Polymorphism (Continued)
Announcements

• Midterm next Thurs (Feb 13) in class
  • Closed-book, Closed-notes
  • Practice questions at bottom of Schedule webpage

• Midterm Review Tues (Feb 11) in class
  • Come prepared with specific questions
Binary Search Trees

Node (key, value, left, right)

BST Property:
keys in left < key < keys in right
Binary Search Trees

BST Property:  \textit{keys in left} < \textit{key} < \textit{keys in right}

Node(“bob”, 13,
  Node(“alice”, 2, Leaf, Leaf),
  Node(“charlie”, 3, Leaf, Leaf))
Exercise: BST Lookup

BST Property: keys in left < key < keys in right

```ocaml
type ('a, 'b) tree =
  | Leaf
  | Node of 'a * 'b * ('a, 'b) tree * ('a, 'b) tree
```

Write a function to lookup keys...

```ocaml
val lookup : 'a -> ('a, 'b) tree -> 'b option
```

Try this at home!
Try this at home!

```
type ('a, 'b) weirdlist =
  Nil
  | Cons 'a * ('b, 'a) weirdlist
```

Pick a well-typed expression from:

(a) Cons(1, Cons(“a”, Cons(3.14, Nil)))
(b) Cons(1, Cons(“a”, Cons(1, Nil)))
(c) Cons(1, Cons(“a”, Cons(“a”, Nil)))
(d) Cons(1, Cons(1, Cons(“a”, Nil)))
(e) Cons(1, Cons(1, Cons(1, Nil)))
• **Container** data structures independent of type!

• Appropriate type is **instantiated** at each **use**

  • Static type checking catches errors early
    - Cannot add **int** key to **string** hashtable

• **Generics**: in Java, C#, VB (borrowed from ML)
What does foo evaluate to?

```ocaml
# let foo x =
  if x > 0 then
    x * x
  else
    failwith "x is negative" ;;
```

(a) Type Error
(b) int -> string = <fun>
(c) int -> int = <fun>
(d) int -> 'a = <fun>
(e) 'a -> 'b = <fun>
failwith

failwith : \a. string \rightarrow \a

• Call to failwith can be instantiated to produce any type

Call to failwith can be instantiated to produce any type

• Can be used to “escape with fatal error”

• Very useful for building up skeleton code

  • Can use failwith “…” for sub-expressions not yet written, but can still type check overall function!
Recall our trusty friend, fold

“fold-right”
\[ \text{a.k.a.} \ \text{List.fold\_right} \]

\[
\begin{align*}
\text{let rec } & \text{foldr } f \ \text{acc} \ \text{xs} = \\
& \text{match } \text{xs} \ \text{with} \\
& | [] \rightarrow \text{acc} \\
& | x::xs' \rightarrow f \ x \ (\text{foldr } f \ \text{acc} \ xs')
\end{align*}
\]

\[ ('a \rightarrow 'b \rightarrow 'b) \rightarrow 'a \ \text{list} \rightarrow 'b \rightarrow 'b \]

≈

\[
\forall 'a, 'b. \ (\text{('a \rightarrow 'b \rightarrow 'b) \rightarrow 'a} \ \text{list} \rightarrow 'b \rightarrow 'b)
\]
Recall our trusty friend, \texttt{fold}

\textbf{“fold-right”}
\texttt{a.k.a. List.fold\_right}

\begin{verbatim}
let rec foldr f acc xs =
  match xs with
  | []     -> acc
  | x::xs' -> f x (foldr f acc xs')
\end{verbatim}

\texttt{(’a \rightarrow ’b \rightarrow ’b) \rightarrow ’a list \rightarrow ’b \rightarrow ’b}

\textbf{“fold-left”}
\texttt{a.k.a. List.fold\_left}

\begin{verbatim}
let foldl f initAcc xs =
  let rec helper acc xs =
    match xs with
    | []     -> acc
    | x::xs’ -> helper (f acc x) xs’
  in
  helper initAcc xs
\end{verbatim}

\texttt{(’a \rightarrow ’b \rightarrow ’a) \rightarrow ’a \rightarrow ’b list \rightarrow ’a}

\begin{itemize}
  \item A bit annoying that the interfaces differ…
  \item Let’s write a version of fold-left of type:
\end{itemize}

\texttt{(’a \rightarrow ’b \rightarrow ’b) \rightarrow ’a list \rightarrow ’b \rightarrow ’b}
Recall our trusty friend, fold

“fold-right”
**a.k.a.** List.fold_right

```ocaml
let rec foldr f acc xs =
  match xs with
  | []     -> acc
  | x::xs'  -> f x (foldr f acc xs')
```

“fold-left”
**a.k.a.** List.fold_left

```ocaml
let foldl f initAcc xs =
  let rec helper acc xs =
    match xs with
    | []     -> acc
    | x::xs'  -> helper (f acc x) xs'
  in
  helper initAcc xs
```

Try this at home!

- Let’s write a version of fold-left of type:

  ```ocaml
  ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
  ```
Type Inference
• **Static types** are an abstract description of, or a prediction about, run-time values

• Polymorphic function and data types are particularly useful
How DOES OCaml figure out all the types?
Especially with polymorphism?!
Type Inference Strategy

Process each binding in order

\[ \text{let } x = e_1 \text{ in } e_2 \]

- Infer a type \( T_1 \) for \( e_1 \) (or \text{TYPE ERROR})
- Remember that \( x : T_1 \) and Infer a type \( T_2 \) for \( e_2 \)
To infer the type of an expression $e$:

1. Recursively traverse $e$ and collect “clues”
   - Clues are equality constraints between types
   - Introduce constraint variables ($K_1, K_2, K_3, ...$) for unknown types, which will be computed

2. Solve constraints
   - If no solution is possible, then TYPE ERROR
   - If a constraint variable $K$ is unconstrained, then generalize to polymorphic type variable (’a, ’b, ’c, ...)
1. Collecting Clues

Simple expressions are simple:

\[
\begin{align*}
1 & : \text{int} \\
\text{true} & : \text{bool} \\
(1, \text{true}) & : (\text{int} \times \text{bool})
\end{align*}
\]
1. Collecting Clues

More complex expressions:

If

\[ T_1 = T_2 = \text{int} \]

Then

\[ e1 + e2 : \text{int} \]
1. Collecting Clues

More complex expressions:

If

\[ T_1 = \text{bool} \]
\[ T_2 = T_3 \]
Then

\[ \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T_2 \]
1. Collecting Clues

More complex expressions:

If

\[ T_0 = \text{“the type of all the patterns”} \]
\[ T_1 = T_2 = \ldots = T_n \]

Then

\[
\text{match } e0 \text{ with }
\begin{align*}
| & \text{pattern1 } \rightarrow \ e1 \\
| & \text{pattern1 } \rightarrow \ e2 \\
| & \ldots \rightarrow \ldots
\end{align*} : T_1
\]
More complex expressions:

If

\[ T_1 \text{ is of the form } T_{11} \rightarrow T_{12} \]

\[ T_2 = T_{11} \]

Then

\[ e_1 \ e_2 : T_{12} \]
1. Collecting Clues

Generate constraint variables for (possibly) polymorphic expressions:

Generate new $K_{in}$ and $K_{out}$

Assume $x : K_{in}$

If

$T_1 = \text{“the solution for } K_{out}\text{”}$

Then

$\text{fun } x \rightarrow e1 : K_{in} \rightarrow K_{out}$
1. Collecting Clues

Generate constraint variables for (possibly) polymorphic expressions:

Generate new $K$

Then

$$\begin{array}{c}
\text{[ ] : K list}
\end{array}$$
2. Solving Constraints

Chase down all equalities (a.k.a. “unification”)

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Solution</th>
<th>Inferred Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_1 = \text{bool}$</td>
<td>$K_1 := \text{bool}$ $K_2 := \text{int}$</td>
<td>foo : $\text{bool} \rightarrow \text{int}$</td>
</tr>
<tr>
<td>$K_2 = \text{int}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$K_1 = \text{bool}$</td>
<td>$K_1 := \text{bool}$</td>
<td>foo : $\text{bool} \rightarrow 'a$</td>
</tr>
<tr>
<td>$K_2 = \text{int}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$K_1 = \text{bool}$</td>
<td></td>
<td>foo : TYPE ERROR</td>
</tr>
<tr>
<td>$K_2 = \text{int}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$K_1 = K_2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Type Inference Strategy

• 2 steps:
  • Generate constraints while traversing expression
  • Solve constraints

• In practice, both steps done simultaneously
  • On paper, as well as in implementation

• Let’s try some examples!
# let x = 2 + 3 ;;
# let x = 2 + 3 ;;
# let y = string_of_int x ;;
```ocaml
# let x = 2 + 3 ;;
# let y = string_of_int x ;;
# let inc z = x + y ;;
```

Constraints

- \( \text{int} = \text{int} \)
- \( \text{string} = \text{int} \)
- \( K_1 = \text{int} \)

Solution

- \( x : \text{int} \)
- \( y : \text{string} \)
- \( \text{inc} : K_z \rightarrow K_1 \)

Inferred Type

- \( K_1 = \text{int} \)

TYPE ERROR
• Notice the **shadowing**

• Rename to keep types/constraints separate
# let x = 2 + 3 ;;
# let y = string_of_int x ;;
# let inc y' = x + y' ;;

### Constraints

\[
\begin{align*}
\text{int} & = \text{int} \\
K_{y'} & = \text{int} \\
K_1 & = \text{int}
\end{align*}
\]

### Solution

\[
\begin{align*}
K_{y'} & := \text{int} \\
K_1 & := \text{int}
\end{align*}
\]

### Inferred Type

\[
\text{inc} : \text{int} \rightarrow \text{int}
\]
```ocaml
# let choose b x y = if b then x else y;;
```

`choose`:

- Constraints: $K_b = \text{bool}$, $K_x = K_y$, $K_x = K_1$
- Solution: $K_b := \text{bool}$, $K_y := K_x$, $K_x := \text{ANY}$, $K_1 := K_x$
- Inferred Type: $\text{choose} : \text{bool} \rightarrow 'a \rightarrow 'a \rightarrow 'a$
What's the type of `foo`?

```ocaml
# let foo b x y =
   if x = y then 0 else 1;;
```

(a) `int`
(b) `bool -> 'a -> 'a -> int`
(c) `'a -> bool -> bool -> int`
(d) `'a -> 'b -> 'b -> int`
(e) Type Error
What’s the type of \texttt{foo}?

\begin{verbatim}
# let foo b x y = if x = y then 0 else 1;;
\end{verbatim}

\texttt{foo} : $K_b \rightarrow K_x \rightarrow K_y \rightarrow K_1$

\textbf{Constraints}

\begin{itemize}
  \item $K_x = K_y$
  \item $K_1 = \text{int}$
\end{itemize}

\textbf{Solution}

\begin{itemize}
  \item $K_x := \text{ANY}$ \hspace{1cm} $K_b := \text{ANY}$
  \item $K_y := K_x$ \hspace{1cm} $K_1 := \text{int}$
\end{itemize}

\textbf{Inferred Type}

\texttt{foo} : $'a \rightarrow 'b \rightarrow 'b \rightarrow \text{int}$
What’s the type of `bar`?

```ocaml
# let bar x = 
    let (z, y) = x in 
    z - y;;
```

(a) `(int -> int) * int`
(b) `int -> int * int`
(c) `int * int -> int`
(d) `int -> int -> int`
(e) **Type Error**
What’s the type of `bar`?

```ocaml
# let bar x =
        let (z, y) = x in
        z - y;;
```

Bar takes `K x` and returns `K_1`:

- Introduce new constraint vars $K_z$ and $K_y$ for unknown tuple type.
What’s the type of `bar`?

```ocaml
# let bar x =
    let (z, y) = x in
    z - y;;
```

**Constraints**

\[
K_x = K_z \times K_y \\
K_z = \text{int} \\
K_y = \text{int} \\
K_1 = \text{int}
\]

**Solution**

\[
K_1 := \text{int} \\
K_y := \text{int} \\
K_z := \text{int} \\
K_x := \text{int} \times \text{int}
\]

**Inferred Type**

`foo : \text{int} \times \text{int} \rightarrow \text{int}`
Recursive Bindings

```ocaml
# let rec cat xs =
  match xs with
  | []      -> ""
  | x::xs'  -> x^(cat xs');
```

Constraints:

- Must be some kind of list
- Introduce new constraint var $K_2$

$K_{xs} = K_2 \text{ list}$
Recursive Bindings

```ocaml
# let rec cat xs =
    match xs with
    | []    -> ""
    | x::xs' -> x^(cat xs');;
```

Constraints

\[ K_{xs} = K_2 \text{ list} \]
\[ K_1 = \text{ string} \]
Recursive Bindings

```ocaml
# let rec cat xs =
    match xs with
    | [] -> ""
    | x::xs' -> x^(cat xs');;
```

Constraints

\[
K_{xs} = K_2 \text{ list} \quad K_{xs'}, = K_2 \text{ list}
\]

\[
K_1 = \text{string}
\]

\[
K_x = K_2
\]
Recursive Bindings

```ocaml
# let rec cat xs =
  match xs with
  | []   -> ""
  | x::xs' -> x^(cat xs');
```

Constraints:

- $K_{xs} = K_2 \text{ list}$
- $K_{xs'} = K_2 \text{ list}$
- $K_1 = \text{ string}$
- $K_x = \text{ string}$
- $K_x = K_2$
- $K_1 = \text{ string}$
# let rec cat xs =
    match xs with
    | [] -> ""
    | x::xs' -> x^(cat xs');;

cat : \( K_{xs} \rightarrow K_1 \)

Constraints

\[
\begin{align*}
K_{xs} &= K_2 \text{ list} \\
K_1 &= \text{ string} \\
K_x &= K_2 \\
K_{xs'} &= K_2 \text{ list} \\
K_x &= \text{ string} \\
K_1 &= \text{ string} \\
\end{align*}
\]

Solution

\[
\begin{align*}
K_1 &: \text{ string} \\
K_{xs} &: \text{ string list} \\
\end{align*}
\]

\( \text{cat} : \text{ string list} \rightarrow \text{ string} \)
# let rec map f xs =

match xs with

| [] -> []
| x::xs' -> (f x)^(map f xs');;

map : K_f \rightarrow K_{xs} \rightarrow K_1

Constraints

\[
\begin{align*}
K_{xs} &= K_2 \text{ list} \\
K_1 &= K_3 \text{ list} \\
K_x &= K_2 \\
K_{xs'} &= K_2 \text{ list} \\
K_1 &= \text{ string}
\end{align*}
\]

Solution

\[
\begin{align*}
K_f &= K_4 \rightarrow K_5 \\
K_x &= K_4 \\
K_5 &= \text{ string} \\
K_f &= K_f \\
K_{xs} &= K_{xs'}
\end{align*}
\]

TYPE ERROR!

\text{string}

not compatible with \( K_3 \text{ list} \)
# let pipe x f = f x;;

pipe : \(K_x \rightarrow K_f \rightarrow K_1\)

Inferred Type

pipe : 'a \rightarrow ('a \rightarrow 'b) \rightarrow 'b

Constraints

\[
\begin{align*}
K_f &= K_2 \rightarrow K_3 \\
K_x &= K_2 \\
K_3 &= K_1
\end{align*}
\]

Solution

\[
\begin{align*}
K_x &= \text{ANY} \\
K_2 &= K_x \\
K_3 &= \text{ANY} \\
K_1 &= K_3 \\
K_f &= K_x \rightarrow K_3
\end{align*}
\]
# let compose f g = g (f x);;

# let fool f g x =
    if f x
    then x
    else g x;;

# let foo2 f g x =
    if f x
    then x
    else foo2 f g (g x);;
What does `bar` evaluate to?

```ocaml
# let bar x = failwith "!!!";;

(a) Type Error
(b) 'a -> string = <fun>
(c) 'a -> 'a = <fun>
(d) 'a -> 'b = <fun>
```