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 systems, Lisp, 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, you can also statically link C libraries into your SBCL runtime, allowing for self-contained deployment of your program as a single executable file. Currently, only SBCL is supported, and has only been tested on Linux.

1. Introduction

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

Bazel is designed to build your software reproducibly — assuming you maintain hermeticity, as Google does. Hermeticity in this context means that the build should only depend on programs and files that are checked into source control; thus, Bazel can see when they are modified, and rebuild; or it can see that they haven't been modified, and reuse cached build artifacts. Reproducibility means that building the same target multiple times from the same source code should deterministically produce the same output every time; this is important for debugging production code as well as for testing in general. To achieve reproducibility, compilers and other build tools have to be tuned to remove sources of non-determinism, such as timestamps, PRNG seeds, hash values dependent on unstable addresses, I/O dependencies, etc. Tools used at Google, including the Lisp compiler, have been tuned accordingly. Bazel further helps enforce determinism by executing each build action in a container whereby the action may only read from declared inputs, and must produce all declared outputs, after which all other temporary files are discarded. Thanks to this isolation, build actions can be easily parallelized, either locally on a multiprocessor, or to a distributed farm of build workers; they can also be cached, making for reliable incremental builds that don't need to recompute outputs and intermediate results that only depend on unmodified source files.

While mainly written in Java, Bazel is also extensible using Skylark, a programming language that is essentially a subset of Python with strict limits on side-effects. Using Skylark, three new rules were written to support building software written in Common Lisp: `lisp_library` for libraries (e.g. alexandria, or cl-ppcre), `lisp_binary` for executable binaries (e.g. the QPX server, or your favorite application), and `lisp_test` for running unit tests or integration tests.

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. Unhappily, building in the current Lisp image means that incremental builds may be affected by all kinds of side-effects. Therefore, the only completely reliable way to get a deterministic build is to build from scratch. Modifying ASDF to build deterministically in isolated processes, while conceptually imaginable, wasn't possible because no one fully understood the dependency model of ASDF — at least, until it was eventually fixed in 2013 (Rideau 2014); such a modification is now possible but the work remains to be done.

A previous attempt to build Lisp code deterministically, XCVB (Brody 2009), was a failure, for social as well as technical reasons, including a lack of a sufficiently smooth transition from previous build systems, and lack of managerial support, until resources were diverted towards ASDF and Bazel.

Meanwhile, QPX itself was built using an ad-hoc script that had to load all of hundreds of files before compiling them and reloading them: this multi-stage build was made necessary because there were circular dependencies between those files; from the point of view of dependencies they 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"; but this cleanup isn't backed by a strong will, and requires splitting files in two (or more), and otherwise reorganizing the code, which in turns necessitates design effort requiring the involvement of senior engineers for whom this is not a high priority. Therefore, the replacement for that build script had to be able to handle that "hairball".

3. Features

The features that we will demonstrate include:

- Building a simple "hello, world" application in Common Lisp using `lisp_binary`.

ELS 2016 May 9–10, Kraków, Poland

- Building a simple library using `lisp_library`.
- Running simple tests using `lisp_test`.
- Including C libraries in a Lisp binary by depending on a `cc_library`.
- More complex dependency graphs using multiple `lisp_library` rules.
- Handling dependency hairballs using the `"multipass"` feature.

4. Discussion

4.1 lisp_binary

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 to the corresponding Lisp/C++ library components before being linked statically into the final binary. Further discussion about compilation related options is to be found with the `lisp_library` rule below.

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.

An example "hello, world" application is simply declared as follows:

```
;-> hello.lisp

(defun main ()
  (format t "Hello, world!~%"))

;-> BUILD

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

lisp_binary(
    name = "hello",
    srcs = ["hello.lisp"]
)
```

The above example contains the `cl-user::main` function that is called at image startup from the `lisp_binary` wrapper specific to the Lisp implementation. The `main` function does not receive any arguments, as access to command line arguments is not unified between Lisp implementations (you can access them in an implementation-specific way or through a portability layer such as the UIOP library). The `BUILD` file contains the system description. First, Bazel needs to load the corresponding definition of the `lisp_binary` rule. It does find the definitions by importing the Lisp rules, from a Bazel extension defined in the known "external repository" `lisp__bazel`, which itself needs to be suitably declared in your project's `WORKSPACE` file. Then it calls the `lisp_binary` rule to create a target named "hello", that uses the source file. The program is compiled, linked, and executed using the following command:

```
> bazel run :hello
```

4.2 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, which is possibly a concatenation of FASL files produced from compilation of each of its Lisp sources (`srcs`).

The attributes of the `lisp_library` rule are Lisp sources `srcs`, Lisp libraries `deps`, C++ sources `csrcs` and libraries `cdeps`, and auxiliary runtime `data` or compile data `compile_data`. The runtime data will be available to all executable targets depending on this library. The compile data is available at 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 will load each of the sources specified in sequence before compiling the next Lisp source; thus, each successive Lisp source can depend on data and info from previous Lisp sources. The `"parallel"` order assures that each of the sources will be compiled without having loaded any of the other Lisp source files. Finally, the `"multipass"` order will load all Lisp sources files first before compiling each one separately; this last option is useful for compiling "hairball" kind of Lisp aggregates.

The Lisp compilation can be modified by specifying reader features using the `features` attribute. The features are set before loading any dependencies or compiling any sources for the target. The features also propagate transitively to any targets that depend on the one library which specified the reader features.

By default the Lisp compilation is strict and any warnings will fail the compilation. In order to compile some "hairy" sources the `nowarn` attribute can be useful: it accepts names of condition types or names of condition-recognizing predicates that identify conditions that must be ignored. There is a set of predefined warning types found in the `bazel.warning` package.

```
;-> alexandria/BUILD

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

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

The above example of a Lisp library is for "alexandria". The library is compiled in the default "serial" `order` because the sources depend on each other. The `visibility` attribute restricts the availability of the rule's target to specified `BUILD` packages. In this example, there is no restriction and the target is "public".

To build the library one needs to invoke the following Bazel command which will produce `alexandria.fasl` file.

```
> bazel build :alexandria
```

The FASL file can be located in the `blaze-genfiles` folder and contains the compiled Lisp sources for the library.

4.3 lisp_test

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 too produces an executable program with a main entry point 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 ()
  (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 `foo/test.lisp` file with a `cl-user::main` referencing the `alexandria:factorial` function and an assertion. The corresponding `BUILD` file has a `foo:test` target defined which depends on the above defined "alexandria" `BUILD` target and library.

The corresponding test can be run using the following Bazel command line:

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

4.4 C dependencies

A `lisp_binary` can directly or transitively depend on C or C++ libraries, that will be statically linked into the executable program. Only basic runtime libraries such as `libc`, `libm`, `libdl` and `libpthread` remain dynamically linked; they must be specially provided by the execution environment.

This static linking makes it more reliable to deploy such a binary on a multitude of hosts in the cloud, without the opportunity to get library dependencies wrong; in particular, static linking helps minimize discrepancies between the test and production environments. This is also a vast improvement over the way Lisp programs with C libraries were typically compiled and deployed.

C and C++ dependencies can be specified via the the `cdeps` rule attribute, which can refer to any `cc_library` that can be built with Bazel. The `csrcs` and `copts` rule attributes also allow to directly specify C or C++ source files, which the Lisp rules will internally translate to a native Bazel `cc_library`.

The 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 build latency; the downside is that Lisp code that runs at Lisp-compilation time cannot call C++ functions.

In order to facilitate linking, all C/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 of the final executable. Computing this file makes the final linking step to only include object files containing referred symbols, out of those provided by the C/C++ libraries; it also allows to statically detect any missing or misspelled C/C++ symbol.

4.5 Parallel compilation

The Bazel Lisp rules compile each file in parallel with other files, after loading its declared dependencies using SBCL's fast interpreter (Katzman 2015). This increases build parallelism and reduces latency, as contrasted to waiting for each dependency to be compiled first before its compilation output may be loaded. This works fine with some restrictions: (a) Some code may use `eval-when` incorrectly and fail to support loading without compiling; it will have to be fixed. (b) Some code may do heavy computations at macro-expansion time and run somewhat slowly because it is not compiled; explicitly calling `compile` may speed up that code.

5. 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* — 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 (synthesized attribute) to be used by other Lisp rules. Each of the Lisp sources are compiled in a separate process, possibly running on different machines. 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 *sources* of the dependencies (not the compiled FASLs) are loaded before compiling an intermediate target; they will thus be processed by the implementation's interpreter rather than compiler. The FASL files are only used when linking the final binary target. The deferred compilation warnings — mostly undefined function warnings — can also only be checked when all FASL sources have been loaded in the final target.

The `lisp_binary` "macro" performs similar tasks as the `lisp_library`. In addition, it will 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`. It will produce a Lisp core containing all the loaded FASL files using the private `_lisp_binary` rule and finally it will combine the runtime and the core to form the final executable by swapping out the Lisp core part in the runtime.

6. 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; but he will may still enjoy the increased reliability and reproducibility of Bazel over traditional build methods.

7. 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.

8. 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`