Damas-Milner Type In

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 th

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

We want the compiler to infer the most general type.

[1] See Java

Implicitly Typed MinHS

Start with our polymorphic MinHS, then:

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

  see whiteboard for why

Typing Rules

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

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

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

Primitive Operators

For convenience

environment.

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

Functions

1 2

Sum Types

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{InR}\ e : \tau_1 + \tau}$$

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:

$$\frac{\quad}{\quad}$$

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

$$\frac{\Gamma \vdash e : \tau \qquad a \notin TV}{\Gamma \vdash e : \forall a.\ \tau}$$

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

The Goal

We want an algorit

- With a clear
- Which term
- Which is fully deterministic.

Typing Rules

$$\frac{\Gamma \quad e : \tau \quad \Gamma \quad e : \tau}{}$$

Can we use the exist

$\texttt{infer} :: \texttt{Context} \rightarrow \texttt{E}$

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

First Problem

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

The rule to add a

$$\cfrac{\cfrac{\cfrac{}{\Gamma \vdash (\texttt{Num}\ 5) : \forall a.\ \forall b.\ \texttt{Int}}}{\Gamma \vdash (\texttt{Num}\ 5) : \forall a.\ \texttt{Int}}}{\Gamma \vdash (\texttt{Num}\ 5) : \texttt{Int}}$$

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

$$\frac{\Gamma \vdash e : \forall a.\tau}{\quad}$$

The above rule can b

break later typing

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

Yet Another Problem

The rule for **re**

In order to infer τ_2 we must provide a context that includ lar. 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 a

use α, β etc. for t

Example

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

$\texttt{Int}$, but no other details of the type have been determined yet.

As we encounter situations where two types should be equal, w e two types to determine what the unknown variables should be.

Example

$\Gamma \vdash \mathsf{fst} : \ a. \ b. \ (a \quad b) \quad a$

$\texttt{Bool})$

$(\alpha \times \beta) \to \alpha \quad \sim \quad (\texttt{In}$

$[\alpha := \texttt{Int}, \beta := \texttt{Boo}$

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 t s no other unifier S' wher

We write $\tau \overset{U}{\sim}$

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 decompos

substitution tha

Inputs

Outputs Type, Substitution

We will write this as $S\Gamma \vdash e : \tau$, to make clear how the o

be reconstructed.

Application, Elimination

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

fresh)

Γ

Example (Whiteboard)

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

Functions

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

Example (Wh t

$(\texttt{Recfun}\ (f.x.\ (\texttt{Pa}$

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

Generalisation

In our typing rules, we could generalise a type to a polymorphic type by introducing a $\forall$ at any point in the t

occur in a *synt*

Consider this exa

$$\textbf{let } f = (\textbf{recfun } f\ x = (x, x)) \textbf{ in}$$

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 ... ey actually play a vita

We define $Gen(\Gamma, \tau) = \forall (TV(\tau) \setminus TV(\Gamma)).\ \tau$

Then we have:

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

Summary

- The rest of the rules are straightforward from their typing rule.
- We've spec y other algorithm *constraint*

- This algorithm is restricted to the Hindley-Milner subs instantiations, and requires that polymorphism is top functions are not first class.

- We still need an algorithm to compute the unifiers.

Unification

(where the `Ty` ... `Unifier` returned is the mgu)

We shall discuss cases for `unify` τ_1 τ_2

Cases

Both type variabl

- $v_1 = v_2$
- $v_1 \neq v_2 \Rightarrow [v_1 := v_2]$

Cases

Both primitive ty

- $C_1 = C_2$
- $C_1 \neq C_2 \Rightarrow$ no unifier

Cases

Both are product t

1. Compute t
2. Compute t
3. Return $S \cup S'$

(same for sum, function types)

Cases

One is a type variabl

- v occurs in
- otherwise $\Rightarrow [v := t]$

Done

- Implemen
- See course w
- You should

  but it requires time to complete.
- Haskell-wise, this code will use a monad to track errors and t

  generate fresh unification variables.