Lecture 9: Type Inference

Polymorphic Data Structures
- Container data structures independent of type!
- Appropriate type is instantiated at each use:
  - `('a, 'b) list`
  - `('a, 'b) tree`
  - `('a, 'b) hashtbl`...
- Appropriate type instantiated at use:
  - No unsafe casting as in C++/Java
- Static type checking catches errors early
  - Cannot add `int` key to `string` hashtable
- Generics: in Java, C#, VB (borrowed from ML)

Other kinds of polymorphisms
- That was OCaml...
- But what about other kinds of polymorphisms..

News
- PA4 Updated
  - Do second question before midterm ...
- “Late Day” = 1 second after 5pm
- Midterm next week in class
Other kinds of polymorphisms

- Sub-type polymorphism
  ```java
  void f(Shape s)
  ```
  - Can pass in any sub-type of Shape
  ```java
  Circle c = new Circle();
  f(c);
  ```

- Parametric polymorphism
  ```java
  void proc elems(list[T])
  ```
  - Can pass in ANY T – this is the kind in OCaml!
  ```java
  bool ShapeEq(T a, T b) T extends Shape
  ```
  - Can call on
    - (Rect, Rect)
    - (Circle, Circle)
  - But not (Rect, Circle)

Bounded polymorphism
- Like parametric, except can provide a bound
  ```java
  void proc elems(list[T]) T extends Printable
  ```
  - Hey… isn’t this subtype polymorphism?
  - No, for example:
    ```java
    bool ShapeEq(T a, T b) T extends Shape
    ```
    - Can call on
      - (Rect, Rect)
      - (Circle, Circle)

Summary of polymorphism

- Subtype
- Parametric
  ```java
  Type Classes: Haskell
  ```
- Bounded = Parametric + Subtype
  (In Java/C#)

Back to OCaml

- Polymorphic types allow us to reuse code
- However, not always obvious from staring at code
- But… Types never entered w/ program!
Type inference

aka: how in the world does Ocaml figure out all the types ???

Polymorphic Types

- Polymorphic types are tricky
- Not always obvious from staring at code
- How to ensure correctness?
- Types (almost) never entered w/ program!

Polymorphic Type Inference

- Computing the types of all expressions
  - At compile time: Statically Typed

- Each binding is processed in order
  - Types are computed for each binding
  - For expression and variable bound to
  - Types used for subsequent bindings

- How is this different from values?
  Values NOT computed statically (e.g. fun values)
Example 1

\[ \text{let } x = 2 + 3;;\]
\[ \text{let } y = \text{string_of_int } x;;\]

Example 2

\[ \text{let } x = 2 + 3;;\]
\[ \text{let } y = \text{string_of_int } x;;\]
\[ \text{let } \text{inc } y = x + y;;\]
\[ \text{let } \text{inc } = \text{fun } y \rightarrow x+y\]

Example 3

\[ T_{\text{foo}} = T_x \rightarrow T_{\text{body}} = \text{int} \times \text{int} \rightarrow \text{int}\]

\[ \text{let foo } x = \]
\[ \text{let } (y,z) = x \text{ in } \]
\[ z-y \]

Example 4

\[ T_{\text{cat}} \equiv T_x \rightarrow T_{\text{body}} = \text{string list} \rightarrow \text{string}\]

\[ \text{let rec } \text{cat } l = \]
\[ \text{match } l \text{ with } \]
\[ [] \rightarrow "" \]
\[ | h::t \rightarrow h^\text{\textdollar}(\text{cat } t)\]

ML doesn’t know what the function does, or even that it finishes only its type!
Example 4
ML doesn’t know what the function does, or even that it finishes only its type!

```
let rec cat l =
  match l with
  | [] -> ""
  | h::t -> h^(cat t)
```

Example 5
(`'a -> 'b) -> 'a list -> 'b list

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

Example 6

```
let compose f g x = f (g x)
```

Example 7

```
let rec fold f cur xs =
  match xs with
  | [] -> cur
  | x::xs' -> fold f (f cur x) xs'
```
(In Class Exercise A)

\[ \text{split : } \alpha \text{ list} \rightarrow (\beta \text{ list} \times \alpha \text{ list}) \]

```plaintext
let rec split xs =
    match xs with
    | []  -> ([], [])
    | [x] -> ([x], [])
    | y::z::xs' ->
      let ys, zs = split xs' in
      (y::ys, z::zs)
```

(In Class Exercise B)

```plaintext
let rec merge xs ys =
    match (xs, ys) with
    | ([],_) -> ys
    | (_,[]) -> xs
    | (x::xs', y::ys') when x\leq y -> x :: (merge xs' ys)
    | (x::xs', y::ys') -> y :: (merge xs ys')
```

(In Class Exercise C)

```plaintext
let rec msort xs =
    match xs with
    | [] -> []
    | x::xs' ->
      let ys, zs = split xs in
      merge (msort ys) (msort zs)
```

Example 11

```plaintext
let foo1 f g x =
    if f x
    then x
    else g x
```
Example 12

```ml
let foo2 f g x =
  if f x
  then x
  else foo2 f g (g x)
```

```scss
val foo2 : 'a -> 'b -> ( 'a -> 'c ) -> 'a -> 'c
```