ASDF: Another System Definition Facility

TC Lispers Meeting

Robert P. Goldman

SIFT, LLC

2009-08-18 Tue

Introduction

- Purpose of ASDF: Support portable design of building of CL systems, across multiple implementations and deployment sites.
  - Somewhat analogous to `make` or `ant`, but CL doesn't have libraries in the same sense as C or Java.
- Developed by Dan Barlow with many outside contributions by now
- Maintained by Gary King

Outline of talk

- ASDF for library users
  - installation
  - use
  - some handy extensions
- ASDF for library authors
- Design of ASDF and how to extend it

Using ASDF-defined libraries

- Very simple!
- Install ASDF
  - You may already have it – try `(require :asdf)` and see if that works.
  - ASDF is available as a tarball, or you may pull from the git repository. See http://www.cliki.net/ASDF
- Load `asdf` in your lisp init file.
- Load an ASDF system as `(asdf:oos 'asdf:load-op` *`system-name`*`)`
  - `oos` is *Operate on System*
  - `load-op` is the name of an *operation class*
  - In most cases, this will compile the system and then load it.
- Look at http://www.cliki.net for pointers to many CL libraries built on ASDF.

Installing an ASDF-defined library

1. ASDF needs a place to look for system definition files.
   - `asdf:*central-registry*`
   - a list of directories in which to look for system definition files
   - likely to be prepopulated — feel free to add your own directories
2. Find the library – some standard ways:
   - get a tarball
   - pull library from a source control system
3. Put the library somewhere
4. Help asdf find it in one of two ways:
   1. Add a symbolic link to the `.asd` file to a directory listed in `*central-registry*` (problematic for windows) or
   2. Add the directory where your source files are to the `*central-registry*`.

Example:

1. I pull CL-JSON from its darcs repo into `~/lisp/cl-json/`
2. I add this to my CL init file:
   `(push "/Users/lisp/cl-json/" asdf:*central-registry*)`
3. `(asdf:oos 'asdf:load-op :cl-json)` does the job!

Example:

1. I pull CL-JSON from its darcs repo into `~/lisp/cl-json/`
2. I add this to my CL init file:
   `(push "/Users/lisp/cl-json/" asdf:*central-registry*)`
3. `(asdf:oos 'asdf:load-op :cl-json)` does the job!

Variant for step 2 — the link farm

- A directory, `~/lisp/asdf-systems/` is in my `*central-registry*`
- I *symlink* `cl-json.asd` into `~/lisp/asdf-systems/`

Enhancements

- asdf-binary-locations
- asdf-install

ASDF Binary Locations

- By default, ASDF drops compiled files in the same directory as the source files.
- Problematic for some use cases, such as:
  - You work with multiple lisp implementations
    - binaries for different implementations have the same file extension;
  - Your source files are in distribution directories that aren't writeable.
- asdf-binary-locations is a simple and elegant extension to ASDF. By default builds separate subdirectories for binaries based on implementation info.
  - E.g., `asd-finder/allegro-8.1m-64bit-macosx-x86-64`
- Notes
  - Developed by Gary King, current ASDF maintainer
  - May someday be packaged with ASDF

ASDF Install

- Helps manage the fuss of finding, unpacking and registering ASDF libraries.
  - Theoretically, works with libraries using other build systems (e.g., `MK-DEFSYSTEM`) – don't try it.
- How to:
  1. Find ASDF-INSTALL on cliki and install it
  2. Configure ASDF-INSTALL to tell it where to put the systems it downloads
  3. (`asdf-install:install` *sysname*)
     - Will try to find a cliki page for *sysname*
     - From there will pull a tarball and a GPG signature
     - Will check the GPG signature and, if it's ok, compile and install
     - Will try to recursively install missing dependencies

Warnings

ASDF is not particularly accommodating to Windows users.

- ASDF has the use of symbolic links in its standard use case
  - Put symlinks from the central registry to the actual `.asd` files containing the system definitions.
  - Windows shortcuts are not symbolic links.
  - There *is* a patch to make ASDF read symlinks, but it has not yet been applied.
- ASDF assumes that it will have access to a Un*x-style shell.

Defining your own ASDF systems

This example *mostly* from the manual...
It's standard to put the ASDF system definition in a separate package to avoid namespace pollution...

```
(defpackage hello-lisp-system
  (:use :common-lisp :asdf))
(in-package :hello-lisp-system)
```

Anatomy of a defsystem

```
(defsystem "hello-lisp"
    :description "hello-lisp: a sample Lisp system."
    :version "0.2" ;; X.Y.Z please!
    :author "Joe User <joe@example.com>"
    :license "Public Domain"
```

Anatomy of a defsystem

```
(defsystem "hello-lisp"
    :description "hello-lisp: a sample Lisp system."
    :version "0.2" ;; X.Y.Z please!
    :author "Joe User <joe@example.com>"
    :license "Public Domain"
    :depends-on (:cl-ppcre (:version :cxml "3.2"))
```

Anatomy of a defsystem

```
(defsystem "hello-lisp"
    :description "hello-lisp: a sample Lisp system."
    :version "0.2" ;; X.Y.Z please!
    :author "Joe User <joe@example.com>"
    :license "Public Domain"
    :depends-on (:cl-ppcre (:version :cxml "3.2"))
    :components ((:file "packages")
                 (:file "macros" :depends-on ("packages"))
                 (:file "hello" :depends-on ("macros"))
```

Anatomy of a defsystem

```
(defsystem "hello-lisp"
    :description "hello-lisp: a sample Lisp system."
    :version "0.2" ;; X.Y.Z please!
    :author "Joe User <joe@example.com>"
    :license "Public Domain"
    :depends-on (:cl-ppcre (:version :cxml "3.2"))
    :components ((:file "packages")
                 (:file "macros" :depends-on ("packages"))
                 (:file "hello" :depends-on ("macros"))
                 (:module "additional"
                    :depends-on ("packages")
                    :components ((:file "packages")
                                 (:file "module-code"
                                    :depends-on "packages")))))
```

Design principles to know

- `*load-truename*` to solve file location problems
- It's not up to ASDF to dictate file locations
- Object-oriented design with two primary components:
  - Systems and
  - Operations
- Plan, then build

`*load-truename*` and file locations

- Previously, system definition schemes didn't help *locate* the files on which to operate.
- Typical expedient: add some logical pathname to your init file, then load.
  - logical pathnames work extremely poorly
  - doesn't allow installer to control locations well
- How do we find the files?
  - Put the `.asd` file *or a symlink* in known location
  - Source files located relative to `*load-truename*`.

Principle: ASDF Shouldn't dictate file locations

- ASDF system definitions are written by developers
- How would a developer know where you want your files put?
- Users should be able to put files where they want them without editing system definition files
  - E.g., ASDF-BINARY-LOCATIONS
  - Done by adding `:around` methods for `output-files`

Object model

- Components
  - System
  - Module
  - cl-source-file, c-source-file, etc.
  - To my mind this inheritance can cause some oddities.
- Operations
  - load-op
  - compile-op
  - load-source-op
  - test-op (somewhat inadequate)

The Protocol

- Generic Functions — mostly on `operation` × `component`.
- `operate` — generate a plan for the operation (`traverse`), and then execute it.
- `traverse` — walk over the system definition with the operator, generating a plan.
  - Plan is a list of `operation` × `component` dotted-pairs.
- `operation-done-p` — has this op already been done? May use `input-files` and `output-files`.
- `input-files` – what files need to be present to perform this operation?
- `output-files` – what files will be created by this operation?
- `perform` — do the operation

Plan, then build

- `traverse` walks the system definition(s) and generates a plan for the operation.
- `traverse` by default does a *postorder* traversal. Example:
  - System `S` containing lisp files `a` and `b`
  - `(traverse load-op S)`
  - `<compile-op a>`, `<compile-op b>`, `<compile-op S>`, `<load-op a>`, `<load-op b>`, `<load-op S>`
  - Note that the operations on the system are *not* wrapped around the operations on the components!
- You can't easily *modulate* operations. This won't work:

```
(defmethod perform :around ((op compile-op)
                            (sys (eql (find-system 'mysys))))
  (let ((*dyn-var* :modulate-loading))
   (call-next-method)))
```

New component types

- Trivial — what if I have code from Franz and they use `.cl` as an extension?
- We can define a new component class

```
(defclass franz-source (cl-source-file)
  ())
(defmethod source-file-type ((c franz-source) (s module))
    "cl")
```

More new component types

- Modulate the behavior of an operation, for your system, using a dynamic variable.
- new component

```
(defclass octal-source (cl-source-file)
 ())
```

- get this new component:

```
(defsystem my-system
  :default-component-class octal-source
  ....)
```

- modify how it's handled:

```
(defmethod perform :around ((op operation) (c octal-source))
  (let ((*read-base* 8))
    (call-next-method)))
```

New operations

- Existing operations are only:
  - load-op
  - compile-op
  - load-source-op
  - test-op (somewhat inadequate)
- Declare a sub-class of `operation`
- Define methods for key parts of the protocol:
  - `operation-done-p`
  - `output-files` — what files does this operation produce (if any)
  - `perform` — how do you do it?

Example operation

Compile a Protege ontology to CL source:

```
(defclass pont-file (cl-source-file)
     ()
  (:documentation "This is an ontology file that must be
built from a protege ontology output file.")
  )

(defclass pont-to-lisp (operation)
     ()
  (:documentation "The process of translating a Protege
ontology to a lisp source file, performed by a perl
script.")
  )
```

Example operation: dependencies

```
(defmethod component-depends-on ((op compile-op) (c pont-file))
  (append (call-next-method)
     `((pont-to-lisp ,(component-name c)))))

(defmethod output-files ((op pont-to-lisp) (c pont-file))
  ;; doing pont-to-lisp on a pont-filename
  ;; should yield this lisp filename...
  (list (component-pathname c)))

(defmethod input-files ((op pont-to-lisp) (c pont-file))
  (list
    (merge-pathnames
      (make-pathname
       :name (component-name c)
       :type "pont" :directory '(:relative :back "onto"))
      (component-pathname c)))))
```

Example operation: perform

```
(defparameter *ptolisp*
    (namestring
     (merge-pathnames
      (make-pathname :name "ptolisp" :type "pl"
             :directory '(:relative :back "onto"))
      *load-truename*)))

(defmethod perform ((op pont-to-lisp) (c pont-file))
  (asdf:run-shell-command "perl ~a ~a > ~a"
              *ptolisp* (namestring (first (input-files op c)))
              (namestring (first (output-files op c)))))
```

Notes and cautions

- You can't easily *wrap* operations in ASDF.
- Adding new operations can work nicely; extending *existing* operations works poorly.
  - If you extend, say, `compile-op`, that won't propagate well through traversal.
  - Making a `compile-op` with a new slot for additional settings is unpleasant.
- There's still no "load this but *don't* compile it" operation.
- I have written this one more times than I can count!

Notes and cautions

- There's no `clean-op`, although we're working on it.
- Soon there will be `asdf:load-system`, `asdf:compile-system` and `asdf:test-system` wrappers.
- Not enough people put `:version` attributes on their systems. This will cause you pain.
  - ASDF's version matching methods are inadequate.
- ASDF defsystem will happily let you write a `:component-pathname` argument of `"foo"` where you should have had `"foo/"`.
- ASDF also supports building foreign code into libraries (e.g., handling C source files).