Loading Multiple Versions of an ASDF System in the Same Lisp Image

Vsevolod Domkin

10th European Lisp Symposium
2017-04-03

It all started with ...

I'm yet to see a name conflict in Lisp that can't be solved by application of **rename-package**

-- a not-exact quote by Xach Beane (?)

Heard about adversarial learning?

Outline

* “Dependency Hell” problem and solutions
* Lisp package system and relevant facilities
* ASDF, its APIs & limitations
* Conflict scenarios
* Our solution to name conflict resolution: its properties, limitations and how it works on/around ASDF
* Conclusions and future work

Module name/version conflicts

Dependency hell arises around shared packages or libraries on which several other packages have dependencies but where they depend on different and incompatible versions of the shared packages. [Wikipedia]

Also known as: “dll hell”, “jar hell”

Its manifestation in the CL ecosystem

* Unrelated packages' name clashes: reported conflict for “bt” nickname in Quicklisp (among 1400+ libraries): bordeaux-threads vs binary-types

* potential conflict between versions of the same system

Solution in most languages

“In a particularly sad situation, you may find that you have two dependencies that depend on very incompatible versions of a common transitive dependency. Ideally, you should try to update one of your dependencies to use a newer version of the shared library. If you can’t do this, a good backup plan is to unleash a tormented wail and weep into your keyboard.”

-- The Nine Circles of Python Dependency Hell (https://goo.gl/2WfTnh)

Java

* custom classloader

```
URLClassLoader clsLoader = URLClassLoader.newInstance(
    new URL[] {new URL("file:/C://Test/test.jar")});
Class cls = clsLoader.loadClass("test.Main");
Method method = cls.getMethod("main", String[].class);
String[]params = new String[2];
method.invoke(null, (Object) params);
```

* OSGi
* project Jigsaw

Javascript

Node.js require

```
// module.js
exports.hello = function() { return "Hello"; }
// main.js
const myModule = require('./module');
let val = myModule.hello();
```

ES6 imports

```
// module.js
export function hello() { return "Hello"; }
// main.js
import {hello} from 'module';
let val = hello();
```

Common Lisp Package facility

* Packages are centrally-accessible dynamic singleton objects that hold references to symbols
* Package namespace is non-hierarchical
* Packages may have nicknames
* Packages may be redefined and renamed

Idea

In the case of a name conflict, use rename-package to alter the name of one the first conflicting package before loading the second one.

Potential pitfalls:

- conflict discovery
- choosing the proper time to rename the package
- limits the subsequent use of **eval** & **intern**

Common Lisp System facility

* Packages provide namespacing, systems provide packaging
* ASDF — a de facto standard
* ASDF in many ways resembles the package system: e.g. **find-system** is modelled after **find-package**
* Package discovery and distribution built on top of ASDF (see Quicklisp)
* A name conflict solution should be built on top of package & system facilities

Limitation of ASDF

Likewise with package, it has a central in-memory registry of known systems with a 1-to-1 name-system correspondence.
ASDF supports system *versioning*, but only marginally:

- only 1 system version may be known
- you can't call **find-system** with a version argument
- ASDF ops (like **load-op**) may take version as argument, but use it passively (as a constraint)

Critique of ASDF

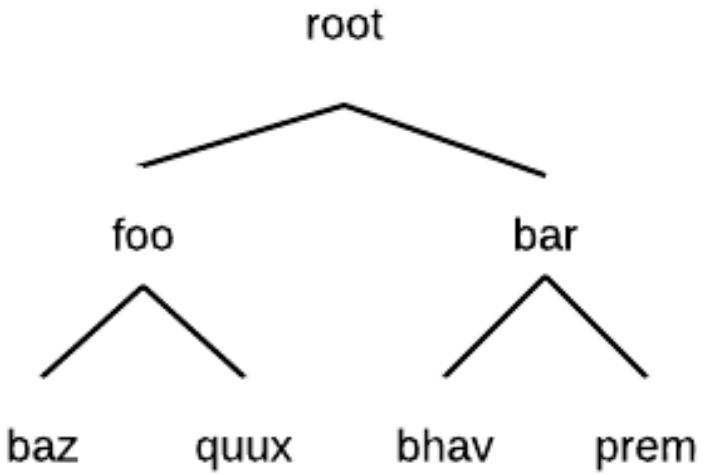
* its ops are not referentially-transparent and not fully-extensible
* it's is a great tool, but it doesn't (yet) realize its potential to become a framework for building on top of it
* its mid-level API is not complete, neither it is documented

ASDF can't

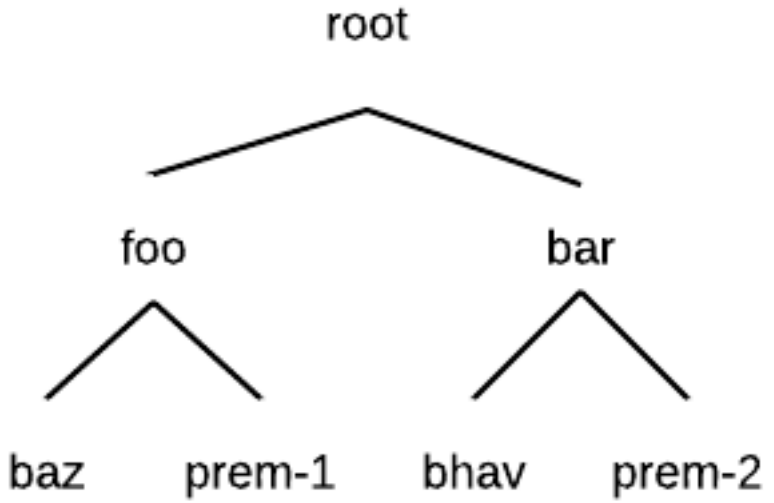
* load a system from a specific filesystem location
* enumerate all potential candidate locations for loading a system
* find a system with a specified version
* load just the source files for the system's components without potentially reloading its dependencies
* read the contents of an ASDF system definition without changing the global state

Basic cases

“zero” (do nothing)

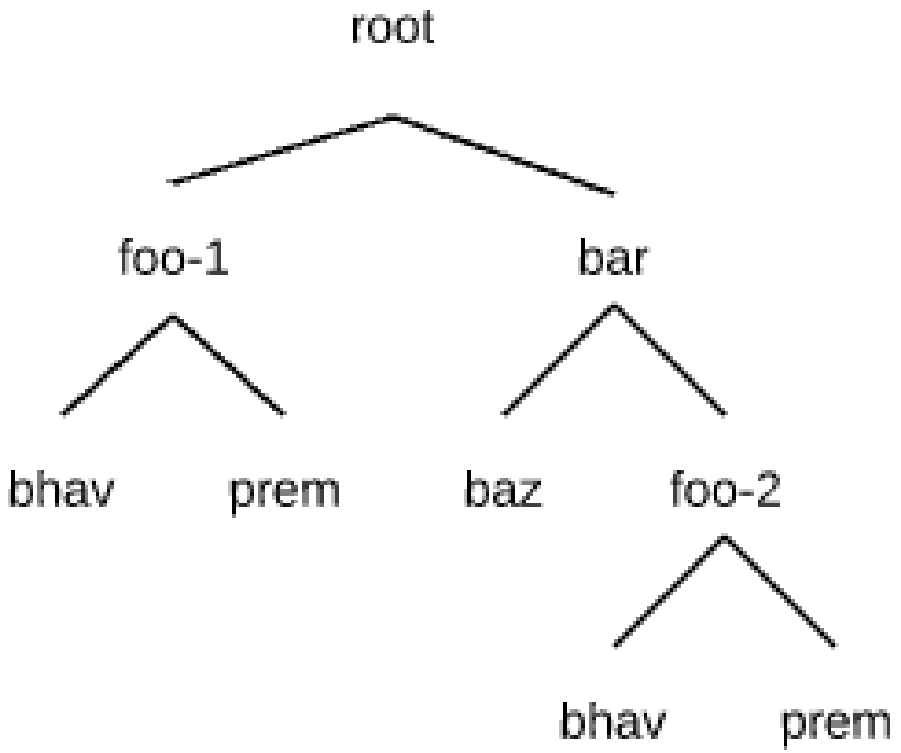


“basic” conflict resolution required

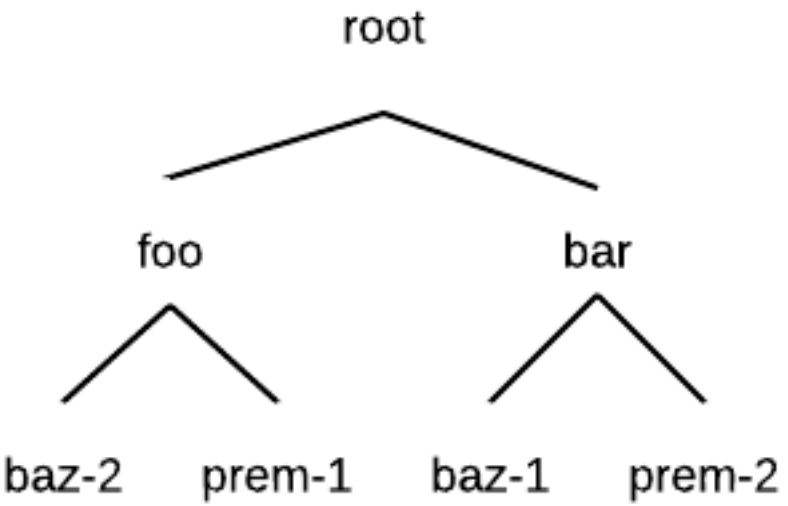


Basic cases (2)

“subroot”

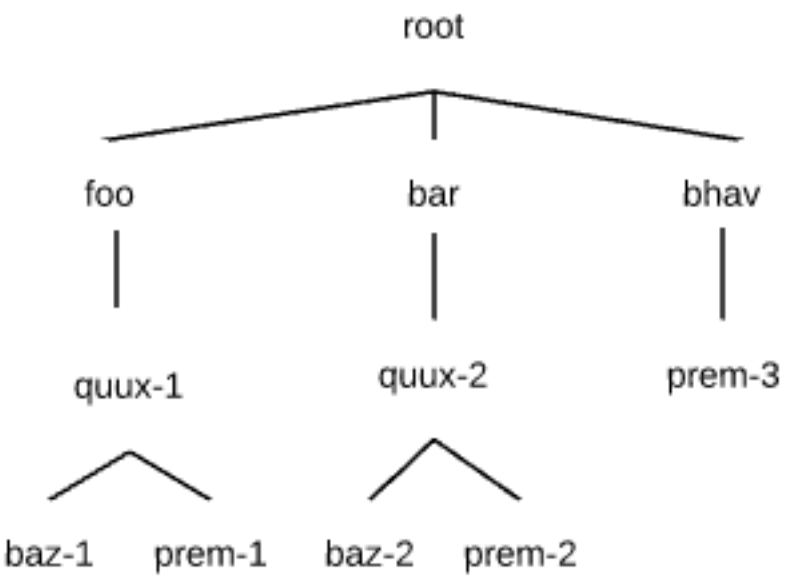


“cross”

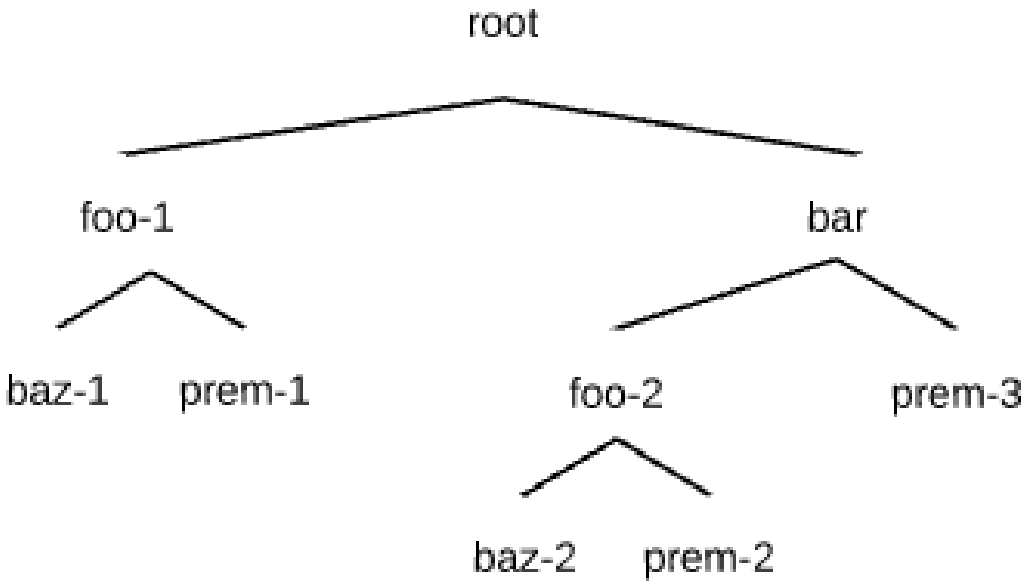


Basic cases (3)

“inter”



“subinter”



Algorithm

1. Assemble the dependency tree for the system to be loaded based on ASDF systems' dependency information and, using it, discover the dependencies, which produce name conflicts.
2. In case of no conflicts, fall back to regular ASDF load sequence.
3. In case of conflicts, for each conflicting system determine the topmost possible user that doesn't have two conflicting dependencies.

Algorithm (2)

4. Determine the load order of systems using topological sort with an additional constraint that, among the children of the current node of the dependency tree, the ones that require conflict resolution will be loaded in proper order.
5. Load the system's components (without loading the dependencies) in the determined order recording the fact of visiting a particular system to avoid reloading of the same dependencies.

Algorithm (3)

6. During the load process, record all package additions and associate them with the system being loaded.
7. After a particular system has been loaded, check whether it was determined as a point of renaming for one or more of its dependencies, and perform the renaming (if necessary).

```
(defun load-system-with-renamings (sys)
  (multiple-value-bind (deps load-order renamings)
      (traverse-dep-tree sys)
    (when (zerop (hash-table-count renamings))
      (return-from load-system-with-renamings
                   (asdf:load-system sys)))
    (let ((already-loaded (make-hash-table :test 'equal))
          (dep-packages (make-hash-table)))
      ;; load dependencies one by one in topological sort
      ;; order renaming packages when necessary and
      ;; caching the results
      (dolist (dep load-order)
        (let ((conflict (detect-conflict)))
          (when (or conflict
                    (not (gethash (sys-name dep)
                                  already-loaded)))
            (renaming-packages
              (if conflict
                  (load-system dep)
                  (load-components
                    (asdf:find-system (sys-name dep)))))
            (unless conflict
              (setf (gethash name already-loaded) t)))))))))
```

```
(defmacro renaming-packages (&body body)
  `(let ((known-packages (list-all-packages)))
     ,@body
     ;; record newly added packages
     (setf (gethash dep dep-packages)
           (set-difference (list-all-packages)
                           known-packages))
     ;; it's safe to rename pending packages now
     (dolist (d (gethash dep renamings))
       (let ((suff (format nil "~:@(~A-~A-~A~)"
                           (sys-version d) (sys-name dep)
                           (gensym))))
         (dolist (pkg (gethash d dep-packages))
           (rename-package
            pkg
            (format nil "~A-~A"
                    (package-name package) suff)
            (mapcar (lambda (nickname)
                      (format nil "~A-~A" nickname suff))
                    (package-nicknames pkg)))))))
```

Limitations of this approach

* passive capture of package changes
* intended for automatic scenarios - may mess up ad hoc interactive workflows
* doesn't handle monkey-patching
* doesn't handle implicit transitive Dependencies
* plus a couple of implementation details

ASDF quiz

* How to get a record of a particular ASDF system when you know its .asd file?

```
(asdf:load-asd asd)
(cdr (asdf:system-registered-p system))
```

ASDF quiz (2)

* How do you find all candidate .asd systems in your “search path”?

```
(defun sysdef-exhaustive-central-registry-search (system)
  (let ((name (asdf:primary-system-name system))
        rez)
    (dolist (dir asdf:*central-registry*)
      (let ((defaults (eval dir)))
        (when (and defaults
                   (uiop:directory-pathname-p defaults))
          (let ((file (asdf::probe-asd
                       name defaults
                       :truename asdf:*resolve-symlinks*)))
            (when file
              (push file rez))))))
    (reverse rez)))
```

ASDF quiz (3)

* How do you load just the system's Components without reloading all of its dependencies (and upgrading ASDF in the process)?

```
(defparameter *loading-with-renamings* nil)

(defmethod asdf:component-depends-on
           :around ((o asdf:prepare-op) (s asdf:system))
  (unless *loading-with-renamings*
    (call-next-method)))

(defun load-components (sys)
  (let ((*loading-with-renamings* t))
    (dolist (c (asdf:module-components sys))
      (asdf:operate 'asdf:load-op c)))
  t)
```

Parting words

* It works :)
* But more work needs to be done to make it not just useful but reusable
* “ASDFx”

Thanks for your attention!

Vsevolod Domkin
http://vseloved.github.io
vseloved @ gmail/twitter/github/…

http://m8nware.com
(m8n)ware — a Lisp company working on cognition-related computing problems