Damas-Milner Type Inference

Johannes Åman Pohjola
UNSW
Term 3 2022
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.

\(^1\text{See Java}\)
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.
- **remove** recursive types, as we can’t infer types for them.

see “whiteboard” for why.
Typing Rules

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

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

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

\[ \Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau \]
\[ \frac{\Gamma \vdash (\text{If } e_1 \ e_2 \ e_3) : \tau}{\text{IF}} \]
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

\[
\begin{align*}
\Gamma & \vdash \text{Recfun} \ (f \cdot x \cdot e) : \tau_1 \rightarrow \tau_2 \\
& \quad \text{\underline{ FUNC }} \\
\end{align*}
\]
Sum Types

\[ \Gamma \vdash e : \tau_1 \quad \text{DISJ}_{I_1} \]
\[ \Gamma \vdash \text{InL} \ e : \tau_1 + \tau_2 \]

\[ \Gamma \vdash e : \tau_2 \quad \text{DISJ}_{I_2} \]
\[ \Gamma \vdash \text{InR} \ e : \tau_1 + \tau_2 \]

Note that we allow the other side of the sum to be any type.
Polymorphism

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

$$\Gamma \vdash e : \forall a. \tau$$

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

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

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

$$\Gamma \vdash e : \forall a. \tau$$

(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} \rightarrow \textit{Expr} \rightarrow \textit{Type}

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

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

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

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

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:

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

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

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 \text{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

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

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

\[ [\alpha := \text{Int}, \beta := \text{Bool}, \gamma := \text{Int}] \]
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 $\tau \sim_\cup \rho$ if $U$ is the mgu of $\tau$ and $\rho$.

**Example (“Whiteboard”)**

- $\alpha \times (\alpha \times \alpha) \sim \beta \times \gamma$
- $(\alpha \times \alpha) \times \beta \sim \beta \times \gamma$
- Int $+ \alpha \sim \alpha + \text{Bool}$
- $(\alpha \times \alpha) \times \alpha \sim \alpha \times (\alpha \times \alpha)$
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.
Application, Elimination

\[ S_1 \Gamma \vdash e_1 : \tau_1 \quad S_2 S_1 \Gamma \vdash e_2 : \tau_2 \quad S_2 \tau_1 \overset{U}{\sim} (\tau_2 \rightarrow \alpha) \]

\[
US_2 S_1 \Gamma \vdash (\text{Apply } e_1 \ e_2) : U\alpha
\]

\[
(x : \forall a_1. \forall a_2. \ldots \forall a_n. \tau) \in \Gamma
\]

\[
\Gamma \vdash x : \tau[a_1 := \alpha_1, a_2 := \alpha_2, \ldots, a_n = \alpha_n]
\]

Example ("Whiteboard")

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

\[ S(\Gamma, x : \alpha_1, f : \alpha_2) \vdash e : \tau \quad S\alpha_2 \sim (S\alpha_1 \to \tau) \]

\[ US\Gamma \vdash (\text{Recfun } (f \cdot x \cdot e)) : U(S\alpha_1 \to \tau) \]

Example ("Whiteboard")

\( (\text{Recfun } (f \cdot x \cdot (\text{Pair } x \ x))) \)

\( (\text{Recfun } (f \cdot x \cdot (\text{Apply } f \ x))) \)
Generalisation

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:

```plaintext
let f = (recfun f x = (x, x)) in (fst (f 4), fst (f True))
```

Where should generalisation happen?
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 $Gen(\Gamma, \tau) = \forall (TV(\tau) \setminus TV(\Gamma)). \tau$

Then we have:

$$
S_1\Gamma \vdash e_1 : \tau \quad S_2(S_1\Gamma, x : Gen(S_1\Gamma, \tau)) \vdash e_2 : \tau' \\
S_2S_1\Gamma \vdash (\text{Let } e_1(x. e_2)) : \tau'
$$
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 \text{Type} arguments do not include any \(\forall\) quantifiers and the \text{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
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.