Evolving ASDF: More Cooperation, Less Coordination

François-René Rideau
ITA Software
fare@itasoftware.com

Robert P. Goldman
SIFT, LLC
rpgoldman@sift.info

Abstract

We present ASDF 2, the current state of the art in CL build systems. From a technical standpoint, ASDF 2 improves upon ASDF by integrating previous common extensions, making configuration easy, and fixing bugs. However the overriding concern driving these changes was social rather than technical: ASDF plays a central role in the CL community and we wanted to reduce the coordination costs that it imposed upon CL programmers. We outline ASDF's history and architecture, explain the link between the social issues we faced and the software features we added, and explore the technical challenges involved and lessons learned, notably involving inplace code upgrade of ASDF itself, backward compatibility, portability, testing and other coding best practices.

Keywords Common Lisp, build infrastructure, interaction design, code evolution, dynamic code update.

1. Introduction

ASDF 2 is the current version of ASDF, "Another System Definition Facility" [4]. ASDF allows Common Lisp (CL) developers to specify how their software should be built from components, and how it should be loaded into the current lisp image. Using CLOS, ASDF can be extended to accommodate operations other than compiling and loading, and components other than CL source files and modules.

ASDF has become the glue that holds the CL community together. The vast majority of CL libraries are delivered with ASDF system descriptions, and assume that ASDF will be present to satisfy their own library dependencies. ASDF has taken that role because of the way it simplifies not just the building and loading, but also the installation of CL libraries.

In the past few years, many in the CL community have expressed a discontent at the current state of ASDF. In November 2009, François-René Rideau, a co-author of this article, while developing an alternative to ASDF (see 7.2), posted an article [15] claiming that ASDF could not be salvaged, but was an evolutionary dead-end, whatever its many technical achievements and shortcomings. The problem was a software matter that had social implications: It was impossible to upgrade a system-provided version of ASDF; therefore before any new feature or bugfix could be considered universally available and usable by developers, some industry-wide synchronization between the many vendors was necessary, at each release, which seemed impossible, practically speaking.

From that analysis, it paradoxically appeared that ASDF was fixable after all, if only it was first given the ability to upgrade from one version to the current version. Shortly after this post, Gary W. King, who was ASDF maintainer, resigned for lack of time resources, and François-René Rideau took over the maintainership of ASDF. With the help of many Lisp hackers (including the second author), he developed ASDF 2: in addition to many technical bug fixes, ASDF 2 tries to identify social problems in how Common Lisp programmers have to interact with the system and with each other, and to offer technical solutions through an API that reduces problematic interactions.

In this paper we discuss a number of interesting technical and social challenges that we had to face while developing the new version of ASDF, and the lessons we learned. The social issues arose from the central role of ASDF — we were determined not to "break the community" with the introduction of ASDF 2 — and imposed many technical challenges. One of the most interesting technical challenges was to develop a portable means to hot upgrade ASDF, as discussed above.

In the immediately following section, we introduce ASDF, explaining what it does and how it does it. We believe this material will be of interest not just to those who have never used ASDF, but even to seasoned users, since experience has shown us that there are several aspects of ASDF that run counter to the intuitions of its users. We begin describing our technical contribution with a section discussing the hot upgrade of ASDF code, since solving this problem was the *sine qua non* for the very existence of ASDF 2. After that, we discuss the semantic changes we made to the user-visible API, most of which had to do with improving the way ASDF is configured so that it can find its input systems and place its output products (notably compiled code). In the following section, we discuss how software engineering best practices applied to our endeavor, after which we discuss one of the most tricky bug fixes (repairing the cornerstone `traverse` function). We wrap up with a discussion of related work, suggestions for future improvements to ASDF, and a brief conclusion.

2. ASDF

2.1 What ASDF does

As a preliminary to a discussion of our work on ASDF 2, this section explains the role and function of ASDF. Readers unfamiliar with ASDF may thus get a sense of what this piece of software is about. Even experienced ASDF users may learn about some of the subtleties of ASDF that they might have to deal with.

In the CL tradition, the unit of software organization is called a *system*. ASDF allows developers to define systems in a declarative way, and enables users to load these systems into the current CL image. ASDF processes system definitions into objects according to a model implemented using the Common Lisp Object System

ILC'10, October 19–21, 2010, Reno/Tahoe, Nevada, USA.

(CLOS). This model is exposed to programmers and extensible by programmers, who may thereby adapt ASDF to their needs.

For most ASDF users, the service that ASDF provides is the `asdf:load-system` function. When ASDF is properly configured, `(asdf:load-system "foo")` will load the system named `"foo"` into the current image; ASDF will first compile the system if necessary, and it will recurse into declared dependencies to ensure they are compiled and loaded.

To that end, it will take the following steps: (1) find the definition of the system `foo` in the ASDF registry; (2) develop a plan to compile and load the system `foo`, first recursing into all its declared dependencies, and then including all of its components; (3) execute this plan.

Most developers load and modify existing systems. Advanced developers define their own systems. More advanced developers extend ASDF's object model: they may define new *components* in addition to Lisp source files, such as C source files or protocol buffer definitions; or they may define new *operations* in addition to compiling and loading, such as documentation generation or regression testing.

ASDF is a central piece of software for the CL community. CL libraries — especially the open source libraries — are overwhelmingly delivered with ASDF system descriptions, and overwhelmingly assume that ASDF will be present to handle their own library dependencies.

One thing ASDF does *not* do is download such library dependencies when they are missing. Other CL programs tackle this problem, while delegating build and load management to ASDF. These programs include the obsolescent `asdf-install`, the popular `clbuild`, the up-and-coming `quicklisp`, and challengers like `desire`, `LibCL` or `repo-install`.

2.2 Analogy with `make`

It is conventional to explain ASDF as the CL analog of `make` [8]: both are used to build software, compile documentation, run tests. However, the analogy is very limited: while both ASDF and `make` are used for such high-level building tasks, they differ in their goals, their design, the constraints they respect, their internal architecture, the concepts that underlie them, and the interfaces they expose to users.

ASDF finds systems and loads systems, problems that are not handled by build tools such as `make` or `ant`. These other build tools don't search for systems; they must be pointed at a system definition in the current or specified directory. Finding systems at build time would be analogous to a subset of `libtool`; finding and loading systems at runtime would correspond to a subset of the Unix dynamic linker `ld.so`. As for loading, the fact that ASDF is available for interactive use makes it analogous to some component of the operating system shell.[1] Loading some systems might thus be similar to importing shell functions, starting a daemon, registering a plugin in your browser, loading code into some master process, etc. Finally, ASDF maintains state in a running (perhaps long-running) Lisp image. If system components have been changed, it is ASDF's job, at the user's request, to generate and execute plans to modify previously-loaded systems in order to accommodate those changes.

ASDF also differs from `make` in terms of how systems are specified. `make` is built around a complex combination of multiple layers of languages — some domain-specific languages and some generalized programming languages. `make` interprets a *makefile* in the following way: A text-substitution preprocessor expands macro definitions in the file, producing a set of pattern-matching rules. An inference engine uses the rules, chaining backwards from the targets it has been directed to build. The rules are annotated with parameterized shell scripts that actually perform the build actions.

`make` is a powerful tool that can express arbitrary programs, but makefiles can grow into a mesh of code that defies any simple analysis. ASDF is a small CL program, and its system definitions are data rather than programs.

Supporting a new file type is relatively easy with `make`, by adding a new rule with an appropriate pattern to recognize filenames with the conventional extension, and corresponding shell commands. Supporting a new file type in ASDF is more involved, requiring one to define a class and a few methods as extensions to the ASDF protocol. Unfortunately, this procedure is complicated by the fact that the ASDF protocol is poorly documented.

On the other hand, when building C programs with `make`, arbitrary side-effects have to be specified in a different language, in the shell layer of the makefile. When building CL programs with ASDF, arbitrary side-effects are specified in Lisp itself inside the components being built. This is because the C compiler and linker are pure functional file-to-file transformers, whereas the CL compiler and loader are imperative languages. The ability Lisp has to do everything without cross-language barriers is a conceptual simplification, but of course, it doesn't save you from the *intrinsic* complexity of such side-effects.

A final difference between ASDF and `make` is that ASDF generates a full plan of the actions required to fulfill all the dependencies of its goal before performing the planned actions. By contrast, `make` performs actions as it traverses its dependency tree [8, 17]. We will discuss this further in Section 2.4.

```
(defsystem "hello-lisp"
  :description "hello-lisp: a sample Lisp system."
  :version "0.3"
  :author "Joe User <joe@example.com>"
  :licence "Public Domain"
  :depends-on (foo-utils)
  :components ((:file "packages")
               (:file "macros"
                 :depends-on ("packages"))
                (:file "classes"
                 :depends-on ("packages"))
                (:file "methods"
                 :depends-on ("macros" "classes"))
                (:module "main"
                 :depends-on ("macros")
                 :serial t
                 :components
                  ((:file "hello")
                   (:file "goodbye")))))
```

Figure 1. Sample ASDF system definition.

2.3 Basic ASDF object model

Figure 1 shows a sample ASDF system definition, illustrating core features of the `defsystem` macro. This form shows how a system can be defined in terms of components (files and sub-modules). It also shows how system dependencies are specified (`hello-lisp depends-on foo-utils`), and gives example metadata (authors, versioning, license, etc.).

The `defsystem` macro parses the system definition into a set of linked objects, all of them instances of (subclasses of) `component`. The main object will be of type `system`. Primitive `components` of systems are typically instances of a subclass of `source-file`, by default `cl-source-file`. There are other predefined component types, including `static-file`, representing files that are distributed with a system, but which do not undergo operations. Systems can be recursively organized in a tree of `modules` that may or may not map to a similar tree of directories.

[1] McDermott also makes this point in the paper about his chunk maintenance system [12].

```
 ((#<COMPILE-OP> . #<CL-SOURCE-FILE "packages">)
  (#<LOAD-OP> . #<CL-SOURCE-FILE "packages">)
  (#<COMPILE-OP> . #<CL-SOURCE-FILE "macros">)
  (#<COMPILE-OP> . #<CL-SOURCE-FILE "classes">)
  (#<LOAD-OP> . #<CL-SOURCE-FILE "macros">)
  (#<LOAD-OP> . #<CL-SOURCE-FILE "classes">)
  (#<COMPILE-OP> . #<CL-SOURCE-FILE "methods">)
  (#<COMPILE-OP> . #<CL-SOURCE-FILE "hello">)
  (#<LOAD-OP> . #<CL-SOURCE-FILE "hello">)
  (#<COMPILE-OP> . #<CL-SOURCE-FILE "goodbye">)
† (#<COMPILE-OP> . #<MODULE "main">)
  (#<COMPILE-OP> . #<SYSTEM "hello-lisp">)
  (#<LOAD-OP> . #<CL-SOURCE-FILE "methods">)
  (#<LOAD-OP> . #<CL-SOURCE-FILE "goodbye">)
† (#<LOAD-OP> . #<MODULE "main">)
† (#<LOAD-OP> . #<SYSTEM "hello-lisp">))
```

Figure 2. Sample `load-op` plan for the system definition in Figure 1. We assume that `foo-utils` is already compiled and loaded.

System management tasks consist of applying the generic function `operate` on parameters specifying an `operation` and a `system`. The `operation` is either a `load-op`, specifying that a system or component is to be loaded into the current image, or a `compile-op` specifying that a system or component is to be compiled into the filesystem. The `system` is designated by a string or a symbol, and `operate` will first find and load the thus named system definition. This is done by the generic function `find-system`, which searches for the system definition in a configurable system registry (and its in-memory cache). The system definition search is one of the aspects of ASDF 1 that we reformed; see Section 4.3.

After a system definition is found, `operate` generates a *plan* for completing the `operation` using the function `traverse` described below. The plan will be a list of steps; the list will be topologically sorted such that all the dependencies of a step appear before that step. If a plan is successfully generated, `operate` will `perform` each of its steps in sequence. If a step fails, ASDF offers the user a chance to correct the step, retry, and continue according to plan, making it easy to fix simple mistakes without interrupting the operation.

2.4 Plan generation

The plan-generating function `traverse` performs a depth-first, postorder traversal over the steps needed to operate on the target system and its dependencies. Each step is a pair of an `operation` and a `component`, What we mean by postorder here is that when applying an `operation` to a `system` (or `module`), the plan will contain steps first to complete the `operation` to the subcomponents, then to perform it on the `system` (or `module`) itself. For example, a `load-op` plan for the system in Figure 1 is given as Figure 2. Note the lines marked with a †, which show that operations on composite components (`modules` and `systems`) are scheduled after operations on their components.

When building a plan, `traverse` skips steps it can prove are not necessary. A step is necessary if it hasn't been done yet or if it is *forced* by a change to one of the component's dependencies. When a *compilation* step is necessary, all the steps that depend on it are *forced* to be necessary, since whatever was done before will be out of date by the time the step is performed. If a step and all the compilation steps transitively required in order to complete it have already been done (by a previous run of ASDF), then the step is not necessary and can be skipped.

To determine whether a step has already been done, `traverse` calls the generic function `operation-done-p` on the `operation` and `component`. For a compilation step, wanted for its effects on the filesystem, `operation-done-p` compares the timestamps of the corresponding `input-files` and `output-files` (if present). For a load step, wanted for its effects on the current Lisp image, there are no `output-files`, and `operation-done-p` compares the timestamp of the `input-files` to the time they were last loaded into the current image (if ever). Steps with neither `input-files` nor `output-files` (e.g., where the component is a `module` or `system`) are never considered necessary unless forced by a dependency or sub-component.

Tracing `traverse` and inspecting the plan it returns is often a good way to debug one's system definitions. Running `traverse` can also be useful as an introspection tool on a system (e.g. to determine what files need be recompiled, and run according tests after recompilation). We have improved `traverse` somewhat in ASDF 2, but have limited the scope of our efforts out of concern for backwards compatibility. See Section 6. `traverse` is not an exported part of the ASDF API, so in theory users and extenders should not depend on it, but that has proven not to work, because of the leakiness of the abstraction. We discuss this further in Section 8.

Importantly, note that system definitions only specify dependencies between steps. These dependencies only specify a partial order, so for a given operation and system there may be multiple possible complete plans. Moreover, the semantics do not dictate a "plan-then-execute" implementation. And indeed, a parallelizing extension to ASDF, POIU, does things differently (see Section 5.5).

2.5 Social role of ASDF

ASDF had a major role in enabling the growth of open source CL libraries, and the renaissance of the CL community. Prior to the existence of ASDF, there have been many efforts at consolidating a Lisp community [20], but it was hard to assemble the pieces of code gathered from over the net, because there was no portable way to specify how to build a system out of components that were not all supplied together. Installing a CL system was often a tricky matter of filling a load file full of logical or physical pathname definitions. Sharing a CL system across a group, even on the same filesystem, often involved fussy coordination of logical pathname definitions. Typically, CL software developers simply developed and delivered enormous, monolithic systems, and there was little opportunity for code reuse across work groups.

ASDF changed all this by providing a *de facto* standard for specifying systems. ASDF made it easy to specify how one system could depend on others. This could now be done simply by naming the systems depended on, and users could trust ASDF to find the requisite system definitions and system contents. Unfortunately, ASDF had several flaws that made this system configuration, and the social software development process, more difficult than need be — though still much easier than doing without ASDF. In this paper, we will discuss some of these flaws, and the steps we have taken to overcome them in ASDF 2.

The CL renaissance is still fragile. For that reason, and because of central role of ASDF in the community, we needed to be extremely careful that our modifications and extensions to ASDF would only strengthen it and not, for example, break the community up into islands running separate and incompatible ASDF versions.

3. Dynamic Code Update

3.1 Social Rationale

As discussed in the introduction, in-memory upgradeability of ASDF was essential to solve the social issues regarding the development, distribution and usage of new versions of ASDF. Only if we guarantee that ASDF can be upgraded if needed can users rely on new features and bug fixes of ASDF.

Previously, there was no portable way to load and configure ASDF unless it had been pre-loaded with your Lisp image (as by `common-lisp-controller` under Debian), and there was no way to upgrade a pre-loaded ASDF with a new version. With ASDF 2, users can install a new version of ASDF into an image with an old version simply by executing the following command, assuming ASDF was properly configured to find the new definition:

```
(asdf:load-system :asdf)
```

In an apparent paradox, **allowing for divergence creates an incentive towards convergence**: because they are confident that ASDF 2 can be upgraded, implementation vendors have less pressure to be conservative and keep a "trusted" old version (which may differ between implementations). Also, with less pressure to get it exactly right, CL implementers need not hesitate to upgrade to the latest upstream release; if they update their code every so often and ASDF does not change too fast, they will quickly converge to the same version, the latest stable release. By removing the requirement for *a priori* coupling of release cycles, we achieve better *a posteriori* coupling of release cycles.

3.2 Technical Challenge

Unlike other build systems, such as `make`, **ASDF is an "in-image" build system managing systems that are compiled and loaded in the current CL image**.

Build tools of other languages typically rely on some external, operating system provided shell to build software that is loaded into virtual machines (*processes* in Unix parlance) distinct from the current one. When using these other languages, all the state necessary to build software is typically kept in the filesystem, and incompatible changes in interfaces or internals of the build system are resolved simply by starting a new virtual machine.

ASDF does not start separate processes for compilation. We believe that there are a number of reasons for this design decision. First, CL implementations historically have run on a vast variety of operating systems, some of which lacked the capability of virtualizing a Lisp process. Previous build systems were designed around that constraint, and ASDF followed their design. Also, even on modern operating systems that allow this virtualization, starting a CL process can sometimes be a relatively expensive process (depending on the individual CL implementation). Moreover, because of the presence of code to be executed at compile time and the dependence on a substantial amount of compile-time state, it is difficult to decompose the process of building a CL system into independent pieces and parcel them out to different processes. Finally, CL programmers often build very extensive state in a long-living CL image, and so prefer to keep them alive. ASDF supports such a use pattern.

Because ASDF performs its build tasks in the user's current Lisp process, upgrading ASDF entails modifying some existing functions and data structures *in situ*, requiring delicate surgery to keep things working as you modify them.

To make things slightly harder, the same source code must be able to both define a fresh ASDF (if it hasn't been loaded yet), or upgrade an existing ASDF installation to the current code (if a previous version already exists). In addition, the code for an ASDF version must recognize the special case when the very same version is already loaded so as to ensure such reloads are idempotent. It does this by relying on a simple version identification string, to be bumped up at every modification of ASDF.

3.3 Rebinding a symbol

The semantics of redefining or overriding a function is not fully specified by the CL standard. The many implementations at the time of standardization may have had explicitly different semantics, the semantic difficulties may have been overlooked, implementers may have called for underspecification as leaving them more room for optimization, or it may have otherwise not been considered appropriate for the committee to standardize a practice that wasn't widely accepted. In writing the code that makes it possible to upgrade ASDF, we encountered two complementary difficulties when rebinding the functional value of symbols.

The first difficulty arises from incompatibilities between new and old function definitions bound to a same symbol when new functions are dynamically called by an old client, with data following the old convention. The second difficulty arises from incompatibilities between the new and old function definitions bound to a same symbol when old functions are statically called by an old client, with data following the new convention.

A *dynamic call* is when the call site dereferences the function bound to the symbol at runtime, and does so at every call. A *static call* is when the compiler or linker dereferences the function bound to the symbol once at compile-time or load-time and prepares the runtime to always directly use the code of that function. An actual implementation need not do the above naively, as long as it behaves in a semantically equivalent way.[2]

The two above difficulties are inherent in redefining functions and are not specific to either CL or ASDF. However, these difficulties are particularly relevant in the case of ASDF, because it drives compilation and loading of Lisp code possibly including new versions of ASDF itself. ASDF's redefined functions are therefore likely to be in the continuation of their own function redefinitions, where the old code will for a short while be a client to the new code. Moreover, these difficulties are compounded by the fact that the CL standard [1, section 3.2.2.3] does not specify whether any particular call will be dynamic or static, unless the function was explicitly declared `notinline`, in which case it should always be dynamic. In practice, implementations may legitimately inline function bodies, cache effective methods for generic function calls, specialize call sites to declared calling conventions, etc. This means that redefining a function requires taking proper precautions against errors that may or may not happen, depending on details of individual CL implementations, the nature of the function being redefined, and the evaluation context.

CL provides a primitive `fmakunbound` that is meant to undo the binding of a symbol to a function. `fmakunbound` ought to clear out any related state and make way for a new definition. Indeed, in the simple case where a function is not referenced in the continuation of the current compile or load, and not exported to code from other files, all references to it will be overridden by newly loaded code. In this case, it is sufficient to `fmakunbound` the function symbol (and possibly re-declaim its type) before redefining it with an incompatible signature. Any inlined or cached version of the function will be overridden by the new definitions in the newly loaded file.

On the other hand, when a function may be referenced in the continuation of the current compile or load, then whether `fmakunbound` will work or not depends on the implementation, the optimization levels, etc. Generic functions in particular may require special MOP operations that haven't been standardized. We cannot work around limitations of MOP standardizations by using a portability layer such as `CLOSER-MOP` [7], lest by doing so we

[2] For instance, assuming functions are seldom rebound, dynamic calls may be implemented just like static calls, except that the value of the binding is a cache that gets invalidated between the time the function is rebound and the time the cache is next used.
As for static calls, they may be implemented not just by linking a call to the proper code value, but also by inlining the body of the target function, or at the other end of the spectrum, by doing a dynamic call to an alpha-converted symbol that will never be rebound.

create a circular dependency between the portability layer (loaded using ASDF) and ASDF itself.

3.4 Shadowing a symbol

In cases where `fmakunbound` will not work and a symbol cannot be rebound, we may instead `unintern` the symbol whose function binding we want to redefine. Doing so we effectively *shadow* the old definition. All existing references to the symbol from old code will continue to point to the old symbol and its existing bindings. The next time same symbol name appears, a new, distinct, symbol will be interned, and further code that is read into the system will refer to this new symbol and its associated bindings.

In addition to cases where bugs and limitations of the implementation prevent `fmakunbound` from working, there are cases where the CL standard doesn't provide any redefinition mechanism, and shadowing the old symbol is the only option available. This is notably the case regarding CL not providing any guaranteed primitive to undo the declaration making the binding of a symbol to a variable *special* (i.e. dynamic, as opposed to lexical, i.e. static), or *constant*, or to undo declarations using a named type that is being redefined. These issues are usually worked around by programmers following syntactic conventions, such as `*ear-muffs*` for special variables and something similar for `+constants+`. There should *never* be a need to turn the `*ear-muffs*` variable into something that is lexically scoped, or to change `+constants+` at all.

The main downside of shadowing as a redefinition mechanism is that it requires that all clients be reloaded and possibly recompiled to be able to use the new interface, even if the code of these clients hasn't changed at all. Indeed, previously loaded clients will continue to use the old symbols and their bindings, and there may be confusion if old and new clients interact while expecting to be talking to the same service.

Another problem is that `unintern` runs the risk of causing "collateral damage." When a symbol has several bindings associated to it, such as a function or macro; variable or constant; type or class or condition; property, etc. All of these bindings will simultaneously become inaccessible when the symbol is uninterned.

Consider a user developing on a CL image that includes ASDF and on top of it ASDF clients such as POIU or CFFI-GROVEL. If the user upgrades ASDF, then she must also reload POIU and CFFI-GROVEL before she may use them. She must do so even if there were no code changes in either of the latter systems, lest the previously loaded system be in an invalid, unusable state. Code in these systems may be linked to obsolete, now-uninterned symbols from the old ASDF. For these client systems to function properly, they must be linked against the symbols from the new ASDF.

Ideally, whether we rebind or shadow would be a matter of the distinction between intension and extension: which symbols we consider intensional fixed entry points that denote some "same" higher meaning when implementation changes underneath, and which symbols denote extensional constant code values, the implementation of which cannot be changed, but that can be shadowed and forgotten.

In practice, which of rebinding and shadowing we use depends on the implementation into which we are loading ASDF, because different implementations have different quirks, bugs and constraints when upgrading code. For instance, rebinding some generic functions fails to flush method caches on SBCL, but shadowing symbols while loading a FASL breaks linker optimizations on ECL and GCL. We barely managed to support the basic use case of upgrading from implementation-provided versions of ASDF to the latest version of ASDF, but we had to deal with more hurdles than we expect the average CL programmer ever to want to deal with.

To make things more complex, in CL, multiple symbols of the same name may coexist, if each of them is interned in a different package. Packages are global flat structures, and each package may import symbols from other packages (without renaming). Packages may "use" other packages that export symbols, which helps automate some of this importing. If `unintern` was causing problems when redefining functions, redefining packages only leads to more madness as uninterned symbols may have been imported by other packages, the package `use` graph may have changed, etc. There again, we may try either to do complex surgery on an existing package object and preserve its relationship to other packages using it or its symbols, or to simply rename it away and shadow it and all its symbols with it, and require client packages to be reloaded to link to the new package object.

ASDF 2 takes care to define the `ASDF` package if it doesn't exist, redefine it properly if it exists, etc. ASDF 2 reuses existing packages and symbols whenever possible, so as not to invalidate previously interned client code, etc. This package wrangling was difficult to get right, and once again, we have to take into account the eager linking done by ECL and GCL. One reason we could make this package wrangling work is that we do not need to blindly handle the general case of upgrading arbitrary package definitions to arbitrary new ones. All we needed to do was to upgrade previous versions of our own packages. This was simplified by the fact that our package does not `use` any other package that is a moving target. If any package `uses` `ASDF` and that somehow causes a clash, it is the responsibility of the authors of that client package to update their code.

3.5 Happy non-issue: Data Upgrade

CL makes dynamic data upgrade extraordinarily easy. Classes can be redefined, slots can be added to them, removed from them, or modified, and all instances will be automatically updated before their next use to fit the new definition. The Common Lisp Object System (CLOS) [6] allows users to control this instance update programmatically by defining methods on `update-instance-for-redefined-class`. We rely on this functionality in ASDF 2 to ensure that previously loaded systems can still be used after an upgrade of ASDF (assuming they are not themselves ASDF extensions). Without such functionality, preserving the existing data (system definitions, etc.) would have been a major undertaking requiring a global rewrite, and requiring much better control over the version of ASDF 1 being replaced than was easily available.

There was one catch with using `update-instance-for-redefined-class` for the purposes of ASDF 2. This was a chicken-and-egg issue between `defclass` and `defmethod` `update-instance-for-redefined-class`: it was difficult to order the definitions so that the same code would work without warning or error in both the case of defining a fresh ASDF and the case of an upgrade from a previous ASDF.

Defining the class before the method may on some implementations cause objects to be upgraded before the method is defined and therefore without the upgrade being properly run. Defining the method before the class in the source code may cause a warning the first time around when the class isn't defined yet. Inserting an introspective check for class existence may cause the method definition not to be statically compiled and emit a warning on some implementations. Protecting the method definition with delayed evaluation (as we finally did) hushes the warning. Unfortunately, it also causes slightly inefficient runtime compilation on some implementations. Nevertheless, it doesn't cause any significant user-visible pause, since the user is compiling ASDF and (presumably lots of other code with it); the slight added delay is not perceptible.

The CL protocol for class redefinition is relatively well-designed and quite effectively handles the difficult problem of schema

upgrade that other programming languages do not dare to tackle. However, the schema upgrade API was written with "upgrade scripts" in mind, and is clumsy to use when writing code that specifies end-result semantics independently of whether the code is an initial definition or an upgrade.

Note that none of this would have been possible if ASDF, like its predecessor `MK-DEFSYSTEM`, had been using pre-CLOS `defstruct`. CL structures do not provide a safe upgrade protocol the way CLOS classes do.

3.6 Towards a better specification

It is to the credit of CL that dynamic code upgrade is possible at all; it is not possible in most programming languages. However, it is possible to support dynamic code upgrade much better. For instance, Erlang solves the issue of dynamic code upgrade by providing syntactic distinction between the two semantically different kinds of calls: calls specifying a syntactically unqualified identifier are always semantically static calls (to a function in the same module), and calls to a syntactically module-qualified identifier (even in the same module) are always semantically dynamic calls. The effect of function redefinition on each call site is therefore perfectly predictable, and this does not prevent optimizations as the CL standard authors might have feared.

It is probably possible to express the Erlang semantics on top of CL, by explicitly `funcall`ing either `(fdefinition 'foo)` or `(load-time-value (fdefinition 'foo))` depending on the call being static or dynamic. However this is cumbersome, non-idiomatic, and most importantly requires existing code to be modified to use the new convention before it may be safely upgraded. Therefore it is not compatible with using existing libraries as black boxes. Future Lisp standards and specifications could learn from Erlang.

CL users could incorporate Erlang-like semantics in a mostly transparent way by layering a CL implementation on top of CL, shadowing the usual reader and evaluator to replace them with something that provides well-defined semantics for hot upgrade, assuming all code is (re)compiled on top of this implementation rather than directly with the underlying implementation. This, however, would be a large, challenging task and not obviously worth the cost. Furthermore, if one were to design and implement what amounts to a new language on top of CL, would it and should it be CL all again? Interestingly, in the presence of concurrent threads within a same Lisp image (as is common nowadays), some model of atomicity or PCLSRing [5] would be required, which also goes beyond the current CL language specification.

Lacking such a better-specified Lisp, possibly implemented atop CL, there are ways to work around these limitations; but not only are they are quite unidiomatic, they require manual management. For instance, we could use some kind of symbol versioning: use completely different symbols any time we would previously redefine things, mark old symbols as obsolete and never reuse them. In other words, add a monotonicity (purity) constraint to our bindings and achieve guaranteed static calls through *manual* alpha-conversion. Therefore function `FOO` would never be redefined, instead `FOO-V2` then `FOO-V3` would be defined and used by client code. This client code in turn would itself need to be renamed with a new version since its contents have changed to use new function names. In a limited way, that is what uninterning symbols does for you, and what renaming away packages would do, etc. However, monotonicity requires new clients not only to be recompiled, but also to be modified any time any code is changed incompatibly.

This latter approach is semantically safe and technically simple, but we didn't adopt it, because of its social implications. This approach requires us to either keep supporting old interfaces, or gratuitously break old programs, all the more gratuitously when the incompatibility with previous interface lies in "extensions" that were conceptually broken and remained (mostly?) unused. Its advantage is that it allows to make interfaces formal where they weren't. Its disadvantage is that it requires to maintain formal interfaces where you mostly don't need them. As part of the CL community, that ascribes a high cost to social interactions and especially to formality in such, we chose to tackle one annoying technical issue that we needed only solve once over a social issue that would crop up every time.

3.7 Lessons Learned

We encountered many technical issues when providing hot upgradability of ASDF. The good news is that it is possible to write hot upgradable code in CL in a reasonably portable way, whereas dynamic code upgrade is not even possible in most programming languages. The bad news is that hot upgrade remains quite tricky, especially *portable* hot upgrade, and it imposes limitations on the code to be upgraded. In order to write hot upgrade code, you have to use application-specific knowledge to determine what is safe and what is not. Furthermore, not only is such dynamic code upgrade not thread safe, it can even damage the operation of a single-threaded environment.

CL support for hot upgrade of code may exist but is anything but seamless. Happily, programmers only need to deal with hot upgrade as an issue for *their own* programs, and so they have the required, application-specific knowledge available; so at least the problem is socially solvable, if technically hard.

In the end, **the general problem with CL is that its semantics are defined in terms of irreversible side-effects to global data structures in the current image.** Not only does this complicate hot upgrade, it also makes semantic analysis, separate compilation, dependency management, and a lot of things much harder than they should be.

4. Interface Modifications

4.1 Objectives and limitations

Our goals in replacing ASDF 1 with ASDF 2 were limited: fixing bugs, pulling in new features that were commonly used but that had to be separately installed, resolving ambiguities in the specified or implemented semantics, cleaning up a few internals, and lowering the barrier to entry for users. While doing this, we aimed above all to maintain backwards compatibility. We did *not* take as our goal designing a better system (but see Section 7.2), making incompatible improvements, etc. We wanted a stable base to build upon and, indeed, a clear sense of what we would be incompatible with before venturing into unexplored territory. We thought our goals were modest, but they turned out to be quite challenging.

4.2 Pathnames

ASDF had a lot of subtle breakage related to pathnames, due to discrepancies between how pathnames behave across vendors and across operating systems. There were cases that one could have expected to work and which actually worked on some implementations that would fail miserably on others.

In particular, the `static-file` component class was wholly broken when used with pathnames without extensions such as the typical `README` (whereas `README.txt` would function correctly). Attempts to use hierarchical pathname strings such as `"foo/bar"` as a component name would fail on most, but not all, implementations. Painfully specifying such hierarchical pathnames as `:pathname` overrides would usually work but still failed in many corner cases.

At the root of these ASDF issues was the fact that **the CL standard leaves many things underspecified about pathnames**. As a

result implementations can and do interpret things in very different ways, depending on what struck each implementer as preferrable on each implementation code base on each operating system. Even restricting oneself to current implementations running on modern operating systems (Windows, Linux, MacOS X, etc.), the discrepancies in pathname implementations made it impossible to write a truly portable ASDF system definition using anything but the most basic pathnames lest it break somewhere.

Some ASDF system definers attempted to work around these problems by explicitly providing pathnames to ASDF using the all-powerful "`#.`" read-time evaluation and `make-pathname` and `merge-pathnames` to build pathnames manually. This provides portably defined syntax, but at the expense of extremely cumbersome definitions. Even adopting this expedient, programmers would often fail to produce a portable system definition, because of the subtle semantics of `merge-pathnames`. In particular, common, apparently reasonably definitions would work on Unix platforms yet could fail spectacularly on Windows machines, yielding pathnames with the wrong host and device. This would happen because of the way `make-pathname` inherits host and device slots from `*default-pathname-defaults*`. CL pathname primitives do not include a portable notion of a relative path independent from a host and device, so there is no portable syntax for specifying such paths.

We therefore built a new `merge-pathnames*` to replace `merge-pathnames`. Unlike the latter, `merge-pathnames*` considers relative pathnames as *not* specifying host and device. Typically modern computer users do not wish to specify hosts or devices when working with relative pathnames, but that is not supported portably by CL pathnames. On top of `merge-pathnames*`, we built and formally specified a mechanism, `merge-component-name-type`, for users to provide such relative pathnames in a portable syntax, and have them merged properly with the rest of a component specification.

In other words, we provided a clean, well-specified abstraction interface on top of CL's pathname interface to the filesystem, reinverting some "abstraction inversions" [3] of that interface so users don't have to go through the pain themselves. Each programmer knows the relative paths for his files and can specify them, but may not know the fine rules of syntax for pathnames on their particular Lisp implementation and operating system. Even if he does, he is extremely unlikely to know what those rules would be on implementations and systems used by other users of his software. With ASDF 1 it was too hard to get pathname specifications right. **With ASDF 2 we made it hard to get pathname specifications wrong**.

4.3 Configuring Input Locations

Another issue with CL software that ASDF has to deal with is that there is no well-defined place where to find source code on any particular operating system.

Using ASDF 1, the end user had to somehow configure the global variable `asdf:*central-registry*` in between the moment ASDF was loaded and the moment it was first used. This was tricky, and required all programmers, even newbies, to be able to insert program snippets in the midst of any source code provided by a third party, so that that code would run properly. Advanced users might add some configuration to their implementation-dependent startup file, but there was no convention whereby distributors of independent software modules could safely add system-wide configuration, or whereby users could extend or override such configuration. At best, some launch scripts (such as `cl-launch` or `buildapp`) might help one specify those things in a portable way as part of the same relatively simple shell command that invokes the Lisp software.

Some developers suggested that we should use the CL notion of *logical pathnames*, arguing that logical pathnames provide the correct kind of abstraction for this task. The idea fell through because logical pathnames have too many limitations preventing their portable use. Worse, the logical pathnames *themselves* need to be configured, which cannot be done portably, so using logical pathnames simply replaces one hairy problem with another even hairier.

A common strategy using ASDF 1 was to minimize changes required to the `asdf:*central-registry*` by setting up "link farms." A single directory, the link farm, would be added to the registry, and that directory would hold the actual state of the configuration, in the form of symlinks to `.asd` files. ASDF would follow those symlinks to the actual files and the associated source code. This strategy wasn't practical on Windows, where symlinks are not traditionally well supported. In ASDF 2, we added support for Windows shortcuts, that are that platform's analog to Unix's symbolic links. Furthermore, the integrity of the link farms was always under threat: if a new version of an installed system was to add or remove some `asd` files, it was up to the user to notice and repair her link farm.

Another strategy, used by one of us (Goldman) was to write a specialized, portable version of the Unix `find` utility, that would find `.asd` files. Using this utility, one could easily initialize ASDF 1's `*central-registry*` from, for example, the top-level working directory for a project's revision control system. In turn, that made it possible for a project team to get a consistent, shared ASDF configuration by loading a single file, without the clumsy expedient of a "link farm" (difficult to maintain), and making it possible for programmers easily to change between different project-specific ASDF configurations. In a somewhat simpler expedient, ITA used a directory of `.../**/*.asd` to locate all relevant directories to push to the central registry.[3]

Based on a discussion with Stelian Ionescu about the proper way that Unix utilities and Debian distributions are configured, we created a new configuration mechanism, the source registry, that solves all the above mentioned configuration issues. We kept the `*central-registry*` mechanism for backwards compatibility, allowing for a smooth upgrade, but that registry is now empty by default.

4.4 Configuring Output Locations

CL compilers produce FASL ("FASt Loading") files, or fasls, that can be loaded faster than source code. FASL files play a role analogous to C object files. Unlike C object files, compiled CL code may contain arbitrary expressions to be evaluated at load time, in addition to simple function definitions as in other languages. CL compilation can be slow enough that you usually want to use FASL files instead of recompiling the source code every time, as some scripting languages do.

Modern Lisp implementations typically include some kind of compiler that produces reasonably fast code, but take some time to optimize their output. Some implementations (such as ECL) also include an interpreter that simply evaluates the input code more or less directly. This is practical for code that only gets run once, while building or interactively. Other implementations (such as SBCL) eschew an interpreter altogether. For such implementations, evaluation of an interactive form is done by compiling it then immediately executing the compiled code.

[3] This simpler approach wasn't available to us, because our directories also contained Java code. The proliferation of subdirectories caused by the Java namespacing protocol made simple tree search unacceptably slow; we needed the equivalent of `find`'s `-prune` directive. (Robert P. Goldman)

Every implementation has a different format for fasl files, reflecting different implementation strategies, etc. Indeed, the format for fasls often varies between different versions of the same implementation. In extreme cases, formats may even vary between installations of identical versions of a single implementation that have been compiled with different options: SBCL compiled with or without thread support, ECL compiled with or without unicode support, Allegro in "modern" or traditional mode, etc.

In this way fasls are unlike C object files, where on any given operating system and processor platform, a standardized Application Binary Interface (ABI) specifies object file format, calling conventions, exception side tables, debugging information, etc.: everything that is needed for producers and users of object files to interoperate. Every CL implementation has its own ABI, and this makes sense, because C is a relatively low-level language designed to be close to the processor, so the mapping from C to processor is simple and direct, whereas CL is a higher level language, which leaves much more leeway in mapping CL to the processor. There are many ways that features like argument passing in presence of optional and keyword arguments, non-local exits, symbol function dereference, fixnum encoding, garbage collection and thread synchronization for garbage collection, etc., can be implemented. These implementation decisions impact the CL ABI and are deeply tied into the guts of the compiler.

Unfortunately, the pathname type (filename extension) for those fasls does not vary as much as the file formats. For example, both Allegro and Steel Bank CL compilers generate files with the `.fasl` extension, but neither compiler's fasls can be loaded into the other's images.

Therefore, for multiple CL implementations to coexist peaceably, we must have a mechanism to allow ASDF to store compiler output in implementation-dependent places. This mechanism will, as a further benefit, allow software to be distributed as source code in shared system-wide directories and compiled by each user to her own cache with her favorite implementation and compilation options, without giving away write access to the shared directories. Shared write access to a directory containing code fragments that will be run by other users is of course a big security liability. Nor will allowing arbitrary programmers, of arbitrary levels of skill, to modify libraries shared by others contribute to overall system reliability.

The mechanism for controlling output locations is based on clever design by the original ASDF author. For `compile-op` operations on `cl-source-file` components, ASDF will direct the CL compiler to compile the file pointed to by the `component-pathname` and write it to the location pointed to by the `output-files` generic function. By defining an `around` method for `output-files`, it is possible to redirect where ASDF stores its outputs: such a method may apply some pathname translations to each of the considered pathnames and substitute the translated pathnames for the original pathnames. All ASDF-defined `perform` methods cooperate with this protocol, and user-defined methods are expected to do so as well.

`ASDF-Binary-Locations` (henceforth `A-B-L`) was a popular piece of software to redirect compiler output for ASDF, developed by Gary King. One problem with `A-B-L` was that output translations could not apply to the `A-B-L` source file itself, since `A-B-L` wasn't configured when it was compiled. Another problem was that, as for input locations above (Section 4.3), any non-standard configuration required code to be executed after the software was loaded but before it was used. Such configuration code had to set various *ad hoc* variables whose values would be combined in a complex way to create a configuration.

Other solutions included `common-lisp-controller` (`C-L-C`) or `cl-launch`, which were simpler and integrated well with other aspects of Unix, though less configurable. `C-L-C` avoided the above issues by making its translation mechanism part of a dumped image, and having a working default system-wide configuration. Unfortunately, `C-L-C` often broke libraries that were not written assuming that they would be controlled by it. Furthermore, CL implementations not distributed with `C-L-C` would not "see" the `C-L-C`-configured libraries. For instance, a programmer on a Debian system with `C-L-C`-integrated SBCL would not be able to easily use her `C-L-C`-managed libraries in a copy of Allegro she installed. `cl-launch` avoided the same issues as it carefully managed the execution, inserting the configuration code at the proper time, and providing a working default configuration that the user could override when invoking the launch script. However it only helped during deployment, not during development.

While ASDF maintainer, Gary King decided to integrate `A-B-L` into ASDF, which solved the first issue with `A-B-L` (loading it). However, this didn't solve the second issue (configuring it). We wanted an output locations mechanism whose configuration would have the same good properties as the the configuration of input locations. This didn't seem possible with the existing `A-B-L` API. Since we had to break backwards compatibility anyway, and since `A-B-L` was not one of the core parts of ASDF for which we guaranteed backwards compatibility, we designed a new mechanism, `ASDF-Output-Translations`.

`ASDF-Output-Translations` (`A-O-T`) is based on a few Domain-Specific Languages (DSLs) that allow to specify pathname patterns and translations in a simple way. We gave it a sensible default configuration that redirected the output of compiled files to an implementation-dependent path under the user's home directory, following the model of `cl-launch` but with improvements:
`~`/.cache/common-lisp/*implementation-id*/*source-path*
For example:

```
/Users/fred/.cache/common-lisp/
  allegro-8.2a-64bit-macosx-x86-64/
  Users/fred/Downloads/spatial-trees-0.2/
```

We ensured that `A-O-T` could express all the cases possible with `A-B-L`, and the other previous solutions. We went beyond this, covering cases not handled by any of the previous solutions. In particular, we improved the handling of Windows implementations, where pathnames have non-trivial host and device components. We provided a function to convert an `A-B-L` configuration into a new `A-O-T` configuration to ease transition. Finally, we provided the developer of ASDF extensions with an interface to specify when the translation should or should not occur.

Despite the backwards incompatibility, the sensible defaults, the upgrade path from previous solutions, and the ability to disable the mechanism in a documented way all conspired to make this new feature one that was widely accepted without much negative feedback. The biggest compatibility problem was with some systems using ASDF extensions that weren't designed for translated output; but they would have been broken by `A-B-L` already. By enabling output file translation by default, we caused these broken systems to **fail early for everyone rather than pass as working for some** and then fail for others.

4.5 Decentralized Configuration

As opposed to the previous central registry, our new source registry is decentralized. The configuration for our new output translations is also decentralized, and the two configuration mechanisms actually share a lot of code. At the user level and the system level, configuration files make it possible to extend or override the configuration. Moreover, at both levels, information is easy to provide in a decentralized way with configuration directories, and is easy to override in a centralized way with configuration files. Finally, the

ability to recurse through a directory hierarchy removes the need to either maintain a link farm or update the configuration as library source trees evolve.

The new decentralized configuration facilities decouple software distribution from software integration. The operating system software distribution (e.g. Debian or Red Hat) or user-level software distribution (e.g. clbuild) can simply provide software modules together with proper configuration as to where to find those modules. The distribution mechanism need not track and modify all the user-controlled programs that may use those modules. Further, the authors of CL programs can now rely on some software distribution mechanism rather than having to include their own module management. They can write scripts that integrate those libraries into programs without having to worry about how the libraries were distributed to the user. The steps of acquiring and using libraries can now be independent for CL programmers as they have for a long time been when using other languages (say C or Python), though with different constraints and through different means.

C programmers usually rely on libraries being installed in some system path, as compiled according to the system-wide ABI. CL lacks both system support and a standardized ABI; which implementation a library will be used with can't be determined at installation time, so this strategy won't work. Moreover, Lisp being a dynamic language favoring interactive development, distributing code as source allows for debugging and documentation habits that would be foiled by binary-only distribution. Source is all the more important since CL doesn't have a notion of "header files" describing the interface of functions independently from their source code, nor much of a standardized documentation system beyond reading the source code (though there are programmatically available documentation strings).

Compared to dynamic "scripting" languages like Python or Perl, CL in practice requires pre-compilation of source code into FASLs. It also has an exploded community divided by many implementations, so there cannot be such thing as a standard distribution of CL software. CL vendors install their bundle software wherever they want, or (like Franz) leave the installation location decision to their customers, and there is no cross-vendor standard for software installation.

The ability for ASDF to easily and predictably override a user's default configuration is also very important. Consider the case of a programmer who is involved in multiple different projects, each depending on its own portfolio of libraries. Each of the projects may require specific versions of libraries, sometimes the latest development software, sometimes an old release, sometimes including local patches, in ways that are incompatible with the version requirements of another project. Such a project may also need to be deterministically compiled from a controlled configuration of library dependencies rather than from whichever configuration sits on a particular computer, or is used by the team members to compile their other projects. Therefore, we make it possible to extend or override ASDF 2's configuration using environment variables; or programmatically from within the Lisp image.

Additionally, we provide a conventional shell-friendly syntax for the most commonly used subset of our configuration DSLs as well as a full-featured Lisp syntax, in the hope of keeping everyday system administration simple.

All this work on configuration, while technically simple, significantly increased the size of ASDF; but it was well worth it, because this **allows each one to contribute what he knows when he knows it, and does not require him to contribute what he doesn't know**. ASDF 1 required expert configuration work to be done by the end user, who needed to know all about the location of the various software modules involved in each project. With ASDF 2, the source-registry and output-translation configuration can be provided in pieces, each piece configured by He Who Knows Best about that piece, without requiring him to also configure things he doesn't know about. Distributors can distribute code and provide configuration information about where code is distributed, either at the system level or the user level. Integrators can write scripts that glue code together without having to worry about code distribution. Individual users can add bits about their user-specific configuration and either inherit system-provided configuration or override it, depending on their needs.

In the end, the aim of our interface modifications was this change in how programmers interact with the machine and with each other, smoothing the need for synchronization between experts and lowering the barriers to entry for newbies.

4.6 Finding data files

An extension related to pathnames above as well as to input and output locations is the question of how to find files distributed with an ASDF system, such as data files. A system containing code that wished to find such files for use *at run time*, often encountered difficulties: code that, for example, attempted to use `*load-pathname*` would be foiled by `A-B-L` or `A-O-T`. There were workarounds, typically involving the use of `(or *compile-file-pathname* *load-pathname*)`, or the definition of a global variable based on `*load-truename*` in a system's `.asd` file; but these were cumbersome and black art that were reinvented over and over. We have added `system-relative-pathname` to the ASDF API to make this easier. It accepts a portable relative pathname syntax and helps find files in their input locations without being foiled by translated output locations.

5. Engineering Best Practices

5.1 Sensible Data Structures

ASDF is (a) a lot of setup and configuration to build a graph that models the software being built, (b) a few simple actions that can be done on components, and (c) a small planning phase that computes from the model a list of such actions.

This planning phase, which is conceptually a simple depth first search but with plenty of cases, used to be encoded in a single humongous function `traverse`. In our bid to understand the algorithm and fix several bugs that affected correctness and/or performance, we broke the function down into smaller, more understandable parts. Along the way, we refactored the code, and reexperienced the fact that **good data structures and algorithms matter**.

ASDF was using the all-purpose Lisp data structure of linked lists as internal representations for sets of components and sequences of operations. This probably made the initial exploratory development of ASDF easy by allowing the original author to reuse the relatively well-endowed builtin "library" of CL functions operating such linked lists. However, linked lists are seldom the correct data structure for large data sets, and indeed, ASDF was extremely slow when running on large systems.

After breaking down the `traverse` function, we saw that while traversing the graph of component dependencies, ASDF 1 used to search for components by name through a linear search in a list, costing linear time per search and quadratic time overall. ASDF 2 uses a per-module hash-table, with constant access time per search, linear cost overall. Similarly in ASDF 1, the pruning of redundant subgraph traversals and the detection of circularity were done with sets implemented as lists, with linear time per operation cost, quadratic time overall cost. ASDF 2 uses hash-tables instead with constant time per operation, for an overall linear cost. Last but not least, in ASDF 1, the `traverse` function was recursively appending lists that are themselves the result of `append` operations from recursive calls to `traverse`, for an overall cubic runtime. ASDF

2 instead recursively accumulates items in a tree, then flattens the tree in the end, for a linear overall cost.

We ultimately gained over an order of magnitude speedup when using ASDF to plan the build of a large system (over 700 files) used at ITA Software. This system was generated by flattening what used to be a hierarchy of systems that had grown ugly cross-directory dependencies.

5.2 Regression Test Suite

When we redid the `traverse` algorithm, we found that there were bugs in some of undocumented features such that these features couldn't possibly have ever been working, much less used. One was the "feature" feature to conditionally depend on another component. Another was specifying a list as the `:force` keyword argument to operate with the intent of specifying systems that have to be rebuilt even if they haven't changed. In both cases, we fixed the code, and inserted a (continuable) error to warn the potential user that such a feature was never supported.

The main point is that if these features had had tests, the author would have caught earlier the fact that they weren't working. In software as complex as ASDF, a seemingly innocuous change can often break things badly in situations that the author of the change failed to consider. In order to avoid such bad changes, it is extremely important to have a good (or even a bad) regression test suite, that will detect changes in behavior and make it obvious when a change is bad that might otherwise have seemed good. To actually detect regressions, it is important that all tests should always pass on all supported implementations, even if some tests have to be disabled or tweaked on some implementations to not crash somewhere.

Testing is paramount. **Without the test suite, we'd be nowhere.** We only regret that we did not muster the time and courage to add test cases for every single feature.

5.3 Backward Compatibility

It is a testament to how central ASDF has become to the whole community that we felt enormous pressure to be backward compatible. We could not have achieved this goal without a regression test suite, and any failure we had in backward compatibility was first our failure to adequately extend our test suite. We did fail to maintain compatibility in many ways, sometimes intentionally, sometimes not. These compatibility failures caused pain for all involved, but it seems that ASDF 2 is welcome overall.

Interestingly, most of our incompatibilities are somehow related to pathname handling. Output pathname translations is now enabled by default, which surprised a few users, and broke a few systems that expected no such translation. The compatibility mode with `A-B-L` requires users to adjust their configuration, though in a straightforward way (also, one common configuration triggers a bug on one implementation). In ASDF 1, the `:pathname` keyword argument of `defsystem` was specially evaluated (other keyword arguments were not). We disabled evaluation of this argument, to make it homogenous with the same keyword argument for other components, which bit some users who switched to our new portable pathname designator syntax. Our new portable way of specifying relative pathnames for components was incompatible in a few corner cases in order to achieve a simple, coherent specification; this broke some existing code.

Our recursive search feature for the source registry has a slight but noticeable performance hit on some implementations, and failed due to a bug in one implementation. Finally, the mechanism by which one customizes a system so that Lisp files may use a different extension from the default `.lisp` has changed, due to our computing component pathnames eagerly rather than lazily.

A few programs that relied on unexported internals of ASDF had to be fixed when we changed the calling conventions of some functions; but we consider we were in our rights, as these interfaces hadn't been exported. We do export them now, however, and promise that we won't change them incompatibly in the future. However, exporting new functions created a namespace management issue for users who import all the symbols from ASDF, as some symbols clashed with symbols from other used libraries.

These incompatibilities were all easy to identify and fix, but it must be noted that we sometimes sacrificed backward compatibility to a greater sanity.

5.4 Portability

While developing ASDF 2, we relearned the value of several well-known software development practices. Here are the lessons we learned in the context of CL.

The first thing we learned to value was portability. For ASDF 2 to be widely adopted, we needed it to be portable to as many CL implementations as possible, adapting to the features, quirks, bugs and limitations of each. At the same time, for software using ASDF to be portable, we also needed to precisely define the semantics of ASDF 2 in a way that leaves as little space for undefined behavior as possible. **We can't simply be "transparent" with respect to semantic discrepancies between underlying implementations; we must abstract those discrepancies away.**

Most importantly, and as mentioned in Section 4.2 above, we had to implement our own well-defined syntax and semantics for specifying and using relative pathnames whose type component may have been independently specified. We also had to either rebind or shadow symbols for redefined functions depending on implementation as discussed in Section 3. In order to eliminate all warnings on all implementations, we had to be careful to include `ignorable` declarations in `defmethod`s, and avoid forward references to as yet undefined and undeclared functions. We also had to revert the previous use of the long form of `define-method-combination`, that some implementations did not support, instead introducing additional generic functions as extension points for which users may define methods. Finally, we wrote several portable wrappers over implementation-specific functions to access the environment, such as `getenv`, and made tens of small implementation-specific adaptations.

There is, however, one place where we took pains to be transparent to cross-platform semantic discrepancies: in the mini-languages to specify pathnames with a shell-friendly syntax, we adopted the same separator for lists of paths as the shells use respectively on Unix and Windows. Thus, we are using "`:`" on Unix vs "`;`" on Windows — indeed "`:`" is used as a device name indicator on Windows and cannot be used as a separator, and "`;`" is traditionally used instead. The reason our previously mentioned rule about abstracting underlying semantic discrepancies doesn't apply here is because the shell configuration is not something done *below* the level of ASDF that we may hide from the user, but something done *above* the level of ASDF. Indeed when the user is using cygwin as a Unix emulation over Windows, then the shell uses "`:`" and so does ASDF.

5.5 Keeping It Simple

In revising ASDF we vowed to follow the KISS principle, and not introduce any unnecessary complexity into the program. However, we did find that a lot of complexity was necessary — more than was originally anticipated by either the original authors or us.

We will maintain ASDF as the minimal but extensible core functionality of a build system for CL software. The principle is that **anything that can be provided as an extension should be provided as an extension and left out of the core**. ASDF has been

```
(defsystem :test-module-depend
  :components
  ((:file "file1")
   (:module "quux"
    :depends-on ("file1")
    :components
    ((:file "file2")
     (:module "file3mod"
      :components
      ((:file "file3")))))))
```

Figure 3. Example system illustrating the module dependency bug.

successfully extended to support such things as FFI generation, syntax extensions, etc. Although we tried to maintain simplicity, we found that configuration had to go into the core of ASDF. You can't bootstrap the configuration of inputs and outputs as an extension, because you need that configuration to locate and compile extensions. Similarly, on Windows, support for shortcuts had to go into the core.

With all the things we added, ASDF almost doubled in size since we started working on ASDF 2, and almost quadrupled since the original author left. Our release 2.008 is 146448 bytes long; 1.369, the last release by Gary King was 77079 bytes long; the last version by Daniel Barlow, in 2004, was 38881 bytes long.

On a positive note, CL helped us keep things simple. Andreas Fuchs wrote POIU [10], an extension for ASDF implementing parallel compilation. Because ASDF was lacking appropriate hooks, he had to redefine many internals of ASDF. These redefinitions were broken by our ASDF 2 modifications. One of us (Faré) suddenly became the maintainer of *both* pieces of software, and took the opportunity to provide from ASDF all the pieces needed for POIU to work without having to redefine anything. This required exposing a new function `component-operation-time`; making `operate` a generic function rather than a simple function, so that POIU could define `:around` methods on it; and adding slots to components to remember original definition-time. This dependency information was not previously retained by ASDF, which only remembered a digest fit for its specific traversal algorithm.

POIU showed CL, and particularly the CLOS object system, at its best. First, it was very impressive that POIU could be written without cooperation from the ASDF maintainers. This was achieved thanks to CLOS class redefinition. Second, the power of CLOS allowed us to keep the interface between POIU and ASDF very simple and informal.

6. Fixing `traverse`

One of the most annoying bugs we fixed for ASDF 2 was a problem with dependencies involving composite components (modules and systems). This bug illustrates issues in the original design of ASDF as well as limitations in our rewrite of it. Ours was a partial fix, and we will have to tackle a full solution in the future.

In ASDF 1, the dependencies of a composite component would fail to trigger recompilation of that component and its dependents (the components that depend on it). Figure 3 gives a system that shows the bug. If one were to load this system, modify `file1.lisp`, and reload the system, ASDF 1 would fail to recompile `file2.lisp` or `file3.lisp` the second time around.

The problem arises because of the way that ASDF models composite components such as `system` or `module`, that contain other components. As per the original design of ASDF, `traverse` generates a plan made of a sequence of elementary steps to `perform`, each a pair of `operation` and `component`. This design can be contrasted with a maybe more intuitive model where instead of a list of elementary steps the plan would have been a tree of composite steps and their constituent sub-steps. A consequence of this design is that the step corresponding to operating on the composite component does not wrap *around* the steps involving its constituents, but is only a synchronization mark scheduled *after* all of them. The `perform` method over such an step does nothing.

So, in ASDF 1, if one was to (compile and) load system `test-module-depend`, touch `file2.lisp`, and then ask whether `compile-op` had been done on module `quux` (containing `file2`), the answer would be "yes" (there are no effects to the `perform` method, there is nothing to do to re-do that, and you should not force any dependency because of that step not having been done yet), although according to the maybe more intuitive model, the answer would be "no" (there are effects required to compile the constituents of that module).

This foils the common desire of ASDF system definers to define `:around` methods for `perform`. Such methods could for instance bind a dynamic variable (such as `*readtable*`) around all of the loading done to a module or system, or establish some handler to catch common conditions, or build and check some datastructure based on effects of operating on sub-components (e.g. exported interfaces). Currently, such things have to be done inside each source file, or by specializing the class of the source files in the module.

Trying to fix composite dependencies in `traverse`, therefore, opened a big can of worms. The rule of good engineering that ASDF fails to adhere to is, **thou shalt tailor thy datastructures to the target problem**, not pick them based on how easy they are to express in the source programming language (and if your programming language isn't expressive enough, you picked the wrong one). CL, with its strong support for cons cells as lists and sets, was certainly misleading here; however ASDF gets some excuse for being a piece of software that cannot use datastructure libraries, since it is the one responsible for loading libraries.

ASDF 1 originally contained special-purpose, *ad hoc*, logic for modules to decide when their components needed to be operated on (since `operation-done-p` could not be used), and this special-purpose logic had some bugs. Faced with buggy code we didn't understand, we considered but quickly decided against a radical reimplementation according to the model we think was more intuitive, that would give wrapping semantics to `perform` on modules. Indeed, we didn't dare make big sweeping changes because we didn't understand the algorithm we were trying to fix, and the consequences such fix would have on client systems that may have depended on the existing model. It is only after refactoring the whole implementation, then later proofreading each other's explanations while writing this article, that we finally understand the ins and outs of the algorithm.

As for the bug illustrated in Figure 3 (which was taken from the ASDF 2 test suite), we also decided to only fix it in the case of modules within a system, but not in the case of systems. If a *system* `a`, depended on by system `b`, changes, that change does *not* trigger recompilation in `b`. We feel that this is not correct behavior. However, some users had come to consider this as a *feature* rather than a bug. We were conservative and preserved this behavior.

The rationale for the existing behavior is that, unlike the files that internal modules depend on, systems *usually* have stable interfaces, and that when their interface changes, either the client systems will change accordingly (in which case ASDF will detect that change and recompile them), or things will otherwise break in obvious ways. However, other users have complained that this is the wrong thing, and that compilers are fast enough that it is better to pay to recompile in the above cases. According to this argument, the recompilation is cheaper than wasting hours at de-

bugging an invalid image that wasn't properly recompiled when a system interface changes in a *subtly* incompatible way that causes an non-obvious bug (such by a modified macro definition). We will probably do what we consider to be the right thing eventually, and sacrifice backward compatibility with a dubious model. See Section 8.

7. Related work

7.1 History

A key inspiration for ASDF was `MK-DEFSYSTEM`, Mark Kantrowitz's portable `DEFSYSTEM` facility [11]. At the time when `MK-DEFSYSTEM` was developed, there was no portable, non-proprietary system definition facility for CL, as Lisp moved off special-purpose platforms and onto general-purpose hardware. Prior to this (and substantially prior to a true *Common* Lisp), there were a number of different system-defining facilities, notably the Symbolics `DEFSYSTEM` [21][4], but people wanting to portably define systems had to rely on `LOAD` scripts and/or `REQUIRE`.

The ASDF manual [4], discussing `MK-DEFSYSTEM` as an inspiration, explains that it was intended to better use modern CL capabilities. Notably, `MK-DEFSYSTEM` is written in pre-CLOS CL, whereas ASDF boasts use of CLOS for features for extensibility, in a way partly inspired by a design by Kent Pitman [14]. However, we argue that a primary reason for ASDF's success was not its CLOS architecture, but its elegant use of `*load-truename*` to solve the social problem of installing and referencing installed Lisp libraries. The problem of *installing* Lisp libraries was not helped by `MK-DEFSYSTEM`, but was substantially eased by ASDF. See Section 4.3 for more discussion of this issue.

BUILD [17] was an earlier Lisp build system (antedating CL) meant to replace the Symbolics `DEFSYSTEM`. It is cited as an influence to `MK-DEFSYSTEM`. BUILD attempted to be more declarative than `make` and other predecessors. It notably introduced the notion of automatically deducing dependencies from a graph of inter-file references, rather than having to manually declare rules that transitively enforce recompilation upon file modification.

7.2 Competing CL build tools

In the modern Lisp world, two alternatives to ASDF provide an interesting contrast. XCVB [16] abandons ASDF's single-image model to bring the pure functional way of building software in separate processes. On the other hand, McDermott's YTools system focuses on image maintenance, allowing the programmer to specify dependencies for units smaller than files, up to and including individual data structures.

XCVB XCVB [16] is a proposed replacement for ASDF. Where ASDF builds software into the current "One True" Lisp world in a context-dependent way, XCVB deterministically builds software into multiple virtual Lisp worlds (as many Unix processes). Image maintenance is achieved through a small system `xcvb-master` (one tenth the size of ASDF 2), that spawns XCVB as an external process to build software, that it subsequently loads in the current image, eliminating interferences between the current image and the build process.

XCVB is currently working and has many features that ASDF doesn't have and could not be evolved to have, including a deterministic build model, cross-compilation, and enforcement of declared dependencies. On the other hand, XCVB requires CL software to be adapted to its build specification format and its slightly stricter build constraints, and it doesn't currently support as many target platforms as does ASDF. Development of XCVB is ongoing, but from the simple one-paragraph idea to a satisfying product, an incredible — and originally unforeseen — amount of work is needed. XCVB has to handle most of the issues that we faced in ASDF 2, and some others as well, from the fine semantics of UNIX signals to distinctions between host and target system that appear in cross-compilation. On the other hand, XCVB has the advantage of clean slate redesign on its side.

Yale tools McDermott's YTools system [12][5] is at the opposite extreme from XCVB. Instead of concentrating on the build aspect of the problem, and starting multiple CL processes to do this most cleanly, McDermott's system focuses on the problem of "maintaining the coherence of a running Lisp." McDermott aims specifically at keeping a *single*, long-lived CL process in a coherent state. While the YTools system can do the same sorts of task as ASDF, McDermott also allows programmers to decompose system definition *below* the file level, allowing them to name arbitrary *chunks*, and define how these chunks are *derived* from each other. So, for example, one might define in a logic programming system, a table for procedural attachment, attaching functions to keywords, maintaining the functional attachment in a hash table. McDermott's system would allow one to record the fact that the hash table (one chunk) is derived from a set of function definitions (other chunks), automating the process of updating the hash table when one of the function definition changes. McDermott uses his chunk maintenance system as the substrate for the YTools File Manager, which plays a role similar to ASDF.

7.3 Build tools beyond CL

There are many related tools for other programming languages; too many to list here. We have already discussed `make` [8] in Section 2.2. Omake [9] is a notable modern take on the `make` concept: of interest to readers of this article are features such as automated dependency analysis, sub-project management, use of cryptographic checksums instead of timestamps, and extension using a DSL (instead of relying on shell commands). Possibly closer to ASDF in flavor are integrated build systems such as CONS [19] and SCONS [18], that aim to unify the entire build chain. These, too, aim to overcome many of the limitations of make. The `ant` system for Java also attempts to go beyond `make`, in particular in using an XML dialect to provide more declarative system specifications [2].

8. Future Directions

There are many as-yet unresolved issues in ASDF, which present opportunities for future improvement. We regarded these issues as out of scope for ASDF 2, either because we didn't have resources to solve them yet, or because they threatened backwards compatibility.

There have been repeated calls to make ASDF system definitions more fully declarative. Currently, ASDF system definitions are not fully declarative, which makes it difficult for ASDF extensions to be able to look into `.asd` files and reason about them. One substantial reason for being non-declarative is that there is no clear protocol for loading ASDF extensions required to interpret a system definition. Right now, system definers that use an ASDF extension must put something like

```
(asdf:load-system "my-asdf-extension")
```

or define their own classes and methods in the `.asd` file. The problem with this is, of course, that after it all bets are off about the readability of the file's contents. In ASDF 2 we extended

[4] Cited by Robbins [17]

[5] This system is, as far as we could tell, nameless. We call it "YTools," because the chunk manager is distributed as part of McDermott's YTools CL utilities library[13].

defsystem with a :defsystem-depends-on argument so developers may declare dependencies required *in order to process the system definition*. However, :defsystem-depends-on hasn't been widely adopted yet, and interacts poorly with previous conventions whereby symbols in extension packages were to be used in the defsystem form. Extension packages cannot be used with :defsystem-depends-on because the defsystem form has to be read before the :defsystem-depends-on argument is processed, causing the symbols to be created. We suspect that other aspects of the ASDF extension protocol may have to be amended before ASDF can be usefully considered declarative.

Another frequent request is provide a clean way to specify "conditional components." These may be used to support files only used on some implementations, or for test files only loaded when doing a test-op, not a simple load-op. Conditional system dependencies could also be added, such as test libraries, or support systems. Currently, implementation-specific changes are typically implemented with "#+" reader-conditions that are invisible to ASDF, and test files are typically handled by defining a separate test system.

Earlier in this paper (see Section 6), we discussed our struggles to fix composite component dependencies in traverse, and the somewhat unsatisfactory partial solution we came up with. Further fixing this problem would involve addressing the semantic problems of operations on composite components. One solution could be to modify the operation-done-p protocol to propagate a timestamp from the transitive dependencies of a build step. Or we could abandon the current model of a generated plan as a sequence of atomic actions for a more intuitive model that allows for nested actions, and may or may not generate a plan in advance. Considering how this may affect backward compatibility, such changes will no happen as part of the current ASDF 2 line, but may be part of a future ASDF 3.

There have been calls to make traverse part of the ASDF API in order to enable more introspection. This would be especially useful for authors of ASDF extensions. It would be wise to resolve the traverse bug before doing this, in order to provide a stable API, especially if we were to choose to add nested steps to ASDF plans.

More modest extensions include adding new standard operations and components to ASDF. There have been calls to add a doc-op, whose purpose would be to build documentation for a system. There *does* exist a test-op, but it has very weak semantics. System definers who implement this operation do not have clear directions about how to, e.g., signal success or failure. This is complicated by the fact the protocol for operate does not provide a standard return value.[6] One component type that has been repeatedly added by system definers (and often in a buggy way) is a "load-only" CL source file. This is a file that is to be loaded but *never* compiled. Previous system defining facilities, including MK-DEFSYSTEM [11], have included such components.

Perhaps the most valuable extension one could make would be to improve ASDF's documentation. We have begun this project, but have not done the thorough, end-to-end rewrite that is needed. A particular need in this area is to clearly document the ASDF protocols for operations and components. ASDF claims to be extensible through its use of CLOS, but in practice this extensibility is limited and problematic. The problem is that there is no clear specification of which methods must be defined when creating either a new component subclass or a new operation subclass. In practice, we find ASDF users doing this empirically, and such definers often find that what seemed to work well for them does not work for others using their new classes.

[6] While this might seem an obvious addition, accumulating such a return value is once again complicated by the fact that operate is done by processing a flat plan structure. For compile and load operations, one can simply raise an error, but one would not want a test operation to raise an error and abort on the first failure; typically one wants a report of all passing and failing tests.

9. Conclusion

From the point of view of computability, there is nothing one can do with ASDF 2 that one couldn't already do with ASDF 1. For that matter, there is nothing one can do with ASDF 1, that one couldn't do without it, and nothing that any piece of software can do that one couldn't do by reimplementing said piece of software. Of course, the implicit assumption above is that of development by a single notional programmer with unlimited resources. The whole point of ASDF is to improve software development in the case of many programmers each with limited resources, where communication between programmers is costly.

Our ambitions in developing ASDF 2 were relatively modest, as indeed were the ambitions behind ASDF itself. Despite this, ASDF has brought something to the CL community that past system build systems failed to achieve: the ability for developers to cooperate with each other, using pre-negotiated conventions to share their work and build large systems across the community without centralized coordination. In revising ASDF 1 to ASDF 2 much of the challenge was to do so in ways that would minimally disrupt this sharing, and ideally would further it. Doing so, we were reminded that software is hard, very hard. Simple ideas can take literally hundreds of iterations to get right.

We would like to conclude with an invitation to the entire CL community to enjoy ASDF 2 and, if they are so inclined, to further contribute. Interested parties can investigate the ASDF source repository. There is documentation, however imperfect. We would encourage CL implementers to please incorporate ASDF 2 into your implementations. To users we say: (1) Enjoy the new features, (2) report bugs to launchpad (https://bugs.launchpad.net/asdf) and (3) help us write better documentation. If you are really enthusiastic, become the new ASDF maintainer! If you do so, though, please be conscious of its central social role, and when changing it "*Primum non nocere*" — First, do no harm. We found this to be harder than it first appeared.

Acknowledgments

We thank Vadim Nasardinov, Ethan Schwartz, Scott McKay, Dan Barlow, and the anonymous reviewers for comments and suggestions.

References

[1] American National Standards Institute and Information Technology Industry Council. *American National Standard for information technology: programming language — Common LISP: ANSI X3.226-1994*. American National Standards Institute, 1996.

[2] ant. Apache ant website. URL http://ant.apache.org/.

[3] H. G. Baker. Critique of DIN kernel lisp definition version 1.2. *Lisp & Symbolic Computation*, 4:371–398, Mar. 1992.

[4] D. Barlow and contributors. *ASDF Manual*, 2001–2010. URL http://common-lisp.net/project/asdf/.

[5] A. Bawden. PCLSRing: Keeping Process State Modular. Technical report, MIT, 1989. URL http://fare.tunes.org/tmp/emergent/pclsr.htm.

[6] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System specification. Document 88-003, X3J13 Standards Commitee (ANSI Common Lisp), June 1988.

[7] P. Costanza. Closer project. web site, 2010. URL http://common-lisp.net/project/closer/.

[8] S. I. Feldman. Make–a program for maintaining computer programs. *Software Practice and Experience*, 9(4):255–65, 1979.

[9] J. Hickey and A. Nogin. OMake: Designing a scalable build process. In *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006*, pages 63–78. Springer, 2006.

[10] ITA Software. QITAB — a collection of free lisp code. URL `http://common-lisp.net/project/qitab/`.

[11] M. Kantrowitz. Portable utilities for Common Lisp. Technical Report CMU-CS-91-143, School of Computer Science, Carnegie-Mellon University, May 1991.

[12] D. McDermott. A framework for maintaining the coherence of a running lisp. In J. White, editor, *International Lisp Conference*, pages 279–288. Association of Lisp Users, June 2005.

[13] D. McDermott. Ytools distribution. web site, 2009. URL `ftp://ftp.cs.yale.edu/pub/mcdermott/software/ytools.tar.gz`.

[14] K. Pitman. The Description of Large Systems. MIT AI Memo 801, MIT AI Lab, September 1984. URL `http://www.nhplace.com/kent/Papers/Large-Systems.html`.

[15] F.-R. Rideau. Software irresponsibility, 2009. URL `http://fare.livejournal.com/149264.html`. Blog posting.

[16] F.-R. Rideau and S. Brody. XCVB: an eXtensible Component Verifier and Builder for Common Lisp. In *International Lisp Conference*, 2009. URL `http://common-lisp.net/projects/xcvb/`.

[17] R. E. Robbins. BUILD: A Tool for Maintaining Consistency in Modular Systems. MIT AI TR 874, MIT AI Laboratory, Nov. 1985. URL `ftp://publications.ai.mit.edu/ai-publications/pdf/AITR-874.pdf`.

[18] SCons. Scons website. URL `http://www.scons.org/`.

[19] B. Sidebotham. Software construction with Cons. *Perl Journal*, 3(1), 1998.

[20] R. C. Waters. The survival of Lisp: either we share, or it dies. *SIGPLAN Lisp Pointers*, VII(1-2):23–26, 1994. ISSN 1045-3563. doi: http://doi.acm.org/10.1145/192590.192600.

[21] D. Weinreb and D. Moon. *Lisp Machine Manual*, 1981.