



Liam O'Connor

CSE, UNSW (and data61)

Semester 2 2018

Implicitly Typed MinHS

Explicitly typed languages are awkward to use¹. Ideally, we'd like the compiler to determine the types for us.

Example

What is the type of this function?

```
recfun  $f$   $x = \text{fst } x + 1$ 
```

We want the compiler to infer the **most general** type.

¹See 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

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

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

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

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

Primitive Operators

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

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

Sum Types

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{InL } e : \tau_1 + \tau_2} \text{DISJ}_{I1}$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{InR } e : \tau_1 + \tau_2} \text{DISJ}_{I2}$$

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:

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

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

$$\frac{\Gamma \vdash e : \tau \quad a \notin TV(\Gamma)}{\Gamma \vdash e : \forall a. \tau} \text{ALLI}$$

(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

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash \text{fst} : \forall a. \forall b. (a \times b) \rightarrow a \quad \dots}{\Gamma \vdash \text{fst} : (\text{Bool} \times \text{Bool}) \rightarrow \text{Bool} \quad \Gamma \vdash (\text{Pair } 1 \text{ True}) : (\text{Int} \times \text{Bool})} \frac{}{\Gamma \vdash (\text{Apply } \text{fst} (\text{Pair } 1 \text{ True})) : ???}$$

Yet Another Problem

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

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

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

$(\text{Int} \times \alpha) \rightarrow \beta$ 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.

Example

$$\frac{\Gamma \vdash \text{fst} : \forall a. \forall b. (a \times b) \rightarrow a \quad \dots}{\Gamma \vdash \text{fst} : (\alpha \times \beta) \rightarrow \alpha \quad \Gamma \vdash (\text{Pair } 1 \text{ True}) : (\text{Int} \times \text{Bool})} \Gamma \vdash (\text{Apply fst } (\text{Pair } 1 \text{ True})) : \gamma$$

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

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

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 $\tau \stackrel{U}{\sim} \rho$ if U is the mgu of τ and ρ .

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

Application, Elimination

$$\frac{S_1\Gamma \vdash e_1 : \tau_1 \quad S_2S_1\Gamma \vdash e_2 : \tau_2 \quad S_2\tau_1 \stackrel{U}{\sim} (\tau_2 \rightarrow \alpha)}{US_2S_1\Gamma \vdash (\text{Apply } e_1 \ e_2) : U\alpha} \quad (\alpha \text{ fresh})$$

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

Example (Whiteboard)

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

Functions

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

Example (Whiteboard)

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

```
(Recfun (f.x. (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:

```
let  $f = (\mathbf{recfun} \ f \ x = (x, x))$  in (fst (f 4), fst (f 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:

$$\frac{S_1\Gamma \vdash e_1 : \tau \quad S_2(S_1\Gamma, x : Gen(S_1\Gamma, \tau)) \vdash e_2 : \tau'}{S_2S_1\Gamma \vdash (\text{Let } e_1(x. e_2)) : \tau'}$$

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

`unify :: Type → Type → 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.
- Assignment 2 will be released next tuesday.
- 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.