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 3

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

Example 4

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

Example 5

```ml
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
```

Deconstructing OCaml

What makes up a language
Key components of a lang

- Units of computation
- Types
- Memory model

Units of computation

In OCaml

- Expressions that evaluate to values
- Everything is an expression
  - int, bool, real
  - if-then-else
  - let-in
  - match
  - fn x -> x+1
  - e1 e2

In Java/Python

- Store and update commands
- Message sends
In Prolog

- Logical facts
- Inference rules

\[
\begin{align*}
\text{Mexican(CARNITAS) } & \text{“Fact”} \\
\text{Food(CARNITAS) } & \text{“Fact”} \\
\text{Mexican(X) } & \land \text{Food(X) } \rightarrow \text{Delicious(X) } \text{“Rule”} \\
\text{Delicious(CARNITAS) } & \text{“Fact”}
\end{align*}
\]

Types

- Used to classify things created by the programmer
- Classification used to check what can be done with/to those things

In OCaml: Static typing

- Types are assigned statically at compile time
- Without computing values
- Rules state when expressions are type-correct

\[
\begin{align*}
e_1 : T_1 & \rightarrow T_2 \\
e_2 : T_1 \\
\end{align*}
\]

\[
\begin{align*}
e_1 \ e_2 : T_2
\end{align*}
\]

In OCaml: Static typing

- How can one reuse code for different types?
  - parametric types: ‘a * ‘b -> ‘b * ‘a
  - implicit forall
- Type “discovered” (inferred) automatically from code
  - less burden on the programmer
In Python: Dynamic typing

- Types assigned to values/objects as they are computed, i.e.: dynamically

- Before an operation is performed, check that operands are compatible with operation

Dynamic vs. Static, OO vs. Func

<table>
<thead>
<tr>
<th></th>
<th>Statically typed</th>
<th>Dynamically typed</th>
</tr>
</thead>
<tbody>
<tr>
<td>OO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functional</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Polymorphism

- Can a language be dynamically typed, but not polymorphic?
- Every dynamically typed language is polymorphic
  - functions just simply work on any datatype that can be operated on at runtime
- Only need explicit polymorphism in statically typed languages to assign at compile time a suitably general polymorphic type

Memory/Data model

aka: what do variables refer to?
Data model in functional langs

- Environment of bindings (phonebook)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>&quot;abc&quot;</td>
</tr>
<tr>
<td>z</td>
<td>(2,3)</td>
</tr>
</tbody>
</table>

- Never change a binding
  - add new bindings at the end of the phonebook

Data model in functional langs

- Variables are names that refer into the phonebook
- Most recent entry looked up during evaluation
- Environment “frozen” inside function value so that the behavior of the function cannot be changed later on (easier reasoning)

Data model in OO langs

- Variables are cells in memory
- Can change them by assigning into them
- Variables point to objects on the heap
- \( x = x + 10 \)

Data model in Prolog

- Variables in Prolog are unknowns to solve for
  - Mexican(CARNITAS)
  - Food(CARNITAS)
  - \( \forall X \) Mexican(X) \( \land \) Food(X) \( \rightarrow \) Delicious(X)
  - Delicious(Y)?

Q: What is delicious?
A: CARNITAS!

Final words on functional programming

What’s the point of all this?
Advantages of functional progs

- Functional programming more concise
  “one line of lisp can replace 20 lines of C”
  (quote from http://www.ddj.com/dept/architect/184445000/bgno-3)

- Recall reverse function in OCaml:
  
  ```
  let reverse = fold (::) [];;
  ```

- How many lines in C, C++?

Don’t be fooled

- Some of the programming assignments made you do certain things using fold in order to force you to think about it, even though using fold was not the easiest way to do it.

- But there are many cases where map and fold make life A LOT EASIER.

Can better reason about progs

- No side effects. Call a function twice with same params, produces same value

- As a result, computations can be reordered more easily

- They can also be parallelized more easily

So what?

- From the authors: “Inspired by similar primitives in LISP and other languages”

- The point is this: programmers who only know Java/C/C++ would probably not have come up with this idea

- Many other similar examples in industry

This stuff is for real: F#

F# = Microsoft’s Ocaml-on-steroids


- Why FP is way cool
- How FP works with Objects (C#)
- How FP allows you to write parallel code
  ... all with an extremely engaging speaker

Remember

- The next time you use google, think of how functional programming has inspired some of the technical ideas behind their engine

- And of course:
  “Free your mind”
  - Morpheus