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 be built its Lisp code base incrementally using its recently open-sourced Bazel built system (Google 2015).

Bazel is designed to build the software reproducibly — assuming you maintain hermeticity. Hermeticity here means that the build depends on sources that are checked into source control. Thus, Bazel can see when they have been modified, and rebuild; otherwise, it will reuse cached build artifacts. Reproducibility means that building the same target multiple times from the same source code 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 timestamps, PRNG seeds, etc. Tools at Google, including the Lisp compiler, are tuned accordingly. 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. Actions are also 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 executable binaries, and `lisp_test` for tests.

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

2. Previous Work

The state of the art for building large Common Lisp applications up until then was ASDF. An evolution of the original Lisp DEFSYSTEM from the 1970s, ASDF builds all the code in the current Lisp image. Building in the current Lisp image 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. Making 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; from the point of view the dependencies formed a big "hairball". An effort was made to chip at the "hairball" by moving files into a "prefix" of files without circular dependencies that the "hairball" depends on, and a "suffix" of files without circular dependencies that depend on the "hairball". The replacement for the build script had to handle the "hairball".

3. Discussion

We will demonstrate how to build a simple test applicatoin using the `lisp_test` rule and how to declare its dependecy on a standrd Lisp library declared with `lisp_library`. Running the application and tests will also be demonstrated. We will discuss how to include C libraries as dependencies in the resulting Lisp binaries and discuss the build feature called `"multipass"` used to compile "hairball" sources.

3.1 lisp_library

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 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.

The rule also accepts other Lisp build options. `order` specifies the order in which the files are loaded and compiled. The default `"serial"` order loads each of the sources in sequence before compiling the next Lisp source. The `"parallel"` order assures that each of the sources is compiled without seeing the other

Lisp source files. Finally, the `"multipass"` order loads all Lisp sources files first before compiling each one separately which is usefull compiling a "hairball" aggregate.

Lisp compilation can be modified by specifying the `features` attribute. The features are set before loading any dependencies or compiling any sources for the target are propagate transitively to any targets depending on the library.

The `nowarn` attribute allow to compile "hairy" sources for which the normal compilation would fail because of compilation and style warning. It accepts condtion names or predicates thereon. A set of predefined warning types is found in the `bazel.warning` package.

```
load("@lisp__bazel//:bazel/rules.bzl",
     "lisp_library")

lisp_library(
    name = "alexandria",
    srcs = [
        "package.lisp",
        "binding.lisp",
        # ... some omitted
        "io.lisp",
    ],
    visibility = ["//visibility:public"])
```

The above example is from the BUILD file of the "alexandria" library. First, Bazel needs to load the corresponding definition of the `lisp_library` rule. It finds the definitions by loading it, from a Bazel extension defined in the "external repository" `lisp__bazel`, which itself needs to be declared in the project's `WORKSPACE` file. The library is compiled in the default "serial" `order`. The `visibility` attribute restricts the availability of the rule's target to specified `BUILD` packages. In this example the target is "public".

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

```
> bazel build :alexandria
```

3.2 lisp_binary and lisp_test

A `lisp_binary` rule is used to link statically an executable with embedded Lisp runtime and Lisp core image. The inputs to the binary are Lisp sources `srcs`, Lisp libraries `deps`, C sources `csrcs` and libraries `cdeps`, and auxiliary compile or runtime data. If Lisp or C sources are specified, those will be compiled into corresponding Lisp/C library components before being statically linked into the final binary.

The produced executable binary can be run as any program. For this purpose the `main` rule attribute specifies the symbol of the entry point, which is the `cl-user::main` function by default.

Last but not least, the Lisp support for Bazel includes the `lisp_test` rule. The test rule is a variation on the `lisp_binary` rule: it also produces an executable program that can be executed on the command line. The special purpose of the test rule is to run tests when invoked with the `bazel test` command.

```
;; foo/test.lisp
(defun main ()
  (format t "Hello, world!~%")
  (assert
    (= 720 (alexandria:factorial 6))))

# foo/BUILD
load("@lisp__bazel//:bazel/rules.bzl",
     "lisp_test")

lisp_test(
  name = "test",
  srcs = ["test.lisp"],
  deps = [
    "@lisp__alexandria//:alexandria"])
```

The above example contains a `cl-user::main` function called at image startup from the `lisp_test` wrapper specific to the Lisp implementation. The `BUILD` file contains the system description. The rule references the above defined "alexandria" `BUILD` target and library. The program is compiled and executed using:

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

3.3 C dependencies

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 a 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 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.

Lisp rules compile C sources and Lisp sources parallel 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.

4. Inside the Lisp rules

The Lisp 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 indirect use of a Skylark "macro" is necessary in order to establish two separate graphs of compilation targets for Lisp and the C counterpart, which are then connected into a final binary executable target. 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 `_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: the FASL files produced by each Library target; all Lisp sources used and their md5 hashes; the Lisp features declared; the deferred warnings from each compilation; the runtime and compilation data from 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 undefined function warnings — are only be checked when all FASL sources have been loaded in the final target.

The `lisp_binary` and `lisp_test` "macros" perform similar tasks as the `lisp_library`. In addition, they compile the C runtime executable and link all but a minimal subset of the C libraries statically using `make_cdeps_library` and Skylark's `native.cc_binary`. They also produce a Lisp core containing the loaded FASL files. Finally, they combine the runtime and the core to form the final executable by swapping out the Lisp core part in the runtime.

5. Speed

Thanks to these rules, the duration of the incremental QPX build went from about 15 minutes to about 90 seconds, or a tenfold improvement, with qualitative effects on developer experience. This however is for a very large project using a large computing cloud for compilation. The open source version of Bazel currently lacks the ability to distribute builds that way, though it can already take advantage of multiple processing cores on a single machine. The typical Lisp user will therefore not experience the speedup when using the Bazel lisp rules. She may still enjoy the increased reliability and reproducibility of Bazel over traditional build methods.

6. Requirements

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

These Common Lisp rules have only been tested on Linux on the x86-64 architecture; but they should be relatively easy to get to work on a different operating system and architecture, as long as they are supported by both SBCL and Bazel: for instance, on MacOS X, and possibly, on Windows or on mobile devices.

Bazel itself is an application written in Java that takes many seconds to start the first time; then it becomes a server that eats several gigabytes of memory, but can start your build instantly. It isn't a lightweight solution for programming Lisp in a small setting; yet it is a robust solution for building quality software in Lisp in an industrial one.

7. 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; we would also like to be able to run tests with Bazel and interactively debug those that fail.

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`