Cabal User Guide

Building and installing packages

After you've unpacked a Cabal package, you can build it by moving into the root directory of the package and running the `cabal` tool there:

```
cabal [command] [option...]
```

The *command* argument selects a particular step in the build/install process.

You can also get a summary of the command syntax with

```
cabal help
```

Alternatively, you can also use the `Setup.hs` or `Setup.lhs` script:

```
runhaskell Setup.hs [command] [option...]
```

For the summary of the command syntax, run:

```
cabal help
```

or

```
    runhaskell Setup.hs --help
```

Building and installing a system package

```
runhaskell Setup.hs configure --ghc
runhaskell Setup.hs build
runhaskell Setup.hs install
```

The first line readies the system to build the tool using GHC; for example, it checks that GHC exists on the system. The second line performs the actual building, while the last both copies the build results to some permanent place and registers the package with GHC.

Building and installing a user package

```
runhaskell Setup.hs configure --user
runhaskell Setup.hs build
runhaskell Setup.hs install
```

The package is installed under the user's home directory and is registered in the user's package database (`--user`).

Installing packages from Hackage

The `cabal` tool also can download, configure, build and install a Hackage package and all of its dependencies in a single step. To do this, run:

```
cabal install [PACKAGE...]
```

To browse the list of available packages, visit the Hackage web site.

Developing with sandboxes

By default, any dependencies of the package are installed into the global or user package databases (e.g. using `cabal install --only-dependencies`). If you're building several different packages that have incompatible dependencies, this can cause the build to fail. One way to avoid this problem is to build each package in an isolated environment ("sandbox"), with a sandbox-local package database. Because sandboxes are per-project, inconsistent dependencies can be simply disallowed.

For more on sandboxes, see also this article.

Sandboxes: basic usage

To initialise a fresh sandbox in the current directory, run `cabal sandbox init`. All subsequent commands (such as `build` and `install`) from this point will use the sandbox.

```
$ cd /path/to/my/haskell/library
$ cabal sandbox init                      # Initialise the sandbox
$ cabal install --only-dependencies       # Install dependencies into the sandbox
$ cabal build                             # Build your package inside the sandbox
```

It can be useful to make a source package available for installation in the sandbox - for example, if your package depends on a patched or an unreleased version of a library. This can be done with the `cabal sandbox add-source` command - think of it as "local Hackage". If an add-source dependency is later modified, it is reinstalled automatically.

```
$ cabal sandbox add-source /my/patched/library # Add a new add-source dependency
$ cabal install --dependencies-only            # Install it into the sandbox
$ cabal build                                  # Build the local package
$ $EDITOR /my/patched/library/Source.hs        # Modify the add-source dependency
$ cabal build                                  # Modified dependency is automatically reinstalled
```

Normally, the sandbox settings (such as optimisation level) are inherited from the main Cabal config file (`$HOME/cabal/config`). Sometimes, though, you need to change some settings specifically for a single sandbox. You can do this by creating a `cabal.config` file in the same directory with your `cabal.sandbox.config` (which was created by `sandbox init`). This file has the same syntax as the main Cabal config file.

```
$ cat cabal.config
documentation: True
constraints: foo == 1.0, bar >= 2.0, baz
$ cabal build                                  # Uses settings from the cabal.config file
```

When you have decided that you no longer want to build your package inside a sandbox, just delete it:

```
$ cabal sandbox delete                       # Built-in command
$ rm -rf .cabal-sandbox cabal.sandbox.config # Alternative manual method
```

Sandboxes: advanced usage

The default behaviour of the `add-source` command is to track modifications done to the added dependency and reinstall the sandbox copy of the package when needed. Sometimes this is not desirable: in these cases you can use `add-source --snapshot`, which disables the change tracking. In addition to `add-source`, there are also `list-sources` and `delete-source` commands.

Sometimes one wants to share a single sandbox between multiple packages. This can be easily done with the `--sandbox` option:

```
$ mkdir -p /path/to/shared-sandbox
$ cd /path/to/shared-sandbox
$ cabal sandbox init --sandbox .
$ cd /path/to/package-a
$ cabal sandbox init --sandbox /path/to/shared-sandbox
$ cd /path/to/package-b
$ cabal sandbox init --sandbox /path/to/shared-sandbox
```

Note that `cabal sandbox init --sandbox .` puts all sandbox files into the current directory. By default, `cabal sandbox init` initialises a new sandbox in a newly-created subdirectory of the current working directory (`./.cabal-sandbox`).

Using multiple different compiler versions simultaneously is also supported, via the `-w` option:

```
$ cabal sandbox init
$ cabal install --only-dependencies -w /path/to/ghc-1 # Install dependencies for both compilers
$ cabal install --only-dependencies -w /path/to/ghc-2
$ cabal configure -w /path/to/ghc-1                   # Build with the first compiler
$ cabal build
$ cabal configure -w /path/to/ghc-2                   # Build with the second compiler
$ cabal build
```

It can be occasionally useful to run the compiler-specific package manager tool (e.g. `ghc-pkg`) tool on the sandbox package DB directly (for example, you may need to unregister some packages). The `cabal sandbox hc-pkg` command is a convenient wrapper that runs the compiler-specific package manager tool with the arguments:

```
$ cabal -v sandbox hc-pkg list
Using a sandbox located at /path/to/.cabal-sandbox
'ghc-pkg' '--global' '--no-user-package-conf'
    '--package-conf=/path/to/.cabal-sandbox/i386-linux-ghc-7.4.2-packages.conf.d'
    'list'
[...]
```

The `--require-sandbox` option makes all sandbox-aware commands (`install`/`build`/etc.) exit with error if there is no sandbox present. This makes it harder to accidentally modify the user package database. The option can be also turned on via the per-user configuration file (`~/.cabal/config`) or the per-project one (`$PROJECT_DIR/cabal.config`). The error can be squelched with `--no-require-sandbox`.

The option `--sandbox-config-file` allows to specify the location of the `cabal.sandbox.config` file (by default, `cabal` searches for it in the current directory). This provides the same functionality as shared sandboxes, but sometimes can be more convenient. Example:

```
$ mkdir my/sandbox
$ cd my/sandbox
$ cabal sandbox init
$ cd /path/to/my/project
$ cabal --sandbox-config-file=/path/to/my/sandbox/cabal.sandbox.config install
# Uses the sandbox located at /path/to/my/sandbox/.cabal-sandbox
$ cd ~
$ cabal --sandbox-config-file=/path/to/my/sandbox/cabal.sandbox.config install
# Still uses the same sandbox
```

The sandbox config file can be also specified via the `CABAL_SANDBOX_CONFIG` environment variable.

Finally, the flag `--ignore-sandbox` lets you temporarily ignore an existing sandbox:

```
$ mkdir my/sandbox
$ cd my/sandbox
$ cabal sandbox init
$ cabal --ignore-sandbox install text
# Installs 'text' in the user package database ('~/.cabal').
```

Creating a binary package

When creating binary packages (e.g. for Red Hat or Debian) one needs to create a tarball that can be sent to another system for unpacking in the root directory:

```
runhaskell Setup.hs configure --prefix=/usr
runhaskell Setup.hs build
runhaskell Setup.hs copy --destdir=/tmp/mypkg
tar -czf mypkg.tar.gz /tmp/mypkg/
```

If the package contains a library, you need two additional steps:

```
runhaskell Setup.hs register --gen-script
runhaskell Setup.hs unregister --gen-script
```

This creates shell scripts `register.sh` and `unregister.sh`, which must also be sent to the target system. After unpacking there, the package must be registered by running the `register.sh` script. The `unregister.sh` script would be used in the uninstall procedure of the package. Similar steps may be used for creating binary packages for Windows.

The following options are understood by all commands:

`--help`, `-h` or `-?`

List the available options for the command.

`--verbose=`*n* or `-v`*n*

Set the verbosity level (0-3). The normal level is 1; a missing *n* defaults to 2.

The various commands and the additional options they support are described below. In the simple build infrastructure, any other options will be reported as errors.

setup configure

Prepare to build the package. Typically, this step checks that the target platform is capable of building the package, and discovers platform-specific features that are needed during the build.

The user may also adjust the behaviour of later stages using the options listed in the following subsections. In the simple build infrastructure, the values supplied via these options are recorded in a private file read by later stages.

If a user-supplied `configure` script is run (see the section on system-dependent parameters or on complex packages), it is passed the `--with-hc-pkg`, `--prefix`, `--bindir`, `--libdir`, `--datadir`, `--libexecdir` and `--sysconfdir` options. In addition the value of the

`--with-compiler` option is passed in a `--with-hc` option and all options specified with `--configure-option=` are passed on.

Programs used for building

The following options govern the programs used to process the source files of a package:

`--ghc` or `-g`, `--jhc`, `--lhc`, `--uhc`

Specify which Haskell implementation to use to build the package. At most one of these flags may be given. If none is given, the implementation under which the setup script was compiled or interpreted is used.

`--with-compiler=`*path* or `-w`*path*

Specify the path to a particular compiler. If given, this must match the implementation selected above. The default is to search for the usual name of the selected implementation.

This flag also sets the default value of the `--with-hc-pkg` option to the package tool for this compiler. Check the output of `setup configure -v` to ensure that it finds the right package tool (or use `--with-hc-pkg` explicitly).

`--with-hc-pkg=`*path*

Specify the path to the package tool, e.g. `ghc-pkg`. The package tool must be compatible with the compiler specified by `--with-compiler`. If this option is omitted, the default value is determined from the compiler selected.

`--with-`*prog*`=`*path*

Specify the path to the program *prog*. Any program known to Cabal can be used in place of *prog*. It can either be a fully path or the name of a program that can be found on the program search path. For example: `--with-ghc=ghc-6.6.1` or `--with-cpphs=/usr/local/bin/cpphs`. The full list of accepted programs is not enumerated in this user guide. Rather, run `cabal install --help` to view the list.

`--`*prog*`-options=`*options*

Specify additional options to the program *prog*. Any program known to Cabal can be used in place of *prog*. For example: `--alex-options="--template=mytemplatedir/"`. The *options* is split into program options based on spaces. Any options containing embedded spaced need to be quoted, for example `--foo-options='--bar="C:\Program File\Bar"'`. As an alternative that takes only one option at a time but avoids the need to quote, use `--`*prog*`-option` instead.

`--`*prog*`-option=`*option*

Specify a single additional option to the program *prog*. For passing an option that contain embedded spaces, such as a file name with embedded spaces, using this rather than `--`*`prog`*`-options` means you do not need an additional level of quoting. Of course if you are using a command shell you may still need to quote, for example `--foo-options="--bar=C:\Program File\Bar"`.

All of the options passed with either `--`*`prog`*`-options` or `--`*`prog`*`-option` are passed in the order they were specified on the configure command line.

Installation paths

The following options govern the location of installed files from a package:

`--prefix=`*dir*

The root of the installation. For example for a global install you might use `/usr/local` on a Unix system, or `C:\Program Files` on a Windows system. The other installation paths are usually subdirectories of *prefix*, but they don't have to be.

In the simple build system, *dir* may contain the following path variables: `$pkgid`, `$pkg`, `$version`, `$compiler`, `$os`, `$arch`, `$abi`, `$abitag`

`--bindir=`*dir*

Executables that the user might invoke are installed here.

In the simple build system, *dir* may contain the following path variables: `$prefix`, `$pkgid`, `$pkg`, `$version`, `$compiler`, `$os`, `$arch`, `$abi`, `$abitag

`--libdir=`*dir*

Object-code libraries are installed here.

In the simple build system, *dir* may contain the following path variables: `$prefix`, `$bindir`, `$pkgid`, `$pkg`, `$version`, `$compiler`, `$os`, `$arch`, `$abi`, `$abitag`

`--libexecdir=`*dir*

Executables that are not expected to be invoked directly by the user are installed here.

In the simple build system, *dir* may contain the following path variables: `$prefix`, `$bindir`, `$libdir`, `$libsubdir`, `$pkgid`, `$pkg`, `$version`, `$compiler`, `$os`, `$arch`, `$abi`, `$abitag`

`--datadir`=*dir*

Architecture-independent data files are installed here.

In the simple build system, *dir* may contain the following path variables: `$prefix`, `$bindir`, `$libdir`, `$libsubdir`, `$pkgid`, `$pkg`, `$version`, `$compiler`, `$os`, `$arch`, `$abi`,

`$abitag`

`--sysconfdir=`*dir*

Installation directory for the configuration files.

In the simple build system, *dir* may contain the following path variables: `$prefix`, `$bindir`, `$libdir`, `$libsubdir`, `$pkgid`, `$pkg`, `$version`, `$compiler`, `$os`, `$arch`, `$abi`, `$abitag`

In addition the simple build system supports the following installation path options:

`--libsubdir=`*dir*

A subdirectory of *libdir* in which libraries are actually installed. For example, in the simple build system on Unix, the default *libdir* is `/usr/local/lib`, and *libsubdir* contains the package identifier and compiler, e.g. `mypkg-0.2/ghc-6.4`, so libraries would be installed in `/usr/local/lib/mypkg-0.2/ghc-6.4`.

dir may contain the following path variables: `$pkgid`, `$pkg`, `$version`, `$compiler`, `$os`, `$arch`, `$abi`, `$abitag`

`--datasubdir=`*dir*

A subdirectory of *datadir* in which data files are actually installed.

dir may contain the following path variables: `$pkgid`, `$pkg`, `$version`, `$compiler`, `$os`, `$arch`, `$abi`, `$abitag`

`--docdir=`*dir*

Documentation files are installed relative to this directory.

dir may contain the following path variables: `$prefix`, `$bindir`, `$libdir`, `$libsubdir`, `$datadir`, `$datasubdir`, `$pkgid`, `$pkg`, `$version`, `$compiler`, `$os`, `$arch`, `$abi`, `$abitag`

`--htmldir=`*dir*

HTML documentation files are installed relative to this directory.

dir may contain the following path variables: `$prefix`, `$bindir`, `$libdir`, `$libsubdir`, `$datadir`, `$datasubdir`, `$docdir`, `$pkgid`, `$pkg`, `$version`, `$compiler`, `$os`, `$arch`, `$abi`, `$abitag`

`--program-prefix=`*prefix*

Prepend *prefix* to installed program names.

prefix may contain the following path variables: `$pkgid`, `$pkg`, `$version`, `$compiler`, `$os`, `$arch`, `$abi`, `$abitag`

`--program-suffix=`*suffix*

Append *suffix* to installed program names. The most obvious use for this is to append the program's version number to make it possible to install several versions of a program at once: `--program-suffix='$version'`.

suffix may contain the following path variables: `$pkgid`, `$pkg`, `$version`, `$compiler`, `$os`, `$arch`, `$abi`, `$abitag`

Path variables in the simple build system

For the simple build system, there are a number of variables that can be used when specifying installation paths. The defaults are also specified in terms of these variables. A number of the variables are actually for other paths, like `$prefix`. This allows paths to be specified relative to each other rather than as absolute paths, which is important for building relocatable packages (see prefix independence).

`$prefix`

The path variable that stands for the root of the installation. For an installation to be relocatable, all other installation paths must be relative to the `$prefix` variable.

`$bindir`

The path variable that expands to the path given by the `--bindir` configure option (or the default).

`$libdir`

As above but for `--libdir`

`$libsubdir`

As above but for `--libsubdir`

`$datadir`

As above but for `--datadir`

`$datasubdir`

As above but for `--datasubdir`

`$docdir`

As above but for `--docdir`

`$pkgid`

The name and version of the package, e.g. `mypkg-0.2`

`$pkg`

The name of the package, e.g. `mypkg`

`$version`

The version of the package, e.g. `0.2`

`$compiler`

The compiler being used to build the package, e.g. `ghc-6.6.1`

`$os`

The operating system of the computer being used to build the package, e.g. `linux`, `windows`, `osx`, `freebsd` or `solaris`

`$arch`

The architecture of the computer being used to build the package, e.g. `i386`, `x86_64`, `ppc` or `sparc`

`$abitag`

An optional tag that a compiler can use for telling incompatible ABI's on the same architecture apart. GHCJS encodes the underlying GHC version in the ABI tag.

`$abi`

A shortcut for getting a path that completely identifies the platform in terms of binary compatibility. Expands to the same value as `$arch-$os-compiler-$abitag` if the compiler uses an abi tag, `$arch-$os-$compiler` if it doesn't.

Paths in the simple build system

For the simple build system, the following defaults apply:

Option	Windows Default	Unix Default
`--prefix` (global)	`C:\Program Files\Haskell`	`/usr/local`
`--prefix` (per-user)	`C:\Documents And Settings\user\Application Data\cabal`	`$HOME/.cabal`
`--bindir`	`$prefix\bin`	`$prefix/bin`
`--libdir`	`$prefix`	`$prefix/lib`
`--libsubdir` (others)	`$pkgid\$compiler`	`$pkgid/$compiler`
`--libexecdir`	`$prefix\$pkgid`	`$prefix/libexec`
`--datadir` (executable)	`$prefix`	`$prefix/share`
`--datadir` (library)	`C:\Program Files\Haskell`	`$prefix/share`
`--datasubdir`	`$pkgid`	`$pkgid`
`--docdir`	`$prefix\doc\$pkgid`	`$datadir/doc/$pkgid`
`--sysconfdir`	`$prefix\etc`	`$prefix/etc`

Option	Windows Default	Unix Default
`--htmldir`	`$docdir\html`	`$docdir/html`
`--program-prefix`	(empty)	(empty)
`--program-suffix`	(empty)	(empty)

Prefix-independence

On Windows it is possible to obtain the pathname of the running program. This means that we can construct an installable executable package that is independent of its absolute install location. The executable can find its auxiliary files by finding its own path and knowing the location of the other files relative to `$bindir`. Prefix-independence is particularly useful: it means the user can choose the install location (i.e. the value of `$prefix`) at install-time, rather than having to bake the path into the binary when it is built.

In order to achieve this, we require that for an executable on Windows, all of `$bindir`, `$libdir`, `$datadir` and `$libexecdir` begin with `$prefix`. If this is not the case then the compiled executable will have baked-in all absolute paths.

The application need do nothing special to achieve prefix-independence. If it finds any files using `getDataFileName` and the other functions provided for the purpose, the files will be accessed relative to the location of the current executable.

A library cannot (currently) be prefix-independent, because it will be linked into an executable whose file system location bears no relation to the library package.

Controlling Flag Assignments

Flag assignments (see the resolution of conditions and flags) can be controlled with the following command line options.

`-f` *flagname* or `-f -`*flagname*

> Force the specified flag to `true` or `false` (if preceded with a `-`). Later specifications for the same flags will override earlier, i.e., specifying `-fdebug -f-debug` is equivalent to `-f-debug`

`--flags=`*flagspecs*

> Same as `-f`, but allows specifying multiple flag assignments at once. The parameter is a space-separated list of flag names (to force a flag to `true`), optionally preceded by a `-` (to force a flag to `false`). For example, `--flags="debug -feature1 feature2"` is equivalent to `-fdebug -f-feature1 -ffeature2`.

Building Test Suites

`--enable-tests`

> Build the test suites defined in the package description file during the `build`

stage. Check for dependencies required by the test suites. If the package is configured with this option, it will be possible to run the test suites with the `test` command after the package is built.

`--disable-tests`

(default) Do not build any test suites during the `build` stage. Do not check for dependencies required only by the test suites. It will not be possible to invoke the `test` command without reconfiguring the package.

`--enable-coverage`

Build libraries and executables (including test suites) with Haskell Program Coverage enabled. Running the test suites will automatically generate coverage reports with HPC.

`--disable-coverage`

(default) Do not enable Haskell Program Coverage.

Miscellaneous options

`--user`

Does a per-user installation. This changes the default installation prefix. It also allow dependencies to be satisfied by the user's package database, in addition to the global database. This also implies a default of `--user` for any subsequent `install` command, as packages registered in the global database should not depend on packages registered in a user's database.

`--global`

(default) Does a global installation. In this case package dependencies must be satisfied by the global package database. All packages in the user's package database will be ignored. Typically the final installation step will require administrative privileges.

`--package-db=`*db*

Allows package dependencies to be satisfied from this additional package database *db* in addition to the global package database. All packages in the user's package database will be ignored. The interpretation of *db* is implementation-specific. Typically it will be a file or directory. Not all implementations support arbitrary package databases.

`--enable-optimization`[=*n*] or `-O`[*n*]

(default) Build with optimization flags (if available). This is appropriate for production use, taking more time to build faster libraries and programs.

The optional *n* value is the optimisation level. Some compilers support multiple optimisation levels. The range is 0 to 2. Level 0 is equivalent to

`--disable-optimization`, level 1 is the default if no *n* parameter is given. Level 2 is higher optimisation if the compiler supports it. Level 2 is likely to lead to longer compile times and bigger generated code.

`--disable-optimization`

Build without optimization. This is suited for development: building will be quicker, but the resulting library or programs will be slower.

`--enable-library-profiling` or `-p`

Request that an additional version of the library with profiling features enabled be built and installed (only for implementations that support profiling).

`--disable-library-profiling`

(default) Do not generate an additional profiling version of the library.

`--enable-profiling`

Any executables generated should have profiling enabled (only for implementations that support profiling). For this to work, all libraries used by these executables must also have been built with profiling support. The library will be built with profiling enabled (if supported) unless `--disable-library-profiling` is specified.

`--disable-profiling`

(default) Do not enable profiling in generated executables.

`--enable-library-vanilla`

(default) Build ordinary libraries (as opposed to profiling libraries). This is independent of the `--enable-library-profiling` option. If you enable both, you get both.

`--disable-library-vanilla`

Do not build ordinary libraries. This is useful in conjunction with `--enable-library-profiling` to build only profiling libraries, rather than profiling and ordinary libraries.

`--enable-library-for-ghci`

(default) Build libraries suitable for use with GHCi.

`--disable-library-for-ghci`

Not all platforms support GHCi and indeed on some platforms, trying to build GHCi libs fails. In such cases this flag can be used as a workaround.

`--enable-split-objs`

Use the GHC `-split-objs` feature when building the library. This reduces the final size of the executables that use the library by allowing them to link with only the bits that they use rather than the entire library. The downside is that building the library takes longer and uses considerably more memory.

`--disable-split-objs`

(default) Do not use the GHC `-split-objs` feature. This makes building the library quicker but the final executables that use the library will be larger.

`--enable-executable-stripping`

(default) When installing binary executable programs, run the `strip` program on the binary. This can considerably reduce the size of the executable binary file. It does this by removing debugging information and symbols. While such extra information is useful for debugging C programs with traditional debuggers it is rarely helpful for debugging binaries produced by Haskell compilers.

Not all Haskell implementations generate native binaries. For such implementations this option has no effect.

`--disable-executable-stripping`

Do not strip binary executables during installation. You might want to use this option if you need to debug a program using gdb, for example if you want to debug the C parts of a program containing both Haskell and C code. Another reason is if your are building a package for a system which has a policy of managing the stripping itself (such as some Linux distributions).

`--enable-shared`

Build shared library. This implies a separate compiler run to generate position independent code as required on most platforms.

`--disable-shared`

(default) Do not build shared library.

`--enable-executable-dynamic`

Link executables dynamically. The executable’s library dependencies should be built as shared objects. This implies `--enable-shared` unless `--disable-shared` is explicitly specified.

`--disable-executable-dynamic`

(default) Link executables statically.

`--configure-option=`*str*

An extra option to an external `configure` script, if one is used (see the section on system-dependent parameters). There can be several of these options.

`--extra-include-dirs`[=*dir*]

An extra directory to search for C header files. You can use this flag multiple times to get a list of directories.

You might need to use this flag if you have standard system header files in a non-standard location that is not mentioned in the package's `.cabal` file. Using this option has the same affect as appending the directory *dir* to the `include-dirs` field in each library and executable in the package's `.cabal` file. The advantage of course is that you do not have to modify the package at all. These extra directories will be used while building the package and for libraries it is also saved in the package registration information and used when compiling modules that use the library.

`--extra-lib-dirs`[=*dir*]

An extra directory to search for system libraries files. You can use this flag multiple times to get a list of directories.

You might need to use this flag if you have standard system libraries in a non-standard location that is not mentioned in the package's `.cabal` file. Using this option has the same affect as appending the directory *dir* to the `extra-lib-dirs` field in each library and executable in the package's `.cabal` file. The advantage of course is that you do not have to modify the package at all. These extra directories will be used while building the package and for libraries it is also saved in the package registration information and used when compiling modules that use the library.

`--allow-newer`[=*pkgs*]

Selectively relax upper bounds in dependencies without editing the package description.

If you want to install a package A that depends on B >= 1.0 && < 2.0, but you have the version 2.0 of B installed, you can compile A against B 2.0 by using `cabal install --allow-newer=B A`. This works for the whole package index: if A also depends on C that in turn depends on B < 2.0, C's dependency on B will be also relaxed.

Example:

```
$ cd foo
$ cabal configure
Resolving dependencies...
cabal: Could not resolve dependencies:
[...]
$ cabal configure --allow-newer
Resolving dependencies...
Configuring foo...
```

Additional examples:

```
# Relax upper bounds in all dependencies.
$ cabal install --allow-newer foo

# Relax upper bounds only in dependencies on bar, baz and quux.
$ cabal install --allow-newer=bar,baz,quux foo

# Relax the upper bound on bar and force bar==2.1.
$ cabal install --allow-newer=bar --constraint="bar==2.1" foo
```

It's also possible to enable `--allow-newer` permanently by setting `allow-newer: True` in the `~/.cabal/config` file.

`--constraint=`*constraint*

Restrict solutions involving a package to a given version range. For example, `cabal install --constraint="bar==2.1"` will only consider install plans that do not use `bar` at all, or `bar` of version 2.1.

As a special case, `cabal install --constraint="bar -none"` prevents `bar` from being used at all (`-none` abbreviates `> 1 && < 1`); `cabal install --constraint="bar installed"` prevents reinstallation of the `bar` package; `cabal install --constraint="bar +foo -baz"` specifies that the flag `foo` should be turned on and the `baz` flag should be turned off.

setup build

Perform any preprocessing or compilation needed to make this package ready for installation.

This command takes the following options:

–*prog*-options=*options*, –*prog*-option=*option*
These are mostly the same as the options configure step. Unlike the options specified at the configure step, any program options specified at the build step are not persistent but are used for that invocation only. They options specified at the build step are in addition not in replacement of any options specified at the configure step.

setup haddock

Build the documentation for the package using haddock. By default, only the documentation for the exposed modules is generated (but see the `--executables` and `--internal` flags below).

This command takes the following options:

`--hoogle`

Generate a file `dist/doc/html/`*pkgid*`.txt`, which can be converted by Hoogle into a database for searching. This is equivalent to running haddock with the `--hoogle` flag.

`--html-location=`*url*

Specify a template for the location of HTML documentation for prerequisite packages. The substitutions (see listing) are applied to the template to obtain a location for each package, which will be used by hyperlinks in the generated documentation. For example, the following command generates links pointing at Hackage pages:

> setup haddock –html-location=‘http://hackage.haskell.org/packages /archive/$pkg/latest/doc/html’

Here the argument is quoted to prevent substitution by the shell. If this option is omitted, the location for each package is obtained using the package tool (e.g. `ghc-pkg`).

`--executables`

Also run haddock for the modules of all the executable programs. By default haddock is run only on the exported modules.

`--internal`

Run haddock for the all modules, including unexposed ones, and make haddock generate documentation for unexported symbols as well.

`--css=`*path*

The argument *path* denotes a CSS file, which is passed to haddock and used to set the style of the generated documentation. This is only needed to override the default style that haddock uses.

`--hyperlink-source`

Generate haddock documentation integrated with HsColour. First, HsColour is run to generate colourised code. Then haddock is run to generate HTML documentation. Each entity shown in the documentation is linked to its definition in the colourised code.

`--hscolour-css=`*path*

The argument *path* denotes a CSS file, which is passed to HsColour as in

> runhaskell Setup.hs hscolour –css=*path*

setup hscolour

Produce colourised code in HTML format using HsColour. Colourised code for exported modules is put in `dist/doc/html/`*pkgid*`/src`.

This command takes the following options:

`--executables`

Also run HsColour on the sources of all executable programs. Colourised code is put in `dist/doc/html/`*pkgid*/*executable*`/src`.

`--css=`*path*

Use the given CSS file for the generated HTML files. The CSS file defines the colours used to colourise code. Note that this copies the given CSS file to the directory with the generated HTML files (renamed to `hscolour.css`) rather than linking to it.

setup install

Copy the files into the install locations and (for library packages) register the package with the compiler, i.e. make the modules it contains available to programs.

The install locations are determined by options to `setup configure`.

This command takes the following options:

`--global`

Register this package in the system-wide database. (This is the default, unless the `--user` option was supplied to the `configure` command.)

`--user`

Register this package in the user's local package database. (This is the default if the `--user` option was supplied to the `configure` command.)

setup copy

Copy the files without registering them. This command is mainly of use to those creating binary packages.

This command takes the following option:

`--destdir=`*path*

Specify the directory under which to place installed files. If this is not given, then the root directory is assumed.

setup register

Register this package with the compiler, i.e. make the modules it contains available to programs. This only makes sense for library packages. Note that the `install` command incorporates this action. The main use of this separate command is in the post-installation step for a binary package.

This command takes the following options:

`--global`

Register this package in the system-wide database. (This is the default.)

`--user`

Register this package in the user's local package database.

`--gen-script`

Instead of registering the package, generate a script containing commands to perform the registration. On Unix, this file is called `register.sh`, on Windows, `register.bat`. This script might be included in a binary bundle, to be run after the bundle is unpacked on the target system.

`--gen-pkg-config`[=*path*]

Instead of registering the package, generate a package registration file. This only applies to compilers that support package registration files which at the moment is only GHC. The file should be used with the compiler's mechanism for registering packages. This option is mainly intended for packaging systems. If possible use the `--gen-script` option instead since it is more portable across Haskell implementations. The *path* is optional and can be used to specify a particular output file to generate. Otherwise, by default the file is the package name and version with a `.conf` extension.

`--inplace`

Registers the package for use directly from the build tree, without needing to install it. This can be useful for testing: there's no need to install the package after modifying it, just recompile and test.

This flag does not create a build-tree-local package database. It still registers the package in one of the user or global databases.

However, there are some caveats. It only works with GHC (currently). It only works if your package doesn't depend on having any supplemental files installed — plain Haskell libraries should be fine.

setup unregister

Deregister this package with the compiler.

This command takes the following options:

`--global`

Deregister this package in the system-wide database. (This is the default.)

`--user`

Deregister this package in the user's local package database.

`--gen-script`

Instead of deregistering the package, generate a script containing commands to perform the deregistration. On Unix, this file is called `unregister.sh`, on Windows, `unregister.bat`. This script might be included in a binary bundle, to be run on the target system.

setup clean

Remove any local files created during the `configure`, `build`, `haddock`, `register` or `unregister` steps, and also any files and directories listed in the `extra-tmp-files` field.

This command takes the following options:

`--save-configure` or `-s`
: Keeps the configuration information so it is not necessary to run the configure step again before building.

setup test

Run the test suites specified in the package description file. Aside from the following flags, Cabal accepts the name of one or more test suites on the command line after `test`. When supplied, Cabal will run only the named test suites, otherwise, Cabal will run all test suites in the package.

`--builddir=`*dir*

The directory where Cabal puts generated build files (default: `dist`). Test logs will be located in the `test` subdirectory.

`--human-log=`*path*

The template used to name human-readable test logs; the path is relative to `dist/test`. By default, logs are named according to the template `$pkgid-$test-suite.log`, so that each test suite will be logged to its own human-readable log file. Template variables allowed are: `$pkgid`, `$compiler`, `$os`, `$arch`, `$abi`, `$abitag`, `$test-suite`, and `$result`.

`--machine-log=`*path*

The path to the machine-readable log, relative to `dist/test`. The default template is `$pkgid.log`. Template variables allowed are: `$pkgid`, `$compiler`, `$os`, `$arch`, `$abi`, `$abitag` and `$result`.

`--show-details=`*filter*

Determines if the results of individual test cases are shown on the terminal. May be `always` (always show), `never` (never show), `failures` (show only failed results), or `streaming` (show all results in real time).

`--test-options=`*options*

Give extra options to the test executables.

`--test-option=`*option*

give an extra option to the test executables. There is no need to quote options containing spaces because a single option is assumed, so options will not be split on spaces.

setup sdist

Create a system- and compiler-independent source distribution in a file *package-version*`.tar.gz` in the `dist` subdirectory, for distribution to package builders. When unpacked, the commands listed in this section will be available.

The files placed in this distribution are the package description file, the setup script, the sources of the modules named in the package description file, and files named in the `license-file`, `main-is`, `c-sources`, `js-sources`, `data-files`, `extra-source-files` and `extra-doc-files` fields.

This command takes the following option:

`--snapshot`

Append today's date (in "YYYYMMDD" format) to the version number for the generated source package. The original package is unaffected.