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

Abstract

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

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques

Keywords Build System, Common Lisp, Bazel, Determinism, Reproducibility, Hermeticity

1. Introduction

Common Lisp, a general-purpose programming language, is used at Google for the well-known low-fare search engine QPX (de Marcken 2001). Google now builds its Lisp code incrementally, using Bazel, its recently open-sourced scalable build system.

Bazel is designed to build software in a reproducible and hermetic way. Hermeticity means all build dependencies, including build tools such as compilers, are kept under source control. Bazel can thus assess what was or wasn't modified, and either rebuild or reuse cached build artifacts. Reproducibility means building the same target multiple times from the same source code produces the same output. Reproducibility 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. Bazel can thus parallelize build actions, either locally or to remote workers. Bazel supports writing software in a mix of languages notably including C++, Java, Python, Go and Javascript. 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. We used Skylark to add support for building software written in Common Lisp.

2. Previous Work

The state of the art so far for building large Common Lisp applications is 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 to guarantee a deterministic build one has to build from scratch. ASDF also lacks good support for multi-language software. 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 build solution had to handle.

3. Building Lisp Software with Bazel

The input to Bazel is a set of `BUILD` files with Python-like syntax, wherein users declaratively specify their software *targets* using *rules*, and *macros* that expand into one or more rules.

Our first Lisp support function, `lisp_library`, declares an intermediate target which can be referenced from other Lisp rules. With SBCL (Steel Bank Common Lisp), `lisp_library` creates a FASt Load (FASL) archive by concatenating the FASL files produced by compiling each of its Lisp sources.

Bazel users specify *attributes* when defining rules. For Lisp rules, these include 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.

Lisp rules have 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 source files, then compiles each one separately in parallel, which is useful to compile a "hairball" aggregate.

For each file it compiles, Bazel will start a new Lisp process that will `load` all the Lisp source files from all the transitive dependencies of the file as well as relevant files from the local rule (as per the `order` attribute). Note that it does not load the compiled FASLs but the actual source files: not only is loading source files faster thanks to SBCL's `fasteval` interpreter (Katzman 2015) that Doug Katzman wrote specifically to speed up building with Bazel; loading source files also means that all compile actions can be run in parallel, whereas loading FASLs would introduce long chains of dependencies between compile actions, with fewer opportunities for parallelism and much higher latency. On the downside, this strategy means that some source files have to be fixed to add missing `:execute` situations in their `eval-when` forms, and optionally to explicitly `compile` any computation-intensive function used at compile-time.

```
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 for the "alexandria" general utility library. First, Bazel loads the definition of 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 all packages.

The following command builds alexandria.fasl and makes it available at a well defined path:

```
bazel build :alexandria
```

Our second Lisp support function, lisp_binary, statically links an executable including both Lisp runtime and Lisp core image. If Lisp or C sources are specified, they will be compiled into corresponding Lisp and C components before being statically linked into the final binary. Our third Lisp support function, lisp_test, is a variation on the lisp_binary rule meant to be invoked with the bazel test command.

```
load("@lisp__bazel//:bazel/rules.bzl",
     "lisp_binary")
lisp_binary(
  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, Lisp function myapp:main will be called with no arguments. The program may be 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 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 as large a speedup when using the Bazel lisp rules.

4. Inside the Lisp rules

Lisp support was implemented using Bazel's *Skylark* extension language. The lisp_binary, lisp_library, and lisp_test functions are implemented as Skylark *macros* calling internal implementation *rules*. A Skylark *macro* is basically a Python function that is executed by Bazel at the time the BUILD file is loaded and invokes the actual Skylark rules as side-effects. A Skylark *rule* consists of an implementation function and a list of attribute specifications that notably define type-checked inputs and outputs for the rule's target. The Lisp support uses macros to establish two separate graphs for each of the Lisp and C parts of the build, that are connected at the final binary targets. Thus, the lisp_library macro calls the _lisp_library rule to create Lisp related actions, and also calls the make_cdeps_library macro 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. The final linking step uses that .lds file to include from C libraries only the referenced objects, and to statically detect any missing or misspelled C symbol. The lisp_binary and the lisp_test macros then create the executable by combining the linked SBCL/C runtime with a core image dumped after loading all FASLs.

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: FASLs from each lisp_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 FASLs are only used when linking the final binary target. The deferred compilation warnings — mostly for undefined functions — are checked only after all FASLs have been loaded into the final target.

5. Requirements

The current version of the Lisp support for Bazel has only been made to work with SBCL on Linux on the x86-64 architecture. It should be relatively easy to get it working on any platform that is supported by both SBCL and Bazel. However, porting to a different Lisp implementation, while possible, will require non-trivial work, especially with respect to linking C libraries into an executable, or reproducing the low latency achieved with SBCL's fasteval interpreter (Katzman 2015).

Bazel itself is an application written in Java. It takes seconds to start for the first time; then it becomes a server that can start an incremental build instantly but consumes gigabytes of memory.

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. Bazel may not be a lightweight solution for writing small programs in Lisp. On the other hand, it has proven to be a robust solution for building a large industrial software projects tended by several groups of developers and implemented in multiple programming languages including Lisp.

Our code can be found at:
http://github.com/qitab/bazelisp

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.

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

Douglas Katzman. SBCL's Fasteval interpreter. 2015.

François-René Rideau. ASDF3, or Why Lisp is Now an Acceptable Scripting Language (extended version). 2014.