Delivering Common Lisp Applications with ASDF 3.3

Pushing the Envelope or Therapeutic Fury?

François-René Rideau, *TUNES Project*

European Lisp Symposium, 2017-04-03

http://github.com/fare/asdf2017/

This Talk

This Talk: A progress report on *ASDF*,

de facto standard build system for *Common Lisp*,

continued evolution of the tradition of *Lisp*,

a language discovered, not created, in *1958*.

Plan

Some Background

Recent ASDF Progress

Lessons for build systems in any language

Some Background

What makes ASDF different

`DEFSYSTEM`: compile & load "systems" *in-image*

C analogs: `make`, `ld.so`, `pkg-config`, `libc`

Primarily designed for CL code

`ASDF`: extensible in CL itself via OO protocol...

... can be made to build anything!

Big focus on backward-compatibility

"If it's not backwards, it's not compatible"

Some History

1976 Unix `Make`

<1981 Lisp Machine `DEFSYSTEM`

1990 `MK-DEFSYSTEM`: portable, pre ANSI

2001 0.5 kloc danb's `ASDF`: extensible OO build

2004 1.1 kloc danb's `ASDF 1.85`: de facto standard

2010 3.3 kloc `ASDF 2`: robust portable configurable

2013 9.7 kloc `ASDF 3`: correct, delivers, `UIOP`

2014 11.3 kloc `ASDF 3.1`: CL as scripting language

2017 12.8 kloc `ASDF 3.2`: link C, `launch-program`

2017? 13.2 kloc `ASDF 3.3`: proper phase separation

Current Limitations

Not declarative enough:

CL has ubiquitous global side-effects

One global set of system versions

One global syntax

Compared to bazel, missing:

cross-compilation, determinism, scalability…

New in ASDF

Previously on this show...

`ASDF 3.1` (2014) ELS, ILC demos:

CL as a scripting language

`Bazelisp` (2016) ELS demo:

scalably build executables

with statically-linked C extensions

2017 Innovations

ASDF 3.2 (January 2017):

Application Delivery with static C libraries

Asynchronous subprocesses with `launch-program`

Source Location Configuration improvements

Deprecation infrastructure

ASDF 3.3 (Real Soon Now 2017):

Proper Phase Separation

Application Delivery with static C libraries

Previously

Extract functions & constants: `:cffi-grovel-file`

Compile & link wrappers: `:cffi-wrapper-file`

New in ASDF 3.2 + cffi-toolchain (2017)

Plain C code to link to: `:c-file`

`cffi-toolchain`: one place to deal with C

Not (yet) a general-purpose C build system

Missing per-system compile and link flags

Example system using C code

```
(defsystem "foo" :depends-on ("cffi")
  :defsystem-depends-on ("cffi-grovel")
  :serial t
  :component
  ((:cffi-grovel-file "interface-extraction")
   (:cffi-wrapper-file "complex-interfaces")
   (:c-file "some-c-code")
   (:cl-source-file "some-lisp")))
```

Loading a system

2001: `(asdf:operate 'asdf:load-op "foo")`

or "short" `(asdf:oos 'asdf:load-op "foo")`

2009: also `(asdf:load-system "foo")`

2013: also `(asdf:oos :load-op "foo")`

2014: also `(asdf:make "foo")`

Making a binary

ASDF 3.0 (2013): image-based delivery

devel. image `(asdf:oos :image-op "foo")`

standalone app. `(asdf:oos :program-op "foo")`

Any C extensions must be dynamically linked

ASDF 3.2 (2017): with static C extensions

```
(asdf:oos :static-image-op "foo")
```

```
(asdf:oos :static-program-op "foo")
```

Demo time!

```
(asdf:make "workout-timer/static")
```

Asynchronous subprocesses

ASDF 3.1 (2014): `run-program`

synchronous subprocesses (Unix *and* Windows)

exit status, optionally error out if not successful

I/O redir.: inject into stdin, capture stdout & stderr

ASDF 3.2 (2017): `launch-program`

asynchronous subprocess (Unix *and* Windows)

exit status, waiting for processes, killing them

I/O redirection, interaction through streams

Asynchronous Limitations

No event loop to which to integrate

No general signal support

Can make do with pipes and macros

Still *way* better than shell programming!

For more serious system programming: `iolib`

It requires a C extension—but that's now easier!

Source Location Configuration: Before

`ASDF 1` (2001): push to `*central-registry*`

early in `~/.sbclrc` — repeat for each impl!

`ASDF 2` (2010): declare hierarchical source-registry

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

Inherit wider configuration, or override it, from CL…

or from shell: `CL_SOURCE_REGISTRY`, XDG vars

Default ∋ `~/.local/share/common-lisp/source/`

`ASDF 3.1` (2014), also `~/common-lisp/`

Source Location Configuration: After

Recursing through large trees can be very slow

2015: `.cl-source-registry.cache` for a `:tree`
Regenerate with a standard `#!/usr/bin/cl` script:
`asdf/tools/cl-source-registry-cache.lisp`

Harkens back to `ASDF 1` style symlink farms, but only for impatient power users with lots of systems

2015: also multicall binaries with `cl-launch`

2016: expose interface to XDG base directory vars
XDG also on Windows, modulo ASDF adaptation

`ASDF 3.2` (2017): the new release has it all

Deprecation Infrastructure

`asdf:run-shell-command` was a *very* bad API

Use `uiop:run-program` instead, as per docstring

In 3.2, using it now issues a `style-warning`

In 3.3, full `warning` if used, **breaks** on SBCL

In 3.4, `cerror` if used, breaks everywhere

In 3.5, `error` if *not deleted yet* from codebase

`uiop/version` makes staged deprecation easy

Part of `UIOP 3.2`, part of `ASDF 3.2` (2017)

Proper Phase Separation

`ASDF` extensions: with CLOS. How to load one?

Using `ASDF`!

What if it itself relies on extensions?

Build in multiple phases.

What if an extension is modified?

Rebuild everything that transitively depends on it.

And what if a library is needed in multiple phases?

Only build it once.

Improper Phase Separation

`ASDF 1` had only two phases: plan, then perform

(that was its least bug—see ASDF 2 & 3 papers)

If *defining* system `foo` depends on `ext`:

`ASDF 1`: `foo.asd` has `(oos 'load-op "ext")`

`ASDF 2`: `:defsystem-depends-on ("ext")`

`ASDF 3`: make it usable despite package issue

Kind of works. `ASDF` unaware it's recursively called

Across phases: extra builds, *missing rebuilds*

Separating Phases

`ASDF 3.3`: loading the asd file is itself an *action*!

`define-op` — for *primary* systems.

Big tricky refactoring of `find-system`:

`find-system > load-asd > operate > perform > load*`

`ASDF 3` had a cache: only call `input-files` once

(its API functions define a pure attribute grammar)

`ASDF 3.3` extends it to a multi-phase *session*

One `plan` per phase, a `session` across phases.

Traversal of the Action Graph

Many kinds of traversals of the graph of *actions*:

`ASDF 1`: mark as needed, in this image

`ASDF 3`: mark as needed, in any previous image

`ASDF 3`: go thru all dependencies, e.g. to get list

`ASDF 3.3`: query whether up-to-date

`ASDF 1`: 1 bit (keep), plus *"magic"* (=bugs)

`ASDF 3`: 2 bit (needed-in-image), plus *timestamp*

`ASDF 3.3`: 3 bit (done), plus *phase*

Proper Phase Separation: Incompatibilities

`:defsystem-depends-on` to systems in same file
(as in the latest `iolib` release)

`clear-system` inside `perform`
(as in lots of systems that use `prove`)

`operate` in a CL file or `perform` method
(temporary exception: `(require …)`)

Now very bad taste: misnamed secondary system
(used all over: once a `ASDF 1` colloquialism)

Proper Phase Separation: How good are we?

Build extensions is a universal need

Most build systems (Make…): on par with `ASDF 1`

Language-specific builds can be greater (Racket…)
but not general-purpose.

Bazel: non-extensible extension language

Proper Phase Separation: How good are we?

Build extensions is a universal need

Most build systems (Make…): on par with `ASDF 1`

Language-specific builds can be greater (Racket…)
but not general-purpose.

Bazel: non-extensible extension language

`ASDF` is on the bleeding edge!?

Lessons and Opportunities

Evolving ASDF

`ASDF` sucks—less

Amazing how much is done with how few klocs

Ceiling: CL's model of global side-effects

Impedes declarativeness, reproducibility, etc.

Evolution is costly (yet consider the alternative)

Gets worse as the code- and user- bases grow

Backward-incompatible change: takes 1-2 years…

Quicklisp: fix it all! And/or issue warnings and wait.

Beyond ASDF?

The ultimate purpose of a build system is:

Division of labor

Opportunity for much a better build system.

What design is worth starting from scratch?

Core: Pure FRP, CLOS-style OO, versioning

plus staging, virtualization, instrumentation

http://j.mp/BuildSystems

Enjoy ASDF!

Common Lisp keeps improving, slowly:

AI, e-commerce, games…

Web, desktop or mobile apps—and now scripts `#!`

`ASDF` also keeps improving, slowly.

If there were demand, it could improve faster…

Donate to `ASDF` through the CLF!

https://common-lisp.net/project/asdf/