approach can work for monomorphic types, but not polymorphic ones. Why not?
First Problem

\[ \Gamma \vdash e : \forall a. \tau \]
\[ \frac{}{\Gamma \vdash e : \tau[a := \rho]} \text{ALLE} \]

The rule to add a \( \forall \)-quantifier can always be applied:

\[ \vdash \]
\[ \frac{}{\Gamma \vdash (\text{Num 5}) : \forall a. \forall b. \text{Int}} \text{ALLE} \]
\[ \frac{}{\Gamma \vdash (\text{Num 5}) : \forall a. \text{Int}} \text{ALLE} \]
\[ \frac{}{\Gamma \vdash (\text{Num 5}) : \text{Int}} \text{ALLE} \]

Read as an algorithm, the rules are non-deterministic – there are many possible rules for a given input. A depth-first search strategy may end up attempting infinite derivations.
Another Problem

\[ \Gamma \vdash e : \forall a. \tau \]
\[ \Gamma \vdash e : \tau[a := \rho] \]

The above rule can be applied at any time to a polymorphic type, even if it would break later typing derivations:

\[ \Gamma \vdash \text{fst} : \forall a. \forall b. (a \times b) \rightarrow a \]
\[ \Gamma \vdash \text{fst} : (\text{Bool} \times \text{Bool}) \rightarrow \text{Bool} \]
\[ \Gamma \vdash (\text{Pair 1 True}) : (\text{Int} \times \text{Bool}) \]
\[ \Gamma \vdash (\text{Apply fst (Pair 1 True)}) : ??? \]
The rule for recfun mentions $\tau_2$ in both input and output positions.

$$
\frac{x : \tau_1, f : \tau_1 \to \tau_2, \Gamma \vdash e : \tau_2}{\Gamma \vdash (\text{Recfun} \ (f \ x \ e)) : \tau_1 \to \tau_2}
$$

In order to infer $\tau_2$ we must provide a context that includes $\tau_2$ — this is circular. Any guess we make for $\tau_2$ could be wrong.
Solution

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 $\alpha, \beta$ etc. for these.

**Example**

$$(\text{Int} \times \alpha) \rightarrow \beta$$ is the type of a function from tuples where the left side 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.
Example

\[\Gamma \vdash \text{fst} : \forall a. \forall b. (a \times b) \rightarrow a\]
\[\Gamma \vdash \text{fst} : (\alpha \times \beta) \rightarrow \alpha\]
\[\Gamma \vdash (\text{Pair 1 True}) : (\text{Int} \times \text{Bool})\]
\[\Gamma \vdash (\text{Apply} \ \text{fst} \ (\text{Pair 1 True})) : \gamma\]
Example

\[
\Gamma \vdash \text{fst} : \forall a. \forall b. (a \times b) \rightarrow a \quad \cdots
\]

\[
\Gamma \vdash \text{fst} : (\alpha \times \beta) \rightarrow \alpha \quad \Gamma \vdash (\text{Pair 1 True}) : (\text{Int} \times \text{Bool})
\]

\[
\Gamma \vdash (\text{Apply fst (Pair 1 True)}) : \gamma
\]

\[
(\alpha \times \beta) \rightarrow \alpha \sim (\text{Int} \times \text{Bool}) \rightarrow \gamma
\]
Example

\[ \Gamma ⊢ \text{fst} : \forall a. \forall b. (a \times b) \rightarrow a \]
\[ \Gamma ⊢ \text{fst} : (\alpha \times \beta) \rightarrow \alpha \]
\[ \Gamma ⊢ (\text{Pair 1 True}) : (\text{Int} \times \text{Bool}) \]
\[ \Gamma ⊢ (\text{Apply} \ \text{fst} \ (\text{Pair 1 True})) : \gamma \]

\( (\alpha \times \beta) \rightarrow \alpha \sim (\text{Int} \times \text{Bool}) \rightarrow \gamma \)

\([\alpha := \text{Int}, \beta := \text{Bool}, \gamma := \text{Int}]\)
We call this substitution a *unifier*.

**Definition**

A substitution $S$ is a *unifier* of two types $\tau$ and $\rho$ iff $S\tau = S\rho$. Furthermore, it is the *most general unifier*, or *mgu*, of $\tau$ and $\rho$ if there is no other unifier $S'$ where $S\tau \sqsubseteq S'\tau$.

We write $\tau \sim U \sim \rho$ if $U$ is the mgu of $\tau$ and $\rho$.

**Example ("Whiteboard")**

- $\alpha \times (\alpha \times \alpha) \sim \beta \times \gamma$
- $(\alpha \times \alpha) \times \beta \sim \beta \times \gamma$
- $\text{Int} + \alpha \sim \alpha + \text{Bool}$
- $(\alpha \times \alpha) \times \alpha \sim \alpha \times (\alpha \times \alpha)$
Back to Type Inference

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 $\Sigma \vdash e : \tau$, to make clear how the original typing judgement may be reconstructed.
Application, Elimination

\[
S_1 \Gamma \vdash e_1 : \tau_1 \quad S_2 S_1 \Gamma \vdash e_2 : \tau_2 \quad S_2 \tau_1 \overset{U}{\sim} (\tau_2 \to \alpha) \\
US_2 S_1 \Gamma \vdash (\text{Apply } e_1 \; e_2) : U\alpha \\
\]

(\alpha \text{ fresh})

\[
(x : \forall a_1. \forall a_2. \ldots \forall a_n. \tau) \in \Gamma \\
\Gamma \vdash x : \tau[a_1 := \alpha_1, a_2 := \alpha_2, \ldots, a_n = \alpha_n] \\
\]

(\alpha_1 \ldots \alpha_n \text{ fresh})

Example ("Whiteboard")

\[(\text{fst} : \forall a \; b. \; (a \times b) \to a) \vdash (\text{Apply } \text{fst} \; (\text{Pair} \; 1 \; 2))\]
Functions

\[
S(\Gamma, x : \alpha_1, f : \alpha_2) \vdash e : \tau \\ S\alpha_2 \ U (S\alpha_1 \rightarrow \tau) \\
US\Gamma \vdash (\text{Recfun } (f.x. e)) : U(S\alpha_1 \rightarrow \tau)
\]

(\alpha_1, \alpha_2 \text{ fresh})

Example ("Whiteboard")

(Recfun (f.x. (Pair x x)))

(Recfun (f.x. (Apply f x)))
In our typing rules, we could generalise a type to a polymorphic type by introducing a $\forall$ at any point. We want to restrict this to only occur in a *syntax-directed* way. Consider this example:

\[
\text{let } f = (\text{recfun } f \ x = (x, x)) \text{ in } (\text{fst } (f \ 4), \text{fst } (f \ \text{True}))
\]

Where should generalisation happen?
Let-generalisation

To make type inference tractable, we will generalise only in `let` expressions.
Let-generalisation

To make type inference tractable, we will generalise only in let expressions.

This means that let expressions are now not just sugar for a function application. They actually play a vital role, as the place where generalisation happens.
Let-generalisation

To make type inference tractable, we will generalise only in let expressions.

This means that let expressions are now not just sugar for a function application. They actually play a vital role, as the place where generalisation happens.

We define \( \text{Gen}(\Gamma, \tau) = \forall (\text{TV}(\tau) \setminus \text{TV}(\Gamma)). \tau \)
Then we have:

\[
\frac{\Gamma \vdash e_1 : \tau \quad S_2(S_1 \Gamma, x : \text{Gen}(S_1 \Gamma, \tau)) \vdash e_2 : \tau'}{S_2S_1 \Gamma \vdash (\text{Let } e_1(x. e_2)) : \tau'}
\]
The rest of the rules are straightforward from their typing rules.

We’ve specified Robin Milner’s algorithm $\mathcal{W}$ for 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.

We still need an algorithm to compute the unifiers.
Unification

\[
\text{unify} :: \text{Type} \rightarrow \text{Type} \rightarrow \text{Maybe Unifier}
\]

(where the Type arguments do not include any \(\forall\) quantifiers and the Unifier returned is the mgu)

We shall discuss cases for \(\text{unify } \tau_1 \ \tau_2\)
Cases

Both type variables: $\tau_1 = v_1$ and $\tau_2 = v_2$:

- $v_1 = v_2 \Rightarrow$ empty unifier
- $v_1 \neq v_2 \Rightarrow [v_1 := v_2]$
Cases

Both 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
Cases

Both are product types $\tau_1 = \tau_{11} \times \tau_{12}$ and $\tau_2 = \tau_{21} \times \tau_{22}$. 
Cases

Both are product types $\tau_1 = \tau_{11} \times \tau_{12}$ and $\tau_2 = \tau_{21} \times \tau_{22}$.

1. Compute the mgu $S$ of $\tau_{11}$ and $\tau_{21}$.
Cases

Both are product types $\tau_1 = \tau_{11} \times \tau_{12}$ and $\tau_2 = \tau_{21} \times \tau_{22}$.

1. Compute the mgu $S$ of $\tau_{11}$ and $\tau_{21}$.
2. Compute the mgu $S'$ of $S\tau_{12}$ and $S\tau_{22}$.
Cases

Both are product types $\tau_1 = \tau_{11} \times \tau_{12}$ and $\tau_2 = \tau_{21} \times \tau_{22}$.

1. Compute the mgu $S$ of $\tau_{11}$ and $\tau_{21}$.
2. Compute the mgu $S'$ of $S\tau_{12}$ and $S\tau_{22}$.
3. Return $S \cup S'$
Both are product types $\tau_1 = \tau_{11} \times \tau_{12}$ and $\tau_2 = \tau_{21} \times \tau_{22}$.

1. Compute the mgu $S$ of $\tau_{11}$ and $\tau_{21}$.
2. Compute the mgu $S'$ of $S\tau_{12}$ and $S\tau_{22}$.
3. Return $S \cup S'$

(same for sum, function types)
Cases

One is a type variable \( v \), the other is just any term \( t \).

- \( v \) occurs in \( t \) \( \Rightarrow \) no unifier

- otherwise \( \Rightarrow [v := t] \)
Implementing this algorithm is the focus of Assignment 2 (out now!)

See course website for deadlines etc.

You should allow plenty of time to tackle it.

Haskell-wise, this code will use a monad to track errors and the state needed to generate fresh unification variables.