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 reproducibly — assuming the build is hermetic, i.e. it depends only on sources that are checked into the source control. Thus, Bazel can see when the sources have been modified, and rebuild; otherwise, it will reuse cached build artifacts. Reproducibility means that building the same target multiple times produces the same output every time, which is important for debugging production code and for testing in general. To achieve reproducibility, compilers and other build tools need to remove sources of non-determinism, such as time-stamps, PRNG seeds, etc. Bazel further enforces determinism by executing each build action in a container whereby the action reads only from declared inputs, produces all declared outputs, after which all temporary files are discarded. Thanks to the isolation, build actions are easily parallelized locally or in a distributed environment and actions can also be cached.

While mainly written in Java, Bazel is extensible using Skylark – essentially 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.

ELS 2016 May 9–10, Kraków, Poland
Copyright © 2016 ACM [to be supplied]. . . $10.00

2. Previous Work

The de facto state of the art for building large Common Lisp applications is ASDF. As an evolution of the original Lisp DEFSYSTEM from the 1970s, ASDF builds all the code in the current Lisp image. This means that incremental builds are affected by all kinds of compilation side-effects. The only completely reliable way to get a deterministic build is to build from scratch. Modifying ASDF to build deterministically in isolated processes, while imaginable, wasn't possible because of the less understood dependency model of ASDF – at least, until fixed in 2013 (Rideau 2014) – for which work remains to be done. Another attempt to build Lisp code deterministically, XCVB (Brody 2009), failed for social and technical reasons when the resources were diverted towards ASDF and Bazel.

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

3. Discussion

We will demonstrate how to build a simple test application using the `lisp_test` rule and how to declare its dependency on a Lisp library declared with `lisp_library`. Running the application and tests will also be demonstrated. We will discuss how to include C dependencies in the resulting Lisp binaries.

A `lisp_library` is useful to declare an intermediate target which can be referenced from other Lisp build rules. Using SBCL, the `lisp_library` creates a fast load (FASL) archive, a concatenation of FASL files produced from compilation of each of its Lisp sources (`srcs`).

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

Other Lisp build options include for example the `order` attribute which specifies the order in which the files are loaded and compiled. The default `"serial"` order loads each of the Lisp text sources in sequence before compiling the next Lisp source. The `"multipass"` order loads all Lisp sources files first before compiling each one separately which is useful to deal with a "hairball" aggregate. The `visibility` attribute can make the rule's target available to other `BUILD` packages.

```
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 needs to load the definition of the `lisp_library` rule from the `lisp__bazel` "external repository" before the `lisp_library` is defined with the target's `name`, `srcs`, and `visibility`.

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

```
> bazel build :alexandria
```

A `lisp_binary` rule is used to link statically an executable with embedded Lisp runtime and Lisp core image. It accepts similar attributes as `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.

```
;; foo/test.lisp
(defun main ()
  (format t "This is a test!~%")
  (assert (alexandria:xor t)))

# foo/BUILD
load("@lisp__bazel//:bazel/rules.bzl",
     "lisp_test")
lisp_test(
  name = "test",
  srcs = ["test.lisp"],
  deps = [
    "@lisp__alexandria//:alexandria"])
```

The above `foo/test.lisp` contains a `main` function called at image start up. The next, `BUILD` file contains the system description with a `lisp_test` target which references the "alexandria" `BUILD` target seen before. The program is compiled and executed using:

```
> bazel test foo:test
```

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 the 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.

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 are constructs that consist of an implementation function, and a list of attribute specifications that notably define type-checked inputs and outputs.

The use of a Skylark "macro" is necessary in order to establish two separate graphs of compilation targets for Lisp and the C counterparts, which are then connected at the final binary targets. So `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 a 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; contrasted to waiting for each dependency to be compiled first before its compilation output may be loaded.

The compilation effects of one source are not seen when compiling other Lisp sources. The Lisp provider 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 only be checked when all FASL files have been loaded into the final target.

5. Requirements

The current version of the build rules work 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 the build instantly. It isn't a lightweight solution for programming Lisp in a small; yet a robust solution for building software in an industrial setting.

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`