Common Architecture for Building Applications and Libraries

User’s Guide

COLLABORATORS

	TITLE : Common Architecture for Building Applications and Libraries		*REFERENCE* :
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 14, 2009	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Abstract

Cabal aims to simplify the distribution of Haskell software. It does this by specifying a number of interfaces between package authors, builders and users, as well as providing a library implementing these interfaces.

User's Guide

1 Introduction

Developers write Cabal packages. These can be for libraries or executables. This involves writing the code obviously and also creating a `.cabal` file. The .cabal file contains some information about the package. Some of this information is needed to actually build the package and some is just useful for identifying the package when it comes to distribution.

```
name:     Foo
version:  1.0

library
  build-depends:   base
  exposed-modules: Foo
```

Users install Cabal packages so they can use them. It is not expected that users will have to modify any of the information in the `.cabal` file. Cabal does provide a number of ways for a user to customise how and where a package is installed. They can decide where a package will be installed, which Haskell implementation to use and whether to build optimised code or build with the ability to profile code.

```
tar -xzf Foo-1.0.tar.gz
cd Foo-1.0
runhaskell Setup configure --with-compiler=ghc-6.4.2 --user
runhaskell Setup build
runhaskell Setup install
```

One of the purposes of Cabal is to make it easier to build a package with different Haskell implementations. So it provides abstractions of features present in different Haskell implementations and wherever possible it is best to take advantage of these to increase portability. Where necessary however it is possible to use specific features of specific implementations. For example one of the pieces of information a package author can put in the package's `.cabal` file is what language extensions the code uses. This is far preferable to specifying flags for a specific compiler as it allows Cabal to pick the right flags for the Haskell implementation that the user picks. It also allows Cabal to figure out if the language extension is even supported by the Haskell implementation that the user picks. Where compiler-specific options are needed however, there is an "escape hatch" available. The developer can specify implementation-specific options and more generally there is a configuration mechanism to customise many aspects of how a package is built depending on the Haskell implementation, the Operating system, computer architecture and user-specified configuration flags.

```
name:     Foo
version:  1.0

library
  build-depends:   base
  exposed-modules: Foo
  extensions:    ForeignFunctionInterface
  ghc-options:     -Wall
  nhc98-options:   -K4m
  if os(windows)
    build-depends: Win32
```

2 Packages

A *package* is the unit of distribution for the Cabal. Its purpose, when installed, is to make available either or both of:

- A library, exposing a number of Haskell modules. A library may also contain *hidden* modules, which are used internally but not available to clients.[1]

[1] Hugs doesn't support module hiding.

- One or more Haskell programs.

However having both a library and executables in a package does not work very well; if the executables depend on the library, they must explicitly list all the modules they directly or indirectly import from that library.

Internally, the package may consist of much more than a bunch of Haskell modules: it may also have C source code and header files, source code meant for preprocessing, documentation, test cases, auxiliary tools etc.

A package is identified by a globally-unique *package name*, which consists of one or more alphanumeric words separated by hyphens. To avoid ambiguity, each of these words should contain at least one letter. Chaos will result if two distinct packages with the same name are installed on the same system, but there is not yet a mechanism for allocating these names. A particular version of the package is distinguished by a *version number*, consisting of a sequence of one or more integers separated by dots. These can be combined to form a single text string called the *package ID*, using a hyphen to separate the name from the version, e.g. ‘`HUnit-1.1`’.

Note
Packages are not part of the Haskell language; they simply populate the hierarchical space of module names. In GHC 6.6 and later a program may contain multiple modules with the same name if they come from separate packages; in all other current Haskell systems packages may not overlap in the modules they provide, including hidden modules.

3 Creating a package

Suppose you have a directory hierarchy containing the source files that make up your package. You will need to add two more files to the root directory of the package:

`package.cabal` a Unicode UTF-8 text file containing a package description (for details of the syntax of this file, see Section 3.1)

`Setup.hs` or `Setup.lhs` a single-module Haskell program to perform various setup tasks (with the interface described in Section 4). This module should import only modules that will be present in all Haskell implementations, including modules of the Cabal library. In most cases it will be trivial, calling on the Cabal library to do most of the work.

Once you have these, you can create a source bundle of this directory for distribution. Building of the package is discussed in Section 4.

Example 3.1 A package containing a simple library

The HUnit package contains a file `HUnit.cabal` containing:

```
Name:   HUnit
Version:  1.1.1
Cabal-Version:  >= 1.2
License:  BSD3
License-File: LICENSE
Author:   Dean Herington
Homepage: http://hunit.sourceforge.net/
Category: Testing
Synopsis: A unit testing framework for Haskell

Library
  Build-Depends:  base
  Exposed-modules:
    Test.HUnit.Base, Test.HUnit.Lang, Test.HUnit.Terminal,
    Test.HUnit.Text, Test.HUnit
  Extensions: CPP
```

and the following `Setup.hs`:

```
import Distribution.Simple
main = defaultMain
```

Example 3.2 A package containing executable programs

```
Name:            TestPackage
Version:         0.0
Cabal-Version:   >= 1.2
License:         BSD3
Author:          Angela Author
Synopsis:        Small package with two programs
Build-Type:      Simple

Executable program1
  Build-Depends:  HUnit
  Main-Is:        Main.hs
  Hs-Source-Dirs: prog1

Executable program2
  Main-Is:        Main.hs
  Build-Depends:  HUnit
  Hs-Source-Dirs: prog2
  Other-Modules:  Utils
```

with `Setup.hs` the same as above.

Example 3.3 A package containing a library and executable programs

```
Name:            TestPackage
Version:         0.0
Cabal-Version:   >= 1.2
License:         BSD3
Author:          Angela Author
Synopsis:        Package with library and two programs
Build-Type:      Simple

Library
  Build-Depends:   HUnit
  Exposed-Modules: A, B, C

Executable program1
  Main-Is:         Main.hs
  Hs-Source-Dirs:  prog1
  Other-Modules:   A, B

Executable program2
  Main-Is:         Main.hs
  Hs-Source-Dirs:  prog2
  Other-Modules:   A, C, Utils
```

with `Setup.hs` the same as above. Note that any library modules required (directly or indirectly) by an executable must be listed again.

The trivial setup script used in these examples uses the *simple build infrastructure* provided by the Cabal library (see Distribution.Simple). The simplicity lies in its interface rather that its implementation. It automatically handles preprocessing with standard preprocessors, and builds packages for all the Haskell implementations (except nhc98, for now).

The simple build infrastructure can also handle packages where building is governed by system-dependent parameters, if you specify a little more (see Section 3.3). A few packages require more elaborate solutions (see Section 3.5).

3.1 Package descriptions

The package description file must have a name ending in '.cabal'. It must be a Unicode text file encoded using valid UTF-8. There must be exactly one such file in the directory. The first part of the name is usually the package name, and some of the tools that operate on Cabal packages require this.

In the package description file, lines whose first non-whitespace characters are '--' are treated as comments and ignored.

This file should contain of a number global property descriptions and several sections.

- The global properties describe the package as a whole, such as name, license, author, etc. (see Section 3.1.1).
- Optionally, a number of *configuration flags* can be declared. These can be used to enable or disable certain features of a package. (see Section 3.1.5).
- The (optional) library section specifies the library properties (see Section 3.1.2) and relevant build information (see Section 3.1.4).
- Following is an arbitrary number of executable sections which describe an executable program and (see Section 3.1.3) relevant build information (see Section 3.1.4).

Each section consists of a number of property descriptions in the form of field/value pairs, with a syntax roughly like mail message headers.

- Case is not significant in field names, but is significant in field values.
- To continue a field value, indent the next line relative to the field name.
- Field names may be indented, but all field values in the same section must use the same indentation.
- Tabs are *not* allowed as indentation characters due to a missing standard interpretation of tab width.
- To get a blank line in a field value, use an indented '.'

The syntax of the value depends on the field. Field types include:

token*, *filename*, *directory Either a sequence of one or more non-space non-comma characters, or a quoted string in Haskell 98 lexical syntax. Unless otherwise stated, relative filenames and directories are interpreted from the package root directory.

freeform*, *URL*, *address An arbitrary, uninterpreted string.

identifier A letter followed by zero or more alphanumerics or underscores.

Modules and preprocessors
Haskell module names listed in the `exposed-modules` and `other-modules` fields may correspond to Haskell source files, i.e. with names ending in '.hs' or '.lhs', or to inputs for various Haskell preprocessors. The simple build infrastructure understands the extensions '.gc' (greencard), '.chs' (c2hs), '.hsc' (**hsc2hs**), '.y' and '.ly' (happy), '.x' (alex) and '.cpphs' (cpphs). When building, Cabal will automatically run the appropriate preprocessor and compile the Haskell module it produces.

Some fields take lists of values, which are optionally separated by commas, except for the `build-depends` field, where the commas are mandatory.

Some fields are marked as required. All others are optional, and unless otherwise specified have empty default values.

3.1.1 Package properties

These fields may occur in the first top-level properties section and describe the package as a whole:

name: *package-name* (required) The unique name of the package (see Section 2), without the version number.

version: *numbers* (required) The package version number, usually consisting of a sequence of natural numbers separated by dots.

cabal-version: *>, <=, etc. & numbers* The version of Cabal required for this package. Since, with Cabal version 1.2 the syntax of package descriptions has changed, this is now a required field. List the field early in your `.cabal` file so that it will appear as a syntax error before any others, since old versions of Cabal unfortunately do not recognize this field.

For compatibility, files written in the old syntax are still recognized. Thus if you don't require features introduced with or after Cabal version 1.2, you may write your package description file using the old syntax. Please consult the user's guide of that Cabal version for a description of that syntax.

build-type: *identifier* The type of build used by this package. Build types are the constructors of the BuildType type, defaulting to `Custom`. If this field is given a value other than `Custom`, some tools such as `cabal-install` will be able to build the package without using the setup script. So if you are just using the default `Setup.hs` then set the build type as `Simple`.

license: *identifier* (default: `AllRightsReserved`) The type of license under which this package is distributed. License names are the constants of the License type.

license-file: *filename* The name of a file containing the precise license for this package. It will be installed with the package.

copyright: *freeform* The content of a copyright notice, typically the name of the holder of the copyright on the package and the year(s) from which copyright is claimed.

For example: `Copyright:  (c) 2006-2007 Joe Bloggs`

author: *freeform* The original author of the package.

maintainer: *address* The current maintainer or maintainers of the package. This is an e-mail address to which users should send bug reports, feature requests and patches.

stability: *freeform* The stability level of the package, e.g. `alpha`, `experimental`, `provisional`, `stable`.

homepage: *URL* The package homepage.

bug-reports: *URL* The URL where users should direct bug reports. This would normally be either:

- A `mailto:` URL, eg for a person or a mailing list.
- An `http:` (or `https:`) URL for an online bug tracking system.

For example Cabal itself uses a web-based bug tracking system

```
bug-reports: http://hackage.haskell.org/trac/hackage/
```

package-url: *URL* The location of a source bundle for the package. The distribution should be a Cabal package.

synopsis: *freeform* A very short description of the package, for use in a table of packages. This is your headline, so keep it short (one line) but as informative as possible. Save space by not including the package name or saying it's written in Haskell.

description: *freeform* Description of the package. This may be several paragraphs, and should be aimed at a Haskell programmer who has never heard of your package before.

For library packages, this field is used as prologue text by **setup haddock** (see Section 4.4), and thus may contain the same markup as haddock documentation comments.

category: *freeform* A classification category for future use by the package catalogue *Hackage*. These categories have not yet been specified, but the upper levels of the module hierarchy make a good start.

tested-with: ***compiler list*** A list of compilers and versions against which the package has been tested (or at least built).

data-files: ***filename list*** A list of files to be installed for run-time use by the package. This is useful for packages that use a large amount of static data, such as tables of values or code templates. For details on how to find these files at run-time, see Section 3.2.

A limited form of * wildcards in file names, for example `data-files: images/*.png` matches all the `.png` files in the `images` directory.

The limitation is that * wildcards are only allowed in place of the file name, not in the directory name or file extension. In particular, wildcards do not include directories contents recursively. Furthermore, if a wildcard is used it must be used with an extension, so `data-files: data/*` is not allowed. When matching a wildcard plus extension, a file's full extension must match exactly, so `*.gz` matches `foo.gz` but not `foo.tar.gz`. A wildcard that does not match any files is an error.

The reason for providing only a very limited form of wildcard is to concisely express the common case of a large number of related files of the same file type without making it too easy to accidentally include unwanted files.

data-dir: ***directory*** The directory where Cabal looks for data files to install, relative to the source directory. By default, Cabal will look in the source directory itself.

extra-source-files: ***filename list*** A list of additional files to be included in source distributions built with **setup sdist** (see Section 4.12).

As with `data-files` it can use a limited form of * wildcards in file names.

extra-tmp-files: ***filename list*** A list of additional files or directories to be removed by **setup clean** (see Section 4.10). These would typically be additional files created by additional hooks, such as the scheme described in Section 3.3.

3.1.2 Library

The library section should contain the following fields:

exposed-modules: ***identifier list*** **(required if this package contains a library)** A list of modules added by this package.

exposed: ***boolean*** **(default: `True`)** Some Haskell compilers (notably GHC) support the notion of packages being 'exposed' or 'hidden' which means the modules they provide can be easily imported without always having to specify which package they come from. However this only works effectively if the modules provided by all exposed packages do not overlap (otherwise a module import would be ambiguous).

Almost all new libraries use hierarchical module names that do not clash, so it is very uncommon to have to use this field. However it may be necessary to set `exposed: False` for some old libraries that use a flat module namespace or where it is known that the exposed modules would clash with other common modules.

The library section may also contain build information fields (see Section 3.1.4).

3.1.3 Executables

Executable sections (if present) describe executable programs contained in the package and must have an argument after the section label, which defines the name of the executable. This is a freeform argument but may not contain spaces.

The executable may be described using the following fields, as well as build information fields (see Section 3.1.4).

main-is: ***filename*** **(required)** The name of the `.hs` or `.lhs` file containing the `Main` module. Note that it is the `.hs` filename that must be listed, even if that file is generated using a preprocessor. The source file must be relative to one of the directories listed in `hs-source-dirs`.

3.1.4 Build information

The following fields may be optionally present in a library or executable section, and give information for the building of the corresponding library or executable. See also Section 3.3 and Section 3.1.5 for a way to supply system-dependent values for these fields.

`build-depends:` ***`package list`*** A list of packages needed to build this one. Each package can be annotated with a version constraint.

Version constraints use the operators `==`, `>=`, `>`, `<`, `<=` and a version number. Multiple constraints can be combined using `&&` or `||`. If no version constraint is specified, any version is assumed to be acceptable. For example:

```
library
  build-depends:
    base >= 2,
    foo >= 1.2 && < 1.3,
    bar
```

Dependencies like `foo >= 1.2 && < 1.3` turn out to be very common because it is recommended practise for package versions to correspond to API versions. There is a special syntax to support this use:

```
build-depends: foo ==1.2.*
```

It is only syntactic sugar. It is exactly equivalent to `foo >= 1.2 && < 1.3`.

`other-modules:` ***`identifier list`*** A list of modules used by the component but not exposed to users. For a library component, these would be hidden modules of the library. For an executable, these would be auxiliary modules to be linked with the file named in the `main-is` field.

> **Note**
> Every module in the package *must* be listed in one of `other-modules`, `exposed-modules` or `main-is` fields.

`hs-source-dirs:` ***`directory list`*** **(default: ‘`.`’)** Root directories for the module hierarchy.

For backwards compatibility, the old variant `hs-source-dir` is also recognized.

`extensions:` ***`identifier list`*** A list of Haskell extensions used by every module. Extension names are the constructors of the Extension type. These determine corresponding compiler options. In particular, `CPP` specifies that Haskell source files are to be preprocessed with a C preprocessor.

Extensions used only by one module may be specified by placing a `LANGUAGE` pragma in the source file affected, e.g.:

```
{-# LANGUAGE CPP, MultiParamTypeClasses #-}
```

> **Note**
> GHC versions prior to 6.6 do not support the `LANGUAGE` pragma.

`build-tools:` ***`program list`*** A list of programs, possibly annotated with versions, needed to build this package, e.g. `c2hs > 0.15, cpphs`. If no version constraint is specified, any version is assumed to be acceptable.

`buildable:` ***`boolean`*** **(default: `True`)** Is the component buildable? Like some of the other fields below, this field is more useful with the slightly more elaborate form of the simple build infrastructure described in Section 3.3.

`ghc-options:` ***`token list`*** Additional options for GHC. You can often achieve the same effect using the `extensions` field, which is preferred.

Options required only by one module may be specified by placing an `OPTIONS_GHC` pragma in the source file affected.

`ghc-prof-options:` ***`token list`*** Additional options for GHC when the package is built with profiling enabled.

ghc-shared-options: ***token list*** Additional options for GHC when the package is built as shared library.

hugs-options: ***token list*** Additional options for Hugs. You can often achieve the same effect using the `extensions` field, which is preferred.

Options required only by one module may be specified by placing an `OPTIONS_HUGS` pragma in the source file affected.

nhc98-options: ***token list*** Additional options for nhc98. You can often achieve the same effect using the `extensions` field, which is preferred.

Options required only by one module may be specified by placing an `OPTIONS_NHC98` pragma in the source file affected.

Warning: Cabal does not currently support building libraries or executables with nhc98 anyway.

includes: ***filename list*** A list of header files to be included in any compilations via C. This field applies to both header files that are already installed on the system and to those coming with the package to be installed. These files typically contain function prototypes for foreign imports used by the package.

install-includes: ***filename list*** A list of header files from this package to be installed into `$libdir/includes` when the package is installed. Files listed in `install-includes:` should be found in relative to the top of the source tree or relative to one of the directories listed in `include-dirs`.

`install-includes` is typically used to name header files that contain prototypes for foreign imports used in Haskell code in this package, for which the C implementations are also provided with the package. Note that to include them when compiling the package itself, they need to be listed in the `includes:` field as well.

include-dirs: ***directory list*** A list of directories to search for header files, when preprocessing with **c2hs**, **hsc2hs**, **ffihugs**, **cpphs** or the C preprocessor, and also when compiling via C.

c-sources: ***filename list*** A list of C source files to be compiled and linked with the Haskell files.

If you use this field, you should also name the C files in `CFILES` pragmas in the Haskell source files that use them, e.g.:

```
{-# CFILES dir/file1.c dir/file2.c #-}
```

These are ignored by the compilers, but needed by Hugs.

extra-libraries: ***token list*** A list of extra libraries to link with.

extra-lib-dirs: ***directory list*** A list of directories to search for libraries.

cc-options: ***token list*** Command-line arguments to be passed to the C compiler. Since the arguments are compiler-dependent, this field is more useful with the setup described in Section 3.3.

ld-options: ***token list*** Command-line arguments to be passed to the linker. Since the arguments are compiler-dependent, this field is more useful with the setup described in Section 3.3.

pkgconfig-depends: ***package list*** A list of pkg-config packages, needed to build this package. They can be annotated with versions, e.g. `gtk+-2.0 >= 2.10, cairo >= 1.0`. If no version constraint is specified, any version is assumed to be acceptable. Cabal uses `pkg-config` to find if the packages are available on the system and to find the extra compilation and linker options needed to use the packages.

If you need to bind to a C library that supports `pkg-config` (use `pkg-config --list-all` to find out if it is supported) then it is much preferable to use this field rather than hard code options into the other fields.

frameworks: ***token list*** On Darwin/MacOS X, a list of frameworks to link to. See Apple's developer documentation for more details on frameworks. This entry is ignored on all other platforms.

3.1.5 Configurations

Library and executable sections may include conditional blocks, which test for various system parameters and configuration flags. The flags mechanism is rather generic, but most of the time a flag represents certain feature, that can be switched on or off by the package user.

Here is an example package description file using configurations:

Example 3.4 A package containing a library and executable programs

```
Name: Test1
Version: 0.0.1
Cabal-Version: >= 1.2
License: BSD3
Author:  Jane Doe
Synopsis: Test package to test configurations
Category: Example

Flag Debug
  Description: Enable debug support
  Default:     False

Flag WebFrontend
  Description: Include API for web frontend.
  -- Cabal checks if the configuration is possible, first
  -- with this flag set to True and if not it tries with False

Library
  Build-Depends:   base
  Exposed-Modules: Testing.Test1
  Extensions:      CPP

  if flag(debug)
    GHC-Options: -DDEBUG
    if !os(windows)
      CC-Options: "-DDEBUG"
    else
      CC-Options: "-DNDEBUG"

  if flag(webfrontend)
    Build-Depends: cgi > 0.42
    Other-Modules: Testing.WebStuff

Executable test1
  Main-is: T1.hs
  Other-Modules: Testing.Test1
  Build-Depends: base

  if flag(debug)
    CC-Options: "-DDEBUG"
    GHC-Options: -DDEBUG
```

3.1.5.1 Layout

Flags, conditionals, library and executable sections use layout to indicate structure. This is very similar to the Haskell layout rule. Entries in a section have to all be indented to the same level which must be more than the section header. Tabs are not allowed to be used for indentation.

As an alternative to using layout you can also use explicit braces `{}`. In this case the indentation of entries in a section does not matter, though different fields within a block must be on different lines. Here is a bit of the above example again, using braces:

Example 3.5 Using explicit braces rather than indentation for layout

```
Name: Test1
Version: 0.0.1
Cabal-Version: >= 1.2
License: BSD3
Author:  Jane Doe
Synopsis: Test package to test configurations
Category: Example

Flag Debug {
  Description: Enable debug support
  Default:     False
}

Library {
  Build-Depends:   base
  Exposed-Modules: Testing.Test1
  Extensions:      CPP
  if flag(debug) {
    GHC-Options: -DDEBUG
    if !os(windows) {
      CC-Options: "-DDEBUG"
    } else {
      CC-Options: "-DNDEBUG"
    }
  }
}
```

3.1.5.2 Configuration Flags

A flag section takes the flag name as an argument and may contain the following fields.

`description:` ***`freeform`*** The description of this flag.

`default:` ***`boolean`*** **(default: `True`)** The default value of this flag.

Note that this value may be overridden in several ways (see Section 4.1.3). The rationale for having flags default to True is that users usually want new features as soon as they are available. Flags representing features that are not (yet) recommended for most users (such as experimental features or debugging support) should therefore explicitly override the default to False.

3.1.5.3 Conditional Blocks

Conditional blocks may appear anywhere inside a library or executable section. They have to follow rather strict formatting rules. Conditional blocks must always be of the shape

```
if condition
     property-descriptions-or-conditionals*
```

or

```
if condition
     property-descriptions-or-conditionals*
else
     property-descriptions-or-conditionals*
```

Note that the `if` and the condition have to be all on the same line.

3.1.5.4 Conditions

Conditions can be formed using boolean tests and the boolean operators `||` (disjunction / logical "or"), `&&` (conjunction / logical "and"), or `!` (negation / logical "not"). The unary `!` takes highest precedence, `||` takes lowest. Precedence levels may be overridden through the use of parentheses. For example, `os(darwin) && !arch(i386) || os(freebsd)` is equivalent to `(os(darwin) && !(arch(i386))) || os(freebsd)`.

The following tests are currently supported.

`os(` *`name`* `)` Tests if the current operating system is *`name`*. The argument is tested against `System.Info.os` on the target system. There is unfortunately some disagreement between Haskell implementations about the standard values of `System.Info.os`. Cabal canonicalises it so that in particular `os(windows)` works on all implementations. If the canonicalised os names match, this test evaluates to true, otherwise false. The match is case-insensitive.

`arch(` *`name`* `)` Tests if the current architecture is *`name`*. The argument is matched against `System.Info.arch` on the target system. If the arch names match, this test evaluates to true, otherwise false. The match is case-insensitive.

`impl(` *`compiler`* `)` Tests for the configured Haskell implementation. An optional version constraint may be specified (for example `impl(ghc >= 6.6.1)`). If the configured implementation is of the right type and matches the version constraint, then this evaluates to true, otherwise false. The match is case-insensitive.

`flag(` *`name`* `)` Evaluates to the current assignment of the flag of the given name. Flag names are case insensitive. Testing for flags that have not been introduced with a flag section is an error.

`true` Constant value true.

`false` Constant value false.

3.1.5.5 Resolution of Conditions and Flags

If a package descriptions specifies configuration flags the package user can control these in several ways (see Section 4.1.3). If the user does not fix the value of a flag, Cabal will try to find a flag assignment in the following way.

- For each flag specified, it will assign its default value, evaluate all conditions with this flag assignment, and check if all dependencies can be satisfied. If this check succeeded, the package will be configured with those flag assignments.
- If dependencies were missing, the last flag (as by the order in which the flags were introduced in the package description) is tried with its alternative value and so on. This continues until either an assignment is found where all dependencies can be satisfied, or all possible flag assignments have been tried.

To put it another way, Cabal does a complete backtracking search to find a satisfiable package configuration. It is only the dependencies specified in the `build-depends` field in conditional blocks that determine if a particular flag assignment is satisfiable (`build-tools` are not considered). The order of the declaration and the default value of the flags determines the search order. Flags overridden on the command line fix the assignment of that flag, so no backtracking will be tried for that flag.

If no suitable flag assignment could be found, the configuration phase will fail and a list of missing dependencies will be printed. Note that this resolution process is exponential in the worst case (i.e., in the case where dependencies cannot be satisfied). There are some optimizations applied internally, but the overall complexity remains unchanged.

3.1.5.6 Meaning of field values when using conditionals

During the configuration phase, a flag assignment is chosen, all conditionals are evaluated, and the package description is combined into a flat package descriptions. If the same field both inside a conditional and outside then they are combined using the following rules.

- Boolean fields are combined using conjunction (logical "and").
- List fields are combined by appending the inner items to the outer items, for example

```
Extensions: CPP
if impl(ghc) || impl(hugs)
  Extensions: MultiParamTypeClasses
```

when compiled using Hugs or GHC will be combined to

```
Extensions: CPP, MultiParamTypeClasses
```

Similarly, if two conditional sections appear at the same nesting level, properties specified in the latter will come after properties specified in the former.

- All other fields must not be specified in ambiguous ways. For example

```
Main-is: Main.hs
if flag(useothermain)
  Main-is: OtherMain.hs
```

will lead to an error. Instead use

```
if flag(useothermain)
  Main-is: OtherMain.hs
else
  Main-is: Main.hs
```

3.1.6 Source Repositories

It is often useful to be able to specify a source revision control repository for a package. Cabal lets you specifying this information in a relatively structured form which enables other tools to interpret and make effective use of the information. For example the information should be sufficient for an automatic tool to checkout the sources.

Cabal supports specifying different information for various common source control systems. Obviously not all automated tools will support all source control systems.

Cabal supports specifying repositories for different use cases. By declaring which case we mean automated tools can be more useful. There are currently two kinds defined:

- The `head` kind refers to the latest development branch of the package. This may be used for example to track activity of a project or as an indication to outside developers what sources to get for making new contributions.
- The `this` kind refers to the branch and tag of a repository that contains the sources for this version or release of a package. For most source control systems this involves specifying a tag, id or hash of some form and perhaps a branch. The purpose is to be able to reconstruct the sources corresponding to a particular package version. This might be used to indicate what sources to get if someone needs to fix a bug in an older branch that is no longer an active head branch.

You can specify one kind or the other or both. As an example here are the repositories for the Cabal library. Note that the `this` kind of repo specifies a tag.

```
source-repository head
  type:     darcs
  location: http://darcs.haskell.org/cabal/

source-repository this
  type:     darcs
  location: http://darcs.haskell.org/cabal-branches/cabal-1.6/
  tag:      1.6.1
```

The exact fields are as follows:

`type:` ***`token`*** The name of the source control system used for this repository. The currently recognised types are:

- darcs
- git
- svn
- cvs
- mercurial (or alias hg)
- bazaar (or alias bzr)
- arch
- monotone

This field is required.

location: ***URL*** The location of the repository. The exact form of this field depends on the repository type. For example:

- for darcs: `http://code.haskell.org/foo/`
- for git: `git://github.com/foo/bar.git`
- for CVS: `anoncvs@cvs.foo.org:/cvs`

This field is required.

module: ***token*** CVS requires a named module, as each CVS server can host multiple named repositories.

This field is required for the CVS repo type and should not be used otherwise.

branch: ***token*** Many source control systems support the notion of a branch, as a distinct concept from having repositories in separate locations. For example CVS, SVN and git use branches while for darcs uses different locations for different branches. If you need to specify a branch to identify a your repository then specify it in this field.

This field is optional.

tag: ***token*** A tag identifies a particular state of a source repository. The tag can be used with a `this` repo kind to identify the state of a repo corresponding to a particular package version or release. The exact form of the tag depends on the repository type.

This field is required for the `this` repo kind.

subdir: ***directory*** Some projects put the sources for multiple packages under a single source repository. This field lets you specify the relative path from the root of the repository to the top directory for the package, ie the directory containing the package's `.cabal` file.

This field is optional. It default to empty which corresponds to the root directory of the repository.

3.2 Accessing data files from package code

The placement on the target system of files listed in the `data-files` field varies between systems, and in some cases one can even move packages around after installation (see Section 4.1.2.3). To enable packages to find these files in a portable way, Cabal generates a module called `Paths_`*`pkgname`* (with any hyphens in *`pkgname`* replaced by underscores) during building, so that it may be imported by modules of the package. This module defines a function

```
getDataFileName :: FilePath -> IO FilePath
```

If the argument is a filename listed in the `data-files` field, the result is the name of the corresponding file on the system on which the program is running.

Note

If you decide to import the `Paths_`*`pkgname`* module then it *must* be listed in the `other-modules` field just like any other module in your package.

The `Paths_`*`pkgname`* module is not platform independent so it does not get included in the source tarballs generated by **sdist**.

3.3 System-dependent parameters

For some packages, especially those interfacing with C libraries, implementation details and the build procedure depend on the build environment. A variant of the simple build infrastructure (the `build-type Configure`) handles many such situations using a slightly longer `Setup.hs`:

```
import Distribution.Simple
main = defaultMainWithHooks defaultUserHooks
```

This program differs from `defaultMain` in two ways:

1. If the package root directory contains a file called `configure`, the configure step will run that. This `configure` program may be a script produced by the autoconf system, or may be hand-written. This program typically discovers information about the system and records it for later steps, e.g. by generating system-dependent header files for inclusion in C source files and preprocessed Haskell source files. (Clearly this won't work for Windows without MSYS or Cygwin: other ideas are needed.)

2. If the package root directory contains a file called *`package`*`.buildinfo` after the configuration step, subsequent steps will read it to obtain additional settings for build information fields (see Section 3.1.4), to be merged with the ones given in the `.cabal` file. In particular, this file may be generated by the `configure` script mentioned above, allowing these settings to vary depending on the build environment.

   The build information file should have the following structure:

   ```
   buildinfo

   executable: name
   buildinfo

   executable: name
   buildinfo

   ...
   ```

   where each *`buildinfo`* consists of settings of fields listed in Section 3.1.4. The first one (if present) relates to the library, while each of the others relate to the named executable. (The names must match the package description, but you don't have to have entries for all of them.)

Neither of these files is required. If they are absent, this setup script is equivalent to `defaultMain`.

Example 3.6 Using autoconf

(This example is for people familiar with the autoconf tools.)
In the X11 package, the file `configure.ac` contains:

```
AC_INIT([Haskell X11 package], [1.1], [libraries@haskell.org], [X11])

# Safety check: Ensure that we are in the correct source directory.
AC_CONFIG_SRCDIR([X11.cabal])

# Header file to place defines in
AC_CONFIG_HEADERS([include/HsX11Config.h])

# Check for X11 include paths and libraries
AC_PATH_XTRA
AC_TRY_CPP([#include <X11/Xlib.h>],,[no_x=yes])

# Build the package if we found X11 stuff
if test "$no_x" = yes
then BUILD_PACKAGE_BOOL=False
else BUILD_PACKAGE_BOOL=True
fi
AC_SUBST([BUILD_PACKAGE_BOOL])

AC_CONFIG_FILES([X11.buildinfo])
AC_OUTPUT
```

Then the setup script will run the `configure` script, which checks for the presence of the X11 libraries and substitutes for variables in the file `X11.buildinfo.in`:

```
buildable: @BUILD_PACKAGE_BOOL@
cc-options: @X_CFLAGS@
ld-options: @X_LIBS@
```

This generates a file `X11.buildinfo` supplying the parameters needed by later stages:

```
buildable: True
cc-options:  -I/usr/X11R6/include
ld-options:  -L/usr/X11R6/lib
```

The `configure` script also generates a header file `include/HsX11Config.h` containing C preprocessor defines recording the results of various tests. This file may be included by C source files and preprocessed Haskell source files in the package.

Note
Packages using these features will also need to list additional files such as `configure`, templates for `.buildinfo` files, files named only in `.buildinfo` files, header files and so on in the `extra-source-files` field, to ensure that they are included in source distributions. They should also list files and directories generated by **configure** in the `extra-tmp-files` field to ensure that they are removed by **setup clean**.

3.4 Conditional compilation

Sometimes you want to write code that works with more than one version of a dependency. You can specify a range of versions for the depenency in the `build-depends`, but how do you then write the code that can use different versions of the API?

Haskell lets you preprocess your code using the C preprocessor (either the real C preprocessor, or `cpphs`). To enable this, add `extensions: CPP` to your package description. When using CPP, Cabal provides some pre-defined macros to let you test the version of dependent packages; for example, suppose your package works with either version 3 or version 4 of the `base` package, you could select the available version in your Haskell modules like this:

```
#if MIN_VERSION_base(4,0,0)
... code that works with base-4 ...
```

```
#else
... code that works with base-3 ...
#endif
```

In general, Cabal supplies a macro `MIN_VERSION_`*`package`*`(A,B,C)` for each package depended on via `build-depends`. This macro is true if the actual version of the package in use is greater than or equal to `A.B.C` (using the conventional ordering on version numbers, which is lexicographic on the sequence, but numeric on each component, so for example 1.2.0 is greater than 1.0.3).

Cabal places the definitions of these macros into an automatically-generated header file, which is included when preprocessing Haskell source code by passing options to the C preprocessor.

3.5 More complex packages

For packages that don't fit the simple schemes described above, you have a few options:

- You can customize the simple build infrastructure using *hooks*. These allow you to perform additional actions before and after each command is run, and also to specify additional preprocessors. See `UserHooks` in Distribution.Simple for the details, but note that this interface is experimental, and likely to change in future releases.
- You could delegate all the work to **make**, though this is unlikely to be very portable. Cabal supports this with the `build-type Make` and a trivial setup library Distribution.Make, which simply parses the command line arguments and invokes **make**. Here `Setup.hs` looks like

  ```
  import Distribution.Make
  main = defaultMain
  ```

  The root directory of the package should contain a `configure` script, and, after that has run, a `Makefile` with a default target that builds the package, plus targets `install`, `register`, `unregister`, `clean`, `dist` and `docs`. Some options to commands are passed through as follows:

  – The `--with-hc-pkg`, `--prefix`, `--bindir`, `--libdir`, `--datadir` and `--libexecdir` options to the `configure` command are passed on to the `configure` script. 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.

  – the `--destdir` option to the `copy` command becomes a setting of a `destdir` variable on the invocation of `make copy`. The supplied `Makefile` should provide a `copy` target, which will probably look like this:

  ```
  copy :
          $(MAKE) install prefix=$(destdir)/$(prefix) \
                          bindir=$(destdir)/$(bindir) \
                          libdir=$(destdir)/$(libdir) \
                          datadir=$(destdir)/$(datadir) \
                          libexecdir=$(destdir)/$(libexecdir)
  ```

- You can write your own setup script conforming to the interface of Section 4, possibly using the Cabal library for part of the work. One option is to copy the source of `Distribution.Simple`, and alter it for your needs. Good luck.

4 Building and installing a package

After you've unpacked a Cabal package, you can build it by moving into the root directory of the package and using the `Setup.hs` or `Setup.lhs` script there:

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

where *`runhaskell`* might be **runhugs**, **runghc** or **runnhc**. The *`command`* argument selects a particular step in the build/install process. You can also get a summary of the command syntax with

`runhaskell Setup.hs --help`

Example 4.1 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.

Example 4.2 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`).

Example 4.3 Creating a binary package

When creating binary packages (e.g. for RedHat 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.

4.1 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 Section 3.3 or Section 3.5), it is passed the `--with-hc-pkg`, `--prefix`, `--bindir`, `--libdir`, `--datadir` and `--libexecdir` 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.

4.1.1 Programs used for building

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

--ghc or **-g**, **--nhc**, **--jhc**, **--hugs** 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`.

--*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 embeded 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 embeded spaces, such as a file name with embeded 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.

4.1.2 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:\ProgramFiles` 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*

--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*

--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*

--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*

--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*

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*

--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*

--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*

--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*

--program-prefix=*prefix* Prepend *prefix* to installed program names.

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

--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*

4.1.2.1 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 Section 4.1.2.3).

$prefix The path variable that stands for the root of the installation.

For an installation to be relocatable, all other instllation 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, eg `mypkg-0.2`

$pkg The name of the package, eg `mypkg`

$version The version of the package, eg `0.2`

$compiler The compiler being used to build the package, eg `ghc-6.6.1`

$os The operating system of the computer being used to build the package, eg `linux`, `windows`, `osx`, `freebsd` or `solaris`

$arch The architecture of the computer being used to build the package, eg `i386`, `x86_64`, `ppc` or `sparc`

4.1.2.2 Paths in the simple build system

For the simple build system, the following defaults apply:

Option	Windows Default	Unix Default
`--prefix` (global installs with the `--global` flag)	`C:\ProgramFiles\Haskell`	`/usr/local`
`--prefix` (per-user installs with the `--user` flag)	`C:\DocumentsAndSettings\user\ApplicationData\cabal`	`$HOME/.cabal`
`--bindir`	`$prefix\bin`	`$prefix/bin`
`--libdir`	`$prefix`	`$prefix/lib`
`--libsubdir` (Hugs)	`hugs\packages\$pkg`	`hugs/packages/$pkg`
`--libsubdir` (others)	`$pkgid\$compiler`	`$pkgid/$compiler`
`--libexecdir`	`$prefix\$pkgid`	`$prefix/libexec`
`--datadir` (executable)	`$prefix`	`$prefix/share`
`--datadir` (library)	`C:\ProgramFiles\Haskell`	`$prefix/share`
`--datasubdir`	`$pkgid`	`$pkgid`
`--docdir`	`$prefix\doc\$pkgid`	`$datadir/doc/$pkgid`
`--htmldir`	`$docdir\html`	`$docdir/html`
`--program-prefix`	(empty)	(empty)
`--program-suffix`	(empty)	(empty)

4.1.2.3 Prefix-independence

On Windows, and when using Hugs on any system, 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 (see Section 3.2), 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.

4.1.3 Controlling Flag Assignments

Flag assignments (see Section 3.1.5.5) 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`.

4.1.4 Miscellaneous options

`--user` Does a per-user installation. This changes the default installation prefix (see Section 4.1.2.2). 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 instllation 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 *`d-b`* 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-executable-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.

`--disable-executable-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` (default) 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 seperate compiler run to generate position independent code as required on most platforms.

--disable-shared (default) Do not build shared library.

--configure-option=*str* An extra option to an external `configure` script, if one is used (see Section 3.3). 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.

In the simple build infrastructure, an additional option is recognized:

--scratchdir=*dir* Specify the directory into which the Hugs output will be placed (default: `dist/scratch`).

4.2 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 (see Section 4.1.1). 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.

4.3 setup makefile

Generate a `Makefile` that may be used to compile the Haskell modules to object code. This command is currently only supported when building libraries, and only if the compiler is GHC.

The makefile command replaces part of the work done by `setup build`. The sequence of commands would typically be:

```
runhaskell Setup.hs makefile
make
runhaskell Setup.hs build
```

where `setup makefile` does the preprocessing, `make` compiles the Haskell modules, and `setup build` performs any final steps, such as building the library archives.

The Makefile does not use GHC's `--make` flag to compile the modules, instead it compiles modules one at a time, using dependency information generated by GHC's `-M` flag. There are two reasons you might therefore want to use `setup makefile`:

- You want to build in parallel using `make -j`. Currently, `setup build` on its own does not support building in parallel.
- You want to build an individual module, pass extra flags to a compilation, or do other non-standard things that `setup build` does not support.

This command takes the following options:

--file=*filename* or **-f *filename*** Specify the output file (default `Makefile`).

4.4 setup haddock

Build the documentation for the package using haddock. By default, only the documentation for the exposed modules is generated (see --executables).

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 listed in Section 4.1.2.2 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 HackageDB 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
```

4.5 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* Copy the CSS file from *path* to `dist/doc/html/pkgid/src/hscolour.css` for exported modules, or to `dist/doc/html/pkgid/executable/src/hscolour.css` for executable programs. The CSS file defines the actual colours used to colourise code. Note that the `hscolour.css` file is required for the code to be actually colourised.

4.6 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** (see Section 4.1.2).

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.)

4.7 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.

4.8 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.

4.9 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.

4.10 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.

4.11 setup test

Run the test suite specified by the `runTests` field of `Distribution.Simple.UserHooks`. See Distribution.Simple for information about creating hooks and using `defaultMainWithHooks`.

4.12 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`, `data-files` and `extra-source-files` fields.

This command takes the following option:

--snapshot Append today's date (in *YYYYMMDD* form) to the version number for the generated source package. The original package is unaffected.

5 Reporting bugs and deficiencies

Please report any flaws or feature requests in the bug tracker.

For general discussion or queries email the libraries mailing list libraries@haskell.org. There is also a development mailing list cabal-devel@haskell.org.

6 Stability of Cabal interfaces

The Cabal library and related infrastructure is still under active development. New features are being added and limitations and bugs are being fixed. This requires internal changes and often user visible changes as well. We therefor cannot promise complete future-proof stability, at least not without halting all development work.

This section documents the aspects of the Cabal interface that we can promise to keep stable and which bits are subject to change.

6.1 Cabal file format

This is backwards compatible and mostly forwards compatible. New fields can be added without breaking older versions of Cabal. Fields can be deprecated without breaking older packages.

6.2 Command-line interface

6.2.1 Very Stable Command-line interfaces

- ./setup configure
  - --prefix
  - --user
  - --ghc, --hugs
  - --verbose
  - --prefix
- ./setup build
- ./setup install
- ./setup register
- ./setup copy

6.2.2 Stable Command-line interfaces

6.2.3 Unstable command-line

6.3 Functions and Types

The Cabal library follows the Package Versioning Policy. This means that within a stable major release, for example 1.2.x, there will be no incompatible API changes. But minor versions increments, for example 1.2.3, indicate compatible API additions.

The Package Versioning Policy does not require any API guarantees between major releases, for example between 1.2.x and 1.4.x. In practise of course not everything changes between major releases. Some parts of the API are more prone to change than others. The rest of this section gives some informal advice on what level of API stability you can expect between major releases.

6.3.1 Very Stable API

- `defaultMain`
- `defaultMainWithHooks defaultUserHooks`
  But regular `defaultMainWithHooks` isn't stable since `UserHooks` changes.

6.3.2 Semi-stable API

- `UserHooks` The hooks API will change in the future
- `Distribution.*` is mostly declarative information about packages and is somewhat stable.

6.3.3 Unstable API

Everything under `Distribution.Simple.*` has no stability guarantee.

6.4 Hackage

The index format is a partly stable interface. It consists of a tar.gz file that contains directories with `.cabal` files in. In future it may contain more kinds of files so do not assume every file is a `.cabal` file. Incompatible revisions to the format would involve bumping the name of the index file, i.e., `00-index.tar.gz`, `01-index.tar.gz` etc.