ASDF 3 TUTORIAL

Building CL Code: How? What? Why?

François-René Rideau
Google
Cambridge, MA

European Lisp Symposium, June 4th 2013

Outline

Outline

Build system

- transform source (for humans) into binary (for machine)
  - a bit like `make` for C
- enable division of labor
  - divide the source into separate components
  - multiple people can collaborate, each making changes to a few components
  - people in different teams, in same team, in same cranium.
- **system**: CL name for top-level unit of software management
  - In other languages they are called: library, package, module, bean, egg, class, archive. . .
- Challenges:
  - **Configuration**: find where is each file needed
  - **Dependencies**: build things in correct order
  - **Incrementality**: re-build iff changed

No build system

- What a manual load file might look like, `this-software-loader.lisp`

```
(load #p"/path/to/library1.lisp")
(defparameter *library2-directory* #p"/path/to/library2/")
(load (merge-pathnames #p"source/loader.lisp"
                       *library2-directory*))
(setf (logical-pathname-translations "LIBRARY3")
      '(("**;*.*.*" #p"/path/to/library3/*.*")))
(load #p"LIBRARY3:load-library3.lisp")
(load (compile-file
       (merge-pathnames "file1.lisp"
                        *this-software-directory*)))
(load (compile-file
       (merge-pathnames "file2.lisp"
                        *this-software-directory*)))
(load (compile-file
       (merge-pathnames "file3.lisp"
                        *this-software-directory*)))
```

Previous example with ASDF

- File `this-software.asd`

```
(defsystem this-software
   :depends-on (library1 library2 library3)
   :components
   ((:file "file1")
    (:file "file2" :depends-on "file1")
    (:file "file3" :depends-on "file1")))
```

Solved by ASDF

- Can find libraries w/o specific configuration
- Can find files *inside* library w/o extra configuration
- Configuration is done separately and uniformly
- dependencies: finer information is captured
- incrementality: only build what's needed
- more: portability, extensibility, etc.

ASDF descends from DEFSYSTEM

- **build system**: compile source files
- **specialized**: oriented toward CL software
  - not geared for arbitrary tasks with dependencies
- **in image**: also load software
  - totally unlike either `make`
  - maintain long-lived system state
- **declarative**: describe system dependencies
  - not imperative instructions on how to build
  - got more declarative as `DEFSYSTEM` grew older

Lisp build system history

- 196x: Manual `load` scripts
- 197x: Lisp Machine `DEFSYSTEM`
  - Chine Nual: components and manual rules
- 198x: kmp's MIT AI Memo 801, rer's MIT AI TR 874.
- 198x: Symbolics `SCT`
  - very elaborate, proprietary
- 1991: `MK-DEFSYSTEM`. 3.6i: 218kB.
  - free, portable, but complex, feature poor, not extensible
- 199x: also `defsystem` of Allegro, LispWorks
- 2002: ASDF, by Dan Barlow et al. 1.85: 38kB. 1.369: 77kB.
  - configurable, extensible, semi-portable.
- 2010: ASDF2, by Faré et al. 2.000: 138kB. 2.26: 198kB.
  - robust, portable, usable, upgradable
  - See "Evolving ASDF: More Cooperation, Less Coordination"
- 2013: ASDF 3, by Faré. 2.27: 409kB. 3.0.1: 459kB.
  - Fix 30-year old bug by making design coherent, new features
- Future: ASDF 4? `quick-build`? `XCVB`? Racket?

ASDF Features

- A simpler, better replacement for `MK-DEFSYSTEM`
- Use CLOS, don't support obsolete platforms
  - focus on SBCL and Unix
  - ported to a handful other implementations
- Inter-system configuration: find systems though `*central-registry*`
  - No need to edit a file for every system any more!
  - Typically, "symlink farms" – but Unix specific
- Intra-system configuration: none needed, use `TRUENAME`
  - Brilliant key idea establishes ASDF dominance
- Extensibility: use of CLOS to model dependencies
  - Example in SB-GROVEL

ASDF success

- Its configuration mechanism was a brilliant innovation
  - Before you laugh, compare to autotools, pkgconfig, etc.
- Extensible CLOS model also innovative, but not fully understood
  - Not by me until I rewrote it, not by Dan Barlow himself.
  - In many ways, a discovery, not an intentional design.
- Became *de facto* standard
  - `quicklisp`: over 700 libraries
- Now a key piece of community infrastructure
- Therefore cursed with **backward-compatibility**
  - if it's not backward. . .

ASDF 1 issues

- Many shortcomings:
  - Not very portable
  - Pathnames horror
  - A lot of bugs outside the common case
  - No standard way to load it
- Yet development stalled:
  - Users wait for new version before to rely on features / bug fixes
  - Implementers wait for users to demand new version before to change and break compatibility
  - Some distributions pre-package CL with ASDF pre-loaded, others don't
  - If an old one is pre-loaded, it's too late to upgrade with a version with bugs fixed

ASDF 2 Features

- **Hot-upgradable**: reverse incentive so development can happen
- **Portable**: 15 implementations, 4 OSes
- **Robust**: Massive bug fixes
  - Massive cleanup of internals. Pathname hell. Corner cases.
- **Faster**: Don't use lists when inappropriate
  - Can now scale to thousands of files
- **Configurable**: by *end-users*, not just developers
  - Domain-Specific Language for better configuration
  - Modular update of configuration
- **Usable**: a whole lot of small missing features
  - `(asdf:load-system :foo)` instead of `(asdf:operate 'asdf:load-op 'foo)`
  - `load-system test-system require-system`
  - `:defsystem-depends-on :force-not :encodings :around-compile :compile-check`

ASDF 3 Features

- Complete refactoring, fixed deep conceptual bugs.
- Deliver your system(s)
  - as single fasl (`fasl-op`)
  - as single lisp source file (`concatenate-source-op`)
  - as an executable program (`program-op`), with runtime hooks
- **Portability**: new library `UIOP`, includes `RUN-PROGRAM`
- **Condition Control**: muffle warnings, keep deferred warnings
- **naming**: multiple systems in `foo.asd`: `foo/bar`, `foo/baz`
- more: `:if-feature build-op force precompiled-system`...

Outline

What ASDF does

- Compile and load Lisp code in current image
- Locates software based on configuration
- Provide extensible object model to developers

What ASDF does not

- Download code (but `quicklisp` does)
- Solve version hell (only checks as specified)
- Build non-Lisp stuff (awkward)

Example minimal ASDF session

```
(require :asdf)
(asdf:load-system :inferior-shell)
(in-package :inferior-shell)
(run `(pipe (echo ,(* 90 137)) (tr "1032" "HOLE")))

;; More:
(run `(grep "Mem" "/proc/meminfo") :output :lines)
(asdf:test-system :inferior-shell)
```

Using ASDF, the safe way

```
;; CLISP alone won't accept :asdf
(require "asdf")

;; active implementations provide ASDF2 or later
#-asdf2 (error "You lose")

;; force ASDF2 to upgrade to your installed ASDF3
(asdf:load-system :asdf)
```

Using ASDF, the hard way

- see `slime/contrib/swank-asdf.lisp`
  - tries hard when the implementation doesn't provide ASDF.
- Even harder: see `lisp/setup.lisp` from `quux` (to be published)
  - configure asdf, twice, to work around cases of unsmooth upgrade.

Using CL-Launch from command-line

```
cl-launch -s this-software -i '(this-software:main)' \
       -- arg1 arg2
```

Using CL-Launch from script

```
#!/bin/sh
":" ; DIR="$(cd $(basename "$0");pwd)" #|
exec cl-launch -l ccl -S "$DIR//:" -i "$0" -- "$@"
exit |#
(some lisp code)
```

Outline

How to configure ASDF

- Source Registry
- Output Translations
- Optimization, verbosity, etc.

Default Installation Paths

- No need to configure if you use defaults
  - `~/.local/share/common-lisp/source/`
  - `/usr/local/share/common-lisp/source/`
  - `/usr/share/common-lisp/source/`
- FASLs under `~/.cache/common-lisp/`

Source Registry, via config file

- `~/.config/common-lisp/source-registry.conf`

```
(:source-registry
  (:directory "/myapp/src")
  (:tree "/home/tunes/cl")
  :inherit-configuration)
```

- Unlike ASDF 1, forgiving of no final /

Source Registry, via modular config file

-
  ```
  ~/.config/common-lisp/source-registry.conf.d/my.conf
  ```

  ```
  (:directory "/myapp/src")
  ```

Source Registry, via environment

```
export CL_SOURCE_REGISTRY=/myapp/src/:/home/tunes/cl//:
```

Source Registry, via Lisp evaluation

```
(asdf:initialize-source-registry
  `(:source-registry
     (:directory ,appdir)
     (:tree ,librootdir)
     :inherit-configuration))
```

Old Style central registry

- `(pushnew #p"/myapp/src/" asdf:*central-registry* :test 'equal)`
- Catch: ASDF 1 was unforgiving if you forgot the trailing /
- Magic: argument actually evaluated.
- ASDF 2 has `asdf::getenv`, now `uiop:getenv`
- No portable place to do it with ASDF 1.
  - e.g. `~/.sbclrc` on SBCL.
- source-registry can be configured in a decentralized way
  - **Each can specify what he knows**,
  - **none need specify what he doesn't**

Output Translations

- Where is the fasl for `foo.lisp` ?
- Multiple implementations and variants may use the same name
  - Allegro 9.0 SMP vs Allegro 9.0 normal
  - SBCL 1.1.0 vs SBCL 1.1.8
  - SBCL 1.1.0 x86 vs SBCL 1.1.8 $x86_{64}$
- Many ASDF1 extensions to move FASLs away, but hard to configure
- No consensus solution on where to put things
- `/src/foo.fasl`
  - `~/.cache/common-lisp/acl-9.0-linux-x86/src/foo.fasl`
  - `~/.cache/common-lisp/sbcl-1.1.8-linux-x64/src/foo.fasl`

Output Translations, via config file

- `~/.config/common-lisp/asdf-output-translations.conf`

```
(:output-translations
  (t (,cache-root :implementation))
  :ignore-inherited-configuration)
```

Output Translations, via modular config file

- `~/.config/common-lisp/`
- `asdf-output-translations.conf.d/foo.conf`

```
("/myapp/src/" ("/var/clcache" :implementation "myapp/src"))
```

Output Translations

- `export ASDF_OUTPUT_TRANSLATIONS=/:/some/cache/dir/:`

  ```
  (asdf:initialize-output-translations
    `(:output-translations
       (t (,cache-root :implementation))
       :ignore-inherited-configuration))
  ```

Output Translations, $PWD/sbcl-1.2-x86/foo.fasl

```
(asdf:initialize-output-translations
  `(:output-translations
    (t (:root :**/ :implementation :*.*.*))
    :ignore-inherited-configuration))
```

Using `quicklisp` and `clbuild`

- `(load "quicklisp/setup.lisp")` does it all
- I'm not sure about `clbuild` — use the source-registry

How do I find a library?

- Just use `quicklisp`
- Google it, search `Cliki`, `cl-user.net`
- Ask the community, e.g. `irc.freenode.net #lisp`

Where do I download it?

- Just use `quicklisp`
- To some place in your source-registry
- zero conf: `~/.local/share/common-lisp/source/`

Build script

- Optimizations: `(declaim (optimize ...)`
- Parameters: `(setf *compile-verbose* nil)`
- easy build script: `sbcl --load build.lisp`
- For portability, use `cl-launch` as above

Outline

Creating Basic ASDF Systems

- `foo.asd`

```
(asdf:defsystem foo
  :components
  ((:file "foo")))
```

Depending on other systems

- foo.asd

```
(defsystem foo
  :depends-on (:alexandria :cl-ppcre)
  :components
  ((:file "foo")))
```

Multiple files

- foo.asd

```
(defsystem foo ...
  :components
  ((:file "pkgdcl")
   (:file "foo" :depends-on ("pkgdcl"))
   (:file "bar" :depends-on ("pkgdcl"))))
```

Typical small system

- foo.asd

```
(defsystem foo ...
  :components
  ((:file "pkgdcl")
   (:file "specials" :depends-on ("pkgdcl"))
   (:file "macros" :depends-on ("pkgdcl"))
   (:file "utils" :depends-on ("macros"))
   (:file "runtime" :depends-on ("specials" "macros"))
   (:file "main" :depends-on ("specials" "macros"))))
```

Bigger system: divided in modules

```
(defsystem foo ...
  :components
  ((:module "base"
      :components ...)
   (:module "runtime"
      :depends-on ("base")
      :components ...)
   ...))
```

Logical Modules, same directory

```
(defsystem foo ...
  :components
  ((:module "base"
       :pathname ""
       :components ...)
   ...))
```

Pathname override

```
(:file "foo/bar")
(:file "foo" :pathname "../sibling-dir/foo")
(:file "foo" :pathname #p"../sibling-dir/foo.LiSP")
```

Sibling directories

```
(:file "../sibling-dir/foo")
(:module "../sibling-dir/foo")
(:file "foo" :pathname "../sibling-dir/foo")
(:file "foo" :pathname #p"../sibling-dir/foo.LiSP")
```

Punting on fine-grained dependencies

```
(defsystem foo
  :serial t
  :components
  ((:file "pkgdcl")
   ...
   (:file "main")))
```

Serial Dependencies

- Scope of `:serial t` is the current module or system
- not its submodules or systems.
- You can easily nest serial / parallel dependencies

Explicit Dependencies

- `:depends-on ("foo" "bar/baz" "quux")`

Good Style

- No `in-package`
- Only `defsystem` forms for `foo`, `foo/bar`
- Any classes, methods from `:defsystem-depends-on`
- No other methods, no side-effect, no pushing features

Other files in a project

- README, LICENSE, TODO, .git, etc.
- Using quickproject
    - Automatically create the skeleton

Outline

Distinct namespaces

- `find-package` vs `find-system`
- A system may or may not define a package of same name

Strategy 1: one package per system

- The traditional way
- system `foo`, package `foo`
- system `cl-foo`, package `foo` (yuck)
- system `cl-foo`, package `cl-foo`
- file `pkgdcl.lisp` or `package.lisp`

Strategy 1b: one package per subsystem

- Whether you subsystem is a second system or a module
- system `foo`, system `foo/bar`
- see `iolib`

Strategy 2: interface vs implementation package

- package `foo`, package `foo-impl`
- same system `foo`, or
- two systems `foo/interface` and `foo/implementation`
- See `cl-protobufs`

Strategy 3: one package per file

- More discipline, reduces mess
- dependencies implicit from defpackage
- See source code of `ASDF 3` itself
- `faslpath`, `quick-build` use it for dependencies!
  - if you :use or :import-from a package, load it first

`uiop:define-package` vs `defpackage`

- Part of UIOP, new in ASDF 3
- Works well with hot-upgrade
- Automation common patterns:
  - `(:mix "foo" "bar")`
  - `(:reexport "foo" "bar")`

.asd file syntax

- ASDF 3: now read in UTF-8 encoding, not `:default`
- ASDF 3: Now read in package `ASDF-USER`, not a temporary package
- Compatibility: NOT binding `*readtable*` and `*print-pprint-dispatch*`
- Deprecated: arbitrary code in `.asd` file
- Recommended: only calls to `defsystem`, use `:defsystem-depends-on`

ASDF-USER

- Issue: avoid name conflict issues between `.asd` files
- Old ASDF 1 & 2 read each file in its own temporary package
- ASDF 3 now all reads them in a common package `ASDF-USER`
- `ASDF-USER :use`'s `ASDF` and `UIOP/PACKAGE`
- Not `UIOP` due to conflict with `RUN-PROGRAM` in `SB-GROVEL`
- ASDF is not the right place for this “innovation”
  - If you’re CL programmer, you know your package discipline
  - If you don’t know your package discipline, you’re screwed anyway

Best package practice

- No need for `(in-package :asdf)` in your `.asd` file
- Read in shared namespace `ASDF-USER` — usual discipline applies
- If you bind new symbols, use `DEFPACKAGE` first.
- On ASDF 3, it `:use`'s `UIOP/PACKAGE` for its `DEFINE-PACKAGE`

Outline

Using Extensions: CFFI Grovel

```
(defsystem foo
  :defsystem-depends-on (:cffi-grovel)
  :depends-on (:cffi)
  :components
  ((:cffi-grovel-file "c-prototypes")
   (:file "lisp-code" :depends-on ("c-prototypes"))))
```

Character encoding, since 2.21

```
(defsystem foo
  :encoding :latin1
  :components
  ((:file "pkgdcl" :encoding :utf-8)
   (:module "russian" :encoding :iso-8859-5
      :components ((:file "bar" :encoding :koi8-r) ...))))
```

- `*default-encoding*` is now `:utf-8` since 2.31
- a boon for most programs, work predictably
- breaks a handful on unmaintained packages in quicklisp

Finalizers, since 2.23

```
(defsystem :asdf-finalizers-test
  :defsystem-depends-on (:asdf-finalizers)
  :around-compile
    "asdf-finalizers:check-finalizers-around-compile"
  :depends-on (:list-of :fare-utils :hu.dwim.stefil)
  :components ((:file "asdf-finalizers-test")))
```

- list-of:

  ```
  (defun foo (l)
    (check-type l (list-of string)))

  (asdf-finalizers:final-forms)
  ```

POIU

- `(asdf:load-system :poiu)`
- `(asdf:load-system :this-software)`
- Compile in a fork, load in current image.
  - Replay compilation errors in current image
- antifuchs 2007-2008: build ASDF systems in parallel
- fare 2009-2013: robust, portable, integrated to ASDF
- Deterministic by default given initial state
  - Faster option: more parallelism
- Can fork on `SBCL`, Single-threaded `CCL`, `CLISP`, `ACL`
  - Graceful fallback if no forking.
- Handle deferred warnings

Outline

Components, Operations, Actions

- `COMPONENT`'s describe your source code
  - e.g. `SYSTEM`, `CL-SOURCE-FILE`, `MODULE`
- `OPERATION`'s are stages of processing to perform on components
  - e.g. `COMPILE-OP`, `LOAD-OP`
- An `ACTION` is a pair of an `OPERATION` and a `COMPONENT`
  - e.g. `(cons (find-operation () 'load-op) (find-component "this-software" "file1"))`
- The dependency graph is a direct acyclic graph of `ACTION`'s
  - It is **not** a graph of components that depend on each other.

Plan first, then perform

- OPERATE calls TRAVERSE then PERFORM-PLAN
  - Factoring out PERFORM-PLAN was a recent change before ASDF 3.
- TRAVERSE walks the dependency graph and returns a plan
  - Traditionally, a LIST of actions to perform in order
  - Can be overridden. POIU returns a representation of the complete graph.
- PERFORM-PLAN walks the plan calling PERFORM-WITH-RESTARTS on each ACTION
  - PERFORM-WITH-RESTARTS sets up proper restarts and calls PERFORM

The graph is computed by `COMPONENT-DEPENDS-ON`

- Misnamed: actions, not components, have dependencies.
- Arguments: an operation designator, component designator
  - e.g. `(COMPONENT-DEPENDS-ON 'LOAD-OP '("this-software" "file2"))`
- CLOS: OO multi-dispatch on two arguments!
- Return a list of lists of operation designator and component designators
  - e.g. `((#<LOAD-OP> #<CL-SOURCE-FILE "this-software" "file1">))`
- CLOS: don't forget to append the `(call-next-method)`
  - we could have used the `APPEND` method combinator, but are not,
  - for historical backward compatibility reasons
- CLOS: inherit from mixins to achieve desired effects
- CLOS makes things very modular. Big win!

Component classes

- Usual classes

```
component
  module
    system
  source-file
    cl-source-file
      cl-source-file.cl
      cl-source-file.lsp
    static-file
    cffi-grovel-file
```

- Usual mixins
  - parent-component, child-component

Typical component tree

```
system
  cl-source-file-1
  cl-source-file-2
  module1
    cl-source-file-3
    cl-source-file-4
  cl-source-file-5
```

Operation classes

- `compile-op`, `load-op`
- `load-source-op`
- new in ASDF 3: `prepare-op`, `prepare-source-op`
- Also new in ASDF3, `bundle-op` and friends:
  - `fasl-op`, `load-fasl-op`
  - `monolithic-fasl-op`, `monolithic-load-fasl-op`
  - `concatenate-source-op`, `load-concatenated-source-op`
  - `program-op`
- Typical operations mixins (ASDF 3):
  - `selfward-operation`
  - `sideway-operation`
  - `downward-operation`
  - `upward-operation`

Action Files

- `OUTPUT-FILES`: output-translations in an `:AROUND` method
- `INPUT-FILES`: automation in `COMPONENT-SELF-DEPENDENCIES`
- An action is `NEEDED-IN-IMAGE-P` iff its `OUTPUT-FILES` is nil
  - Otherwise, it need not be `PERFORM`'ed again in current image if files up to date
  - Important notion implicit in ASDF 1&2, introduced by `POIU`
- ASDF 3's `TRAVERSE` may visit an action twice
  - once with `NEEDED-IN-IMAGE-P NIL` and oncep with it `T`

Outline

ASDF 2.26 was stable

- ASDF had been completely rewritten since ASDF 1
  - Now made portable, robust, usable, etc.
  - Everything had been touched except trivial things
- But core dependency traversal algorithm unchanged
  - To fix bugs, refactored out of spaghetti code, but
  - functionally equivalent, modulo bug fixes
- `TRAVERSE` was the holy relic passed by Dan Barlow
  - I didn't grok the design, it felt slightly wrong.
  - Couldn't change anything by fear of backward compatibility
- Remained only one bug to procrastinate on
  - All other bugs were wishlist items made difficult by current design

Failure to propagate dependency changes

- lp#479522 changes fail to trigger a rebuild across systems
  - explicitly disabled in `TRAVERSE`
  - In olden days, some have argued for the former bug as a "feature"
  - It was only a crock to work around lack of `:force-not`
- When you enable the obvious fix, it only works in current session
  - system2 depends-on system1
  - in one session, change system1, recompile it
  - in another session, compile system2 that didn't change
  - ASDF 1 and 2 fail to recompile system2

Not just between systems!

- More common failure mode:
  - Use a stateful macro, such as `DEFPACKAGE`'s `:use`
  - have `file1` define the macro, `file2` use it
  - modify `file1`, `file2` is not recompiled
- Other common failure mode:
  - have file1, file2, file3 with serial dependencies
  - file1 has changed, file3 hasn't
  - file2 completely breaks the build
  - you fix file2, and restart the build
  - ASDF 2 fails to recompile file3

Decades Old Dependency Bugs

- Cause: ASDF only checked timestamp for files of action
  - Doesn't even *try* to propagate timestamp from dependencies! lp#1087609
  - Need-to-recompile may be propagated only from current session
- Bug present in 1991 MK-DEFSYSTEM and the original 197X DEFSYSTEM
- *Optional* fix in Symbolics, Allegro, LispWorks defsystem
  - offer a different kind of dependencies than the default
  - broken by default (backward compatibility?)
  - not a complete fix in LispWorks
- Fixing the bug requires a complete rewrite of ASDF's `TRAVERSE`
  - Twice. Because then you find you need a correct dependency model
  - along which to correctly propagate timestamps.

Why never reported before?

- Usually not THAT big an issue
  - Most Lispers hack on one small system at once.
  - Usually you *interactively* use the `CONTINUE` restart after fixing bug.
  - When you change `file1`, you often need to change `file3`, too, anyway.
  - In doubt, you `:force` a build from clean or erase all the fasls.
- Now given in large systems built in batch with stateful macros... Ouch.
  - false positives and negatives waste time in building and testing
  - uncontroled non-determinism in testing is bad
  - Not your typical Lisp development style!

Live Programming vs Dead Programs

- Live Programming: code is mutable
  - Short feedback "OODA" loop. Low overhead (meta)computing.
- Dead Programs: code is immutable
  - Easier to analyze before it's run. Too late to debug afterwards.
- Both matter for the same reason:
  - **programmer interaction is a scarce resource**
  - On-line, adj.: The idea that a human being should always be accessible to a computer.
- Computing systems of the future should support both in synergy.
  - Live style to metaprogram dead style programs.
  - Zombie programs that resurrect on-demand.

Outline

Solution: road to ASDF3

- Propagate timestamps
  - This in turn necessitates a complete graph representation
- Introduce `prepare-op`
  - This means refactoring downward propagation away from `TRAVERSE`
- Refactor `traverse` and the `operation` classes
  - This means reorganizing the source code
- Split the code into files so it makes sense
  - Implement `monolithic-concatenate-source-op`
  - Merge in and fix the `asdf-bundle` infrastructure
  - Recursively use new `traverse` to walk the partial plan for an action
- It now makes sense to have a separate portability layer
  - Implement `UIOP`, spend time making it a quality library
- Many cleanups and new features are now unlocked
  - Spend a lot of time implementing them robustly
- Some new features are oh so slightly backward incompatible
  - Spend a lot of time fighting the community, and losing

PREPARE-OP

- introduced to fix a conceptual bug in the ASDF object model.
- “load the dependencies of a component and its parents”
- explicitly `depends-on`’ed by `LOAD-OP` and `COMPILE-OP`
- Propagates *upward* in the component hierarchy, not *downward*
- `TRAVERSE` special cases such dependencies no more

`TRAVERSE` was gutted out

- Not only bug fixes, but much simpler, sensible semantics
  - Now propagating timestamps along a graph and that only
  - Refactored into reusable higher-order functions and objects
- The object model now actually makes sense, and can be extended
  - No more implicit descending into children components
  - Inherit from `downward-operation` for such propagation
- methods take a `plan` object, NIL for actual action
  - Informed by interface-passing-style and experience with `POIU`
  - Was necessary to get `BUNDLE-OP` right portably
- Many many thanks to antifuch's `POIU`

COMPONENT-DEPENDS-ON is now more powerful

- can express dependencies on arbitrary operation objects
- Supported: depend not just on siblings
- Supported: express arbitrary build graphs
- Deprecated: operations with different options
- Deprecated: depending on component in other system

COMPONENT-DO-FIRST is no more

- It used to specify some dependencies that were skipped
- if no re-build was triggered based on local timestamps;
- ASDF 1 didn't let the users control it,
- ASDF 2 only let you control it since 2.017 or so.
- In ASDF 3, `NEEDED-IN-IMAGE-P` mechanism supersedes `COMPONENT-DO-FIRST`
- `COMPONENT-DEPENDS-ON` is used for all dependencies.
- Use `:in-order-to` everywhere you used to use `:do-first`, if ever.

IF-FEATURE

- new attribute of COMPONENT
  - accepts an arbitrary feature expression
  - e.g. `:if-feature (:and :sbcl (:or :x86 :x86-64))`
  - Beware: no magic reading in keyword package — use : syntax
- Replaces the misguided `:if-component-dep-fails` attribute of MODULE
  - could not be salvaged when refactoring TRAVERSE
  - Dropped that attribute and the accompanying `:feature` feature
  - *Limited* backward compatibility just for SB-GROVEL and co.

Outline

Performance

- ASDF3 ~70% slower than ASDF2
  - Slightly faster when `*RESOLVE-SYMLINKS*` is false (default true)
  - ASDF2 much faster than ASDF1: don't (ab)use LIST data structures
- Underneath, ASDF3 does much more work, correctly
- Cache expensive computations in hash-table in dynamic variable

One package per file

- ASDF 3 was rewritten in the style of `faslpath` and `quick-build`
- Each file has its own `DEFPACKAGE`
- Actually uses `UIOP/PACKAGE:DEFINE-PACKAGE` for hot-upgrade and reexport
- Future: actually support `faslpath` or `quick-build` dependencies?

CONCATENATE-SOURCE-OP

- build a single Lisp file from all the source in a system
- Variant `MONOLITHIC-CONCATENATE-SOURCE-OP` to transclude dependencies
- Used by ASDF itself to split it in multiple files
    - ASDF has more than doubled in size between ASDF 2.26 and ASDF 3.0.1
    - Had already increased manifold since ASDF 1.
    - It just does that much more work.
    - The ASDF 1 bits have actually been much simplified.

`ASDF-BUNDLE` was merged into ASDF.

- Fewer headaches for users of ECL
- More features for users of other implementations
- Can create a single fasl per system with `fasl-op`
- Makes software delivery easier.
- Support for pre-compiled systems.
- SBCL patch to use that for contribs.

PROGRAM-OP

- create standalone executables on supported implementations
- Supported: `clisp ccl cmucl ecl lispworks sbcl scl`
- See example in `test/hello-world-example.asd`
- Uses image hooks above.

`BUILD-OP`

- A generic operation that will do the "right thing" for each system
- Not super supported yet, but the future(?)
- TODO: `generic-load-op`, `build-op`, etc.

FORCE and FORCE-NOT

- Fixed `:force` to actually work as advertised by ASDF 1.
- Accepts `:all`, `t`, or a list of system names
- Also implemented `:force-not` and based on it `require-system`
- Can't force builtin systems (e.g. `SB-BSD-SOCKETS`)
- WARNING: rpg may revert that `FORCE` has precedence over `FORCE-NOT`

System FOO/BAR/BAZ

- name be recognized by defsystem as located in `foo.asd`
- Somewhat backward compatible
  - in ASDF2, you had to manually ensure `foo.asd` was loaded beforehand
  - in ASDF3, works automatically
- Allows sensible way to define multiple systems in an `.asd` file.
- See `iolib.asd`
- Internals: grep for function `primary-system-name`

Deferred warnings

- Don't drop info on yet undefined functions
- Supported: `allegro ccl cmucl sbcl scl`
- Disabled by default.
- Enable it: `#+asdf3 (setf asdf::*warnings-file-type* (asdf::warnings-file-type))`
- Dump info for `foo.lisp` in `foo.sbcl-warnings`
- Checked at the end of the build on each system
- In a method to `PERFORM (COMPILE-OP SYSTEM)`
- As if a `WITH-COMPILATION-UNIT` around each system

TRUENAME resolution

- Now can be reliably turned off:
- `(setf asdf:*resolve-symlinks* nil)`
- Useful if `TRUENAME` is slow or bogus on your OS
- Necessary if using symlinks to content-addressed storage
  - e.g. the Google build system

VERSION strings

- Warnings if you don't follow the convention of `VERSION-SATISFIES`
- Regex: "[0-9]+(
  .[0-9]+)+"
- version-satisfies now uses uiop:version<= for comparison
- No more checking for a same major version number
- Was undocumented behavior since ASDF 1, still in version-compatible-p

`:VERSION` spec in `DEFSYSTEM`

- Now also accept `(:read-file-form <path> :at <formpath>)`
- Now also accept `(:read-file-line <path> :at <linenum>)`
- `:at` optional, defaults to 0, 0-based
- `<formpath>` as per `UIOP:ACCESS-AT`
- e.g. `(:read-file-form "specials.lisp" :at (2 2))`
- same as `(:read-file-form "specials.lisp" :at (third third))`
- Easier to manage versioning from master location
- See `poiu.asd`, `poiu.lisp`

Self-Upgrade

- ASDF 3 will always start by automatically upgrade itself
- Proviso against downgrade, with warning
- Just have the `asdf/` tree somewhere in your `source-registry`
- Only sane way to deal with potential upgrade
- Otherwise, if any recursive dependency loads ASDF, *kaboom*
- not algorithmically detectable: `.asd` files not declarative

Deprecated COMPONENT-PROPERTY

- also the :PROPERTIES initarg of DEFSYSTEM
  - Still works for now
  - To be retired before a hypothetical future ASDF 4.
- Used by few, never with any name convention.
  - Recommended instead: use DEFCLASS a subclass of ASDF:SYSTEM
    to add new slots and/or initargs. Then use :defsystem-depends-on and :class in defsystem
- We added :homepage :bug-tracker :mailto :long-name to defsystem
  - The only common metadata used, though never in the same way

DEFSYSTEM Internals

- Completely refactored. Many renamings after checking Quicklisp.
- Some sorry features were excised
- `OPERATION-DONE-P` is simplified and now well-specified
- `FIND-COMPONENT` will pass component objects through
- a corresponding `FIND-OPERATION` replaces `MAKE-SUB-OPERATION`

Convenience methods

- Added to many exported generic functions:
- `input-files output-files component-depends-on operate ...`
- You can e.g.: `(input-files 'compile-op '(system1 "file1"))`
- Instead of `(input-files (make-instance 'compile-op) (find-component 'system1 "file1"))`
- Makes it much easier to interact with ASDF at the REPL
- Debugging ASDF extensions and modifications easier

inline-methods can now be unqualified

- Fixes lp#485393
- Great for defining test-op methods:
  - `(defsystem foo/test ... :perform (test-op (o s) (symbol-call :foo-test :run-tests)))`
- NB: Unhappily, this is works in ASDF 3 but is circular in ASDF2:
  - `(defsystem foo ... :in-order-to ((test-op (test-op foo/test))))`

`:ASDF3` in `*features*`

- `#+asdf3` present since pre-release ASDF 2.27
- Typically used in `:depends-on (#-asdf3 :asdf-driver)`
- Can protect code not supported in all of ASDF 1, ASDF 2
- No support for ASDF $<$ 2.014.6 (original Quicklisp ASDF)

SLIME support

- Significantly enhanced (Use 2013-02 or later)
- For around-compile hook support, in `~/.swank.lisp` add:
- `(in-package :swank)`
- `(pushnew 'try-compile-file-with-asdf *compile-file-for-emacs-hook*)`

Documentation

- `asdf.texinfo` only covers the `DEFSYSTEM` part
- It doesn't cover new operations or internals
- `UIOP` is only documented in docstrings
- All in all, very limited. But examples abound.

Tests

- Regression test framework massively improved
- Regression-driven, with plenty of new test cases
- Still far cry from covering all desired behavior
- UIOP largely untested
- Automated tests: `abcl allegro allegromodern ccl clisp`
- `cmucl ecl ecl_bytecodes lispworks sbcl scl xcl`
- Manual tests: `gcl2.6 genera lispworks-personal-edition`
- Untested on `cormancl mkcl rmcl`

Outline

UIOP

- “Utilities for Implementation- and OS- Portability”
- a separately-usable library for Common Lisp runtime support.
- Pathnames, Filesystem, `RUN-PROGRAM`, compilation, image. . .
- Formerly known as `ASDF-DRIVER`, formerly `ASDF-UTILS`
- Includes bits from `ASDF`, `XCVB-DRIVER`, `TRIVIAL-BACKTRACE`, etc.
- Transcluded in `asdf.lisp` thanks to `MONOLITHIC-CONCATENATE-SOURCE-OP`
- Also more portable alias `:asdf-driver` for versions before 2.32
- Use it: `:depends-on (#-asdf3 :asdf-driver)` or if you insist `:depends-on (:uiop)`

Portability

- Updates on each and every implementation
- 9 active: `abcl allegro ccl clisp cmucl ecl lispworks sbcl scl`
- 6 mostly dead: `gcl2.6 genera xcl cormancl rmcl mkcl`
- Variants: `allegromodern lispworks-personal-edition ecl_bytecodes`
- Festering horror: pathnames.
- Worst: “logical” pathnames.

CL Pathnames: THE HORROR!

- CLHS horribly misdesigned. Countless bugs in ASDF and CL implementations.
- **FAIL:** #p"foo/bar" can never be portable (separator OS dependent)
  - Pray your `*default-pathname-defaults*` isn't "logical"
- **FAIL:** no sure way to make a non-wildcard pathname
  - Pray your filesystem doesn't contain files with * in name
- **FAIL:** even `MAKE-PATHNAME` isn't portable
  - Host, device, `:unspecific`, wildcard escaping, etc.
- **FAIL:** even `MERGE-PATHNAMES` isn't portable
  - Host and device defaulting **will** bite you eventually
- **FAIL:** No portability across implementations on a same OS
- **FAIL:** logical pathnames are unusable in practice. Avoid.
  - Not portable, inefficient, not modular, unusable `DIRECTORY`...
  - If you can initialize them portably, you don't need to use them.
- **FAIL:** Can never be fixed
  - implementers each maintain their own backward-compatibility
  - users can't portably fix it and hook into `OPEN`, `LOAD`, `#P`, etc.

Semi-solution: `UIOP/PATHNAME`

- Don't use #P"foo/bar", have your own string parser
- ASDF uses `PARSE-UNIX-NAMESTRING` for relative path specs
  - So path specs are portable, even when not on Unix,
  - as long as you don't use in names any character that is
  - a valid separator, wildcard or escape on *any* platform.
- Do our own pathname type defaulting.
- Use `MERGE-PATHNAMES*`, `MAKE-PATHNAME*` instead of CLHS primitives
- `SUBPATHNAME`, `PARSE-UNIX-NAMESTRING`, `PARSE-NATIVE-NAMESTRING`
- `ENSURE-PATHNAME`
- Many more working around CLHS braindeadness
- Supersedes `cl-fad`
- Still, can't save you from impl-dep wild pathnames

DEFINE-PACKAGE

- In package `UIOP/PACKAGE`, also exported from `UIOP`
- A better `DEFPACKAGE` variant
- Works well for hot upgrade, fixes existing packages
- Has `(:mix pkg1 pkg2 pkg3 ...)` instead of `(:use ...)`
- Also has `(:reexport pkg1 pkg2 pkg3 ...)`
- Also has `PACKAGE-DEFINITION-FORM` to inspect current package state
- Still within limitations of CL packages.

UIOP/IMAGE, image lifecycle support

- Included in UOIP
- Must call RESTORE-IMAGE early during program initialization
- Done implicitly by DUMP-IMAGE with :executable t
- Will initialize *COMMAND-LINE-ARGUMENTS* and more
- REGISTER-IMAGE-RESTORE-HOOK,
  REGISTER-IMAGE-DUMP-HOOK

RUN-PROGRAM

- replaces the broken old misdesigned `RUN-SHELL-COMMAND`
  - Do NOT use `RUN-SHELL-COMMAND`
  - Misdesign copied from `MK-DEFSYSTEM`
- `RUN-PROGRAM` portable to *all* Windows & Unix CL (not Genera)
- Can sensibly capture output, via `SLURP-INPUT-STREAM`
- `(run-program '("ls" "-l") :output :lines)`
- Supersedes `XCVB-DRIVER:RUN-PROGRAM/`
- Higher-level interface available in system `inferior-shell`

Conditions control

- Will selectively muffled conditions
- Muffle `*UNINTERESTING-COMPILER-CONDITIONS*` around `COMPILE-FILE`
- Muffle `*UNINTERESTING-LOADER-CONDITIONS*` around `LOAD`
- Muffle `*UNINTERESTING-CONDITIONS*` around either
- Empty by default for backward-compatibility by user demand
- Suggested: `(setf uiop:*uninteresting-conditions* (uiop:*usual-uninteresting-conditions*))`
- Supersedes code from `XCVB-DRIVER`, `QRes`, `QPX`

COMPILE-FILE*

- On ASDF3, does the Right Thing(tm) on all implementations
- Supports output-translation, deferred-warnings, etc.
- Supports ECL and MKCL linkable object in addition to FASL
- Supports `.lib` in CLISP, `CFASL` in SBCL, etc.

UIOP-DEBUG

- load favorite debugging primitives in current package
- Put path to yours in `uiop/utility:*uiop-debug-utility*`
- See mine in `uoip/contrib/debug.lisp`
- `(DBG :tag expr1 expr2 ...  last-expr)`

Also in `UIOP`

- common-lisp: compatibility with obsolete CL implementations
- utilities: plenty of general-purpose utilities
- filesystem: `chdir`, `directory-files`, etc.
- stream: `with-safe-io-syntax`, `format!`, `with-temporary-file`
- os: `getenv`, etc.
- configuration: help with configuration

Documentation

- `UIOP` is only documented in docstrings

Outline

How to implement an extension

- define new `component` and/or `operation` subclasses
- define appropriate methods:
  - at least `component-depends-on`, `input-files`, `output-files`, `perform`
  - also `operation-description` for debugging.
- see `cffi/grovel/asdf.lisp`
- see `cl-protobufs/asdf-support.lisp`

Troubleshooting ASDF

- Look at error messages
- Look at the backtrace
- Trace relevant functions
  - `perform-plan`, `perform`
  - `input-files`, `output-files`

Often requested: load-only component class

- some kind of `CL-SOURCE-FILE` for which `LOAD-OP` means `LOAD-SOURCE-OP`
- Beware: defeats executable creation!
- Maybe instead you want run-time evaluation in your Lisp file:
- `(foo '(some data))` or even `(eval '(some expression))`

Support other languages?

- Can they be loaded in-image?
- Yes: CL becomes a platform (e.g. use `cl-python`)
- No: second class citizens

Dependency generation?

- `asdf-dependency-grovel`

Components of type `SYSTEM` ?

- Yes: that's what `ASDF:DEFSYSTEM` does!
  - use `:depends-on (foo)`
- No: `mk-defsystem` idiom, not supported
  - do NOT use `:components ((:system foo))`

Horror .asd file?

- `mcclim.asd` before ASDF 3 refactoring
- `gbbopen.asd` is still pretty complex
- Really, any `.asd` file with non `defsystem` forms.

Outline

Future Work?

- More declarative `DEFSYSTEM`
  - Forbid or specially treat `.asd` files with forms beside `defsystem`
- Keep deferred warnings by default?
  - Must fix tens of systems in quicklisp that would fail on SBCL.
- Make further cleanups to the object model?
  - Never going to happen: if it's not backward...
- Document!
- Move to `XCVB`, `quick-build`
  - or move to Racket? R7RS?

Lessons Learned

- ASDF design discovered by evolution, not intelligent design
  - Big design constraint was interactive development in live image
- It is possible to write code portably in CL, by using UIOP.
  - Whether it's a good idea is a different question
- Some things in CL can never be fixed. e.g. pathnames.
  - Not even possible to start thinking of better
  - namespace management, continuations, type systems, etc.
- The test suite matters a whole lot
  - TODO: automate tests with `quicklisp` and `cl-test-grid`

ASDF 3 is now available in stores near you

- `http://common-lisp.net/project/asdf/`
- Download and install in your source registry
    - Demand it from your implementation vendors!
    - Meanwhile, ASDF 2 ubiquitous at long last.
- ASDF 3 needs new maintainers
    - Must remain backward compatible — be gentle with it!