Admin stuff

- PA 4 due Friday Feb 11th 5pm
- Midterm Tue Feb 8th or Th Feb 10th
  - covers all of Ocaml, but only PA1-PA3

Recap of last week

- High-level iteration: Map and Fold
- Research corner: compiler correctness

Parametric types

aka: what’s up with those ‘a ???

Polymorphism

- Poly = many, morph = kind
- ‘a and ‘b are type variables!
- For-all types: For all ‘a, ‘b: ‘a * ‘b → ‘b * ‘a
- ‘a,’b can be instantiated with any type:
  w/ int,string : int * string → string * int
  w/ char, int list : char * int list → int list * char
  w/ int! int , bool : (int ! int) * bool → bool * (int ! int)

What is the deal with ‘a ?

These meta-functions have strange types:

map: ('a → 'b) → 'a list → 'b list
filter: ('a → bool) → 'a list → 'a list

Why?

Instantiation at Use

map: ('a → 'b) → 'a list → 'b list
let f x = x + 10;;
let fm = map f;;

let f x = x" like";;
let fm = map f ["cat"; "dog"; "burrito"];;
Instantiation at Use: be Careful

map: 

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

let \(f x = x^\text{"like"};\);
let \(fm = \text{map } f [1;2;3;4];\);

Polymorphic ML types

- Implicit for-all at the “left” of all types
  - Never printed out, or specified
  - map: For all \(\text{'a , 'b} \rightarrow \text{'a list} \rightarrow \text{'b list}\)

  Typing rule:

\[
\begin{align*}
\text{e: } T \text{'a,T} \quad e: T \text{'a,T} \\
\text{e: } \forall \text{'a,T} \\
\text{e: } \forall \text{'a,T} \quad \text{where } T \text{ does not contain type vars already in } T
\end{align*}
\]

Example

\[
\begin{align*}
\text{swap}(1,\text{"a"}) : \text{string} * \text{int}
\end{align*}
\]

Polymorphism enables Reuse

- Can reuse generic functions:
  - swap: \(\text{\text{'a} * \text{'b}} \rightarrow \text{\text{'b} * \text{'a}}\)
  - rev: \(\text{\text{'a list}} \rightarrow \text{\text{'a list}}\)
  - length: \(\text{\text{'a list}} \rightarrow \text{\text{int}}\)
  - filter: \(\text{\text{'a} → \text{bool}} \rightarrow \text{\text{'a list}} \rightarrow \text{\text{'a list}}\)
  - partition: \(\text{\text{'a} → \text{bool}} \rightarrow \text{\text{'a list}} \rightarrow \text{\text{('a list * \text{'a list)}}\}
  - map: \(\text{\text{'a} → \text{'b}} \rightarrow \text{\text{'a list}} \rightarrow \text{\text{'b list}}\)
Not just functions ...

• Data types are also polymorphic!

```ocaml
type 'a list =
  | Nil
  | Cons of ('a * 'a list)
```

• Type is instantiated for each use:

```ocaml
Cons(1,Cons(2,Nil)) : int list
Cons("a",Cons("b",Nil)) : string list
Cons((1,2),Cons((3,4),Nil)) : (int*int) list
Nil : 'a list
```

Datatypes with many type variables

• Multiple type variables

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

• Type is instantiated for each use:

```ocaml
Leaf("joe",1) : (string,int) tree
Leaf("william",2) : (string,int) tree
Node(…,...) : (string,int) tree
Node(Leaf("joe",1),Leaf(3.14, "pi")): (string,int) tree
```

Polymorphic Data Structures

• Container data structures independent of type!
• Appropriate type is instantiated at each use:

```ocaml
'a list
| ('a , 'b) Tree
| ('a , 'b) Hashtbl ...
```

• Appropriate type instantiated at use
  - No casting
• Static type checking catches errors early
  - Cannot add int key to string hashtable
• Generics: feature of Java,C#...

Other kinds of polymorphisms

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

• Sub-type polymorphism
  void f(Shape s)
  - Can pass in any sub-type of Shape

• Parametric polymorphism
  void proc_elements(list[T])
  - Can pass in ANY T
  - this is the kind in OCaml!

Other kinds of polymorphisms

• Bounded polymorphism
  - Like parametric, except can provide a bound
  void proc_elements(list[T]) WHERE T <= Printable
  - Hey... isn’t this subtype polymorphism?
  - Can’t I just do?
  void proc_elements(list[Printable])
  - Yes, in this case, but in general...

  \( \text{print} (T) \) \( \text{where } T \text{ is Printable} \)

  \( \text{print} (*p).\text{edit} \)

Other kinds of polymorphisms

• Bounded polymorphism
  - Say we have:
    \[ \text{print}\_\text{elem}\(T\) \text{WHERE } T \text{ <= Printable} \]
  - and we have
    • a Car car which is printable, and
    • a Shark shark which is printable
  - The following typechecks:
    • \(\text{print}\_\text{elem}\(\text{car}\).steering\_\text{wheel} = \text{null}\)
    • \(\text{print}\_\text{elem}\(\text{shark}\).teeth = \text{null}\)
  - But not if \(\text{print}\_\text{elem}\) returns Printable

Other kinds of polymorphisms

• Bounded polymorphism
  - Or as another example:
    \[ \text{bool ShapeEq}(T a, T b) \text{WHERE } T \text{ <= Shape} \]
  - Can call on
    • (Rect, Rect)
    • (Circle, Circle)
  - But not (Rect, Circle)

F-bounded polymorphism

```c
class Comparable<T>{
    bool less_than(T other) { return ... }
}

int main{
    int list[Comparable<T>]
    // Use Polymorphic Comparable Classes
    return 0
}
```
**F-bounded polymorphism**

```java
interface Comparable<T> {
    bool less_than(T);
}

void sort(list<T extends Comparable<T>>) {
    ...
}
```

**Summary of polymorphism**

- Subtype
- Parametric
- Bounded
- F-bounded

**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 for me???

**Example 1**

```ocaml
let x = 2 + 3;;
let y = string_of_int x;;
```

**Example 2**

```ocaml
let x = 2 + 3;;
let inc y = x + y;;
```
Example 2

```
let x = 2 + 3;;
let inc y = x + y;;
```

Example 3

```
let foo x =
  let (y,z) = x in
  z - y;;
```

Example 4

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

Example 5

```
let rec map f l =
  match l with
  | []    -> []
  | h::t  -> (f h)::(map f t)
```
Example 5

```ocaml
let rec map f l =
  match l with
  | [] -> []
  | h::t -> (f h)::(map f t)
```

Inferring types with ‘a

- Introduce unknown type vars
- Figure out equalities that must hold, and solve these equalities
- Remaining types vars get a forall and thus become the ‘a, ‘b, etc.

Example 6

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

Example 7

```ocaml
let rec fold f cur l =
  match l with
  | [] -> cur
  | h::t -> fold f (f h cur) t
```