Introduction to HOT languages

(no, sorry, not X-rated, but rather “higher order & typed”)
[with more than a bit of OCaml]

slides at: http://www.bononia.it/~zack/courses/somfosset0607/ocaml_hot.pdf

Master in Tecnologie del

Software Libero ed Open Source

http://www.almaweb.unibo.it/os_presentazione.html

Claudio Sacerdoti Coen <sacerdot@cs.unibo.it>

Stefano Zacchiroli <zack@cs.unibo.it>

Alma Graduate School – Università degli Studi di Bologna

13/10/2007

Outline

1. biodiversity in programming
2. why learn HOT programming languages
3. an OCaml tutorial (live)
4. (some) functional programming concepts
5. what's next?

Biodiversity in programming

- There is more than one way to skin a cat!
  - most of them in academia only ...
  - neither macho nor commercially supported
- If all you have is an hammer, everything becomes a nail!
  - but with a big hammer with many spare parts you do not miss the screwdriver
- Languages constrain the way we think!
- Everything is obvious… after you see it!

Biodiversity != niches

- Niches require ad-hoc languages
  - Operating systems and C
  - Interactive theorem provers and ML/Haskell
  - Artificial intelligence and Prolog/Lisp
- But most programs are outside niches!
  - Most (all?) languages can compete
  - Correctness and safety are the problems, not control and efficiency

The “commercial” world ...

- C, Pascal:
  - imperative, almost alike, same weak type system
- C++, Java, C#, Visual Basic, Delphi:
  - class based
- C++, Java:
  - templates/generics (recently)
- Scripting languages: even less typing
- Good language == bad language with large library

HOT languages

- HOT = Higher Order and Typed
- Higher Order == functional
  - Untyped: Lisp, Scheme, Miranda, ...
  - Typed: Standard ML, OCaml, Haskell, ...
  - Dependently typed: DML, Cayenne, Epigram, ...
- Typed ==
  - strongly typed, really!
  - highly polymorphic

Why Learn OCaml?

Or, Why Your Current Programming Language Sucks

This part of the talk is based on the slides of Brian Hurt, available here:
http://www.bogonomicon.org/bblog/ocaml.sxi

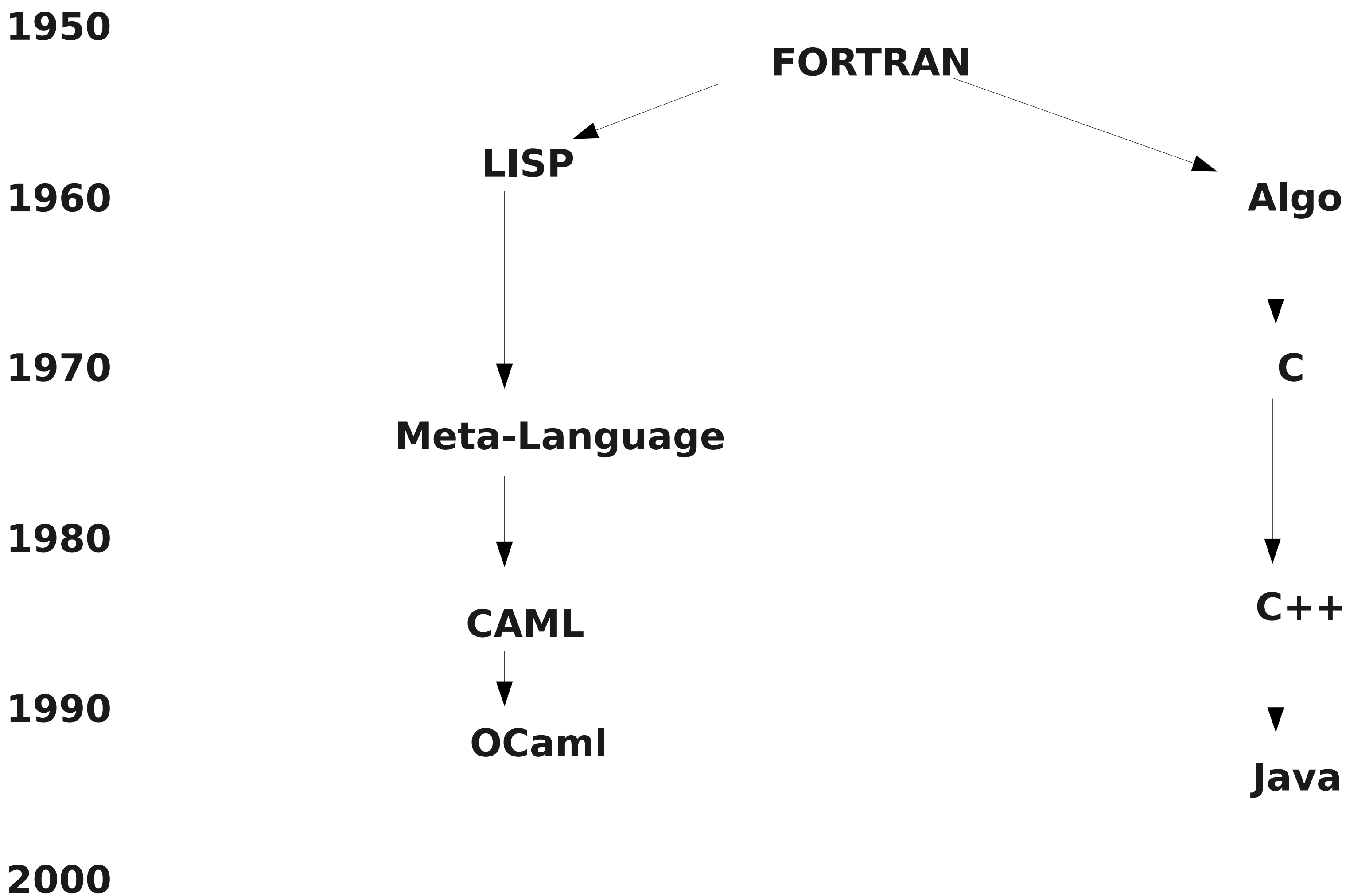
Parental Advisory:

Contains Strong Opinions

OCaml brochure

- OCaml (i.e. Objective Caml)
  - is an advanced, pragmatic, programming language
  - uses theoretic advances that have happened in the last 30 years
  - is not a theoretical/experimental language — it was designed to do real work in
- References
  - http://caml.inria.fr
  - Debian binary package “ocaml”

OCaml pedigree



OCaml **is not** ...

- ... a scripting language
  - doesn't compete with: Perl, Shell script, TCL/TK, ...
- ... a systems language
  - things not to write in OCaml:
    - operating systems
      - even if crazy people do that http://dst.purevoid.org/ :-)
    - device drivers
    - embedded software
      - where space is a real concern
    - hard realtime systems
    - anything that needs to talk directly to hardware

OCaml **is** ...

- ... an applications language ...
  - compete with: Java, C++, C#, Python, C (when used for apps)
- ... for writing **large-scale apps**

Use the Right Tool (tm) for the Job

(This is the best advice I will give you
[several times] in this part of the talk)

Why large-scale apps are different?

Large-Scale Apps

- *Lots of Code* (30KLOC or more)
- *Lots of Developers* (> 5 ?)
- *Maintenance* is a real concern
  - Application will have a long life
  - New developers will need to maintain code written by developers who left the project, company, continent, planet, and/or plane of existence

Lots of Code

- Lots of code makes it difficult to navigate
  - More screens to be looked at to figure anything out
  - Easy to lose or duplicate code
- Short is better
  - @LANGUAGE@ should help expressing algorithms in **as few lines as possible**
  - caveat: code still needs to be **readable**
    - remember Perl adagio “write once, read never”?
- Two further aspects: complexity & immutability

Lots of Code — Complexity

- complexity
  - Is a function of the number of possible interactions the programmer needs to worry about
  - Number of possible interactions goes up with the *square* of the number of lines of code
    - We've already addressed this
  - Side effects cause unexpected interactions (aka bugs)
- @LANGUAGE@ should help **avoiding side effects**

Lots of Code — Immutability

- changing other code's data behind it's back is not playing nice
  - creates a dependency on change presence/absence
  - violates ~~OO~~ good design principles (encapsulation)
- cloning/copying is not a valid work around
  - Too much memory wasted
  - Too much CPU wasted
- @LANGUAGE@ should enforce (or at least enable) **immutability**

Maintenance

- the only thing constant is change
  - programs are never complete, just abandoned
- incomplete/inconsistent changes make for bugs
  - you've found 461 places you needed to fix — are there 462?
- @LANGUAGE@ should enforce **complete and consistent changes**

Use the Right Tool (tm) for the Job

(I said you'd see this again)

Executive Summary

- @LANGUAGE@ = OCaml
- OCaml allows you to:
  - write code faster
  - spend less time debugging
  - have more maintainable code
  - *without* sacrificing performance!

This leaves us with one question...

How?

OCaml Features

(We'll explain all of them and why they're good in a bit)

- Garbage Collection
- Exceptions
- Bounds checking on Arrays
- References, not Pointers
- Everything is a Reference
- Strong static typing
  - Expressive Type System
  - Type Inference
- Three different ways to run code
  - Interpreted
  - Virtual Machine
  - Compiled to Native
- Immutability as default

OCaml Features (cont.)

- Multi-paradigm support
  - Functional
  - Object Oriented
  - Imperative/Procedural
- Higher Order Functions
- Variant types (no null)
- Builtin types- tuple, list, record
- Pattern Matching

Manual Memory Management

- free/malloc-like memory management
  - does not interact well with large scale-apps
    - increases complexity of code
    - takes large part of development time (~ 40%)
  - can be slow
    - free/malloc are O(*heapsize*) on the average
    - increases cache misses (heap fragmentation)
  - wastes memory
    - heap fragmentation
    - blocks book-keeping

Garbage Collection

- reference counting GC
  - easy to implement, so popular (perl, python, ruby, ...)
  - issues with circular data structures
  - expensive in terms of CPU cycles
    - reference counters book-keeping
  - heap still fragmented

Garbage Collection (cont.)

- generational copying GC
  - based on the “generational hypothesis”
    - the objects most recently created in the runtime system are also those most likely to quickly become unreachable
  - fast allocation
  - heap is always compact
  - cache conscious data placement

Garbage Collection (cont.)

- Java GC (generational copying)
  - Java: only "popular" language with decent GC
  - allocation still expensive — at least according to all the Java programmers I talk to
  - long GC pauses
- OCaml GC (generational copying)
  - very fast allocation
    - common case is 5 assembly instructions on x86
  - no long GC pauses

Exceptions

- same basic capabilities as Java, C++
- way faster — ~20 clock cycles total between setting up the try block, and doing the throw
  - C++ exceptions are slow — you have to unwind the stack
  - Java's stack trace requirement means you can't do tail call optimization
- GC picks up the garbage

Bounds Checking on Array Accesses

- Fencepost (off-by-1) errors are very common
- Bounds checking is often very cheap
  - Most checks can be eliminated by the compiler

```
for i = 1 to (Array.length a) - 1 do
   a.(i) <- 0
done
```

- Of course OCaml bounds checks it's array accesses!

OCaml has references, not pointers

- No pointer arithmetic
  - This is why you can't use it to bang on hardware
- No random memory corruption either
- Same as Java Objects

Everything is a Reference

- Any type can be added to any data structure
  - no more Java-like Int, Double, etc.
  - the same object code works for all types
    - no code bloat like C++ templates
  - OCaml automatically "unboxes" the fundamental types- ints, chars, etc., and stores them in place of their pointers
    - efficiency is not lost
- Allows for true universal types (∀-types)
  - works like void * tricks ... but is type safe!

Strong Compile-time Type Checking

- Finding bugs at compile time cheap, debugging code expensive (time consuming)
  - Especially since type checking tells you the file and line the bug is at
  - Simply firing up a debugger and recreating the problem takes longer than fixing a bug detected at compile time
- OCaml gives you strong static type checking, but without the bondage and discipline aspects.

It's not quite true that once your OCaml code compiles, it's correct

- ... but it's surprisingly close to being true!
  - OCaml detects many logic errors as type errors
    - forgotten cases
    - conditions not checked for
    - incorrect function arguments
    - violated constraints (especially with modules)
  - all code gets checked
    - all branches, even not taken ones
    - code gets checked automatically
      - compiler does checks — no extra work for the programmer

“ of ” relationship

- Like “is-a” or “has-a”, objects can have “of” relationships
  - e.g.: list *of* foo, tree *of* array *of* float, etc.
  - can express “universal types”
    - OCaml can easily express types like
      - “for any types a and b (which can be the same or different types), this function takes a list of type a's, and a function which converts a type a to a type b, and returns a list of type b's”
      - In OCaml, that type would look like:

        ```
        'a list -> ('a -> 'b) -> 'b list
        ```
      - OCaml allows you to express complex types **concisely**
    - Universal types are the default, not the exception

C++ and Java type *checking*

- Little more advanced than Algol-68
- Java: cast to/from Object pattern sucks
  - Totally defeats static type checking
  - Run time type checking -> CPU/memory penalties
  - Allows programmer to hide errors
  - Verbose to boot
  - (now fairly better with generics)
- C++ templates suck
  - Horrid syntax
  - Templates the exception, not the rule
  - Still verbose

OCaml has type *inference*

- compiler can figure out what type a variable has from context
  - programmer does not need to specify the types of (most) variables and functions
    - clearer code (not confused by redundant type specifications)
    - more likely to be correct
    - compiler can even generate type annotations for those types which need them (you, lazy guys!)
  - this is considered a major advantage of run time type checking
    - but keeps the benefits of static type checking!

Running OCaml code

- 3 different ways to run OCaml code
  1. interpreted
  2. compiled to bytecode + virtual machine
  3. compiled to native executable

OCaml Toplevel Interpreter

- Lisp/Python-like
- Advantages
  - Fast turn around (no need to build/run)
    - Can be used for scripts
  - Instant feedback
    - Good for experiments, exploration, and one-off programs
- Disadvantages
  - Customer needs OCaml installed to run the code
  - Slow
    - Interpreter needs to compile code constantly
    - No optimizations
  - More memory needed
    - Compiler/UI needed

The OCaml Virtual Machine

- Like Java, C# (.NET)
- Advantages
  - Byte code highly portable
  - Byte code is small
  - Compiles faster than native
  - Don't need to ship source
  - Don't need to compile source at runtime (faster than interpreted)
- Disadvantages
  - Customer needs to have OCaml runtime installed
  - Slower than native

Compiling to Native Code

- Like C/C++
- Advantages
  - Fastest way to execute OCaml code
    - Close to C performance
  - Customer doesn't need anything of OCaml installed to run OCaml code

- Disadvantages
  - Not all systems support compiling to native code
    - currently: alpha, amd64, ~~arm~~, hppa, x86, ia64, ppc, sparc
  - Native code not very portable
    - Can't run code compiled for x86 on a Sparc
    - Can't run Windows code (natively) on Linux

OCaml native code performance

- Official statement — within a factor of 2 of C's
  - Hard to measure — lies, damned lies, and benchmarks
- Yes, C++ does have a performance hit
  - Need to add code to handle exceptions wether you use them or not (someone else might have to - like operator::new())
  - more C++ features -> less performance
    - Virtual functions == indirect calls
    - Templates == code bloat == more cache misses

OCaml native code performance (cont.)

- OCaml code sometimes faster than C
  - Better algorithms
  - Copying garbage collection reduces cache misses, and is a negative performance cost (it speeds the program up)

Immutability is the Default

- Decreases code inter-dependencies
  - A function can not “accidentally” change it's arguments
    - Use tuples to return multiple values — say what you mean
- Eliminates the need for deep copies
  - Just pass the data structure around
  - Reusing objects isn't always faster — what you gain in the straight aways (not allocating new objects) you lose in the turns (needing to clone objects to prevent modifications)

Immutability and Allocation

- Instead of changing a data structure, allocate a new data structure just like the old, except for the one change
  - Since the old data structure can not change, you can resuse most of it.
  - Functions can return the new, modified, data structure, and let the caller decide which (new or old) to use.
- Immutability means you allocate a lot
  - Allocate new objects, instead of reusing old ones
  - statistics: OCaml programs allocate about 1 word every 6 instructions -- an insane amount of allocations!
- This means speed of allocation is important
  - Fortunately, OCaml has an insanely fast allocator, so this isn't a performance hit.

OCaml is a Multi-paradigm Language

- Supports:
  - functional (Lisp, ML)
  - Object Oriented (Java, C++, C#, Python, ...)
  - procedural (C, Pascal)
- No one paradigm is right for all problems
  - If all you have is a hammer, everything looks like a nail

Use the Right Tool (tm) for the Job

(This means use the right paradigm for the job too!)

Higher Order Functions

- Fifty-cent word for some simple concepts:
  - Partial function evaluation
    - If a function has *n* arguments, you can supply *k*<*n* values and get a function with *n*-*k* arguments
  - Inner functions (like Pascal, Algol, GCC)
  - Anonymous local functions easy to define
  - Functions can be passed around like variables
    - Inner functions can be returned, and they keep the stack frame they execute in
    - AKA continuations

Higher Order Functions Combine State and Functions

- Replaces “doit” classes popular with Java
  - MouseClickEvent, KeypressEvent, etc.
  - An interface with a single function (“doit”) which the caller implements and instantiates
  - The class is the state associated with the function
- Good C programmers pass state pointers to callbacks
  - These are void *'s which are passed, uninspected, to the callback function
  - Works like the this pointer for a “doit” class

Higher Order Functions Simplify APIs

- No need to define special classes for every call back
- Easier to “glue” disseperate APIs together
  - any function can be a call back
  - easy to overcome mismatched argument lists
- Say what you mean

Data Structure Comprehensions

- Functions which do something to the entire data structure
  - Iter- call a function on every member
    - Example use: printing the data structure
  - Fold- accumulate a value over the data structure
    - Example use: Vector length function
  - Map- convert the data structure
    - Example use: Vector scale function
- Many algorithms can be expressed entirely as comprehensions
  - Why keep writing the same loops?
- Easy to write and use if you have HOF, painful otherwise

Variant (or *algebraic*) datatypes

- C's enums on steroids
  - They are not ints!
    - Typesafe- can not cast to/from ints
    - What does APPLE + ORANGE mean? BANANAs?
  - Can contain data
    - Work like Eckel's Java Enums
    - Easy way to do simple data structures
  - How OCaml does nulls
      - Not all data types can have nulls- programmer chooses which
      - It's a compile-time error if you don't handle the null case
        - Bye bye null pointer exception!

How do you hold different types in the same data structures?

- common question asked by people used to run time type checking
  - often because they use lists when they should use tuples, structures, or objects
- answer: use a variant type!
  - Tag each element with what type it is
  - Compiler makes sure you handle all cases
    - A huge help in maintainance when adding new cases
  - If all types can not exist in all locations, you are using the wrong data structure!

OCaml Has Rich Data Structures

Built-in support:

- Tuples
- Lists
- Records
- Arrays
- Objects
- Modules

Standard Library:

- Hash Tables
- Maps
- Sets
- Queues
- Stacks

“Use the right tool for the job” means use the right data structure!

- Many programming languages encourage you to use only one data structure
  - Lists (Lisp)
  - Associative Arrays (Perl)
  - Objects (Java)
- By supplying multiple data structures (and making it easy to add your own), OCaml encourages you to use the right data structure
  - But you have to know your data structures!

Pattern Matching

- Switch/case statements on steroids
- Syntactic sugar, but...
- Allows you to express complicated algorithms compactly
  - Balancing algorithm for red-black trees becomes simple enough to use as an example

Nice song and dance, but what proof do you have?

The Computer Language Shootout Benchmarks

- collection of micro-benchmarks written in many different languages
  - http://shootout.alioth.debian.org/
  - compares LOC, run times, and memory
- not a perfect comparison
  - small benchmarks are not represenitive of large projects
  - lies, damned lies, and benchmarks
  - we will show you 2004 data
- results are surprising
  - scores in brackets

Top 10 Fastest Languages

(least CPU usage overall)

1. C (GCC)	[752]
2. **OCaml** (native code)	[751]
3. SML (mlton)	[751]
4. C++ (G++)	[743]
5. SML (smlnj)	[736]
6. Common Lisp (cmucl)	[734]
7. Scheme (bigloo)	[730]
8. **OCaml** (bytecode)	[718]
9. Java (Blackdown/Sun)	[703]
10. Pike	[647]
13. Python	[578]
14. Perl	[577]
15. Ruby	[546]

Top 10 Concise Languages
(fewest lines of code overall)

1. **OCaml** (both)	[584]
2. Ruby	[582]
3. Scheme (guile)	[578]
4. Python	[559]
5. Pike	[556]
6. Perl	[556]
7. Common Lisp (cmucl)	[514]
8. Scheme (bigloo)	[506]
9. Lua	[492]
10. TCL	[478]
11. Java	[468]
16. C++	[435]
23. C	[315]

Top 10 Smallest Footprints
(least memory usage overall)

1. C (GCC)	[739]
2. **OCaml** (native code)	[719]
3. C++ (G++)	[715]
4. SML (mlton)	[713]
5. **OCaml** (byte code)	[709]
6. Forth	[649]
7. Python	[643]
8. Lua	[626]
9. Perl	[624]
10. Pike	[611]
11. Ruby	[609]
27. Java (Blackdown/Sun)	[290]

An OCaml tutorial
(live)

have `fun () ->`

All that glitters is not gold

Good reasons *not* to use OCaml

- ... no, we are not going crazy
  - ... but in some respects far better than OCaml can be done, let's see some of them
- OCaml *is* HOT, but doesn't know the meaning of “marketing”
  1. open source, but bound to the (INRIA) cathedral development model
     - external patches are seldomly considered (strong opinions there as well) and philosophical/design change proposals are never
     - the standard library is ridiculously small
       - paradox: in OCaml is damned easy to code complex tasks and sometimes damned tedious to code simple ones

Good reasons *not* to use OCaml (cont.)

- lack of “marketing” (cont.):
  2. (practically) no dynamic linking
  3. ABI compatibility breaks with every release / interface change (including comments!)
     - not such a big deal, but entails a source based distribution
  4. no (GNU) team player
     - e.g.: hard to mix with autotools, no cooperation w gcc pipeline, ...
  5. concrete syntax *is* important: other languages have got this, why we haven't?

Good reasons *not* to use OCaml (cont.)

- some technical and philosophical deficiencies:
  1. no real concurrency of OCaml code, since the garbage collector is not distributed and has a global lock
  2. TIMTOWTDI ... (yet another Perl's adagio: there is more than one way to do it),
     ... but There Are *Too Many* Ways To Do It

- but still ... OCaml *is* HOT :-)

A Functional Programmer's Toolkit

Functional programming techniques

- as imperative programming, functional programming (FP) has its well-established techniques
- a minimal functional programmer toolkit necessarily includes:
  1. (tail) recursion
  2. “container” manipulation
     - iteration, transformation, filtering, ...
  3. “container” folding

Recursion: beware of the stack!

- we all (now) know recursion

```
let rec mk_list = function
  | 0 -> []
  | n -> n :: mk_list (n-1)
```

  val mk_list : int -> int list

  – let's try it on a (not so) large input

  ```
  # mk_list 1_000_000;;
  Stack overflow during evaluation (looping
  recursion?).
  ```

  – “bug”: each time fact is recursively invoked, the *activation record* of the previos invocation can't be removed from the stack

    - sooner or later the stack will explode

Tail recursion

- recursive calls can be in *tail position*
  - i.e. the return value of the whole function is the same of that particular recursive invocation (or *tail call*)
- tail calls can be optimized by the compiler: the generated code can *reuse* the current activation record
  - recursive invocations no longer require more stack space than a single function invocation

Tail recursion (cont.)

- tail recursive version of mk_list

  ```
  let rec mk_list acc = function
    | 0 -> acc
    | n -> mk_list (n::acc) (n-1)
  val mk_list : int list -> int -> int list
  ```

  - where has the base case value gone? you have to provide it at 1st invocation time (have a look at the inferred type ...)
  - now the following does work:

    ```
    # mk_list [] 1_000_000;;
    (* long output snipped *)
    ```

  - beware: the result is in reverse order!

Tail recursion (cont.)

- a frequent idiom is to bundle the base case value together with an auxiliary function
  - encapsulation and the desired type are back
  - yet another version of mk_list

```
let mk_list n =
  let rec aux acc = function
    | 0 -> acc
    | n -> aux (n::acc) (n-1) in
  List.rev (aux []) n
```

    - a posteriori processing before returning is possible
    - η-contraction is quite common

Containers vs inductive types

- “containers” are mirrored in HOT languages by inductive datatypes
  - container manipulation (often) asks the programmer to follow explicit flow control patterns, e.g.:
    - to visit an array use an indexed for loop
    - to visit a list/set/bag/... use a while on an iterator
  - inductive datatypes are conceptually associated to recursors on them
    - using recursors the control flow is implicit and the programmer only needs to care about the actual operation she wants to perform on containees

Iterators (iter)

- iterators: the simplest recursors
  - they apply a function returning unit to each containee
  - the functional version of a for(each) loop

    List.iter : ('a -> unit) -> 'a list -> unit

    List.iter print_int [1;2;3;4;5]
  - iterators are provided for built-in types, but you can do them by yourself (and for your own types!)

```
type 'a my_list = Nil | Cons of 'a * my_list
let rec my_iter f = function
  | Nil -> ()
  | Cons (hd, tl) -> f hd ; my_iter f tl
```

  - … in fact they can even be automatically generated …

Containee transformation (map)

- a “map” recursor transforms a container to an isomorphic one, applying a local transformation to each containee
  - functional version of a container copy (on steroids)

List.map : ('a -> 'b) -> 'a list -> 'b list

```
List.map (fun x -> x+1) [1;2;3;4;5] ;;

List.map ((+) 1) [1;2;3;4;5] ;;   (* how elegant ... *)

let rec my_map f = function
  | [] -> []
  | hd :: tl -> f hd :: my_map f tl
  (* question: is this tail recursive? *)
```

Selection (filter)

- a *predicate* on a value of type *t* can be represented as a function *f: t -> bool*
  - intuition: applying a predicate to a value returns true if the value satisfies the predicate
- a filter recursor selects all values satisfying a given predicate

List.filter : ('a -> bool) -> 'a list -> 'a list

```
List.filter (fun x -> x mod 2 =0) [1;2;3;4;5]
```

```
let rec my_filter p =
  | [] -> []
  | hd :: tl when p hd -> hd :: my_filter p tl
  | hd :: tl -> my_filter p tl
```

Predicate algebra

- when working with predicates some predicate operators can come handy

```
let (&~) p1 p2 = fun x -> p1 x && p2 x
```

val (&~): ('a -> bool) -> ('a -> bool) -> ('a -> bool)

```
let (|~) p1 p2 = fun x -> p1 x || p2 x
```

val (|~): ('a -> bool) -> ('a -> bool) -> ('a -> bool)

```
let (!~) p = fun x -> not (p x)
```

val (!~): ('a -> bool) -> ('a -> bool)

  - e.g.

```
let even = fun x -> x mod 2 = 0
let div_by n = fun x -> x mod n = 0
List.filter (even  &~  !~ (div_by 5)) [5;6;7;8;9;10]
```

Container folding (fold)

- the recursors we have seen so far are unable to compute *aggregate values* dependent on containees
  - but this is a frequent need, e.g.:
    - List.length: given an 'a list, compute its length
    - list_sum: given an int list, sum up all its elements
    - or even List.rev: given a list, reverse it
  - though we can write recursive functions for all the above needs (but we are back to explicit flow control!), a generic recursors on top of which implement them does exist: *fold*

Fold

- intuition
  - a fold recursor “consumes” a container one step at a time (with one step for each containee), building incrementally the final result
  - at each step the new “final” result is built using the current element and the previous “final” result
    - how the incremental construction is actually implemented is a (functional) parameter of fold ...
    - ... as well as the initial “final” result, which is needed to bootstrap the process

Fold (cont.)

- common variants of (list) fold: left/right
  - fold on lists

    List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

    - intuition

      fold_left f init [e1; e2; …; en] = (f … (f (f init e1) e2) … en)

    - sample usage

```
let list_sum =
   List.fold_left (fun acc e -> acc + e) 0 [1;2;3;4;5]
let list_sum = List.fold_left (+) 0 [1;2;3;4;5] (* elegance? *)
let list_length l = List.fold_left (fun acc _ -> acc + 1) 0 l
let list_iter f l = List.fold_left (fun _ e -> f e ; ()) () l
let list_rev l = List.fold_left (fun acc e -> e :: acc) [] l
let list_map f l =
   List.rev (List.fold_left (fun acc e -> f e :: acc) [] l)
```

Fold (cont.)

– fold on lists

List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

- do it by yourself

```
let rec my_fold_left f curr = function
  | [] -> curr
  | hd :: tl -> my_fold_left f (f curr hd) tl
```

Fold (cont.)

- fold on lists

  List.fold_right: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

  - intuition

    ```
    fold_right f [e1; e2; ...; en] init = (f e1 (f e2 (... (f en init) ...)
    ```

  - sample usage

    ```
    let list_sum = List.fold_right (+) [1;2;3;4;5] 0
    ```

    ```
    let list_map f l =
      List.fold_right (fun e acc -> f e :: acc) l [] (* no List.rev *)
    ```

  - do it by yourself

    ```
    let rec my_fold_right f l init =
      match l with
      | [] -> curr
      | hd :: tl ->  f hd (my_fold_right f tl init)
    ```

    ```
    (* beware: not tail recursive [like many implementation] *)
    ```

Recursors as a concept

- Remember: recursors is a concept, not a specific implementation of them in some library
  - you can (and should!) develop your own recursors
    - as a specialized version of the usual recursors on some built-in or available inductive datatypes
    - for your home made inductive datatypes
  - benefit: keep separate the visit logics from the business logics of doing what you want with containees

Funct. data structures and algo.

- How can I go back & forth in a list in O(1)?
- Answer: use a zipper!

```
type 'a zipper = 'a list * 'a list   (* past, future *)

let next = function
    l,[] -> assert false
  | l,he::tl -> he::l,tl

let prev = function
    [],l -> assert false
  | he::tl,l -> tl,he::l

let z = [5;4;3;2;1],[6;7;8;9;10]
next (next z) = [7;6;5;4;3;2;1],[8;9;10]
```

CSC' mantras

- Avoid cluttering the namespace!
- Scope hides functions needed only once
- Scope avoids passing too many parameters to auxiliary functions
- Scope avoids passing constant parameters around
- let f k1 k2 x1 x2 =

```
let f k1 k2 x1 x2 =
   let rec aux1 x1 x2 = ... in
    ...
   let rec auxn x1 x2= ... in
       auxn x1 x2
```

CSC' mantras

- Only high-level meaningful functions (even with large types) should be exported
- There should be no more than a few ways to compose functions together
- Compound functions should not be exported
  - array_of_list: 'a list -> 'a array
  - optimize_list: data list -> data list
  - optimize_array: data array -> data array

The many shapes of Polymorphism

What is polymorphism?

- polymorphic = that does different things
- but what does it mean, really?
  - overloading
  - generic/templates
  - late binding
    (in class based object oriented languages)

What is polymorphism?

- overloading (C++, Java, ...)
  - int plus(int x, int y) != float plus(float x, float y)
  - “totally unrelated” code in different memory locations
  - not unrelated for the human being: lack of abstraction? (cfr. Haskell type classes)
  - resolved at compile time
  - resolution can be schizophrenic
  - prevents type inference

What is polymorphism?

- generics/templates:

  ```
  static <T>
  void fromArrayToCollection(T[] a, Collection<T> c)
  { for (T o : a) { c.add(o); } }
  ```

  - one source code that works on different types (any type?)
  - no more safe/unsafe down-casts
  - do we need such an horrible syntax?
  - how are they implemented?

Implementation of generics/templates

- C++ templates:
  - different data types have different memory representations
  - impossible to have compiled code that works uniformly on every type
  - at compile time, one compiled code for every type instance in the source code
  - large executables, horribly long symbol names, performance penalties (cache misses)

Implementation of generics/templates

- Java generics:
  - primitive data types have ad-hoc memory representations
  - all objects represented uniformly via references
  - one compiled code that works uniformly on every class type
  - small executables, no performance penalties
  - int vs Int

Typing of generics/templates

- Late addition to Algol68 type system
- Late addition to Algol68/C/Pascal/Modula syntax
- Requires type abstraction and partial application
  - List <Int> l;
- Academic type systems got it right in the 70s!

System-F polymorphism

- Typing a la System-F:

  ```
  id (A: Type, a: A) : A  { return a; }

  smap (A: Type, f: forall B:Type. B -> B, l: list A) : list A

  smap (int, id, [1; 2; 3]) = [1; 2; 3]
  ```

- Explicit type abstractions/applications in terms
- No type-inference
  - all variables should be typed

Hindley-Milner polymorphism

- Abstractions can be everywhere in System-F:
  - smap: forall A:Type. list A -> (forall B:Type. B -> B) -> list B
- Hindley-Milner polymorphism:
  - quantifications on types only in front
  - map: forall A,B:Type. list A -> (A -> B) -> list B
    map: list 'A -> ('A -> 'B) -> list 'B
  - no type abstractions/applications in terms
  - type inference is decidable
    - variables need not be typed

OCaml polymorphism

- OCaml implements Hindley-Milner
- Most functions are typable in Hindley-Milner
- System-F types
  - available when needed
  - require explicit quantification
  - only in record fields/object methods (why?)

Example:

- type ('a,'b) r = {label: 'b -> 'a * 'b }
  let mk_r x = {label = fun y -> x,y }
  let o = mk_r 2   (* o has type (int,_'b) r *)
  let x = o.label 5 (* x = 2,5
  o has type (int,int) r*)
  let y = o.label “ciao” (* ERROR!!! *)
- type 'a doit = { label: 'b. 'b -> 'a * 'b };
  let mk_r x = {label = fun y -> x,y }
  let o = mk_r 2   (* o has type int r *)
  let x = o.label 5 (* x = 2,5 *)
  let y = o.label “ciao” (* y = 2,”ciao” *)

What is polymorphism?

- In class-based object oriented languages:
  - OOP = state + incapsulation + inheritance + overriding + subtyping + late binding
  - class Point { method move() { ... } }
    class ColoredPoint inherits Point
        { method move() { ... } }
    void force(Point c) { c.move(); }
    ColoredPoint c = new ColoredPoint;
    force(c);
  - do we need all ingredients together?

Incapsulation + Late binding

- (Private) data and methods that act on the data are put in a first class object
- First class objects can be stored, passed around, etc.
- Methods code is related to the instance
- In functional programming:
  - functions are first class objects
  - they can be assembled in containers
  - they can share (immutable) private data
  - we can invoke a function in a container

Example

- type tower = (float -> float) * (float -> float)
                        (* volume *)   (* lateral surface *)
- let new_circle r =
    (fun h -> pi *. r *. r *. h), (* volume *)
    (fun h -> 2. *. pi *. r *. h) (* lateral surface *)
  new_circle : float -> tower
- let new_square r =
    (fun h -> r *. r *. h),   (* volume *)
    (fun h -> 4.0 *. r *. h) (* lateral surface *)
  new_square : float -> tower
- let c = new_circle 8.0   (* c: tower *)
  let s = new_square 3.0   (* s: tower *)
  let res = fst c 10. -. fst s 10. (* volume *)

Inheritance + code reusal + overriding + subtyping

- Two uses of inheritance:
  - to reuse code
    - in functional programming: just copy a container, changing the fields that must be overridden
    - let o = (fun () -> “Hello”), (fun () -> “World”)
let o' = fst o, (fun () -> “Mom”)
    - efficient, because of sharing

Inheritance + code reusal + overriding + subtyping

- Two uses of inheritance:
  - for inheritance
    - why?
      - let f o =
        “Object “ ^ o.print ^ “ holds “ ^ string_of_int o.value
      - o.print must make sense; o.value must make sense
      - why do we need interface printable and valuable?
      - why do we inherit from interfaces if NO CODE must be reused?
    - with generic polymorphism:
      - f has type < print : string; value : int; ... >
        where ... stands for any list of other method
      - e.g. we can use an object of type
        < save: unit; print: string; move: unit; value: int>

State + Incapsulation

- Global variables are bad, bad, bad
- Protect mutable variables inside objects
- In functional languages:
  - incapsulation is given by scope
  - state (= mutable variable) can be added
  - let new_account () =
    let password = ref “change me” in
    (fun p -> password := p) (* set password *),
    (fun p -> !password = p) (* check password *)
    new_account: unit -> (string -> unit)*(string -> bool)

Mutable status

Mutable status: why?

- Mutable status is bad
  - less correct code, harder to debug
  - no sharing, not reentrant, complex backtracking
  - less intuitive code (e.g. w.r.t. fold, map, etc.)
- So why?
  - for reactive programming
    (to store data between events/commands)
  - for non algebraic data structures
    (e.g. graphs)

Mutable status: how?

- Do NOT make everything mutable
- Introduce the type of mutable cells:
  let x = ref 0   (* x has type int ref *)
  – x is a constant (a reference) to “ref 0”
  – “ref 0” is a memory cell that currently holds the value 0
- Assignment via a reference: x := 1
  – x is unchanged: it still points to the same cell
  – the content of the cell is changed
- Dereferencing: x := !x + 1

Mutable status: how?

- let x = ref 0
  let y = x
  - x and y are equal constant references to the same cell “ref 0”
- let f c = c := !c + 1 in f x
  - the constant x is passed by value to f
  - everything is passed by value in OCaml!
  - the cell can be equally reached by x and c
  - this is C++/Java call-by-reference
  - THIS IS BAD!

Mutable status: how?

- Functions should be side-effect free
- In C++/Java side-effects used to return multiple values
- Say what you mean!
- Instead of

```
let f c d =
    d := !c + !d;
    c := !c + 1;
in f x y
```

  use

```
let f c d = c+d, c+1 in
let x',y' = f !x !y in
    x := x'; y := y'
```

Mutable status: how?

- Wait a minute! If functions should be side-effect free, where can I use mutable cells?
- Answer 1: only a few mutable cells to store the status between different callback invocations!
  - ```
    let status = ref 0
    let button_pressed () =
        status := do_something(!status)
    ```
    ```

Mutable status: how?

- Wait a minute! If functions should be side-effect free, where can I use mutable cells?
- Answer 2: inside cells to implement non algebraic data structures
  - Example 1 (|3| <====> |5|):

    ```
    type cell = { v: int; neighbours: cell list ref }
    let c1 = {v = 3; neighbours = ref [] }
    let c2 = {v = 5; neighbours = ref [c1] }
    c1.neighbours := [c2]
    ```

Mutable status: how?

- Wait a minute! If functions should be side-effect free, where can I use mutable cells?
- Answer 2: inside cells to implement non algebraic data structures
  - Example 2 (fixed number of arcs):

```
type cell = { v: int; neighbours: cell ref list}
let ref c1 = { v = 3; neighbours = [ref c2] }
and     c2 = { v = 5; neighbours = [ref c1] }
c1.neighbours := []           (* ERROR! *)
(fst c1.neighbours) := c1  (* OK! *)
```

Mutable status: how?

- Wait a minute! If functions should be side-effect free, where can I use mutable cells?
- Answer 3: to implement static shared function variables / friend functions
  - ```
    let new_option () =
        let value = ref 0 in
        (fun v -> value := v),  (* set *)
        (fun () -> !value)      (* get *)

    let set, get = new_option ()
    (* set : int -> unit ; get : unit -> int *)
    ```
    ```

Modules

Abstract Data Types

- Abstract Data Type = data type whose representation is unknow
- No ADTs, no modularity
  - When the implementation changes, all the code changes
- OO languages: objects are ADTs because fields (and methods) can be private == not in the interface
- ADTs without objects are possible!

Modules

- A module is made of an implementation and an interface (module type)
- The module type restricts the interface of the implementation
  - module M =

```
module M =
struct
    type set = int list
    let empty = []
    let add1 x l = x::l
    let addn l l' = l @ l'
    let union = addn
end
```

```
module type M =
sig
    type set

    val add1 : int -> t -> t
    val addn: int list -> t -> t
    val union: t -> t -> t
end
```

When concrete data types are more handy

- Pattern matching is only allowed on algebraic data types
- Views: functions from an ADT T to an algebraic data type T'
  - module type M = sig

```
module type M = sig
  type set
  type set' =
    Choice of int * set
  val set'_of_set : set -> set'
  val empty: set
  val add: int -> set -> set
end
```

  - let rec iter f s=

```
let rec iter f s=
  match
   set'_of_set s
  with
   Choice(a,s') =>
    f a;
    iter f s'
```

When concrete data types are more handy

- Pattern matching is only allowed on algebraic data types
- Private types: semi-abstract data types
  - ```
    private type ordered_list :=
        Nil
      | Cons of int * ordered_list
    val nil : ordered_list
    val cons : int -> ordered_list -> ordered_list
    ```
  - ```
    let nil = Nil
    let cons x l =
      match l with
         [] -> [x]
       | he::_ -> if x <= he then Cons (x,l) else raise E
    ```

Modules as Namespaces

- Modules can also define a namespace
- ```
  module HashTable = struct
      module Key = struct
          type t = int
          let hash n = n mod 10
      end
      let table = ([| |] : Key.t list)
      let add x v =
          let h = Key.hash x in
          table.(h) <- (x,v)::table.(h)
  end
  let hash_13 = HashTable.Key.hash 13
  ```
  ```

Functors

Verbosity of generic polymorphism

- H.o. generic functions very good for code reusal...
- ... but too verbose!

```
hashtable_add:
    ('key -> 'key -> 'bool) ->
    ('key -> int) ->
    ('key,'value) hashtable ->
    'key -> 'value -> ('key,'value) hashtable
```

- Partial solution:

```
let htbl_add = hashtable_add inteq inthash
let htbl_del = hashtable_del inteq inthash
...
```

Functors

- Instead of abstracting many functions one at a time, abstract all of them AT ONCE

```
module type Key = sig
    type t
    val eq: t -> t -> bool
    val hash: t -> int
end
module HashTbl(K: Key) = struct
    let hashtable_add tbl k v =
        let hash = Key.hash k in
        ...
end
module String = type t = string  let eq = ... end
module StringHash = HashTbl(String)
```

Functors = H.O. Modules

- A functor is a function from a module to another module
- As functions, functors are typed
- Unlike functions, functors are not first class objects
- Many details on the type system omitted here

What's next?

We do NOT need more functions!

- Keep it simple, stupid!
  Aka Adding new operators / instructions / expressions is bad!
  - Perl: write once, read never => throw away soon
- Higher order functions, recursion, algebraic data types, references and data hiding is already too much
- Syntax and semantics of OCaml already too cluttered

We DO need more types!

- More/better types mean:
  - more polymorphism => more code reusal, more abstract code, easier to understand
    - type 'a list = Nil | Cons of 'a * 'a list
      type 'a tree = Empty | Node of 'a * 'a tree * 'a tree
      super_map: forall 'a 'T. ('a -> 'b) -> 'a 'T -> 'b 'T
    - System-F types (require type annotations)
  - more properties checked at compile-time
    - type list n =
          Nil : list 0
      | Cons : int * list m -> list (m+1)
    - hd Nil   (* ERROR: Nil has type list 0, hd requires
              list (m+1) for some m *)
    - List.nth n l   (* ERROR if l has type list m, m < n)

Status of the art

- Languages with stronger type disciplines are among us...
  - ```
    let rec f = function
        `A -> `B
      | `B y -> `C (f y)
      |   x -> `D x
    ```
- ... but they do not speak to us
  - ```
    f : ([> `A | `B of 'a ] as 'a) ->
        ([> `B | `C of 'b | `D of 'a ] as 'b)
    ```

Status of the art

- Or type checking becomes undecidable
  - because when the type totally captures the specification, type checking becomes proving that the program is correct
  - E.g.:
    ```
    append: list n -> list m -> list (n + m)
    tl: list (n + 1) -> list n
    fun (l : list n) -> tl (append l (Cons 5 l))

    well typed iff exists m s.t.
              m + 1 = n + (n + 1)
    ```

Dependent types

- What is the type of
  “if x = 0 then 3 else true” ?

Dependent types

- What is the type of
  “if x = 0 then 3 else true” ?
- That's simple:
  “if x = 0 then nat else bool” !
- Types can depend on the value of terms!
- Aka Dependent Types
- DML, Cayenne, Epigram
- Coq, PVS, Matita, ... (even more dependent types)

Dependently typed programs

- The type of the output of a function may depend on the value of the input
  - split:
    ```
    forall l: list int.
      if even (length l) then
          list int * list int
      else
          list int * int * list int
    ```
  - hd:
    ```
      forall l: list int.
          if l = [] then unit else int
    ```
  - cfr. hd: list int -> int option

Dependently typed programs

- The type of the second element of a pair can depend on the value of the first element
- c : exists n: int. list n
  (0, []) : exists n: int. list n
  (1, [4]): exists n: int. list n
  ...
- fun ((x,l) : exists n:int. list n) ->
      match x,l with
          0,[] -> 0
       | n,he::tl -> he

Dependently typed programs

- Proofs are programs
  - a proof of A => B is a transformation of a proof of A into a proof of B
  - a proof of A => B is a function of type proof A -> proof B
- Dependent types can fully capture a specification
- certified_sort:

```
forall l: list int.
    exists l': list int.
        (same_elements l l'  &&
         ordered l')
```

Dependently typed programs

- certified_sort:
  ```
  forall l: list int.
      exists l': list int.
          (same_elements l l'  &&
           ordered l')
  ```
- let sort l = fst (certified_sort l)
  sort : list int -> list int
- But: writing by hand the function that proves (same_elements l l' && ordered l') is extremely difficult (> 10x man-month)

Conclusions

Wrap-up

- bottom line(s):
  1. diversity is good: use the right ~~tool~~ programming language for the right job
  2. a compiler is a programmer's best friend: let him does as much as he can
- some references
  - http://caml.inria.fr
  - *Elements of Functional Programming*, Chris Reade
  - *Developing applications with Objective Caml*, Emmanuel Chailloux et al.
  - *Practical OCaml*, Joshua B. Smith

Thank you.