Building Common Lisp programs using Bazel
or Correct, Fast, Deterministic Builds for Lisp

James Y. Knight
Google
jyknight@google.com

François-René Rideau
Google
tunes@google.com

Andrzej Walczak
Google
czak@google.com

Keywords Build System, Common Lisp, Bazel, Deterministic, Reproducible, Hermetic

Abstract

We will demonstrate how to build Common Lisp programs using Bazel, Google's hermetic and reproducible build system. Unlike the previous state of the art for building Lisp programs, Bazel ensures that incremental builds are always both fast and correct. With Bazel, one can statically link C libraries into your SBCL runtime, making the executable file self-contained.

1. Introduction

Common Lisp, a universal programming language, is notably used at Google for the well-known QPX server application (de Marcken 2001). Google can built its Lisp code base incrementally using its recently open-sourced Bazel built system (Google 2015).

Bazel is designed to build software in a reproducible and hermetic way. Hermeticity means that all build dependencies are checked into a source control. Reproducibility means building the same target multiple times from the same source code produces the same output. Thus, Bazel can assess what was or wasn't modified, and either rebuild or reuse cached artifacts. Reproducibility also facilitates testing and debugging production code. Bazel further enforces determinism by executing each build action in a container wherein only declared inputs may be read, and any non-declared output is discarded. Thanks to isolation, build actions can be parallelized, either locally or in a remote worker farm. Compilers and other build tools must be tuned to remove sources of non-determinism, such as timestamps, PRNG seeds, etc.

While mainly written in Java, Bazel is extensible using *Skylark* — a subset of Python with strict limits on side-effects. Using Skylark, three new rules were added to support building software written in Common Lisp: `lisp_library` for libraries (e.g. alexandria), `lisp_binary` for executables, and `lisp_test` for tests.

2. Previous Work

The previous state of the art for building large Common Lisp applications was ASDF (Rideau 2014). A descendent of the original Lisp DEFSYSTEM from the 1970s, ASDF builds all the code in the current Lisp image; incremental builds may therefore be affected by all kinds of potential side-effects in the current image; and only a build from scratch can guarantee to be deterministic. An attempt to build Lisp code deterministically, XCVB (Brody 2009), failed for social and technical reasons, though it had a working prototype.

Meanwhile, QPX was built using an ad-hoc script loading hundreds of source files before compiling them and reloading the resulting FASLs. The multi-stage build was necessary because of circular dependencies between the files; the dependencies formed a big "hairball" which any replacement for the build script had to handle.

3. Discussion

Let's discuss how to build a simple test application using the `lisp_binary` rule and how to declare its dependency on a Lisp library with `lisp_library`. Including C libraries as dependencies in the resulting Lisp binaries, as well as running the applications and test will also be mentioned.

A `lisp_library` is useful to declare an intermediate target which can be referenced from other Lisp BUILD rules. With SBCL, the `lisp_library` creates a FASt Load (FASL) archive by concatenating the FASL files produced by compiling each of its Lisp sources.

The attributes of the `lisp_library` rule are the Lisp sources `srcs`, Lisp libraries `deps`, C sources `csrcs` and C libraries `cdeps`, auxiliary `data` available at runtime to all executable targets depending on the library, and auxiliary `compile_data` available at compile-time while building.

The rule has additional build options. The `order` attribute notably specifies a build strategy; the default `"serial"` order loads each source file in sequence before compiling the next file. The `"parallel"` order compiles all files in parallel without loading other ones. The `"multipass"` order first loads all sources files, then compiles each one separately in parallel which is useful to compile a "hairball" aggregate.

```
load("@lisp__bazel//:bazel/rules.bzl",
     "lisp_library")
lisp_library(
    name = "alexandria",
    srcs = ["package.lisp",
        # ...
        "io.lisp"],
    visibility = ["//visibility:public"])
```

The above example is from the `BUILD` file of the "alexandria" library. First, Bazel loads the definition of the `lisp_library` from its conventional *build label* using the `lisp__bazel` "external repository". The `visibility` attribute indicates which `BUILD` packages are allowed to reference the rule's target — in this case, it is visible to any package.

ELS 2016 May 9–10, Kraków, Poland

The `alexandria.fasl` file can be located in the `bazel-genfiles` folder by issuing the command:

```
bazel build :alexandria
```

A `lisp_binary` rule is used to statically link an executable including both Lisp runtime and Lisp core image. It accepts attributes similar to `lisp_library`. If Lisp or C sources are specified, those will be compiled into corresponding Lisp and C components before being statically linked into the final binary. The `lisp_test` rule is a variation on the `lisp_binary` rule with the special purpose to be invoked with the `bazel test` command.

```
load("@lisp__bazel//:bazel/rules.bzl",
     "lisp_binary")
lisp_test(
  name = "myapp",
  srcs = ["myapp.lisp"],
  main = "myapp:main",
  deps = [
    "@lisp__alexandria//:alexandria"])
```

This `BUILD` file contains a `lisp_binary` target which references the "alexandria" `BUILD` target seen before. At startup, function `myapp:main` will be called with no arguments. The program is compiled and executed using:

```
bazel run :myapp
```

A `lisp_binary` can directly or transitively depend on C or C++ libraries. Static linking of the libraries makes it more reliable to deploy such a binary on multiple hosts in the cloud, without the opportunity to get library dependencies wrong; in particular, it helps to minimize discrepancies between test and production environments. C and C++ dependencies can be specified via the `cdeps` rule attribute, which can refer to any `cc_library` built with Bazel. The `csrcs` and `copts` rule attributes allow to directly specify C source files for which an internal C `BUILD` target will be generated.

Thanks to these build rules, the duration of the incremental QPX build went from about 15 minutes to about 90 seconds, with qualitative effects on developer experience. However, this is for a large project using a computing cloud for compilation. The open source version of Bazel currently lacks the ability to distribute builds, though it can already take advantage of multiple cores on a single machine. The typical Lisp user will therefore not experience a speedup when using the Bazel lisp rules; but he may enjoy the reproducibility.

4. Inside the Lisp rules

The build rules have been implemented using Bazel's extension language *Skylark*. The implementation has several levels, starting with Skylark *macros* which are basically Python functions. The `lisp_binary`, `lisp_library`, and `lisp_test` rules are implemented as macros invoking Skylark *rules*. Skylark rules consist of an implementation function and a list of attribute specifications that notably define type-checked inputs and outputs for the rule's target.

The use of a Skylark "macro" is necessary in order to establish two separate graphs of compilation for Lisp and the C counterparts, which are then connected at the final binary targets. So, the `lisp_library` "macro" calls `_lisp_library` rule to create Lisp related actions and also calls the `make_cdeps_library` to create the C related targets using Skylark's `native.cc_library`.

The rules compile C sources and Lisp sources in parallel to each other, and the resulting compilation outputs are combined together in the last step. This improves the build parallelism and reduces the latency. In order to facilitate linking, all C symbols referred to at Lisp-compilation time are dumped into a linker script file. That `.lds` file is then used to link the SBCL/C runtime part for the final executable. It makes the final linking step only include object files containing referred symbols, out of those provided by the C libraries; it also allows to statically detect any missing or misspelled C symbol. The `lisp_binary` and the `lisp_test` "macro" combines the SBCL/C runtime with the FASL based core by swapping the Lisp core part out in the runtime binary.

The `_lisp_library` rule implementation computes the transitive dependencies from referenced targets, compiles the sources using Skylark's `ctx.action`, and returns a Lisp *provider* structure to be used by other Lisp rules. Each of the Lisp sources are compiled in a separate process, possibly running on different machines. This increases build parallelism and reduces the latency when contrasted with waiting for each dependency to be compiled first before its compilation output is loaded. The compilation effects of one source are not seen when compiling other Lisp sources.

The Lisp provider structure contains transitive information about: FASL files from each Library target; all Lisp sources and reader features declared; deferred warnings from each compilation; the runtime and compilation data for each library. The Lisp text *sources* of the dependencies are loaded before compiling an intermediate target. The FASL files are only used when linking the final binary target. The deferred compilation warnings — mostly for undefined functions — are checked only after all FASL files have been loaded into the final target.

5. Requirements

The current version of the build rules works only with SBCL. Porting to a different Lisp implementation, while possible, requires non-trivial work, especially with respect to linking C libraries into an executable, or reproducing the low latency that achieved with SBCL's fasteval interpreter (Katzman 2015).

The rules have only been tested on Linux on the x86-64 architecture; they should be relatively easy to get working on different platforms, as long as they are supported by both SBCL and Bazel.

Bazel itself is an application written in Java taking seconds to start for the first time; then it becomes a server consuming gigabytes of memory, and can start an incremental build instantly. It isn't a lightweight solution for programming Lisp in a small; yet it is a robust solution for building software in the large.

6. Conclusion and Future Work

We have demonstrated simultaneously how Common Lisp applications can be built in a fast and robust way, and how Bazel can be extended to reasonably support a new language unforeseen by its authors.

In the future, we may want to add Lisp-side support for interactively controlling Bazel: we would like to be able to build code, and load the result code into the current image, without reloading unmodified fasls and object files.

Bibliography

François-René Rideau and Spencer Brody. XCVB: an eXtensible Component Verifier and Builder for Common Lisp. 2009. `http://common-lisp.net/projects/xcvb/`

Carl de Marcken. Carl de Marcken: Inside Orbitz. 2001. `http://www.paulgraham.com/carl.html`

Google. Bazel. 2015. `http://bazel.io/`

Douglas Katzman. SBCL's Fasteval. 2015. `https://github.com/sbcl/sbcl/tree/master/src/interpreter`

François-René Rideau. ASDF3, or Why Lisp is Now an Acceptable Scripting Language (extended version). 2014. `http://fare.tunes.org/files/asdf3/asdf3-2014.html`