Delivering Common Lisp Applications with ASDF 3.3

Robert P. Goldman
SIFT
rpgoldman@sift.net

Elias Pipping
FU Berlin
elias.pipping@fu-berlin.de

François-René Rideau
TUNES
fare@tunes.org

Abstract

ASDF is the *de facto* standard build system for Common Lisp (CL). In this paper, we discuss the most important improvements in ASDF versions 3.2 and 3.3. ASDF's ability to deliver applications as a single executable file now allows the static linking of arbitrary code written in C. We substantially improved ASDF's portability library UIOP, so its interface to spawn and control external processes now supports asynchronous processes. ASDF permits programmers to extend ASDF's build processes in an object-oriented way; until ASDF 3.2, however, ASDF did not correctly handle updates to these extensions during incremental builds. Fixing this involved managing the multiple phases in an ASDF build session. We also improved ASDF's source finding: it provides more usable default behaviors without any configuration; power users willing to manage its location caching can speed it up; and it offers better compliance with standard configuration locations.

CCS Concepts • **Software and its engineering** → *Software maintenance tools*;

Keywords ASDF, Build System, Common Lisp, Portability, Application Delivery, Demo

1 Introduction

Common Lisp (CL) is a general-purpose programming language with over ten active implementations on Linux, Windows, macOS, etc. ASDF, the *de facto* standard build system for CL, has matured from a wildly successful experiment to a universally used, robust, portable tool. While doing so, ASDF has maintained backward compatibility through many major changes, from Daniel Barlow's original ASDF in 2002 to François-René Rideau's largely rewritten versions, ASDF 2 in 2010, ASDF 3 in 2013, and now ASDF 3.3 in 2017. ASDF is provided as a loadable extension by all actively maintained CL implementations; it also serves as the system loading infrastructure for Quicklisp, a growing collection of now over 1,400 CL libraries. In this paper, we present some of the most notable improvements made to ASDF since we last reported on it [4], focusing on improvements to application delivery and subprocess management, better handling of ASDF extensions, and source location configuration refinements.

2 Application Delivery

ASDF 3 introduced *bundle operations*, a portable way to deliver a software system as a single, bundled file. This single file can be either: (1) a source file, concatenating all the source code; (2) a FASL file, combining all compiled code; (3) a saved image; or (4) a standalone application. In the first two cases, the programmer controls whether or not the bundle includes with a system all the other systems it transitively depends on.

We made bundle operations stable and robust across all active CL implementations and operating systems. We also extended these operations so that ASDF 3.2 supports single-file delivery of applications that incorporate arbitrary C code and libraries. This feature works in conjunction with CFFI-toolchain, an extension which we added to the *de facto* standard foreign function interface CFFI. CFFI-toolchain statically links arbitrary C code into the Lisp runtime. As of 2017, this feature works on three implementations: CLISP, ECL, and SBCL.

Loading a large Lisp application, either from source or from compiled files, can take several seconds. This delay may be unacceptable in use cases such as small utility programs, or filters in a Unix pipe chain. ASDF 3 can reduce this latency by delivering a standalone executable that can start in tens of milliseconds. However, such executables each occupy tens or hundreds of megabytes on disk and in memory; this size can be prohibitive when deploying a large number of small utilities. One solution is to deliver a "multicall binary" à la Busybox: a single binary includes several programs; the binary can be symlinked or hardlinked with multiple names, and will select which entry point to run based on the name used to invoke it. Zach Beane's `buildapp` has supported such binaries since 2010, but `buildapp` only works on SBCL, and more recently CCL. `cl-launch`, a portable interface between the Unix shell and all CL implementations, also has supported multicall binaries since 2015.

3 Subprocess Management

ASDF has always supported the ability to synchronously execute commands in a subprocess. Originally, ASDF 1 copied over a function `run-shell-command` from its predecessor `mk-defsystem` [2]; but it could not reliably capture command output, it had a baroque calling convention, and was not portable (especially to Windows). ASDF 3 introduced the function `run-program` that fixed all these issues, as part of its portability library UIOP. By ASDF 3.1 `run-program` provided a full-fledged portable interface to synchronously execute commands in subprocesses: users can redirect and transform input, output, and error-output; by default, `run-program` will throw CL conditions when a command fails, but users can tell it to `:ignore-exit-status`, access and handle exit code themselves.

ASDF 3.2 introduces support for asynchronously running programs, using new functions `launch-program`, `wait-process`, and `terminate-process`. These functions, available on capable implementations and platforms only, were written by Elias Pipping, who refactored, extended and exposed logic previously used in the implementation of `run-program`.

With `run-program` and now `launch-program`, CL can be used to portably write all kind of programs for which one might previously have used a shell script. Except CL's rich data structures, higher-order functions, sophisticated object system, restartable conditions and macros beat the offering of its scripting alternatives [4] [5].

ELS 2017, Brussel, Belgium

4 Build Model Correctness

The original ASDF 1 introduced a simple "plan-then-perform" model for building software. It also introduced an extensible class hierarchy so ASDF could be extended in Lisp itself to support more than just compiling Lisp files. For example, some extensions support interfacing with C code.

Unfortunately, these two features were at odds with one another: to load a program that uses an ASDF extension, one would in a first phase use ASDF to plan then perform loading the extension; and one would in a second phase plan then perform loading the target program. Of course, there could be more than just two phases: some extensions could themselves require other extensions in order to load, etc. Moreover, the same libraries could be used in several phases.

In practice, this simple approach was effective in building software from scratch, though not necessarily as efficient as possible since libraries could sometimes unnecessarily be compiled or loaded more than once. However, in the case of an incremental build, ASDF would overlook that a change in one phase could affect the build in a later phase, and fail to invalidate and re-perform actions accordingly. Indeed it failed to even consider loading a system definition as an action that may be invalidated and re-performed when it depended on code that had changed. The user was then responsible for diagnosing the failure and forcing a rebuild from scratch.

ASDF 3.3 fixes this issue. It supports a notion of session wherein code is built and loaded in multiple phases. It tracks the status of traversed actions across phases of a session, whereby an action can independently be (1) considered up-to-date or not at the start of the session, (2) considered done or not for the session, and (3) considered needed or not during the session, and if so, for how early a phase. When ASDF 3.3 checks whether an action is still valid from previous sessions, it uses a special traversal that carefully avoids either loading system definitions or performing any other actions that are potentially either out-of-date or unneeded for the session.

Build extensions are a common user need, though most build systems fail to offer proper dependency tracking when they change. Those build systems that do implement proper phase separation to track these dependencies are usually language-specific build systems (like ASDF), but most of them (unlike ASDF) only deal with staging macros or extensions inside the language, not with building arbitrary code outside the language. An interesting case is Bazel, which does maintain a strict plan-then-perform model yet allows user-provided extensions (e.g. to support Lisp [6]). However, its extensions, written in a safe restricted DSL, are not themselves subject to extension using the build system.

Fixing the build model in ASDF 3.3 led to subtle backward-incompatible changes. Libraries available on Quicklisp were inspected, and their authors contacted if they depended on modified functionality or abandoned internals. Those libraries that are still maintained were fixed.

5 Source Location Configuration

In 2010, ASDF 2 introduced a basic principle for all configuration: *allow each person to contribute what they know when they know it, and do not require anyone to contribute what they do not know* [1]. In particular, everything should "just work" by default for end-users, without any need for configuration, but configuration should be possible for "power users" and unusual applications.

ASDF 3.1, now offered by all active implementations, includes `~/common-lisp/` as well as `~/.local/share/common-lisp/` in its source registry by default; there is thus always an obvious place in which to drop source code such that ASDF will find it: under the former for code meant to be visible to end-users, under the latter for code meant to be hidden from them.

ASDF 2 and later consult the XDG Base Directory environment variables [3] when locating its configuration. Since 2015, ASDF exposes a configuration interface so all Lisp programs may similarly respect this Unix standard for locating configuration files. The mechanism is also made available on macOS and Windows, though with ASDF-specific interpretations of the standard: XDG makes assumption about filesystem layout that do not always have a direct equivalent on macOS, and even less so on Windows.

Finally, a concern for users with a large number of systems available as source code was that ASDF could spend several seconds the first time you used it just to recursively scan filesystem trees in the source-registry for `.asd` files — a consequence of how the decentralized ASDF system namespace is overly decoupled from any filesystem hierarchy. Since 2014, ASDF provides a script `tools/cl-source-registry-cache.lisp` that will scan a tree in advance and create a file `.cl-source-registry.cache` with the results, that ASDF will consult. Power users who use this script can get scanning results at startup in milliseconds; the price they pay is having to re-run this script (or otherwise edit the file) whenever they install new software or remove old software. This is reminiscent of the bad old days before ASDF 2, when power users each had to write their own script to do something equivalent to manage "link farms", directories full of symlinks to `.asd` files. But at least, there is now a standardized script for power users to do that, whereas things just work without any such trouble for normal users.

6 Conclusions and Future Work

We have demonstrated improvements in how ASDF can be used to portably and robustly deliver software written in CL. While the implementation is specific to CL, many of the same techniques could be applied to other languages.

In the future, there are many features we might want to add, in dimensions where ASDF lags behind other build systems such as Bazel: support for cross-compilation to other platforms, reproducible distributed builds, building software written in languages other than CL, integration with non-Lisp build systems, etc.

Bibliography

[1] François-René Rideau and Robert Goldman. Evolving ASDF: More Cooperation, Less Coordination. In *Proc. ILC*, 2010.

[2] Mark Kantrowitz. Defsystem: A Portable Make Facility for Common Lisp. 1990.

[3] Waldo Bastian, Ryan Lortie and Lennart Poettering. XDG Base Directory Specification. 2010.

[4] François-René Rideau. Why Lisp is Now an Acceptable Scripting Language. In *Proc. ELS*, 2014.

[5] François-René Rideau. Common Lisp as a Scripting Language, 2015 edition. 2015.

[6] James Y. Knight, François-René Rideau and Andrzej Walczak. Building Common Lisp programs using Bazel or Correct, Fast, Deterministic Builds for Lisp. In *Proc. ELS*, 2016.