TYPE INference
Last Time: ML Polymorphism

The type for map looks like this:

\[
\text{map : } (\forall a \to b) \to (a \to b) \to a \to b
\]

This type includes an implicit quantifier at the outermost level. So really, map's type is this one:

\[
\text{map : } \forall a, b. (a \to b) \to (a \to b) \to a \to b
\]

To use a value with type \(\forall a, b, c. t\), we first substitute types for parameters \(a, b, c\). eg:

\[
\text{map } (\text{fun } x \rightarrow x + 1) [2;3;4]
\]

here, we substitute \([\text{int}/a][\text{int}/b]\) in map's type and then use map at type \((\text{int} \to \text{int}) \to \text{int list} \to \text{int list}\)
Last Time

Type Checking (Simple Types)

A function \texttt{check : context -> exp -> type}

- requires function arguments to be annotated with types
- specified using formal rules. eg, the rule for function call:

\[
\begin{array}{c}
\frac{\Gamma |- e_1 : t_1 \rightarrow t_2 \quad \Gamma |- e_2 : t_1}{\Gamma |- e_1 \ e_2 : t_2}
\end{array}
\]
Type Inference (Simple Types)

A function \texttt{infer : context -> exp -> ann_exp * type * constraints}

- Generates constraints (equations between types)
- Solves those constraints to find a solution (i.e.: a substitution)
- An example rule:

\[\begin{align*}
\Gamma &\vdash u_1 \Longrightarrow e_1 : t_1, q_1 \\
\Gamma &\vdash u_2 \Longrightarrow e_2 : t_2, q_2 \\
\Gamma &\vdash u_1 u_2 \Longrightarrow e_1 e_2 : a, q_1 U q_2 U \{t_1 = t_2 \rightarrow a\}
\end{align*}\]
Type Inference (Simple Types)

A function \( \text{infer} : \text{context} \rightarrow \text{exp} \rightarrow \text{ann\_exp} \times \text{type} \times \text{constraints} \)

- Generates constraints (equations between types)
- Solves those constraints to find a solution (i.e., a substitution)
- An example rule:

\[
\begin{align*}
G \vdash u_1 \Rightarrow e_1 : t_1, q_1 \\
G \vdash u_2 \Rightarrow e_2 : t_2, q_2 \\
\hline
G \vdash u_1 u_2 \Rightarrow e_1 e_2 : a, q_1 \cup q_2 \cup \{t_1 = t_2 \rightarrow a\}
\end{align*}
\]

Up Next: How to find solutions to sets of type equations.
SOLVING CONSTRAINTS
Solving Constraints

A solution to a system of type constraints is a substitution $S$ — a function from type variables to types.

Given a set of constraints:

\[
\begin{align*}
t_1 &= t_2 \\
t_3 &= t_4 \\
t_5 &= t_6 \\
&\ldots
\end{align*}
\]

$S$ is a solution to these constraints when it makes LHS and RHS of each equation equal. ie:

\[
\begin{align*}
S(t_1) \text{ and } S(t_2) \text{ must be identical} \\
S(t_3) \text{ and } S(t_4) \text{ must be identical} \\
S(t_5) \text{ and } S(t_6) \text{ must be identical} \\
&\ldots
\end{align*}
\]
Example Constraints & Solution

constraints:

<table>
<thead>
<tr>
<th>a = b -&gt; c</th>
</tr>
</thead>
<tbody>
<tr>
<td>c = int -&gt; bool</td>
</tr>
</tbody>
</table>
Example Constraints & Solution

constraints:

\[ a = b \rightarrow c \]
\[ c = \text{int} \rightarrow \text{bool} \]

solution S:

\[ b \rightarrow (\text{int} \rightarrow \text{bool})/a \text{ int} \rightarrow \text{bool}/c \]
\[ b/b \]
Example Constraints & Solution

constraints:

\[ a = b \rightarrow c \]
\[ c = \text{int} \rightarrow \text{bool} \]

solution S:

\[ b \rightarrow (\text{int} \rightarrow \text{bool})/a \]
\[ \text{int} \rightarrow \text{bool}/c \]
\[ b/b \]

Why is this a solution?

\[ S(a) = S(b \rightarrow c) = b \rightarrow (\text{int} \rightarrow \text{bool}) \]
\[ S(c) = S(\text{int} \rightarrow \text{bool}) = \text{int} \rightarrow \text{bool} \]
Example Constraints & Solution

constraints:

\[
\begin{align*}
a &= b \rightarrow c \\
c &= \text{int} \rightarrow \text{bool}
\end{align*}
\]

solution S:

\[
\begin{align*}
b &\rightarrow (\text{int} \rightarrow \text{bool})/a \\
\text{int} &\rightarrow \text{bool}/c \\
b &/b
\end{align*}
\]

solution S2:

\[
\begin{align*}
\text{int} &\rightarrow (\text{int} \rightarrow \text{bool})/a \\
\text{int} &\rightarrow \text{bool}/c \\
\text{int} &/b
\end{align*}
\]

We say that S is a more general solution than S2 because for all type t, \( S2(t) = U(S(t)) \) when U is the substitution [int/b]
Why do we like more general solutions?

constraints:
\[ a = b \rightarrow c \]
\[ c = \text{int} \rightarrow \text{bool} \]

solution S:
\[ b \rightarrow (\text{int} \rightarrow \text{bool})/a \]
\[ \text{int} \rightarrow \text{bool}/c \]
\[ b/b \]

solution S2:
\[ \text{int} \rightarrow (\text{int} \rightarrow \text{bool})/a \]
\[ \text{int} \rightarrow \text{bool}/c \]
\[ \text{int}/b \]

Consider this program, which might have generated the above constraints:

let f : a =
  fun (x:b) : c ->
  fun n -> n < 10)

Fact 1: *Any solution* to the constraints gives rise to a *sound* type for f.
- ie: f won’t crash if we give it any type that arises from a solution

Fact 2: If solution S is *more general* than S2 then f can be *used* in at least as many contexts (without the program crashing) if f has type S(a) than if f has type S2(a).
Why do we like more general solutions?

Consider this program, which might have generated the above constraints:

```
let f : a =
  fun (x:b) : c ->
    fun n -> n < 10)
```

Fact 2: If solution S is more general than S2 then f can be used in at least as many contexts (without the program crashing) if f has type S(a) than if f has type S2(a).

eg: with S, “f true” will type check but with S2, it won’t
It turns out, there is always a best solution, which we can call a *principle solution*. This is a pretty fortunate property – it means we can prove a kind of “completeness” property for ML type inference.

The best solution is (at least as) preferred as any other solution.
Example 1

- \( q = \{a=\text{int}, \ b=a\} \)
- principal solution \( S:\)
Example 1

- \( q = \{a=\text{int}, \ b=a\} \)

- principal solution \( S \):
  
  - \( S(a) = S(b) = \text{int} \)
  
  - \( S(c) = c \) (for all \( c \) other than \( a,b \))
Example 2

- $q = \{a=\text{int}, \ b=a, \ b=\text{bool}\}$
- principal solution $S$: 
Example 2

- $q = \{a=\text{int}, b=a, b=\text{bool}\}$
- principal solution $S$:
  - does not exist (there is no solution to $q$)
Unification: An algorithm that provides the principal solution to a set of constraints (if one exists)

- Unification systematically simplifies a set of constraints, yielding a substitution
  
  • Starting state of unification process: (I,q)
  • Final state of unification process: (S, {})

Unification
Unification simplifies equations step-by-step until
• there are no equations left to simplify, or
• we find basic equations are inconsistent and we fail

type ustate = substitution * constraints

unify_step : ustate -> ustate
Unification simplifies equations step-by-step until
• there are no equations left to simplify, or
• we find basic equations are inconsistent and we fail

\[
\text{type } \text{ustate} = \text{substitution} \times \text{constraints}
\]

\[
\text{unify_step} : \text{ustate} \rightarrow \text{ustate}
\]

\[
\text{unify_step} \ (S, \ \{\text{bool}=\text{bool}\} \ U \ q) \ = \ (S, q)
\]

\[
\text{unify_step} \ (S, \ \{\text{int}=\text{int}\} \ U \ q) \ = \ (S, q)
\]
Unification

Unification simplifies equations step-by-step until

• there are no equations left to simplify, or
• we find basic equations are inconsistent and we fail

\[
\text{type ustate} = \text{substitution} \ast \text{constraints}
\]

\[
\text{unify\_step} : \text{ustate} \rightarrow \text{ustate}
\]

\[
\text{unify\_step} (S, \{\text{bool}=\text{bool}\} \cup q) = (S, q)
\]

\[
\text{unify\_step} (S, \{\text{int}=\text{int}\} \cup q) = (S, q)
\]

\[
\text{unify\_step} (S, \{a=a\} \cup q) = (S, q)
\]
Unification simplifies equations step-by-step until
• there are no equations left to simplify, or
• we find basic equations are inconsistent and we fail

unify_step (S, \{A \rightarrow B = C \rightarrow D\} U q)
= (S, \{A = C, B = D\} U q)
Unification

Unification simplifies equations step-by-step until
• there are no equations left to simplify, or
• we find basic equations are inconsistent and we fail

\[
\text{type ustate} = \text{substitution} * \text{constraints} \\
\text{unify\_step} : \text{ustate} \rightarrow \text{ustate}
\]

\[
\text{unify\_step} \left(S, \{A \rightarrow B = C \rightarrow D\} \cup q\right) = \left(S, \{A = C, B = D\} \cup q\right)
\]
unify_step (S, \{a=s\} U q) = ([s/a] o S, [s/a]q)

when a is not in FreeVars(s)
Unification

\[
\text{unify\_step}\ (S, \ \{a=s\} \ U \ q) = ([s/a] \circ S, \ [s/a]q)
\]

when \(a\) is not in \(\text{FreeVars}(s)\)

the substitution \(S'\) defined to:
do \(S\) then substitute \(s\) for \(a\)

the constraints \(q'\) defined to:
be like \(q\) except \(s\) replacing \(a\)
Recall this program from assignment #1:

```haskell
fun x -> x x
```

It generates the constraints: \( a \rightarrow a = a \)

What is the solution to \( \{ a = a \rightarrow a \} \)?
Recall this program from assignment #1:

```
fun x -> x x
```

It generates the constraints: \( a \rightarrow a = a \)

What is the solution to \( \{a = a \rightarrow a\} \)?

There is none!

Notice that \( a \) does appear in \( \text{FreeVars}(s) \).

Whenever \( a \) appears in \( \text{FreeVars}(s) \) and \( s \) is not just \( a \), there is no solution to the system of constraints.
Recall this program from assignment #1:

\[
\text{fun } x \rightarrow x \times x
\]

It generates the constraints: \( a \rightarrow a = a \)

What is the solution to \( \{ a = a \rightarrow a \} \)?

There is none!

"when a is not in FreeVars(s)" is known as the "occurs check"
Recall: unification simplifies equations step-by-step until

- there are no equations left to simplify:

\[(S, \{ \})\]

no constraints left. S is the final solution!
Irreducible States

Recall: unification simplifies equations step-by-step until
• there are no equations left to simplify:

\[(S, \{\})\]

no constraints left. 
S is the final solution!

• or we find basic equations are inconsistent:
  • int = bool
  • s1 -> s2 = int
  • s1 -> s2 = bool
  • a = s \quad (s \text{ contains } a)

(or is symmetric to one of the above)

In the latter case, the program does not type check.
TYPE INFERENCE
MORE DETAILS
Where do we introduce polymorphic values? Consider:

\[ g \left( \text{fun } x \rightarrow 3 \right) \]

It is tempting to do something like this:

\[ (\text{fun } x \rightarrow 3) : \forall a. a \rightarrow \text{int} \]

\[ g : (\forall a. a \rightarrow \text{int}) \rightarrow \text{int} \]

But recall last lecture: OCaml doesn’t have those sorts of types. If we aren’t careful, we run into decidability issues.
Where do we introduce polymorphic values?

In ML languages: Only when values bound in "let declarations"

```
g (fun x -> 3)                                No polymorphism for fun x -> 3!
```

```
let f : forall a. a -> int = fun x -> 3 in
  (f 7, f true)                                  Yes polymorphism for f!
```
Generalization

let f : forall a. a -> int = fun x -> 3 in (f 7, f true)  

Yes polymorphism for f!

How do we use polymorphic values with type forall a. a -> int?

Each time we use them, during inference generate a fresh type variable b and use f with this type: b -> int

Because we pick a fresh variable (b, c, d, e, ...) each time, those variables can be constrained separately and take on separate types.

eg, in the first case int and in the second case bool

Using a polymorphic value by substituting a type t for a is called **type instantiation**.
Generalization: More rules!

Where do we introduce polymorphic values?

```
let x = v
```

General rule:
- if \( v \) is a value (or guaranteed to evaluate to a value without effects)
  - OCaml has some rules for this
- and \( v \) has type scheme \( s \)
- and \( s \) has free variables \( a, b, c, \ldots \)
- and \( a, b, c, \ldots \) do not appear in the types of other values in the context
- then \( x \) can have type \( \forall a, b, c. s \)
Let Polymorphism

Where do we introduce polymorphic values?

let x = v

General rule:
• if v is a value (or guaranteed to evaluate to a value without effects)
  • OCaml has some rules for “guaranteed to evaluate to a value”
• and v has type scheme s
• and s has free variables a, b, c, ...
• and a, b, c, ... do not appear in the types of other values in the context
• then x can have type forall a, b, c. s

That’s a hell of a lot more complicated than you thought, eh?
Consider this function $f$ – a fancy identity function:

```ocaml
let f = fun x ->
    let y = x in
    y
```

A sensible type for $f$ would be:

$$f : \forall a. a \to a$$
Consider this function $f$ – a fancy identity function:

```
let f = fun x ->
    let y = x in
    y
```

A bad (unsound) type for $f$ would be:

```
f : forall a, b. a -> b
```
Consider this function $f$ – a fancy identity function:

```
let f = fun x ->
  let y = x in
  y
```

A bad (unsound) type for $f$ would be:

$f : \forall a, b. a \rightarrow b$

$(f \text{ true}) + 7$

goes wrong! but if $f$ can have the bad type, it all type checks. This *counterexample* to soundness shows that $f$ can’t possible be given the bad type safely
Now, consider doing type inference:

```ocaml
let f = fun x -> let y = x in y
```

```ocaml
x : a
```
Now, consider doing type inference:

```
let f = fun x -> let y = x in y
```

suppose we generalize and allow $y : \forall a. a$
Now, consider doing type inference:

```
let f = fun x -> let y = x in y
```

then we can use `y` as if it has any type, such as `y : b`.

suppose we generalize and allow `y : \forall a.a`
Let's consider doing type inference:

```
let f = fun x -> let y = x in y
```

We suppose we generalize and allow `y : forall a.a`. Then we can use `y` as if it has any type, such as `y : b`.

But now we have inferred that `(fun x -> ...) : a -> b` and if we generalize again, `f : forall a,b. a -> b`.

That's the bad type!
Now, consider doing type inference:

```
let f = fun x -> let y = x in y
```

If we suppose we generalize and allow

```
y :forall a.a
```

this was the bad step — `y` can’t really have any type at all. Its type has got to be the same as whatever the argument `x` is.

`x` was in the context when we tried to generalize `y`!
The Value Restriction

let x = v

this has got to be a value to enable polymorphic generalization
Unsound Generalization Again

let \( x = \text{ref} [\] \) in

\[ x : \forall a . a \text{ list ref} \]

not a value!
let x = ref [] in
x := [true];

x : forall a . a list ref

use x at type bool as if x : bool list ref

not a value!
let $x = \text{ref} \; []$ in

$x := [\text{true}];$

$\text{List.hd} \; (!x) + 3$

$x : \forall a \cdot \text{a list ref}$

use $x$ at type $\text{bool}$ as if $x : \text{bool list ref}$

use $x$ at type $\text{int}$ as if $x : \text{int list ref}$

and we crash ....
What does OCaml do?

```ocaml
let x = ref [] in
```

`x : '_weak1 list ref`

A “weak” type variable can’t be generalized means “I don’t know what type this is but it can only be one particular type.”

Look for the “_” to begin a type variable name.
What does OCaml do?

```ocaml
let x = ref [] in
x := [true];
```

The "weak" type variable is now fixed as a bool and can't be anything else.

bool was substituted for '_weak during type inference.
What does OCaml do?

```ocaml
let x = ref [] in
x := [true];

List.hd (!x) + 3
```

```
x : '_weak1 list ref

x : bool list ref

Error: This expression has type bool but an expression was expected of type int
```

type error ...

One other example

notice that the RHS is now a value – it happens to be a function value but any sort of value will do

let x = fun () -> ref [] in

now generalization is allowed

x : forall 'a. unit -> 'a list ref
One other example

notice that the RHS is now a value – it happens to be a function value but any sort of value will do

let x = fun () -> ref [] in

x () := [true];

now generalization is allowed

x : forall 'a. unit -> 'a list ref

x () : bool list ref
One other example

notice that the RHS is now a value – it happens to be a function value but any sort of value will do

```
let x = fun () -> ref [] in
x () := [true];

List.hd (!x ()) + 3
```

what is the result of this program?

now generalization is allowed

x : forall 'a. unit -> 'a list ref

x () : bool list ref

x () : int list ref
One other example

notice that the RHS is now a value – it happens to be a function value but any sort of value will do

let x = \( \text{fun } () \rightarrow \text{ref [ ] in} \)

\( x () : \text{forall 'a. unit } \rightarrow 'a \text{ list ref} \)

\( x () : \text{bool list ref} \)

\( x () : \text{int list ref} \)

now generalization is allowed

\[
\text{List.hd (!x ()) + 3}
\]

what is the result of this program?

List.hd raises an exception because it is applied to the empty list. why?
One other example

notice that the RHS is now a value – it happens to be a function value but any sort of value will do

let \( x = \text{fun () -> ref []} \) in

\( x () := [\text{true}]; \)

\( \text{List.hd (!x () + 3} \)

what is the result of this program?

List.hd raises an exception because it is applied to the empty list. why?
And yet another example

let f = g x

Can we give f a (strong) polymorphic type?

I don’t see any references around ...
And yet another example

let \( f = g \ x \)

Can we give \( f \) a (strong) polymorphic type?

I don’t see any references around ...

No – \( g \) could contain references. “\( g \ x \)” is not a value. \( f \) will have a weakly polymorphic type (at best) in OCaml.

Watch for this in your assignment.
And yet another example

Sometimes, you can change this:

\[
\text{let } f = g \ x
\]

to something like:

\[
\text{let } f = \text{fun } () \rightarrow g \ x
\]

this: \[
\text{let } f = \text{fun } () \rightarrow g \ x
\]

or this: \[
\text{let } f () = g \ x
\]

Now the right-hand side is a value (a function value)
TYPE INFERENCE: THINGS TO REMEMBER
Type Inference: Things to remember

**Declarative algorithm**: Given a context $G$, and untyped term $u$:

- Find $e$, $t$, $q$ such that $G |- u ==> e : t$, $q$
  - understand the constraints that need to be generated

- Find substitution $S$ that acts as a solution to $q$ via unification
  - if no solution exists, there is no way to type check the expression
  - unification will find the best (ie, the *principle*) solution if one exists

- Apply $S$ to $e$, ie our solution is $S(e)$
  - $S(e)$ contains schematic type variables $a,b,c$, etc

- If desired, use the type checking algorithm to validate
In order to introduce polymorphic quantifiers, remember:

- Quantifiers must be on the outside only
  - this is called “prenex” quantification
  - otherwise, type inference may become undecidable

- Quantifiers can only be introduced at let bindings:
  - let \( x = v \)
  - only the type variables that do not appear in the environment may be generalized
  - if \( x \) has type \( \forall a.t \), when \( x \) is used, generate fresh variable \( b \) and assume \( x \) has type \( t[b/a] \), continue type inference.

- The expression on the right-hand side must be a value
  - no references or exceptions or function calls that might contain such things
TYPE SYSTEMS:
ONE MORE THING THAT IS REALLY NIFTY
Type Checking Rules

\[ x_1 : t_1 \ldots x_n : t_n \vdash x_i : t_i \]

```
use an assumption from the context
```

\[ G, x : t_1 \vdash e : t_2 \]

```
a function has type \( t_1 \to t_2 \)
if when you assume \( x : t_1 \), you can show the body has type \( t_2 \)
```

\[ G \vdash \lambda x : t. e : t_1 \to t_2 \]

```
show a call has type \( t_2 \)
by proving the function has type \( t_1 \to t_2 \) and the argument has type \( t_1 \)
```

\[ G \vdash e_1 : t_1 \to t_2 \quad G \vdash e_2 : t_1 \]

\[ G \vdash e_1 e_2 : t_2 \]
Remarkably, these type checking rules are also the rules of basic (constructive) logic.
Instead of thinking of “A -> B” as a function type, think of it as the logical formula “A implies B”
Logical Rules

\[ x_1 : t_1 \ldots x_n : t_n \vdash x_i : t_i \]

“use an assumption from the context”

\[ \Gamma, x : t_1 \vdash e : t_2 \quad \Gamma \vdash \lambda x : t. e : t_1 \rightarrow t_2 \]

“prove \( t_1 \rightarrow t_2 \) by assuming \( t_1 \), and proving \( t_2 \)”

\[ \Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1 \]

\[ \Gamma \vdash e_1 \ e_2 : t_2 \]

“prove \( t_2 \) by proving \( t_1 \rightarrow t_2 \) and by proving \( t_1 \)”

“modus ponens”
When presenting rules of logic, it is common to leave out the expressions.
Logical Proofs

Rules:

\[
A_1, \ldots, A_n \vdash A_i
\]

\[
G, A \vdash B
\]

\[
A, A \vdash B \vdash A \rightarrow B
\]

\[
A, A \rightarrow B \rightarrow B
\]

\[
A \vdash (A \rightarrow B) \rightarrow B
\]

\[
\vdash A \rightarrow (A \rightarrow B) \rightarrow B
\]

A Proof:

The Corresponding Program:

\[
\lambda x : A. \lambda f : A \rightarrow B. f x
\]
The Curry-Howard Isomorphism is the observation that proofs and programs have similar structure.
### Curry-Howard Isomorphism

<table>
<thead>
<tr>
<th>Concept in Programming Languages</th>
<th>Concept in Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
<td>proof</td>
</tr>
<tr>
<td>type</td>
<td>theorem</td>
</tr>
<tr>
<td>inhabited type</td>
<td>true theorem</td>
</tr>
<tr>
<td>function type</td>
<td>implication</td>
</tr>
<tr>
<td>pair type</td>
<td>conjunction</td>
</tr>
<tr>
<td>union type (ie: data type)</td>
<td>disjunction</td>
</tr>
<tr>
<td>universal polymorphism</td>
<td>universal quantifier</td>
</tr>
<tr>
<td>program execution</td>
<td>proof simplification</td>
</tr>
</tbody>
</table>
There is much more to the Curry-Howard isomorphism.


The Curry-Howard isomorphism suggests ideas developed in logic may be useful in understanding programming languages and vice versa.

Many theorem proving/verification environments are based on the interplay between logic and programming.

Logicians were developing programming language concepts before computers existed!