Damas-Milner Type Inference

Christine Rizkallah
CSE, UNSW
Term 3 2020

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?

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

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 $\forall$-quantified types.
- remove recursive types, as we can't infer types for them.

  see whiteboard for why

Typing Rules

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\textsc{Var}$$

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\texttt{Pair}\ e_1\ e_2) : \tau_1 \times \tau_2}\textsc{Conj}_I$$

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

Primitive Operators

For convenience, we treat prim ops as functions, and place their types in the environment.

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

Functions

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

Sum Types

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{InL}\ e : \tau_1 + \tau_2}\ \textsc{Disj}_{\mathrm{I1}}$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{InR}\ e : \tau_1 + \tau_2}\ \textsc{Disj}_{\mathrm{I2}}$$

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:

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

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

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

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

The Goal

We want an algorithm for type inference:

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

Typing Rules

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\texttt{Pair}\ e_1\ e_2) : \tau_1 \times \tau_2}$$

Can we use the existing typing rules as our algorithm?

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

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

First Problem

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

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

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash (\texttt{Num}\ 5) : \forall a.\ \forall b.\ \texttt{Int}}\ \textsc{All}_\mathrm{E}}{\dfrac{\Gamma \vdash (\texttt{Num}\ 5) : \forall a.\ \texttt{Int}}{\Gamma \vdash (\texttt{Num}\ 5) : \texttt{Int}}\ \textsc{All}_\mathrm{E}}$$

This makes the rules give rise to a non-deterministic algorithm – there are many possible rules for a given input. Furthermore, as it can always be applied, a depth-first search strategy may end up attempting infinite derivations.

Another Problem

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

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

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

Yet Another Problem

The rule for **recfun** mentions τ_2 in both input and output positions.

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

In order to infer τ_2 we must provide a context that includes τ_2 — this is circular. Any guess we make for τ_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 α, β etc. for these variables.

Example

$(\texttt{Int} \times \alpha) \rightarrow \beta$ is the type of a function from tuples where the left side of the tuple is $\texttt{Int}$, 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

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

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

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

Unification

We call this substitution a *unifier*.

Definition

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.

Example (Whiteboard)

- $\alpha \times (\alpha \times \alpha) \quad \sim \quad \beta \times \gamma$
- $(\alpha \times \alpha) \times \beta \quad \sim \quad \beta \times \gamma$
- $\texttt{Int} + \alpha \quad \sim \quad \alpha + \texttt{Bool}$
- $(\alpha \times \alpha) \times \alpha \quad \sim \quad \alpha \times (\alpha \times \alpha)$

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

Application, Elimination

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

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

Example (Whiteboard)

$$(\mathsf{fst} : \forall a\ b.\ (a \times b) \to a) \vdash (\texttt{Apply}\ \mathsf{fst}\ (\texttt{Pair}\ 1\ 2))$$

Functions

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

Example (Whiteboard)

$$(\mathtt{Recfun}\ (f.x.\ (\mathtt{Pair}\ x\ x)))$$

$$(\mathtt{Recfun}\ (f.x.\ (\mathtt{Apply}\ f\ x)))$$

Generalisation

In our typing rules, we could generalise a type to a polymorphic type by introducing a $\forall$ at any point in the typing derivation. We want to be able to restrict this to only occur in a *syntax-directed* way.

Consider this example:

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

Where should generalisation happen?

Let-generalisation

To make type inference tractable, we restrict generalisation to only occur when a binding is added to the context via a **let** expression.

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.

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

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

Summary

- The rest of the rules are straightforward from their typing rule.
- 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

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

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

We shall discuss cases for `unify` τ_1 τ_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 $\tau_2 = \tau_{21} \times \tau_{22}$.

1. Compute the mgu S of τ_{11} and τ_{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 $\Rightarrow [v := t]$

Done

- Implementing this algorithm will be the focus of Assignment 2.
- See course website for deadlines etc.
- You should allow plenty of time to tackle it. You will be given a generous deadline but it requires time to complete.
- Haskell-wise, this code will use a monad to track errors and the state needed to generate fresh unification variables.