The GHC packaging ecosystem

Duncan Coutts — duncan@well-typed.com

GHC Contributors' Workshop 2023 —

Outline

- History
- Components overview: ghc-pkg, Cabal, cabal-install/stack, hackage-server and their relationship
- Focus on package dbs, package envs, ghc-pkg
- Focus on Cabal
- Focus on cabal-install
- Focus on hackage-server
- Q&A

But please interrupt with questions as we go!

History

Why look at history?

- To understand why things are the way they are now: what the context was; what problems were being solved.
- To understand what can or should be changed
- To understand what context is different now and thus what changes might be sensible

History

In the beginning...

There was ghc (and hugs and nhc98)

- several libraries bundled with ghc
- 3 libraries distributed independently of ghc (IIRC)
- much pressure to bundle more libraries with ghc

And the people saw that it was **not** good.

History

In the beginning…

There was ghc (and hugs and nhc98)

- several libraries bundled with ghc
- 3 libraries distributed independently of ghc (IIRC)
- much pressure to bundle more libraries with ghc

And the people saw that it was **not** good.

This was the problem, and **The Haskell Library and Tools Infrastructure Project** was going to solve it.

History

Timeline of early history

2003-09-29	Isaac Jones proposes “The Library Infrastructure Project” at the 2003 Haskell Implementors Meeting
2004-04-26	Isaac Jones makes first Cabal commit (darcs of course)
2004-09-01	Isaac Jones announces Cabal 0.1
2005-04-xx	David Himmelstrup develops `cabal-get`
2005-07-26	(Duncan’s first Cabal patch!)
2005-10-23	Isaac Jones publishes Cabal paper at TFP ’05
2005-04-30	`cabal-get` adopted and renamed to `cabal-install`
2006-09-14	first GHC hackathon (Portland)
2006-09-24	first packages uploaded to ‘HackageDB’
2006-10-11	(Duncan’s first release as Cabal release manager! Cabal-1.1.6)
2007-01-10	first Haskell hackathon (Oxford)

History

Original problems identified (Isaac Jones, 2003):

- Difficult to distribute (binary) Haskell libraries
- No standard build system
- Multiple compilers
- Language extensions and libraries a moving target
- No way to express dependencies on libs, compilers or versions
- No central repository for packages / libraries
- Hard to build Linux distro packages

History

Context at the time

- Standard build system interface:
  `./configure && make && make install`
- Linux distros are the state of the art in package distribution
- Distinction in roles between package author vs distro packager

History

Unclear at the time if the ‘Right Way’™ to do things is:

- for the package to adapt to the environment, classic `./configure` style
- make the environment adapt to the package, distro package manager style
- or a hybrid?

Package manager style needs **lots** of **accurate** metadata about dependencies.

Much effort in Linux distro packaging went into reverse engineering dependency metadata from `./configure` style packages.

There were strong views both ways

History

Was also unclear who should be ‘in control’

- `./configure` puts decisions in the hands of the package author:
  e.g. use of optional dependencies found on the system
- Can be quite unclear how `./configure` decides things or how to influence its choices
- Distro packaging people want to be in control:
  e.g. location of install, use of pre-existing system libraries

For a new packaging system we could ask package authors to declare metadata useful for distro packaging

History

Cabal was/is a specification! (plus some implementation)

- spec notionally agreed between Haskell compiler authors, tool authors and package authors
- spec requires support from compilers
- authors of ghc, nhc98 and hugs all co-authors of initial Cabal spec
- specifies `${compiler}-pkg` CLI for package registry
- specifies installed package metadata file format
  e.g. input to `ghc-pkg register`
- specifies source package metadata file format: `.cabal` files

For example: explains why `ghc-pkg` depends on Cabal lib for file formats

Components overview: ghc & ghc-pkg

ghc uses a **package environment** to satisfy module imports

A package environment is an ephemeral **set** of packages drawn from a list of **package databases**

A package database is **set** of packages, represented on disk

ghc only reads package databases, never modifies

The ghc-pkg tool used to edit package databases

The ghc/ghc-pkg notion of a package is really a **library**, tools/pre-processors/applications are not registered

Components overview: Cabal

Cabal **specification** defines:

- the notion of a package for distribution
  (not the same as a ghc package!)
- the `.cabal` file format
- the `Setup.hs` CLI
- the `ghc-pkg` CLI
- the `ghc-pkg` 'installed package info' input file format

In theory this is compiler agnostic. In practice, hugs, nhc98 RIP

Components overview: Cabal

`Cabal` **library** provides

- data types, parser and pretty printer for the `.cabal` file format and the `ghc-pkg` input format
- list of all known Haskell language extensions (in all compilers)
- the above now split out into `Cabal-syntax` library
- impl of the `Setup.hs` CLI using the 'Simple' build system
- 'Simple' build system support for multiple compilers
- hooks and utilities for custom `Setup.hs` scripts

These days the `Setup.hs` CLI is rarely invoked manually

Components overview: `cabal-install`

`cabal-install` is a CLI for building and managing packages

- intended for use by human Haskell devs, as well as by tools
- implementation defined, not defined by specification
- mix of build tool and package management tool
- modern incarnation is based on **projects**
- a project is an environment for building a related set of packages
- supports a solver-based workflow for building projects
  (or fixed versions for (mostly))
- supports reproducible builds by index snapshots or freeze files
- fetches Haskell dependencies from remote sources: hackage archives, source control repos, local and remote tarballs

Components overview: Hackage

Hackage archive format

- notionally defined by specification, implementation independent
- static file archives using generic HTTP servers are possible
- used by `cabal-install` and other tools
- provides an index of `.cabal` files
- provides package tarballs
- uses TUF-based cryptographic security system
- supports trustless mirrors
- append-only index format intended for incremental update

Components overview: hackage-server

hackage-server is an implementation of a hackage archive

- standalone HTTP server application
- lots of features beyond serving archive
- users, permissions
- package uploads
- limited post-release editing of package metadata
- package browsing and search
- package haddock documentation
- used for the public hackage.haskell.org
- running private instances is possible

Components overview: `stack` and `stackage`

`stack` is an alternative to `cabal-install`

- different philosophy and intended workflows
- focus on reproducible project builds
- based on curated collections of packages
- fixed versions of packages, from collections or selected manually
- intended for a non-solver based workflow
  (but can also use `cabal-install` to use solver)

`stackage` is sort-of an alternative to hackage

- `stackage` hosts curated versioned subsets of the public hackage
- stackage snapshots have a single version of each package
- all packages in a snapshot **should** work together
- stackage format is custom (based on git)

ghc's concept of a package

ghc's concept of a package

- a library
- an **installed** package
- a **binary** library built against **specific** dependencies
- also called a 'unit' within ghc code

Registration information for a package includes

- all the information needed to compile and link against this library
- other metadata not needed by ghc (but available to other tools)
- backpack unit info…
- some historical cruft
- defined by `data InstalledPackageInfo` from `Cabal-syntax`

How ghc identifies a package

Two kinds of package identifier

- source package identifier: name and version
- installed package identifier: **opaque string identifier**

ghc **uniquely identifies** packages by installed package identifier

Allows for multiple installed instances of the same source package, e.g. compiled against different dependencies

Example

id: `network-3.1.2.7-2d72df787370b1f16d6201e748b2276c2e8...`
name: network
version: 3.1.2.7

ghc package environments

A package environment is a concept used within ghc

- a set of packages (by unique installed package id)
- hopefully closed and coherent
- used to resolve module imports during compilation
- ephemeral, constructed and used within a ghc invocation

Constructing a package environment is sadly non-trivial

- packages selected from combination of package databases
- command line flags to select by source package id, or installed package id
- complicated ‘shadowing’ algorithm (to support old use cases)

ghc package dbs

A ghc package database:

- is a set of registered packages
- unique primary key is the installed package id
- not closed, dependencies can be satisfied from other databases
- has an on-disk representation

Multiple package dbs

- global package db: system wide in multi-user installations, contains all libraries bundled with ghc
- user package db: per-user db in multi-user installations, initially empty
- other specific package dbs: specified on ghc command line

ghc package dbs

On-disk representation:

- `${pkgdb}/${pkgid}.conf`: human readable registration files
- `${pkgdb}/package.cache`: a binary cache of all the packages
- `${pkgdb}/package.cache.lock`: lock file for modification

The binary cache:

- for efficiency
- ghc **only** reads the binary cache
- written by `ghc-pkg`
- avoids ghc depending on `Cabal-syntax` library

Historical note: “Cabal Hell”

Historically:

- no separate concept of package environment
- the ghc package db(s) defined the environment
- **constraint** that there be at most one version of each source package in each db (or stack of dbs)

Result was blamed on Cabal, aka “Cabal Hell”

- installing new packages could break old ones
- or installing new packages have to be forced to be consistent with existing ones (which might be impossible)
- result depends on the order of installation – non-deterministic

Grumpy Haskell developers

Historical note: “Cabal Hell”

Constraint

there can be at most one version of each package in each db

This is the problem! This is not a reasonable constraint!

Historical note: “Cabal Hell”

Constraint

there can be at most one version of each package in each db

This is the problem! This is not a reasonable constraint!

- Must allow multiple instances of the same version of the same source package
- Leads to design with source and installed package identifiers
- But still want coherent environments for compiling
- Leads to design with environment as a subset of package db
- Still some historical cruft in ghc from the old design
  e.g. exposed vs hidden packages, complex shadowing algorithm
- Backpack also relies on this design

Persisting package environments

What environment do you get when you run `ghci` manually?

Selecting coherent environment from package db needs tool support.

Package environment files

- subset of ghc package cli flags
- select package dbs and package ids within

Example:

~/.ghc/x86_64-linux-8.10.7/environments/default

```
clear-package-db
global-package-db
package-db /home/duncan/.cabal/store/ghc-8.10.7/package.db
package-id base-4.14.3.0
...
package-id network-3.1.2.7-2d72df787370b1f16d6201e748b2276c2e8...
```

Persisting package environments

Various ways to select a package environment

- explicitly by file `-package-env ${file}`
- explicitly by name `-package-env ${name}` from `$XDG_DATA_HOME/ghc/arch-os-version/environments/name`
- implicitly via `GHC_ENVIRONMENT` shell environment variable
- implicitly via local file `.ghc.environment.arch-os-version`
- `$XDG_DATA_HOME/ghc/arch-os-version/environments/default`

Mechanism intended for tools to manage env files.

Allows ‘entering’ an environment for a shell session, or for a particular directory.

ghc-pkg

ghc-pkg tool:

- manages package dbs: register, unregister
- provides interface to tools
- somewhat encapsulated package db file format
- uses `Cabal-syntax` library to parse input
- does some sanity checking
- some historical cruft

Also supports an idempotent mode

- add or remove `${pkgid}.conf` file directly within package db dir
- run ghc-pkg `recache`
- idempotent mode suits package management tools

Cabal packages

Cabal concept of a package

- unit of **distribution**
- a **source** package
- versioned
- somewhat flexible, can be built against a range of dependencies
- consists of several **components**: libraries, executables, tests
- one ‘public’ library, but recent support for multiple libraries
- distribution metadata: authors, license, homepage etc
- intended to be automagically translated into distro/system packages, which limits expressiveness somewhat

Roughly: Cabal component $==$ ghc package/unit.

Cabal-syntax library

Cabal-syntax library

- defines the .cabal file format
- defines the ghc-pkg input format
- relatively recently split out of Cabal lib
- historically bootstrapping was a big deal
- defines types, parser and pretty printers

Less heavy dependency than Cabal library

But still chunky: ≈ 16 kloc

Cabal library

Cabal library

- implements the `Setup.hs` CLI
- implements the so-called ‘Simple’ build system
- ≈ 25 kloc

‘Simple’ build system

- Historically “for simple packages”
- embodies a **lot** of knowledge on building Haskell code
- portable across OSs, even Windows
- multi-compiler design, still supports ghcjs and uhc
- not a great build system
- delegates to ghc `--make`

Cabal build phases

Cabal build phases: `./Setup $cmd`

configure	find compiler, tools, check and resolve dependencies
build	pre-process, compile and link all components
install	move files into an image dir or final location
register	register libraries with the compiler

Modelled on classic `./configure && make && make install`

Originally designed to be a human interface

Now called by other tools (with hundreds of flags!)

Setup.hs

With hindsight, `Setup.hs` was a design mistake

Original concept

- each package has its own build system
  (this was true at the time for the 3 pre-existing libraries!)
- standard CLI interface to the package's build system
- simple packages use the 'Simple' build system in Cabal
- complex packages use their own, or customised one from Cabal

In practice, **all** packages use the 'Simple' build system, or 'Custom' which is Simple with custom pre and post hooks.

We pay the costs but do not exercise the flexibility

Setup.hs

Problems with `Setup.hs`

- would like to use one build system across all packages
- incompatible with using GHCi across multiple packages
- incompatible with IDEs
- slow: configure step repeated for every package
- custom `Setup.hs` scripts are fragile
- CLI had to grow lots of features to support package managers
- CLI no longer a human interface
- incompatible with building package components independently
- hard to build in parallel
- blocks or complicates various other features

Flexible dependencies

Packages and distribution are to enable large scale distributed development

Packages evolve over time

Packages can often work with a range of versions of a dependency

Flexibility is often necessary to enable reuse

Flexible dependencies

Each component specifies dependencies with an optional version range

Example

build-depends: bytestring >= 0.9 && < 0.12

When using a package in a context we have to decide exact versions of dependencies. Can choose manually or automagically.

Optional dependencies

Cabal file format supports conditionals and per-package flags

These can express optional dependencies

Example

```
if compiler(ghc)
   build-depends: ghc-prim
if flag(feature-foo)
   build-depends: foo
   ghc-options: -DENABLE_FOO
```

Package flags can be controlled by the user or selected automagically

cabal-install

`cabal-install` is intended as a tool for developers for building code

Necessarily involves dependency and package management

The modern incarnation is project based, steals ideas from nix

Dependency resolution

Deciding what versions of dependencies to use is tricky

First version of `cabal-install`

- would recursively download and build packages from hackage
- näive algorithm to decide versions of dependencies
- could scale to **several** package dependencies!

Dependency resolution

Why is this a hard problem?

Types!

Passing something of type T between two libraries requires the libraries agree on the type of T, hence both must be built against the same version of the library that defines T.

This introduces equality constraints in selecting package versions.

In general this turns out to be an **NP-complete** problem!

Dependency resolution

`cabal-install` uses a solver

- custom solver
- constraint based
- plus soft preferences for heuristics
- ‘only’ 4kloc
- evolved over the years, especially performance
- now scales to large projects
- but still can be slow
- error messages are ok but not great

Dependency resolution

Dependency resolution also has to resolve values of per-package flags

Example

```
if flag(feature-foo)
   build-depends: foo
   ghc-options: -DENABLE_FOO
```

The user can fix a choice, or the solver can find a choice that works.

This allows picking an optional dependency by default if it works.

Projects and packages

A cabal project:

- contains a set of local packages
- defines an environment for local packages and dependencies
- allows setting configuration for all packages (local or deps)
- isolated from other projects
- opportunistic sharing with other projects
- can support (mostly) reproducible builds

Nix style package management

cabal-install steals ideas from nix

- package identifier is a hash of package config plus dependencies
- package store provides caching and sharing between projects
- single ghc package db covers whole store
- relies on ghc's installed package identifiers
- ghc environment per project

Gets many of the nix advantages

- installing new packages never breaks existing packages
- switching environments is cheap
- concurrent environments / projects
- deterministic: doesn't depend on what is installed already
- mostly reproducible, but cannot cover system dependencies

Solves most causes of "Cabal Hell"

`cabal-install` internal flow

Most commands follow the same phases

- discovery phase: establish what is in the project
- solving phase: run the solver to make an install plan
- planning phase: elaborate the install plan with full details
- building phase: execute the detailed install plan
- post-build phase: update state and report failures

Reasonably nice separation

- Decide what to do, then do it
- Most complexity is in solving and planning
- elaborated install plan is a useful artefact for debugging cabal-install

Caching is used to skip solving and planning, and skip building components that are already up to date.

Hackage

Crucial infrastructure to enable large scale distributed development

Centralised **distribution** of releases

- never intended to replace github or project homepages
- upstream stable archive for downstream distros
- crucial that package tarballs are stable (immutable)

Enables some light-touch quality assurance

- various checks at upload time
- post-upload editing of package metadata

Hackage archive format

Hackage archive format

- package tarballs
- index tarball of all package .cabal files

Archive directory layout

01-index.tar
01-index.tar.gz
${pkgname}/${version}/${pkgname}-${version}.tar.gz
root.json snapshot.json timestamp.json mirrors.json

Index tarball layout

${pkgname}/${version}/${pkgname}.cabal
${pkgname}/${version}/package.json

Hackage archive format

Designed to be served over HTTP

- download index to use in solving and planning
- only download package tarballs as needed

Index as (compressed) tarball format

- standardised format, portable, stable, 3rd party tools
- append only
- incremental update by fetching the tail with HTTP range request
- supports arbitrary snapshots of index by using prefixes

Hackage security

Hackage has a (partial) security system

- based on ‘The Update System’ (TUF) design
- prevents man-in-the-middle, replay and rollback attacks
- enables safe use of untrusted mirrors
- TUF design customised for Hackage (index format)
- does not (yet) include per-package author signing
- implemented in `hackage-security` package

Hackage mirrors

Mirrors

- `mirrors.json` can publish known mirrors
- `hackage-security` package will use mirrors automagically
- `hackage-security` used by `cabal-install` and other tools
- `hackage-mirror` client can incrementally mirror
- `hackage.haskell.org` has a couple public mirrors

Post-release editing of package metadata

Possible to update `.cabal` file in index without uploading new tarball

Why:

- enables distro maintenance role,
  centrally rather than duplicating per distro
- quicker fixing of minor problems

How:

- package tarball not modified (tarballs **must** have stable hash)
- tar format supports file replacement by appending new file
- original and all updates available in the index
- `cabal-install` uses the latest `.cabal` file in the index for solving, and overwrites the the `.cabal` file after unpacking
- compatible with `cabal-install` stable index snapshots

Simple package search

Large scale distributed development requires that developers be aware of what is available to reuse

So package search is important!

`hackage-server` has a simple full text search

- uses `tokenize` lexer, `snowball` stemmer, `full-text-search`
- `full-text-search` lib uses BM25 ranking algorithm
- multiple weighted text attributes: package name, synopsis and description
- should be extended to non-text attributes like ‘package rank’

hackage-server

Original `hackage-scripts` used CGI and file system as database

`hackage-server` rewrite as a http server

- "all in one" design, http server and data store
- based on Happstack http server library
- uses `acid-state` for most state
- source and documentation tarballs stored on disk
- 'only' *approx* 30kloc

hackage-server

Internal design is very modular

- perhaps unnecessarily modular for its size
- URLs and data store for each feature is separate
- features can be turned on/off
- features combined at top level at runtime

These days might do it differently

- whole application DB schema
- whole application URL scheme

Q & A