OCaml @ Debian

Stefano Zacchiroli
<zack@debian.org>

$ whoami

- Stefano “Zack” Zacchiroli <zack@debian.org>
  - http://www.bononia.it/~zack
- PhD student
  - @ Computer Science Dept., Uny. of Bologna (Italy)
  - proof assistants, logical frameworks, other crazy stuff ...
- DD work, package maintainance
  - MathML stuff, OCaml libraries and tools, VIM

Outline

- why learn OCaml?
  - OCaml features
- packaging OCaml software
  - OCaml-specific packaging issues
  - a taste of OCaml packaging policy
  - open issues

Why Learn OCaml?

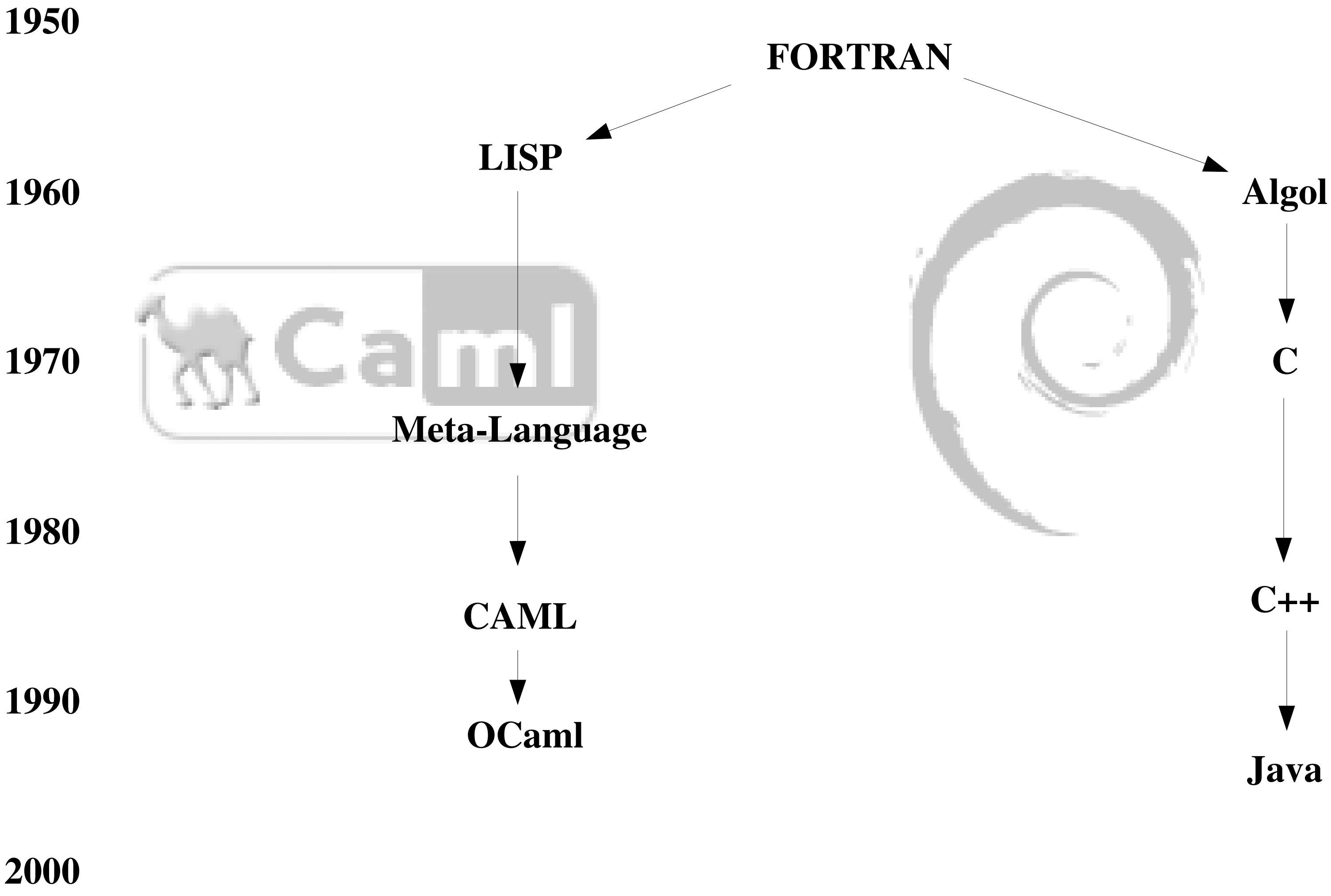
Or, When Your Current Programming Language Sucks

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

About OCaml

- OCaml (i.e. Objective Caml)
  - general-purpose language
  - type safety w/o sacrificing performance
  - very expressive, yet easy to learn and use
  - supports functional, imperative, and object-oriented programming styles
- References
  - http://caml.inria.fr
  - Debian binary package “ocaml”

OCaml pedigree



1950

FORTRAN

LISP

1960

Algol

1970

C

Meta-Language

1980

CAML

C++

1990

OCaml

Java

2000

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 exist 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#, C (when used for apps)
- ... for writing **large-scale apps**
  - lots of code
  - lots of developer
  - maintainance is a real concern

Executive summary

- OCaml allows you to:
  - write code faster
  - wpend 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)

- type system
  - expressive type system
  - strong static typing
  - type inference
- pattern matching
- garbage collection
- exceptions
- bounds-checked arrays
- 3 ways to run code
  - interpreter
  - byte code + VM
  - native code
- multi-paradigm
  - functional
  - procedural
  - object oriented

Expressive type system

- Built-in types: int, string, float, ...
- Type constructors
  - tuples, records, arrays, ...
  - lists
    - real polymorphism: “compile once use many” vs “compile many use once”
  - variant types (AKA C on steroids)
    - pattern matching (AKA C switch on steroids)
  - arrow types
    - higher order functions

Strong static typing

- 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.

Strong static typing

- it's not quite true the 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

Type inference

- compiler can figure out what type a variable has from the context
    - programmer does not need to specify the types of most variables and functions
        - less typing
        - 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 (for the truly lazy programmer)
    - this is considered a major advantage of run time type checking
        - and keep the benefits of static type checking!

Garbage collection

- manual memory management
  - sucks! : increases complexity of code, takes large part of development time (~ 40%), fragments heap, ...
- automatic GC is far better
  - reference counting
    - trivial to implement, widely used, ... still slow
  - generational copying
    - good idea, but Java did it wrong (long GC pauses, slow allocation)
    - OCaml did it right (allocation on the average in 5 CPU cycles, no long GC pauses)

Multi-paradigm

- OCaml is mainly a functional programming language, still:
  - procedural/imperative constructs are supported
  - OO programming is supported
    - interfaces
    - abstract methods and classes
    - multiple inheritance
    - functional objects
    - on-the-fly objects

Bells and whistles

- exceptions
  - same basic capabilities as Java, C++, but faster
    - tail calls are possible, no need to unwind the stack
- bound checking on arrays
  - most checks removed at compile time
- value immutability as default
  - sharing for free

Running OCaml code

- 3 different ways to run OCaml code
  - interpreter
    - python/lisps-like read eval loop
  - compiled to bytecode + virtual machine
    - portability (*NIX, Mac, M$ Win)
    - small code footprint
  - compiled to native executable
    - performance
    - available on: alpha, amd64, arm, hppa, x86, ia64, ppc, sparc

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 used
  - not a perfect comparison
    - small benchmarks are not represenitive of large projects
    - lies, damned lies, and benchmark
    - I'll show you 2004 data
- results are surprising

Top 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 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 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]

Packaging OCaml software

Why DDs have to care about OCaml?

- several free software projects uses OCaml
  - sw you may have heard about, written in it:
    - *Unison* (file synchronizer)
    - *MLdonkey* (P2P client)
    - *ara* (Debian packages database search engine)
    - *Active-DVI* (TeX-based presenter)
    - *Coq* (proof assistant)
    - *Debian From Scratch*
    - *Polygen* (random sentence generator)
    - *FreeRP* (full-featured Web-based ERP)
    - *CDuce* (XML programming language)

Why DDs have to care about OCaml?

- we need to properly handle OCaml in Debian so that our users:
  - could use applications written in OCaml
  - could develop their own OCaml apps

Debian OCaml Maintainers Task Force

- a group of DDs born to help maintainance of OCaml related packages
    - coordinate efforts on the debian-ocaml-maint@lists.debian.org mailing list
    - has an alioth project http://pkg-ocaml-maint.alioth.debian.org/
    - wrote and maintains (a draft of) the Debian OCaml Packaging Policy
    - collaboratively maintains several OCaml related debian packages
    - will be very happy to welcome your contribution :-)

OCaml distribution

- OCaml distribution ships several components
  - bytecode interpreter
  - interactive read-eval loop
  - compilers (bytecode executables)
    - ocamlc (ocaml -> bytecode)
    - ocamlopt (ocaml -> nativecode)
  - optimized compilers (nativecode executables)
    - ocamlc.opt (ocaml -> bytecode)
    - ocamlopt.opt (ocaml -> nativecode)

OCaml distribution

- OCaml distribution components cont'd:
  - other developers' tools (debugger, profiler, ...)
  - standard library (both bytecode and nativecode objects)
    - includes X bindings which pulls in the whole xlibs dependencies
  - shared objects for C library bindings contained in the standard library (e.g. Unix module)

OCaml Debianization

- OCaml distribution spans several binary packages
  - ocaml-base{,-nox}
    - bytecode interpreter
    - standard library stub libraries
  - ocaml{,-nox}
    - compilers
    - developers' tools
    - standard library
  - ocaml-interp
    - interactive toplevel
  - ocaml-native-compilers
    - optimized compilers

Type safety constraints

- In order to ensure type safety
  - objects (both byte and nativecode) compiled by different version of the compiler can't be linked together
    - this is because OCaml has no runtime type information and in-memory representations of data structures may change between versions
    - run-time performances have a cost!
  - bytecode built with version X of the compiler can be run only by version X of the bytecode interpreter
    - same reason as above

Virtual packages

- Debian's dependencies should enforce those constraints
- each package of the OCaml Debianization provides a virtual package <package-name>-<version>
  - e.g.: ocaml-base-3.08, ocaml-3.08

Packaging OCaml apps

- Let's assume you find “Wonderful” on the web, a GPL-ed application written in OCaml and want to Debianize it. You've a choice:
  - create an “Architecture: all” package containing ocaml bytecode excutables
  - create an “Architecture: any” package containing either ocaml bytecode executables or native code ones

“Arch: all” OCaml apps

- Congratulations!
  - your package will be portable on all debian architectures and wont use any buildd clock cycle
- Dependencies:
  - a Dependency on ocaml-base-{nox,}-<version>
  - a Build-Dependency on ocaml-{nox,}-<version>
    - this dep is not strictly necessary for the package to be built properly, but ensures compiler and interpreter versions to be in sync

“Arch: all” OCaml apps

- Caveats
  - in debian/rules you've to be sure the app you're packaging build bytecode executables instead of native code ones
    - a widespread convention among OCaml apps is to use make's “all” target (i.e. “make all”) to build bytecode executables and “opt” target to build native code ones
    - you can verify this setting
      - looking at the build log: “ocamlc” should be invoked instaed of “ocamlopt”
      - looking at the generated executables, they should start with a `#!/usr/bin/ocamlrun` shebang line

“Arch: any” OCaml apps

- congratulations!
  - your package executables will be as fast as lightning
- unfortunately ...
  - OCaml native code compiler do not have backends for all archs supported by debian — supported archs:
    - alpha, amd64, arm, hppa, i386, ia64, powerpc, sparc

“Arch: any” OCaml apps

- byte/native code conditional building
  - you've to check at package build time if native code compilation is available
    - if so build native code using ocamlopt
    - if not build byecode using ocamlc
  - a meaningful test is to verify if ocamlopt executable is available on the building machine, e.g.:

```
build-stamp:
     dh_testdir
     $(MAKE) all
     if [ -x /usr/bin/ocamlopt ]; then $(MAKE) opt; else true; fi
     touch build-stamp
```

“Arch: any” OCaml apps

- dependencies:
  - uhm ... here we've a problem: the same package should depend on ocaml-base{,-nox} only on some arch
    - those for which native compilation is not available
  - “clean” solution (in our opinion): 2 binary packages
    - wonderful
      - architectures: all with native compilation available
      - conflicts/replaces: wonderful-byte
    - wonderful-byte
      - architecture: all
      - provides/conflicts/replaces: wonderful
      - depends: ocaml-base{,-nox}

“Arch: any” OCaml apps

- sample debian/control (from “spamoracle”)

```
Package: spamoracle
Architecture: alpha amd64 arm hppa i386 ia64 powerpc sparc
Depends: ${shlibs:Depends}
Conflicts: spamoracle-byte
Replaces: spamoracle-byte

Package: spamoracle-byte
Architecture: all
Depends: ${shlibs:Depends}, ocaml-base-nox-3.08.3
Provides: spamoracle
Conflicts: spamoracle
Replaces: spamoracle
```

Be nice to auto-builders

- let's suppose Wonderful takes hours to build
        - hey: it's a wonderful app, it must span several KLOC!
    - in order to reduce the auto-builders load ocamlc.opt/ocamlopt.opt (shipped by ocaml-native-compilers) should be used instead of ocamlc/ocamlopt (shipped by ocaml)
    - of course ... they're not available on all arch! :-(
- ocaml-best-compilers is the package for you

Be nice to auto-builders

- ocaml-best-compilers
  - on arch supporting native code compilation (and hence optimized compilers) is provided by ocaml-native-compilers
  - on other archs is provided by ocaml-nox
  - **bug**: ATM ocaml-best-compilers is not versioned
    - thus you should build depend on both it **and** ocaml{,-nox}-<version> to ensure compiler/interpreter compatibility
    - this will change in the near future

Packaging OCaml libs

- let's assume now that next version of “Wonderful” depends on an OCaml library “Wow” ... of course not yet Debianized
  - you, skilled DD, decide to package it for Debian!
  - two scenarios have to be considered
    - Wow is a pure OCaml library
    - Wow is a mixed C/OCaml library
      - e.g. OCaml binding for a C library

Pure OCaml libs

- just create a “libwow-ocaml-dev” binary package
  - installing everything in a directory just below the Debian OCaml standard library directory: /usr/lib/ocaml/<version>
    - e.g. /usr/lib/ocaml/3.08/wow/
- follow the same advice on byte/native code conditional building we already discussed

Pure OCaml libs

- caveats (as usual):
  - on arch not supporting native code compilation only bytecode objects will be generated (and should be installed) while on other archs both byte and nativecode will
  - usually upstream's make “install” is smart enough to decide what to install
    - otherwise you can use the following rule of thumb to decide what should be installed
      - bytecode objects: *.cmi, *.cmo, *.cma
      - nativecode objects: *.cmx, *.cmxa, *.a, *.o

Mixed C/OCaml libs

- both byte and native OCaml code can be linked with C code
  - bindings of existing C libraries
  - implementation of C-specific parts (e.g. hw I/O)
- kinds of linking with C code:
  - static linking
    - no run-time dependencies / non-portable executables
  - dynamic linking (since OCaml 3.03)
    - run-time dependencies / portable (bytecode) executables

Mixed C/OCaml libs

- dynamic linking of C code requires a .so (usually named dll<libname>.so) that must be available at runtime
- in order to be found by the ocaml interpreter .so s must be located in the stublibs/ sub-directory of the ocaml standard library directory
  - e.g. /usr/lib/ocaml/3.08/stublibs/dllwow.so

Mixed C/OCaml libs

- packages shipping mixed C/OCaml libs should thus be split as follows
  - libwow-ocaml
    - runtime part of the library, basically the .so
    - depends: ocaml-base{,-nox}
  - libwow-ocaml-dev
    - development part of the library, basically everything else
    - depends: ocaml{,-nox}
  - other details in the sample ...

Mixed C/OCaml libs

- sample debian/rules (from “libzip-ocaml{,-dev}”)

```
Package: libzip-ocaml
Depends: ocaml-base-nox-3.08.3, ${shlibs:Depends}

Package: libzip-ocaml-dev
Depends: ocaml-nox-3.08.3, zlib1g-dev (>> 1.1.4),
          libzip-ocaml (= ${Source-Version})
```

OCaml libs dependencies

- on-library and inter-library deps represent a challenge for the Debian deps management
  - in order to preserve type-safety OCaml objects linking an external library includes md5sums of their modules interfaces
  - each change to interfaces (no matter if it is only an adjunct or not) will make for link time incompatibility
    - run-time performances have a cost!

Link time incompatibility

- example:
  - libwow-ocaml-dev 1.0 ships WowBasic interface with md5sum X
  - liburka-ocaml-dev 1.0 is built against WowBasic and internally stores X md5sum for it
  - libwow-ocaml-dev 1.1 is released and changes WowBasic md5sum to Y
  - linking an app against libwow 1.1 and liburka 1.0 will fail with an error message like
    - The files afile.cmi and anotherfile.cmi make inconsistent assumptions over interface WowBasic

OCaml libs dependencies

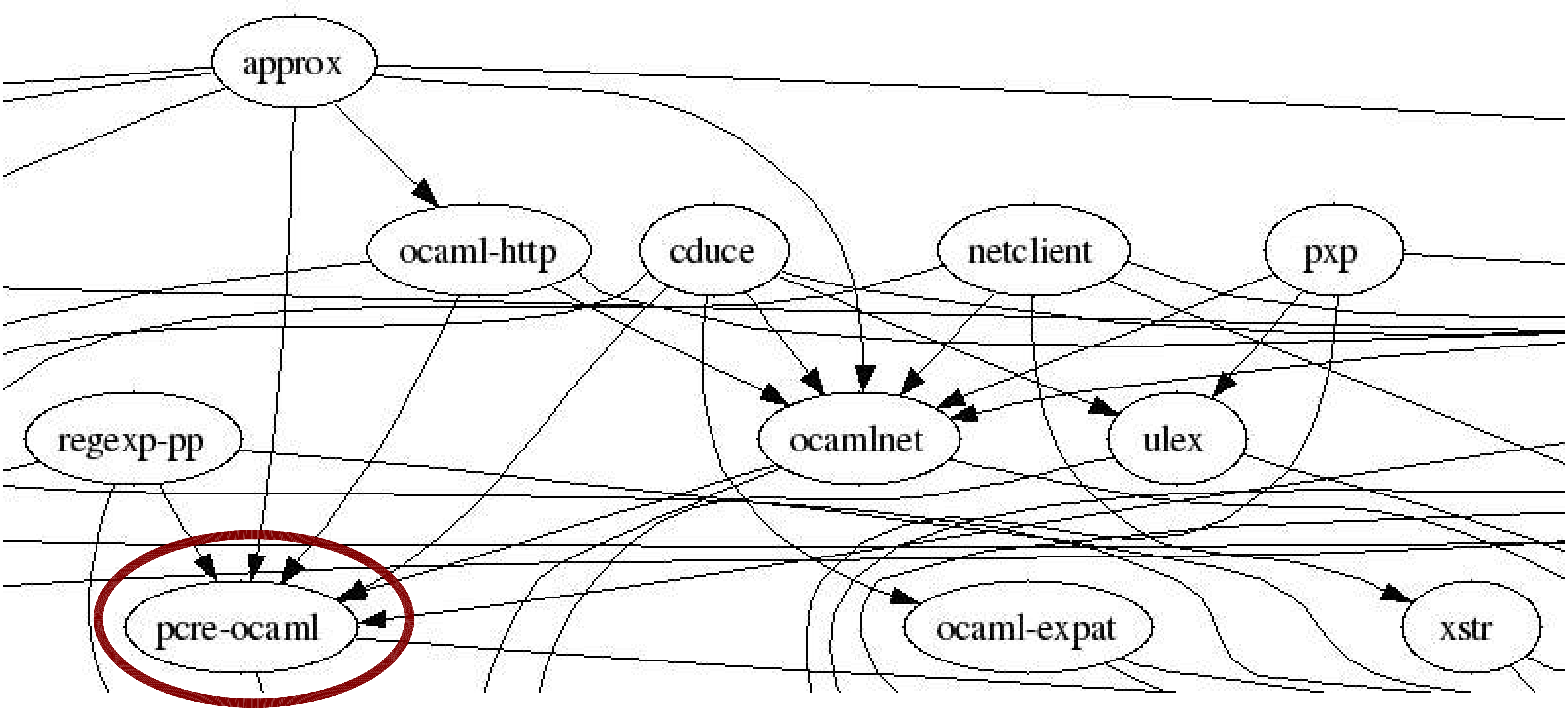
- analysis
  - Debian versioned dependencies are not enough
    - we need to express constraints like “depends on a version of libwow-ocaml-dev whose md5sums are that0, that1, and that2”
  - current solution
    - depends and build-depends on libwow-ocaml-dev >= x.y.z where x.y.z is the least version known to ship the right interface
    - each time an interface change, its maintainer inform maintainers of all depending packages asking rebuilding and dependencies fix :-(

OCaml libs dependencies

- issues with the current solution
  1. dependencies must be manually filled and bumped
  2. packages should be manually rebuilt each time an interface md5sum change
     - this happens quite often ...
     - ... and can be really painful on packages which are at the bottom of the dependency graph
     - let's have a look at the dependency graph ...

(Part of)

the OCaml packages build-dep graph



- each time pcre-ocaml releases you can hear ocaml maintainers screaming!

OCaml libs dependencies

- Etch solution
  - dh_ocaml
    - a new debhelper
    - maintains an “OCaml md5sums registry” of all installed OCaml interfaces with information on owner package and its version
    - given a set of OCaml objects extract from them information on which md5sums they need and, looking up the registry, compute package dependencies
    - create postinst/prerm scripts for registry book-keeping

OCaml libs dependencies

- the Etch solution
  - addresses issue 1. (manually filling of dependencies)
  - does not address issue 2. (manual rebuilding of depending packages)
- ... feel free to suggest any (good) idea

Caml

The End.