Type Checking

COS 326

David Walker

Princeton University
Implementing an Interpreter

```
let x = 3 in
x + x
```

- Parsing
- Let ("x", Num 3, Binop(Plus, Var "x", Var "x"))
- Evaluation
- Num 6
- Pretty Printing
- 6

```
Num 6
```

```
Pretty Printing
```

```
6
```
Implementing an Interpreter

```
let x = 3 in
x + x
```

- **Parsing**
- **Type Checking**
- **Evaluation**
- **Pretty Printing**

```
Let ("x",
Num 3,
Binop(Plus, Var "x", Var "x"))
```

```
Num 6
```

```
Pretty Printing
```

```
6
```

```
Type Checking
```

```
Parsing
```

```
Evaluation
```

```
Num 6
```

```
Pretty Printing
```

```
6
```
type t = IntT | BoolT | ArrT of t * t

type x = string (* variables *)
type c = Int of int | Bool of bool
type o = Plus | Minus | LessThan

type e =
    Const of c
  | Op of e * o * e
  | Var of x
  | If of e * e * e
  | Fun of x * typ * e
  | Call of e * e
  | Let of x * e * e
type \( t \) = \text{IntT} \mid \text{BoolT} \mid \text{ArrT of } t \times t \\

type \( x \) = \text{string} (* \text{variables} *) \\

type \( c \) = \text{Int of int} \mid \text{Bool of bool} \\

type \( o \) = \text{Plus} \mid \text{Minus} \mid \text{LessThan} \\

type \( e \) = 
  \begin{array}{l}
    \text{Const of } c \\
    \mid \text{Op of } e \times o \times e \\
    \mid \text{Var of } x \\
    \mid \text{If of } e \times e \times e \\
    \mid \text{Fun of } x \times \text{typ} \times e \\
    \mid \text{Call of } e \times e \\
    \mid \text{Let of } x \times e \times e \\
  \end{array}

\text{Notice that we require a type annotation here.}

\text{We'll see why this is required for our type checking algorithm later.}
Language Syntax (BNF Definition)

t ::=
  int | bool | t -> t

b -- ranges over booleans
n -- ranges over integers
x -- ranges over variable names
c ::= n | b
o ::= + | - | <
e ::= c | e o e | x | if e then e else e | λx:t.e | e e | let x = e in e

type t = IntT | BoolT | ArrT of t * t

type x = string (* variables *)
type c = Int of int | Bool of bool
type o = Plus | Minus | LessThan

type e =
  Const of c
  | Op of e * o * e
  | Var of x
  | If of e * e * e
  | Fun of x * typ * e
  | Call of e * e
  | Let of x * e * e
When defining how evaluation worked, we used this notation:

\[
\begin{align*}
e_1 &\rightarrow^* \lambda x.e \\
e_2 &\rightarrow^* v_2 \\
e[v_2/x] &\rightarrow^* v \\
e_1 e_2 &\rightarrow^* v
\end{align*}
\]

In English:

“if \( e_1 \) evaluates to a function with argument \( x \) and body \( e \) and \( e_2 \) evaluates to a value \( v_2 \) and \( e \) with \( v_2 \) substituted for \( x \) evaluates to \( v \) then \( e_1 \) applied to \( e_2 \) evaluates to \( v \)”

And we were also able to translate each rule into 1 case of a function in OCaml. Together all the rules formed the basis for an interpreter for the language.
This notation:

\[ e \rightarrow^* v \]

was read in English as "e evaluates to v."

It described a relation between two things – an expression e and a value v. (And e was related to v whenever e evaluated to v.)

Note also that we usually thought of e on the left as "given" and the v on the right as computed from e (according to the rules).
The typing judgement

This notation:

\[ G \vdash e : t \]

is read in English as "e has type t in context G." It is going to define how type checking works.

It describes a relation between three things – a type checking context G, an expression e, and a type t.

We are going to think of G and e as given, and we are going to compute t. The typing rules are going to tell us how.
What is the type checking context \( G \)?

Technically, I'm going to treat \( G \) as if it were a (partial) function that maps variable names to types. Notation:

\[
\begin{align*}
G(x) & \quad \text{-- look up x's type in } G \\
G, x : t & \quad \text{-- extend } G \text{ so that } x \text{ maps to } t
\end{align*}
\]

When \( G \) is empty, I'm just going to omit it. So I'll sometimes just write: \( \vdash e : t \)
Here's an example context:

x:int, y:bool, z:int

Think of a context as a series of "assumptions" or "hypotheses"

Read it as the assumption that "x has type int, y has type bool and z has type int"

In the substitution model, if you assumed x has type int, that means that when you run the code, you had better actually wind up substituting an integer for x.
Typing Contexts and Free Variables

One more bit of intuition:

If an expression e contains free variables x, y, and z then we need to supply a context G that contains types for at least x, y and z. If we don't, we won't be able to type check e.
Goal: Give rules that define the relation "G |- e : t". 

To do that, we are going to give one rule for every sort of expression.

(We can turn each rule into a case of a recursive function that implements it pretty directly.)
Typing Contexts and Free Variables

Rule for constant booleans:

\[ G \vdash b : \text{bool} \]

English:
“boolean constants \( b \) \textit{always} have type \text{bool}, no matter what the context \( G \) is"
Typing Contexts and Free Variables

```plaintext
t ::= int | bool | t -> t

c ::= n | b

o ::= + | - | <

e ::= c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e

Rule for constant integers:

\[ G |- n : \text{int} \]

English:

“integer constants n *always* have type int, no matter what the context G is"
Typing Contexts and Free Variables

t ::= int | bool | t -> t

c ::= n | b

o ::= + | - | <

e ::= c
| e o e
| x
| if e then e else e
| \(\lambda\) x : t . e
| e e
| let x = e in e

Rule for operators:

\[
\begin{align*}
G |- e_1 : t_1 & \quad G |- e_2 : t_2 \quad \text{optype}(o) = (t_1, t_2, t_3) \\
\hline
G |- e_1 o e_2 : t_3
\end{align*}
\]

where

\[
\begin{align*}
\text{optype (+)} &= (\text{int}, \text{int}, \text{int}) \\
\text{optype (-)} &= (\text{int}, \text{int}, \text{int}) \\
\text{optype (<)} &= (\text{int}, \text{int}, \text{bool})
\end{align*}
\]

English:

“\(e_1 o e_2\) has type \(t_3\), if \(e_1\) has type \(t_1\), \(e_2\) has type \(t_2\) and \(o\) is an operator that takes arguments of type \(t_1\) and \(t_2\) and returns a value of type \(t_3\)"
Typing Contexts and Free Variables

\[ t ::= \text{int} \mid \text{bool} \mid t \rightarrow t \]
\[ c ::= n \mid b \]
\[ o ::= + \mid - \mid < \]
\[ e ::= c \mid e \circ e \mid x \mid \text{if } e \text{ then } e \text{ else } e \mid \lambda x: t. e \mid e \circ e \mid \text{let } x = e \text{ in } e \]

**Rule for variables:**

\[
G \vdash x : G(x) 
\]

**English:**

“variable \( x \) has the type given by the context"

Note: this rule explains (part) of why the context needs to provide types for all of the free variables in an expression.
Typing Contexts and Free Variables

<table>
<thead>
<tr>
<th>t ::= int</th>
<th>bool</th>
<th>t -&gt; t</th>
</tr>
</thead>
<tbody>
<tr>
<td>c ::= n</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>o ::= +</td>
<td>-</td>
<td>&lt;</td>
</tr>
<tr>
<td>e ::= c</td>
<td>e o e</td>
<td>x</td>
</tr>
</tbody>
</table>

**Rule for if:**

\[
\begin{align*}
&G |- e1 : \text{bool} && G |- e2 : t && G |- e3 : t \\
&G |- \text{if e1 then e2 else e3 : t}
\end{align*}
\]

**English:**

“if e1 has type bool and e2 has type t and e3 has (the same) type t then e1 then e2 else e3 has type t"
Typing Contexts and Free Variables

Rule for functions:

$$\frac{G, x:t |- e : t2}{G |- \lambda x:t.e : t \rightarrow t2}$$

English:

"if G extended with x:t proves e has type t2 then \( \lambda x:t.e \) has type t -> t2"
Typing Contexts and Free Variables

\[ t ::= \text{int} | \text{bool} | t \rightarrow t \]
\[ c ::= n | b \]
\[ o ::= + | - | < \]
\[ e ::= \]
\[ c \]
\[ | e \circ e \]
\[ | x \]
\[ | \text{if } e \text{ then } e \text{ else } e \]
\[ | \lambda x : t . e \]
\[ | e \ e \]
\[ | \text{let } x = e \text{ in } e \]

Rule for function call:

\[
\begin{align*}
G |\vdash e_1 : t_1 \rightarrow t_2 & \quad G |\vdash e_2 : t_1 \\
G |\vdash e_1 \ e_2 : t_2
\end{align*}
\]

English:

“if \( G \) extended with \( x : t \) proves \( e \) has type \( t_2 \) then \( \lambda x : t . e \) has type \( t \rightarrow t_2 \)"
Typing Contexts and Free Variables

\[ t ::= \text{int} \mid \text{bool} \mid t \rightarrow t \]

\[ c ::= n \mid b \]

\[ o ::= + \mid - \mid < \]

\[ e ::= \]
\[ c \]
\[ | e \circ e \]
\[ | x \]
\[ | \text{if e then e else e} \]
\[ | \lambda x : t . e \]
\[ | e \circ e \]
\[ | \text{let x = e in e} \]

Rule for let:

\[
\begin{align*}
G & \vdash e_1 : t_1 & G,x : t_1 & \vdash e_2 : t_2 \\
G & \vdash \text{let x = e}_1 \text{ in e}_2 : t_2
\end{align*}
\]

English:

“if e_1 has type t_1 and G extended with x:t_1 proves e_2 has type t_2 then let x = e_1 in e_2 has type t_2 "
A typing derivation is a "proof" that an expression is well-typed in a particular context.

Such proofs consist of a tree of valid rules, with no obligations left unfulfilled at the top of the tree. (ie: no axioms left over).

notice that “int” is associated with x in the context

\[
\begin{align*}
G, x: \text{int} & \vdash x : \text{int} \quad G, x: \text{int} \vdash 2 : \text{int} \\
G, x: \text{int} & \vdash x + 2 : \text{int} \\
G & \vdash \lambda x: \text{int. } x + 2 : \text{int} \to \text{int}
\end{align*}
\]
Key Properties

Good type systems are *sound*.

- ie, well-typed programs have "well-defined" evaluation
  - ie, our interpreter should not raise an exception part-way through because it doesn't know how to continue evaluation
  - colloquial phrase: “sound type systems do not go wrong”

Examples of OCaml expressions that go wrong:

- true + 3 (addition of booleans not defined)
- let (x,y) = 17 in ... (can’t extract fields of int)
- true (17) (can’t use a bool as if it is a function)

Sound type systems *accurately* predict run time behavior

- if e : int and e terminates then e evaluates to an integer
Soundness = Progress + Preservation

Proving soundness boils down to two theorems:

**Progress Theorem:**
If |- e : t then either:
(1) e is a value, or
(2) e --> e'

**Preservation Theorem:**
If |- e : t and e --> e' then |- e' : t

See COS 510 for proofs of these theorems.

But you have most of the necessary techniques:
Proof by induction on the structure of ...
... various inductive data types. :-}
The typing rules also define an algorithm for... type checking...

If you view G and e as inputs, the rules for “G |- e : t” tell you how to compute t (see demo code)
TYPE INFERENCE
For three distinct and complete achievements:

1. LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;

2. ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;

3. CCS, a general theory of concurrency.

In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.
Robin Milner

For three distinct and complete achievements:

1. LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;

2. ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;

3. CCS, a general theory of concurrency.

In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.

We will be studying Hindley-Milner type inference. Discovered by Hindley, rediscovered by Milner. Formalized by Damas. Broken several times when effects were added to ML.
The ML language and type system is designed to support a very strong form of type inference.

```
let rec map f l =
    match l with
    [ ] -> [ ]
    | hd::tl -> f hd :: map f tl
```

It’s very convenient we don’t have to annotate f and l with their types, as is required by our type checking algorithm.
The ML language and type system is designed to support a very strong form of type inference.

```
let rec map f l =
    match l with
    [ ] -> [ ]
  | hd::tl -> f hd :: map f tl
```

ML finds this type for map:

```
map : ('a -> 'b) -> 'a list -> 'b list
```
The ML language and type system is designed to support a very strong form of type inference.

```ml
let rec map f l =
  match l with
  [ ] -> [ ]
| hd::tl -> f hd :: map f tl
```

ML finds this type for map:

```
map : ('a -> 'b) -> 'a list -> 'b list
```

which is really an abbreviation for this type:

```
map : forall 'a,'b.('a -> 'b) -> 'a list -> 'b list
```
We call this type the *principle type (scheme)* for `map`.

Any other ML-style type you can give `map` is *an instance* of this type, meaning we can obtain the other types via *substitution* of types for parameters from the principle type.

**Eg:**
- `(bool -> int) -> bool list -> int list`
- `('a -> int) -> 'a list -> int list`
- `('a -> 'a) -> 'a list -> 'a list`
Principle types are great:

- the type inference engine can make a *best choice* for the type to give an expression
- the engine doesn't have to guess (and won't have to guess wrong)

The fact that principle types exist is surprisingly brittle. If you change ML's type system a little bit in either direction, it can fall apart.
Suppose we take out polymorphic types and need a type for id:

```ocaml
let id x = x
```

Then the compiler might guess that id has one (and only one) of these types:

```ocaml
id : bool -> bool
id : int -> int
```
Suppose we take out polymorphic types and need a type for id:

```ocaml
let id x = x
```

Then the compiler might guess that id has one (and only one) of these types:

```
id : bool -> bool
id : int -> int
```

But later on, one of the following code snippets won't type check:

```
id true
id 3
```

So whatever choice is made, a different one might have been better.
We showed that removing types from the language causes a failure of principle types.

Does adding more types always make type inference easier?
We showed that removing types from the language causes a failure of principle types.

Does adding more types always make type inference easier?

Nope!
OCaml has universal types on the outside ("prenex quantification"): 

forall 'a,'b. (('a -> 'b) -> 'a list -> 'b list)

It does not have types like this:

((forall 'a.'a -> int) -> int -> bool)

argument type has its own polymorphic quantifier
Consider this program:

```ocaml
let f g = (g true, g 3)
```

notice that parameter g is used inside f as if:

- 1. it’s argument can have type bool, \textit{AND}
- 2. it’s argument can have type int
Consider this program:

```plaintext
let f g = (g true, g 3)
```

notice that parameter g is used inside f as if:

1. it’s argument can have type bool, AND
2. it’s argument can have type int

Does the following type work?

```plaintext
('a -> int) -> int * int
```
Consider this program:

\[
\text{let } f \ g = (g \text{ true}, \ g \ 3)
\]

notice that parameter \( g \) is used inside \( f \) as if:

1. it’s argument can have type \( \text{bool} \), \textbf{AND}
2. it’s argument can have type \( \text{int} \)

Does the following type work?

\[
(\ 'a \rightarrow \text{int} ) \rightarrow \text{int} \ast \text{int}
\]

\textbf{NO:} this says \( g \)'s argument can be any type \( 'a \) (it could be \( \text{int} \) or \( \text{bool} \))

\textit{Consider} \( g \) is (fun \( x \rightarrow x + 2 \) : int -> int.
Unfortunately, \( f \ g \) goes wrong when \( g \) applied to true inside \( f \).
Consider this program again:

```ocaml
let f g = (g true, g 3)
```

We might want to give it this type:

```ocaml
f : (forall a.a->a) -> bool * int
```

Notice that the universal quantifier appears left of ->
System F is a lot like OCaml, except that it allows universal quantifiers in any position. It could type check f.

```
let f g = (g true, g 3)
```

```
f : (forall a.a->a) -> bool * int
```

Unfortunately, type inference in System F is undecidable.
System F is a lot like OCaml, except that it allows universal quantifiers in any position. It could type check f.

\[
\text{let } f \ g = (g \text{ true}, \ g \text{ 3})
\]

\[
f : (\forall a. a \rightarrow a) \rightarrow \text{bool} \times \text{int}
\]

Unfortunately, type inference in System F is undecideable.

Developed in 1972 by logician Jean Yves-Girard who was interested in the consistency of a logic of 2nd-order arithmetic.

Rediscovered as programming language by John Reynolds in 1974.
Even seemingly small changes can effect type inference.

Suppose "\+" operated on both floats and ints. What type for this?

```
let f x = x + x
```
Language Design for Type Inference

Even seemingly small changes can effect type inference.

Suppose "+" operated on both floats and ints. What type for this?

```ocaml
let f x = x + x

f : int -> int

f : float -> float
```
Even seemingly small changes can effect type inference.

Suppose "+" operated on both floats and ints. What type for this?

```plaintext
let f x = x + x

f : int -> int

f : float -> float

f : 'a -> 'a
```
Language Design for Type Inference

Even seemingly small changes can effect type inference.

Suppose "+" operated on both floats and ints. What type for this?

```plaintext
let f x = x + x

f : int -> int ?
f : float -> float ?
f : 'a -> 'a ?
```

No type in OCaml's type system works. In Haskell:

```plaintext
f : Num 'a => 'a -> 'a
```
INFERRING SIMPLE TYPES
A type scheme contains type variables that may be filled in during type inference

\[ s ::= a \mid \text{int} \mid \text{bool} \mid s \to s \]

A term scheme is a term that contains type schemes rather than proper types. eg, for functions:

\[
\text{fun (x:s) -> e}
\]

\[
\text{let rec f (x:s) : s = e}
\]
Two Algorithms for Inferring Types

Algorithm 1:
• Declarative; generates constraints to be solved later
• Easier to understand
• Easier to prove correct
• Less efficient, not used in practice

Algorithm 2:
• Imperative; solves constraints and updates as-you-go
• Harder to understand
• Harder to prove correct
• More efficient, used in practice
• See: http://okmij.org/ftp/ML/generalization.html
Algorithm 1

1) Add distinct variables in all places type schemes are needed
Algorithm 1

1) Add distinct variables in all places type schemes are needed

2) Generate constraints (equations between types) that must be satisfied in order for an expression to type check

   • Notice the difference between this and the type checking algorithm from last time. Last time, we tried to:
     • eagerly deduce the concrete type when checking every expression
     • reject programs when types didn't match. eg:

       \[ f \ e \quad \text{-- f's argument type must equal e} \]

   • This time, we'll collect up equations like:

     \[ a \to b = c \]
Algorithm 1

1) Add distinct variables in all places type schemes are needed

2) Generate constraints (equations between types) that must be satisfied in order for an expression to type check
   - Notice the difference between this and the type checking algorithm from last time. Last time, we tried to:
     - eagerly deduce the concrete type when checking every expression
     - reject programs when types didn't match. eg:

       \[
       \text{f e} \quad \text{-- f's argument type must equal e}
       \]
   - This time, we'll collect up equations like:

     \[
     \text{a -> b = c}
     \]

3) Solve the equations, generating substitutions of types for var's
Example: Inferring types for map

```ocaml
let rec map f l =
    match l with
    | [] -> []
    | hd::tl -> f hd :: map f tl
```

let rec map (f:a) (l:b) : c =
    match l with
    | [] -> []
    | hd::tl -> f hd :: map f tl
let rec map (f:a) (l:b) : c =
  match l with
  | []  -> []
  | hd::tl -> f hd :: map f tl
Step 2: Generate Constraints

```
let rec map (f:a) (l:b) : c =
  match l with
  | [] -> []
  | hd::tl -> f hd ::: map f tl
```

final constraints:
- \( b = b' \) list
- \( b = b'' \) list
- \( b = b''' \) list
- \( a = a \)
- \( b = b''' \) list
- \( a = b'' \rightarrow a' \)
- \( c = c' \) list
- \( a' = c' \)
- \( d \) list = \( c' \) list
- \( d \) list = \( c \)
Step 3: Solve Constraints

```
let rec map (f:a) (l:b) : c =
match l with
    [] -> []
| hd::tl -> f hd ::: map f tl
```

**final constraints:**

\[
\begin{align*}
b &= b' \text{ list} \\
b &= b'' \text{ list} \\
b &= b''' \text{ list} \\
a &= a \\
b &= b''' \text{ list} \\
a &= b'' \rightarrow a' \\
c &= c' \text{ list} \\
a' &= c' \\
d \text{ list} &= c' \text{ list} \\
d \text{ list} &= c
\end{align*}
\]

**final solution:**

\[
\begin{align*}
[b' \rightarrow c'/a] \\
[b' \text{ list}/b] \\
[c' \text{ list}/c]
\end{align*}
\]
Step 3: Solve Constraints

let rec map (f:a) (l:b) : c =
match l with
    [] -> []
| hd::tl -> f hd :: map f tl

final solution:
[b' -> c'/a]
[b' list/b]
[c' list/c]

let rec map (f:b' -> c') (l:b' list) : c' list =
match l with
    [] -> []
| hd::tl -> f hd :: map f tl
Step 3: Solve Constraints

```ml
let rec map (f:a) (l:b) : c =
    match l with
    | [] -> []
    | hd::tl -> f hd :: map f tl
```

renaming type variables:

```ml
let rec map (f:a -> b) (l:a list) : b list =
    match l with
    | [] -> []
    | hd::tl -> f hd :: map f tl
```
Type Inference Details

Type constraints are sets of equations between type schemes
- q ::= \{s_{11} = s_{12}, \ldots, s_{n1} = s_{n2}\}

- eg: \{b = b’ list, a = b \rightarrow c\}
Syntax-directed constraint generation

- our algorithm crawls over abstract syntax of untyped expressions and generates
  - a term scheme
  - a set of constraints
Syntax-directed constraint generation

- Our algorithm crawls over abstract syntax of untyped expressions and generates
  - a term scheme
  - a set of constraints

Algorithm defined as set of inference rules:

\[ G |--- u \Rightarrow e : t, q \]

- context
- unannotated expression
- annotated expression
- type (scheme)
- constraints that must be solved
Syntax-directed constraint generation

- our algorithm crawls over abstract syntax of untyped expressions and generates
  - a term scheme
  - a set of constraints

Algorithm defined as set of inference rules:

- \( G |- u \Rightarrow e : t, q \)

constraints that must be solved

context

unannotated expression

annotated expression

type (scheme)

in OCaml:

\[
\text{gen : ctxt} \rightarrow \text{exp} \rightarrow \text{ann_exp} \ast \text{scheme} \ast \text{constraints}
\]
Constraint Generation

Simple rules:

- $G \vdash x \Rightarrow x : s, \{\} \quad (\text{if } G(x) = s)$

- $G \vdash 3 \Rightarrow 3 : \text{int}, \{\} \quad (\text{same for other ints})$

- $G \vdash \text{true} \Rightarrow \text{true} : \text{bool}, \{\}$

- $G \vdash \text{false} \Rightarrow \text{false} : \text{bool}, \{\}$
If statements

G |-- u1 ==> e1 : t1, q1
G |-- u2 ==> e2 : t2, q2
G |-- u3 ==> e3 : t3, q3

G |-- if u1 then u2 else u3 ==> if e1 then e2 else e3

: a, q1 U q2 U q3 U \{t1 = bool, a = t2, a = t3\}
G |-- u1 ==> e1 : t1, q1
G |-- u2 ==> e2 : t2, q2             (for a fresh a)
---------------------------------------------
G |-- u1 u2 ==> e1 e2 : a, q1 U q2 U \{t1 = t2 -> a\}
Function Declaration

\[
G, x : a |\vdash u \Rightarrow e : t, q \quad \text{ (for fresh } a) \\
\hline \\
G |\vdash \text{fun } x \rightarrow e \Rightarrow \text{fun } (x : a) \rightarrow e : a \rightarrow b, \quad q \cup \{t = b\}
\]
G, f : a -> b, x : a |-- u ==> e : t, q           (for fresh a,b)

---------------------------------------------
G |-- rec f(x) = u ==> rec f (x : a) : b = e    :   a -> b, q U {t = b}
A solution to a system of type constraints is a substitution $S$

- a function from type variables to types
- assume substitutions are defined on all type variables:
  - $S(a) = a$ (for almost all variables $a$)
  - $S(a) = s$ (for some type scheme $s$)
- $\text{dom}(S) = \text{set of variables s.t. } S(a) \neq a$
Solving Constraints

A solution to a system of type constraints is a substitution $S$

- a function from type variables to types
- assume substitutions are defined on all type variables:
  - $S(a) = a$ (for almost all variables $a$)
  - $S(a) = s$ (for some type scheme $s$)
- $\text{dom}(S) =$ set of variables s.t. $S(a) \neq a$

We can also apply a substitution $S$ to a full type scheme $s$.

apply: $[ \text{int}/a, \text{int}\rightarrow\text{bool}/b ]$

to: $b \rightarrow a \rightarrow b$

returns: $(\text{int}\rightarrow\text{bool}) \rightarrow \text{int} \rightarrow (\text{int}\rightarrow\text{bool})$
When is a substitution $S$ a solution to a set of constraints?

Constraints: \{ $s_1 = s_2$, $s_3 = s_4$, $s_5 = s_6$, ... \}

When the substitution makes both sides of all equations the same.

Eg:

```
constraints:
 a = b -> c
 c = int -> bool
```
Substitutions

When is a substitution $S$ a solution to a set of constraints?

Constraints: \{ $s_1 = s_2$, $s_3 = s_4$, $s_5 = s_6$, ... \}

When the substitution makes both sides of all equations the same.

Eg:

<table>
<thead>
<tr>
<th>constraints:</th>
<th>solution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = b \rightarrow c$</td>
<td>$b \rightarrow (\text{int} \rightarrow \text{bool})/a$</td>
</tr>
<tr>
<td>$c = \text{int} \rightarrow \text{bool}$</td>
<td>$\text{int} \rightarrow \text{bool}/c$</td>
</tr>
<tr>
<td>$b/b$</td>
<td>$b/b$</td>
</tr>
</tbody>
</table>
When is a substitution $S$ a solution to a set of constraints?

Constraints: \{ $s_1 = s_2$, $s_3 = s_4$, $s_5 = s_6$, ... \}

When the substitution makes both sides of all equations the same.

Eg:

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

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

constraints with solution applied:  
\begin{align*}
b &\rightarrow (\text{int} \rightarrow \text{bool}) = b \rightarrow (\text{int} \rightarrow \text{bool}) \\
\text{int} &\rightarrow \text{bool} = \text{int} \rightarrow \text{bool}
\end{align*}
When is a substitution $S$ a solution to a set of constraints?

Constraints:  \{ s_1 = s_2, s_3 = s_4, s_5 = s_6, ... \}

When the substitution makes both sides of all equations the same.

A second solution

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

solution 1:
\[
\begin{align*}
b & \rightarrow (\text{int} \rightarrow \text{bool})/a \\
\text{int} & \rightarrow \text{bool}/c \\
b & /b
\end{align*}
\]

solution 2:
\[
\begin{align*}
\text{int} & \rightarrow (\text{int} \rightarrow \text{bool})/a \\
\text{int} & \rightarrow \text{bool}/c \\
\text{int} & /b
\end{align*}
\]
When is one solution better than another to a set of constraints?

constraints:

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

solution 1:

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

type b \rightarrow c with solution applied:

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

solution 2:

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

type b \rightarrow c with solution applied:

\[
\text{int} \rightarrow (\text{int} \rightarrow \text{bool})
\]
Solution 1 is "more general" – there is more flex.
Solution 2 is "more concrete"
We prefer solution 1.
Solution 1 is "more general" – there is more flex.
Solution 2 is "more concrete"
We prefer the more general (less concrete) solution 1.
Technically, we prefer T to S if there exists another substitution U and for all types t, $S(t) = U(T(t))$
There is always a *best* solution, which we can a *principle solution*. The best solution is (at least as) preferred as any other solution.
Examples

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 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

Unification simplifies equations step-by-step until

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

```plaintext
type ustate = substitution * constraints

unify_step : ustate -> ustate

unify_step (S, {bool=bool} U q)  =  (S, q)
unify_step (S, {int=int}      U 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

\[
type \text{ ustate} = \text{substitution} \times \text{constraints} \\
\text{unify\_step} : \text{ustate} -> \text{ustate}
\]

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

Unification simplifies equations step-by-step until

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

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

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

\[
\text{unify\_step} (S, \{A \rightarrow B = C \rightarrow D\} \cup q) = (S, \{A = C, B = D\} \cup q)
\]
Unification simplifies equations step-by-step until

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

Unification

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

\[
\text{unify\_step} \ (S, \ \{A \rightarrow B = C \rightarrow D\} \ U \ q)
\]

\[
= (S, \ \{A = C, \ B = D\} \ U \ q)
\]
Unification

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

*when a is not in FreeVars(s)*
Unification

Unify step \( S, \{a=s\} \cup q \) = ([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:

\[
\text{fun x -> x x}
\]

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

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

```
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:

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

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:
  - \(\text{int} = \text{bool}\)
  - \(s_1 \rightarrow s_2 = \text{int}\)
  - \(s_1 \rightarrow s_2 = \text{bool}\)
  - \(a = s\) \((s\) contains \(a)\)

(or is symmetric to one of the above)

In the latter case, the program does not type check.
TYPE INFERENCE
MORE DETAILS
Generalization

Where do we introduce polymorphic values? Consider:

```plaintext
g (fun x -> 3)
```

It is tempting to do something like this:

```plaintext
(fun x -> 3) : forall a. a -> int
```

```plaintext
g : (forall a. a -> int) -> int
```

But recall the beginning of the lecture:
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"

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

No polymorphism for \text{fun } x \rightarrow 3!

\[
\text{let } f : \forall a. a \rightarrow a = \text{fun } x \rightarrow 3 \text{ in } g f
\]

Yes polymorphism for \text{f}!
Where do we introduce polymorphic values?

**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 \( a, b, c, ... \)
- and \( a, b, c, ... \) do not appear in the types of other values in the context
- then \( x \) can have type \( \forall a, b, c. s \)
Let Polymorphism

Where do we introduce polymorphic values?

let x = v

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, ...
• 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?
Unsound Generalization Example

Consider this function f – a fancy identity function:

\[
\text{let } f = \text{fun } x \rightarrow \text{let } y = x \text{ in } y
\]

A sensible type for f would be:

\[
f : \text{forall } a. \ a \rightarrow a
\]
Unsound Generalization Example

Consider this function f – a fancy identity function:

\[
\text{let } f = \text{fun } x \rightarrow \text{let } y = x \text{ in } y
\]

A bad (unsound) type for f would be:

\[
f : \text{forall } a, b. \ a \rightarrow b
\]
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 -> b
```

```
(f 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:

\[
\text{let } f = \text{fun } x \rightarrow \text{let } y = x \text{ in } y
\]

\[x : a\]
Now, consider doing type inference:

```plaintext
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
```

Suppose we generalize and allow $y : \forall a.a$.

Then we can use $y$ as if it has any type, such as $y : b$. 

$x : a$
Now, consider doing type inference:

```ml
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 \( y : \text{forall } a.a \)

but now we have inferred that (\( \text{fun } x \to ... \)) : \( a \to b \)

and if we generalize again, \( f : \text{forall } a,b. \ a \to b \)

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

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

Suppose we generalize and allow `y : forall a.a`.

This was the bad step – `y` can’t really have any type at all. It’s 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

not a value!

let x = ref [] in

x : forall a . a list ref
Unsound Generalization Again

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

not a value!

x : forall a . a list ref
use x at type bool as if x : bool list ref
Unsound Generalization Again

```ocaml
let x = ref [] in
x := [true];
List.hd (!x) + 3
```

- `x : forall a . a list ref`
- `use x at type bool as if x : bool list ref`
- `use x at type int as if x : 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

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

now generalization is allowed

```ocaml
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?

```
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 = fun () -> ref [] in
  x () := [true];
  List.hd (!x ()) + 3
```

now generalization is allowed

```
x : forall 'a. unit -> 'a list ref
x () : bool list ref
x () : int list ref
```

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

what is the result of this program?

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

- Apply S to e, ie our solution is S(e)
  - S(e) contains schematic type variables a,b,c, etc that may be instantiated with any type

- Since S is principal, S(e) characterizes all reconstructions.

- 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

– The expression on the right-hand side must be a value
  • no references or exceptions