hs-to-coq Documentation

Release 0.1

Stephanie Weirich

Jun 02, 2021

Contents:

1 Installation **1**
- 1.1 Latest release . . . 1
- 1.2 Development version . . . 1
- 1.3 Coq Requirements . . . 1
- 1.4 Test your hs-to-coq installation . . . 2
- 1.5 Troubleshooting . . . 2

2 Quickstart **5**
- 2.1 Translating a single Haskell file . . . 5
- 2.2 Local Edit files . . . 6
- 2.3 Additional Coq definitions . . . 6
- 2.4 Strict vs. permissive translation . . . 6
- 2.5 Proofs . . . 6
- 2.6 Translating a multi-file project . . . 7
- 2.7 Avoiding `base` . . . 7

3 Edit Files **9**
- 3.1 General format of edit files . . . 9
- 3.2 Skipping Haskell . . . 9
- 3.3 Adding Coq Commands . . . 15
- 3.4 Renaming and Rewriting . . . 16
- 3.5 Extra information . . . 18
- 3.6 Termination edits . . . 21
- 3.7 Mutual recursion edits . . . 23
- 3.8 Invariant Edit . . . 26
- 3.9 Meta-edits . . . 29
- 3.10 Deprecated edits . . . 30

4 Identifiers and Notation **31**
- 4.1 Coq keywords . . . 31
- 4.2 Operators . . . 31
- 4.3 Notation . . . 31
- 4.4 String Notation . . . 32

5 Interface files **33**

6 Indices and tables **35**

Index 37

CHAPTER 1

Installation

1.1 Latest release

Not yet released. Eventually *hs-to-coq* will be available on hackage.

1.2 Development version

The source code for the development version is available via github. This repository includes everything that you need.

```
$ git clone https://github.com/antalsz/hs-to-coq.git
$ cd hs-to-coq
```

The recommended way of building *hs-to-coq* is to use the *stack* tool. If you have not setup stack before

```
$ stack setup
```

To build `hs-to-coq`

```
$ stack build
```

To compile *hs-to-coq* from scratch, we recommend GHC 8.4.1.

1.3 Coq Requirements

This repository comes with a Coq version of the Haskell base library, used by the output of `hs-to-coq`.

You must have *Coq 8.8.1* and *ssreflect* to build the base library. You can install these tools using opam.

```
$ opam repo add coq-released https://coq.inria.fr/opam/released
$ opam update
$ opam install coq.8.8.1 coq-mathcomp-ssreflect.1.6.4
```

Once installed, you can build the base library from the project root with

```
$ make -C base
```

Th directory base-thy contains auxillary definitions and lemmas, such as lawful type-class instances. You can build these with

```
$ make -C base-thy
```

1.4 Test your hs-to-coq installation

To test whether your *hs-to-coq* installation is successful, you can try to compile the examples that are distributed with the tool.

Some examples use git submodules, so run

```
$ git submodule update --init --recursive
```

once first to download all dependencies.

Then, compile all of the examples with

```
$ cd examples
$ ./boot.sh
```

The flag *noclean* will recompile everything without first deleting the old versions.

```
$ ./boot.sh noclean
```

The flag *quick* is like the above but doesn't run the tests.

```
$ ./boot.sh quick
```

1.5 Troubleshooting

1. lndir: command not found

On Mac OSX, you may need to install XQuartz and imake to run *examples/boot.sh*.

If you get the error message while trying to run *./boot.sh*:

```
> lndir: command not found
```

The following commands will install XQuartz and imake through *brew*:

```
$ brew cask install xquartz
$ brew install imake
```

Depending on your *brew cask* setup, you may also need to update your $PATH variable.

```
$ export PATH=$PATH:/usr/X11/bin >> ~/.bash_profile
```

2. git submodule update –init –recursive gives error fatal: Needed a single revision

Try removing the submodule directory that the error was triggered on, and run the command again. (i.e. If the error was on *examples/wc/wc*, a *rm -rf examples/wc/wc* followed by a *git submodule update –init –recursive* will do the trick.

CHAPTER 2

Quickstart

The easiest way to see how to use `hs-to-coq` is to look at the Makefiles in each of the subdirectories in the examples subdirectory of the repository.

2.1 Translating a single Haskell file

You can run *hs-to-coq* on a single Haskell module (called *Main.hs*) with the following command.

```
$ stack exec -- hs-to-coq -e hs-to-coq/base/edits Main.hs --iface-dir hs-to-coq/base -
↪o .
```

Adjust the paths to the edits and base files in the `hs-to-coq` repository accordingly. This invocation uses the following commandline options.

`-e` `<editfile>`

The `-e` command line argument tells `hs-to-coq` to use edits from the specified edit file, and can be used multiple times. In almost every case, you'll want to include `-e hs-to-coq/base/edits`, the edit file distributed with the base library.

`--iface-dir` `<dir>`

The `--iface-dir` command line argument tells `hs-to-coq` where to find the interface files for the translated files in the `base` library. These interface files contain extra information about produced during translation and are needed to translate any modules that use the `base` libraries.

`-o` `<dir>`

The `-o` argument specifies the output directory for the generated `.v` files. In this case, it is the current directory.

An example translated in this way is simple. Check out the `Makefile` in the `examples/simple` subdirectory to see how `hs-to-coq` is invoked with these arguments.

2.2 Local Edit files

Often, a particular file will require its own set of edits. These edits can be provided with additional uses of the `-e <editfile>` command line argument.

An example that uses a local edit file is intervals, as described in Joachim Breitner's blog post.

Any number of edit files may be provided to `hs-to-coq`.

2.3 Additional Coq definitions

Sometimes an `hs-to-coq` translation requires the addition of Coq definitions to the output. These definitions can be specified via the `preamble` and `midamble` arguments.

`-p` `<preamble-file>`

Inserts the Coq definitions from the specified file at the beginning of the output.

For example, the rle example uses a preamble to add additional imports to the output.

`-m` `<midamble-file>`

Inserts the Coq definitions from the specified file in the output after the type definitions but before any of the translated code or instance declarations.

Only one preamble and one midamble can be provided to `hs-to-coq`.

2.4 Strict vs. permissive translation

What should `hs-to-coq` do when it can't translate a definition? By default, it will throw an error and stop translating. But you can make this behavior more permissive if you want.

`--strict`

`-S`

The default option: any definition that can't be translated will stop the whole development process.

`--permissive`

`-P`

In permissive mode, `hs-to-coq` will either attempt to axiomatize or skip failing definitions when possible. This is particularly useful during development.

2.5 Proofs

Once you have translated your module with `hs-to-coq`, you will want to prove stuff about it. However, if your module includes definitions from `base`, you need to set up a `_CoqProject` file so that `coq` can find all of the necessary definitions.

The Makefile in the rle example demonstrates how this file can be constructed automatically. The proofs in this example use a lemma called `map_map` from the base library.

2.6 Translating a multi-file project

Larger examples include containers and transformers.

These examples use a `Makefile` to translate each module in the library individually, using edit files, preambles and midambles specific to each module. It also includes the addition of manually written Coq files to the library.

For this scale of project, we recommend starting with one of the Makefiles above and editing it to suit your application.

2.7 Avoiding `base`

`hs-to-coq` is designed to automatically use definitions from the `base` library. However, it is sometimes possible to translate small examples so that they are self contained and only require definitions from Coq's standard library.

An example project that takes this approach is: https://github.com/mit-plv/riscv-coq

CHAPTER 3

Edit Files

The edit files contain declarations that control the output of `hs-to-coq` on various Haskell files.

3.1 General format of edit files

Edit files are plain text files. Empty lines and lines starting with # are ignored. Otherwise, the files are consist of a sequence of *edits*. Most form of edits are exactly one line long, but some edits can span multiple lines, and must be terminated with a period.

Make sure that your edit file ends with a newline.

3.1.1 Qualified names

A *qualified_name* is the Coq name with module prefix. Names must always be qualified, because edit files are not bound to a specific module (even though you may want to have a separate edit for each Haskell module).

Reserved names have an underscore appended and renames (see below) have already been applied.

More details about how `hs-to-coq` treats see Section *Identifiers and Notation*.

3.1.2 Gallina expressions

Some edits contain Gallina expressions (i.e. Coq code). The parser is pretty limited. In particular, it does not know anything about operator precedence or associativity, so add plenty of parentheses!

3.2 Skipping Haskell

Sometimes, `hs-to-coq` should ignore various Haskell declarations, because they are not translatable, or they are out-of-scope, or for other reasons.

3.2.1 `skip` – skip a function, type, or type class instance

Format:

skip *qualified_name*

Effect: During translation, ignore the declaration of the function, value, type, or type class instance with the given *qualified_name*. The name must be the translated Coq name, not the original Haskell name (if those differ).

This does not affect the translation of *uses* of the given name. This means that you can use other methods, e.g. a preamble, to make it available.

You can skip type class instances, but as they do not have names in Haskell, you must use the name `hs-to-coq` generates for them. The name generation is systematic, but you might want to first attempt the translation and check the output for the precise name.

To skip data type constructors, see `skip constructor`; to skip type classes, see `skip class`; to skip type class methods, see `skip methods`. They are not unified here because those effects are more powerful. You can also skip whole modules with `skip module`.

Examples:

```
skip Data.Function.fix_ # Note the mangled name!
skip GHC.Base.String
skip GHC.Real.Fractional
skip Data.Monoid.Show__Last # an instance
```

3.2.2 `skip constructor` – skip a constructor of a data type

Format:

skip constructor *qualified_name*

Effect: During translation, ignore the given data type constructor. Any equation of a function, alternative of a case statement, or pattern guard that pattern-matches on that constructor is also skipped. List comprehensions that bind to that constructor become `[]`.

As with `skip`, this does not affect the translation of *uses* of the constructor. This means that you must either make it available in a preamble or elide it with other edits. Additionally, matching against the constructor in `do` notation will cause a translation failure.

These skipped constructors are not stored in the generated metadata, so you need to include the `skip constructor` edits in all downstream modules.

Examples:

```
skip constructor Core.Cast
skip constructor Core.Tick
skip constructor Core.Type_
skip constructor Core.Coercion
```

3.2.3 `skip class` – skip a type class and all its instances

Format:

skip class *qualified_class*

Effect:

Omit the given type class and all its instances.

These skipped classes are not stored in the generated metadata, so you need to include the `skip class` edits in all downstream modules.

Examples:

```
skip class GHC.Base.Alternative
skip class Data.Data.Data
```

3.2.4 `skip method` – skip a method

Format:

skip method *qualified_class method*

Effect: Omit the given method from the its class declaration, and also from all instances.

Examples:

```
skip method GHC.Base.Monad fail
```

3.2.5 `skip equation` – skip one equation of a function definition

Format:

skip equation *qualified_function pattern ...*

Effect: Skip the equation of the function definition whose arguments are the specified patterns. Guards are not considered, only the patterns themselves.

For example, consider the following (silly) function definition:

```
redundant :: Maybe Bool -> Maybe Bool -> Bool
redundant (Just True)  _         = False
redundant (Just False) _         = True
redundant _            _         = True
redundant _            (Just b) = b
```

The last case is redundant, so Coq will reject this definition. However, we can add the following edit:

```
skip equation ModuleName.redundant _ (Some b)
```

And the last case will be deleted on the Coq side:

```
Definition redundant : option bool -> option bool -> bool :=
  fun arg_0__ arg_1__ =>
    match arg_0__, arg_1__ with
    | Some true, _ => false
    | Some false, _ => true
    | _, _ => true
    end.
```

Note that you have to use the translated name (`Some` vs. `Just`), and most constructor names will be fully qualified.

Why would you want this? This edit is most useful in tandem with `skip constructor` (which see). Suppose we have a function where the final catch-all case can only match skipped constructors, such as

```
data T = TranslateMe
       | SkipMe

function :: T -> Bool
function TranslateMe = True
function _           = False
```

Then, on skipping `SkipMe`, this function's _ case will be redundant, and Coq would reject it. We can fix this with

```
skip equation ModuleName.function _
```

to translate just the `TranslateMe` case.

See also `skip case pattern` for the equivalent edit for `case` and lambda-case expressions.

Examples:

```
skip equation ModuleName.redundant _ (Some b)
skip equation Core.hasSomeUnfolding _
```

3.2.6 `skip case pattern` – skip one alternative of a `case` expression

Format:

skip case pattern *pattern*

Effect: Skip any alternative of a `case` expression (or a lambda-case expression) which matches against the given pattern. Guards are not considered, only the pattern itself.

For example, consider the following (silly) function definition:

```
redundant :: Bool -> Bool
redundant b = not (case b of
                    True  -> False
                    False -> True
                    _     -> True)
```

The last case is redundant, so Coq will reject this definition. However, we can add the following edit:

```
in ModuleName.redundant skip case pattern _
```

And the last case will be deleted on the Coq side (reformatted):

```
Definition redundant : bool -> bool :=
  fun b => negb (match b with
                 | true => false
                 | false => true
                end).
```

You can use an arbitrary pattern, not simply _; constructor names must be fully qualified and the names used must be those that appear *after* renaming.

Why would you want this? This edit is most useful in tandem with `skip constructor` (which see); see the discussion in `skip equation` for a worked example (with a named function).

This edit is unusual in that you *very likely* want to use it with the `in` meta-edit to scope its effects to within a specific definition. However, this isn't mandatory; if, for some reason, you want to skip every _ in every `case`, then `skip case pattern _` will do what you want.

See also `skip equation` for the equivalent edit for named functions.

Examples:

```
in ModuleName.redundant skip case pattern _
```

3.2.7 `skip module` – skip a module import

Format:

skip module *module*

Effect: Do not generate an `Require` statemnt for *module*.

This is mostly useful during development: `hs-to-coq` automatically requires the modules of all names it encounters, in the beginning of the resulting file. If there are names from modules that you do not intent to translate, Coq will already abort there. It is more convenient to have it fail when the name is actually encountered, to then decide how to fix it (e.g. using `skip`, `rename` or `rewrite`).

In the end, all mentions of names in the give module ought to be gone, in which case `hs-to-coq` would not generate an `Require` statement anyways. So in complete formalizations, this edit should not be needed.

Examples:

```
skip module GHC.Show
```

3.2.8 `axiomatize module` – replace all definitions in a module with axioms

Format:

axiomatize module *module*

Effect: Replaces all definitions in a module with axioms.

This translates type and type class definitions, and then produces axioms for variable bindings and type class instances which have the translated types. Any types that are `redefine`d are correctly redefined; any bindings or instances that are `skip`ped don't have axioms generated. If you want to override the axiomatization for a single definition and actually translate it, you can use the `unaxiomatize definition` edit.

The `axiomatize module` edit is useful if you want to stub out a dependency of a module you are actually interested in.

See also `axiomatize definition`.

Examples:

```
axiomatize module TrieMap
```

3.2.9 `axiomatize original module name` – replace all definitions in a module with axioms, using the pre-renaming module name

Format:

axiomatize original module name *module*

Effect: You probably do not need to use this edit; it's only important when using `rename module` to merge multiple modules into one. If you are doing this, however, and wish you could use `axiomatize module` on *some* of the input modules but not others, then `axiomatize original module name` is the edit for you.

The behavior of `axiomatize original module name` is the same as that of `axiomatize module`, except for how it picks which module to axiomatize. While every other edit operates in terms of the Coq name after any renamings from the edits have been applied, `axiomatize original module name` checks the *original*, pre-`rename module`, form of the module name. Most of the time, this would be confusing, and `axiomatize module` would be preferable.

However, if you have used `rename module` to merge two (or more) modules into one, but you only want one of them (or some other strict subset) to be axiomatized, then `axiomatize original module name` is the only way to get this behavior.

Examples:

```
axiomatize original module name Part1
rename module Part1 Whole
rename module Part2 Whole
```

3.2.10 `axiomatize definition` – replace a value definition with an axiom

Format:

axiomatize definition *qualified_name*

Effect: Replaces a single definition with an axiom.

This takes the name of a value-level definition and, when translating it, translates only the type and generates an axiom with that type.

See also `axiomatize module`, and also `redefine Axiom` for type-level axiomatization.

Examples:

```
axiomatize definition GHC.Prim.primitiveFunction
```

3.2.11 `unaxiomatize definition` – override whole-module axiomatization on a case-by-case basis

Format:

unaxiomatize definition *qualified_name*

Effect: Translates a single definition, `axiomatize module` notwithstanding.

If the module containing the given value-level definition is being axiomatized, then this definition will be translated in the usual way.

If a definition is both `unaxiomatize`d and `skip`ped, then it will simply be skipped. But please don't do this :-)

Examples:

```
axiomatize module TrieMap
unaxiomatize definition TrieMap.insertTM
unaxiomatize definition TrieMap.deleteTM
```

3.3 Adding Coq Commands

3.3.1 `add` – inject a definition

Format:

add *module coq_definition*

Effect: Add a Coq definition to *module*. The definition can be a `Definition`, a `Fixpoint`, an `Inductive`, an `Instance`, an `Axiom`, or a `Theorem` (with a `Proof`).

The name in the definition should be fully qualified. (If it is not, some dependency calculations inside `hs-to-coq` might go wrong – but this is not always critical.)

Our Coq parser is dramatically incomplete, and you may need to add parentheses or pick a simpler syntactic representation of terms to get them to parse correctly or at all. One example is that `hs-to-coq` does not understand the associativity of the function arrow when parsing: `a -> b -> c` will not parse, and needs to be given as `a -> (b -> c)`.

When providing a `Theorem` – or a `Lemma`, a `Remark`, a `Fact`, a `Corollary`, a `Proposition`, or an `Example` – it must be immediately followed by `Proof.`, some unparsed text (newlines are permitted), and then the word `Qed`, `Defined`, or `Admitted`.

This is a multi-line edit and needs to be terminated by a period (as is natural when writing a *coq_definition*).

Examples:

```
add Data.Foldable Definition Data.Foldable.elem {f} `{(Foldable f)} {a} `{(GHC.
↪Base.Eq_ a)} :
  a -> (f a -> bool) :=
  fun x xs => Data.Foldable.any (fun y => x GHC.Base.== y) xs.

add Data.Monoid Instance Unpeel_Last a : GHC.Prim.Unpeel (Last a) (option a) :=
  GHC.Prim.Build_Unpeel _ _ getLast Mk_Last.
```

3.3.2 `add type` – inject a definition into the type component

Format:

add type *module coq_definition*

Effect: Add a Coq definition to the *type* component of a *module*. The definition can be as above, and need not be a type definition. However, it is inserted before the *midamble* section and will appear grouped with the type and class definitions.

3.3.3 `import` – inject an `Import` statement

Format:

import module *module*

Effect: Inject a `Import` statement into the Coq code, which makes the definitions from the given module available unqualified.

When used to import the hs-to-coq base library, this makes the output look more like standard Haskell.

Note, however, that Coq's module system lacks the `import ... hiding` construct so all definitions from the module must be made available.

Examples:

```
import module Prelude
```

3.4 Renaming and Rewriting

3.4.1 `rename type` – rename a type

Format:

rename type *qualified_name* = *qualified_name*

Effect: Change the name of a Haskell type, at both definition and use sites.

Examples:

```
rename type GHC.Types.[] = list
rename type GHC.Natural.Natural = Coq.Numbers.BinNums.N
```

3.4.2 `rename value` – rename a value

Format:

rename value *qualified_name* = *qualified_name*

Effect: Change the name of a Haskell value (function, data constructor), at both definition and use sites.

Note: When renaming a name in its definition, you should not change the module.

Examples:

```
rename value Data.Foldable.length = Coq.Lists.List.length      # use Coq␣
↪primitive
rename value GHC.Base.++          = Coq.Init.Datatypes.app     # operators ok
rename value Data.Monoid.First    = Data.Monoid.Mk_First       # resolve␣
↪punning
```

3.4.3 `rename module` – change a module name

Format:

rename module *module module*

Effect: Change the name of a Haskell module, affecting the filename of the generated Coq module.

Note: If two modules are renamed to the same name, they will be combined into a single joint module, as long as they are processed during the same execution of `hs-to-coq`. This feature is useful to translate mutually recursive modules.

Examples:

```
rename module Type MyType
rename module Data.Semigroup.Internal Data.SemigroupInternal
```

3.4.4 `alias module` – abbreviate a module name

Format:

alias module *module module*

Effect: Abbreviate a module name with an alias. All occurrences of the alias in the current edits file are expanded to the original name. Aliases do not affect the generated Coq code.

Examples:

```
alias module Seq Data.Sequence.Internal
order Seq.Functor__Seq Seq.Applicative__Seq

# Equivalent to
order Data.Sequence.Internal.Functor__Seq Data.Sequence.Internal.Applicative_
↪_Seq
```

3.4.5 `rewrite` – replace Haskell subexpressions

Format:

rewrite forall *vars*, *expression* = *expression*

Effect:

Pattern-matches a sub-expression and replaces it with the right-hand side after substituting all variables.

The pattern-matching is unhygienic: if you mention a variable `x` in the pattern but not in the list of variables (*vars*), then the rewrite rule will only match if there is actually is a variable named `x`.

Examples:

```
## work around laziness
rewrite forall xs x, (GHC.List.zip xs (GHC.List.repeat x)) = (GHC.Base.map␣
↪(fun y => pair y x) xs)
rewrite forall x, GHC.Magic.lazy x = x

## replace with Coq library function
rewrite forall x y, GHC.List.replicate x y = Coq.Lists.List.repeat y x

## skip debugging code
rewrite forall x, andb Util.debugIsOn x = false

## create dummy strings to ignore particular definitions
## note empty variable list
rewrite forall , Outputable.empty = (GHC.Base.hs_string__ "Outputable.empty")
```

3.4.6 `redefine` – override a Coq definition

Format:

redefine *Coq_definition*

Effect: Combines the **skip** and **add** edits.

You can use `redefine Axiom ...` to replace a type-level definition with an axiom; for value-level definitions, please use `axiomatize definition` instead.

Examples:

```
redefine Definition GHC.Base.map {A B :Type} (f : A -> B) xs := Coq.Lists.
↪List.map f xs.
```

3.4.7 `collapse let` – if a definition is just a `let`-expression, inline it

Format:

collapse let *qualified_name*

Effect: If a converted definition is of the form

```
Definition outer := let inner := definition in inner.
```

then convert it to simply

```
Definition outer := definition.
```

Both `outer` and `inner` can have arguments; `inner` can have a type annotation, but it's ignored.

Additionally, if `definition` is a non-mutual fixpoint `fix f args := body`, the recursive calls to `f` in `body` are rewritten to direct calls to `outer`.

This is particularly important for mutual recursion: if `inner` is mutually recursive with another top-level function, then if `outer` has no arguments, it would otherwise appear not to be a function and would thus cause conversion to fail, as Coq doesn't support recursion through non-functions.

Examples:

```
collapse let CoreFVs.freeVars
```

3.5 Extra information

3.5.1 `data kinds` – Declare kinds of type arguments to Inductive datatypes

Format:

data kinds *qualified_name Coq_types*

Effect:

Haskell programmers rarely include kind signatures on inductive datatypes. This usually isn't a problem, but for higher-order parameters, some phantom type parameters, or poly-kinded type parameters, Coq does not necessarily automatically infer the right types. In these cases, the information can be included in an edit.

Examples:

```
# The edit file's Coq parser needs parentheses
data kinds Control.Applicative.WrappedArrow (Type -> (Type -> Type))

# Multiple kinds are separated with commas
data kinds Data.Functor.Reverse.Reverse   (Type -> Type), Type
data kinds Data.Functor.Constant.Constant Type,           Type
```

3.5.2 `polykinds` – Declare polymorphic kind variables to Inductive datatypes

Format:

polykinds *qualified_name name*

Effect:

For Haskell programs written with the `PolyKind` extension, the user can provide the polymorphic kind variables to help hs-to-coq to include those kind variables.

Examples:

```
polykinds Data.Monoid.Ap k

data kinds Data.Monoid.Ap (k -> Type), k
```

3.5.3 `class kinds` – Declare kinds of type arguments to type classes

Format:

class kinds *qualified_name Coq_types*

Effect:

Like `data kinds`, but for classes.

Examples:

```
class kinds Control.Arrow.Arrow (Type -> (Type -> Type))
```

3.5.4 `delete unused type variables` – Remove unused type variables from a declaration

Format:

delete unused type variables *qualified_name*

Effect:

Don't translate binders for any type variables that aren't visibly used in the specified definition.

An explanation: sometimes, poly-kinded Haskell data types have extra invisible type parameters. For instance, in `Data.Functor.Const`, we have the type

```
newtype Const a b = Const { getConst :: a }
```

which is secretly

```
newtype Const {k} (a :: Type) (b :: k) = Const { getConst :: a }
```

Often, such as here, this doesn't show up in the translated Coq code; we get

```
Inductive Const a b : Type := Mk_Const (getConst : a) : Const a b.
```

(And, as in Haskell 2010, `b` is inferred to have kind `Type`.) Sometimes it does, in which case we can fix it using `data kinds`. But either way, we still introduce spurious kind variables in the translation sometimes. For example, the derived `Eq` instance for `Const` is translated to

```
Program Instance Eq___Const {a} {k} {b} `{GHC.Base.Eq_ a}
   : GHC.Base.Eq_ (Const a b : GHC.Prim.TYPE GHC.Types.LiftedRep) :=
  fun _ k =>
    k {| GHC.Base.op_zeze____ := Eq___Const_op_zeze__ ;
         GHC.Base.op_zsze____ := Eq___Const_op_zsze__ |}.
```

The implicit argument `{k}` isn't useful in the Coq code, and causes a type-checking failure when its type cannot be determined. We can avoid this with

```
delete unused type variables Data.Functor.Const.Eq___Const
```

which will drop the `{k}` and leave the definition with just the `{a}` and `{b}` it needs:

```
Program Instance Eq___Const {a} {b} `{GHC.Base.Eq_ a}
   : GHC.Base.Eq_ (Const a b : GHC.Prim.TYPE GHC.Types.LiftedRep) :=
  fun _ k =>
    k {| GHC.Base.op_zeze____ := Eq___Const_op_zeze__ ;
         GHC.Base.op_zsze____ := Eq___Const_op_zsze__ |}.
```

Examples:

```
delete unused type variables Data.Functor.Const.Eq___Const
```

3.5.5 `order` – reorder output

Format:

order *qualified_name* ...

Effect: `hs-to-coq` topologically sorts definitions so that they appear in dependency order. However, this sorting is not always correct — type classes introduce implicit dependencies that are invisible to `hs-to-coq`. This edit adds a new ordering constraint into the topological sort so that the output definitions appear in the order indicate in this edit.

You can order more than two definitions at the same time:

```
order Foo.foo Foo.bar Foo.baz
```

is equivalent to

```
order Foo.foo Foo.bar
order Foo.bar Foo.baz
```

Examples:

```
order GHC.Base.Functor__arrow GHC.Base.Applicative__arrow_op_ztzg__ GHC.Base.
↪Applicative__arrow GHC.Base.Monad__arrow_return_ GHC.Base.Monad__arrow GHC.Base.
↪Alternative__arrow GHC.Base.MonadPlus__arrow
```

3.5.6 `promote` – promote a definition from the term level to the type level

Format:

promote *qualified_name* ...

Effect: *hs-to-coq* divides Haskell definitions into two "levels": type-level and term-level. Type-level definitions always appear above term-level definitions in the Coq output. The `promote` edit moves a term-level definition to the type level. It also recursively moves the transitive closure of all definitions on which the specified definition depends.

Examples:

```
promote MyModule.foo
```

3.5.7 `polyrec` – Make a function polymorphic recursive

Format:

polyrec *qualified_name*

Effect: Translates a function which calls itself recursively at a different type.

Examples:

```
polyrec MyModule.foo
```

3.5.8 `manual notation` – Indicate presence of manual notation

Format:

manual notation *name*

Effect: If your preamble includes custom notation (usually for operators), you need to indicate this using this edit. See Section *Identifiers and Notation* for more information about how `hs-to-coq` implements custom notation.

Examples:

```
manual notation GHC.Base
```

3.5.9 `set type` – Specify a type for a binding

Format:

set type *qualified_name* : *Coq_type*
set type *qualified_name* no type

Effect: Sets the type of the given definition to the given type, or omits the type if `no type` is specified.

Examples:

```
set type Example.int_to_int : Z -> Z
set type Example.inferred no type
in CoreUtils.stripTicksE set type go_b : (b * Core.Expr b) -> (b * Core.Expr b)
```

3.6 Termination edits

3.6.1 `coinductive` – use a coinductive instead of an inductive datatype

Format:

coinductive *qualified_name*

Effect:

Examples:

3.6.2 `termination` – hints for termination proofs

Format:

termination *qualified_name termination_argument*

Examples:

```
termination MyModule.foo {struct arg_33__}
termination MyModule.foo {struct 3}
termination MyModule.foo {measure myExpr}
termination MyModule.foo {wf myRel myExpr}
termination MyModule.foo {corecursive}
termination MyModule.foo {deferred}
```

Effect:

By default, `hs-to-coq` translates recursive definitions using Coq's `fix` operator, which requires that the recursion is obviously structurally recursive. This is not always the right choice, and a `termination` edit tells `hs-to-coq` to construct the recursive definition differently, where *termination_argument* is one of the following:

- **corecursive**

  This causes `hs-to-coq` to use `cofix` instead of `fix`.

- **{ struct** *qualified_name* **}**

  Coq's `fix` operator usually determines the recursive argument automatically, but also supports the user to specify it explicitly. This *termination_argument* is just passed along to Coq's `fix`.

  { struct *number* **}**

  This variant specifies the position of the recursive argument, counting from 1 and ignoring type arguments. (Note: this variant is not actually valid Coq syntax, it is only allowed in edits files.)

- **{ measure** *expr* **}**

  { measure *expr* **(** *relation* **) }**

  { wf *relation expr* **}**

  With one of these forms for *termination_argument*, `hs-to-coq` uses `Program Fixpoint` to declare the function, passing these termination arguments along. See the documentation of `Program Fixpoint` for their precise meaning.

  The *expr* is a Coq expression that mentions the parameters of the current functions. These often have names generated by `hs-to-coq` – look at the generated Coq code to see what they are.

  `Program Fixpoint` only supports top-level declaration. When these termination edits are applied to local definitions, `hs-to-coq` therefore uses the fixed-point operator `wfFix1` defined in `GHC.Wf` in our `base` library.

  A side effect of these edits is that the definition (or the enclosing definition) is defines using `Program`, which leaves proof obligations to the user. These should be discharged using the `obligations` edit (see below).

- **deferred**

  This causes `hs-to-coq` to use the axiom `deferredFix` from the module `GHC.DeferredFix` to translate the recursive definition. This defers the termination proof until the verification stage, where the axiom `deferredFix_eq_on` is needed to learn anything about the recursive function, and this axion requires an (extensional) termination proof.

  See the file `GHC/DeferredFix.v` for more details.

Examples:

```
termination Memo.mkTrie corecursive

termination Memo.lookupTrie { measure arg_1__ (Coq.NArith.BinNat.N.lt) }
obligations Memo.lookupTrie solve_lookupTrie

termination Data.Set.Internal.link {measure (Nat.add (set_size arg_1__) (set_
↪size arg_2__))}
obligations Data.Set.Internal.link termination_by_omega

in Data.IntSet.Internal.foldlBits  termination go  {measure (Coq.NArith.
↪BinNat.N.to_nat arg_0__)}
obligations Data.IntSet.Internal.foldlBits BitTerminationProofs.termination_
↪foldl


termination QuickSort.quicksort deferred
```

3.6.3 `obligations` – Proof obligations in `Program` mode

Format:

obligations *qualified_name tactic*

Effect: The specified definition is now defined using `Program`, and is followed by

```
Solve Obligations with (tactic).
```

with the specified tactic.

This is most commonly used with with the `termination` hint, but can be useful on its own: For example, `Program` mode automatically applies or unwraps sigma types, which may leave proof obligations.

The `{ measure ... }` termination argument of the `termination` edit always causes `Program` to be used. If no `obligations` edit is specified, then all obligations are solved with `Admit Obligations.`.

The `tactic` is drawn from a very simple subset of Ltac, featuring identifiers, identifiers with `@`, application, numbers, underscore, `;`, and `||`. Anything richer should go in the preamble or midamble.

3.7 Mutual recursion edits

3.7.1 `inline mutual` – Move mutually-recursive functions into `let`-bindings

Format:

inline mutual *qualified_name*

Effect: The specified definition must be part of a mutually recursive set of definitions. Instead of being defined as another mutual fixpoint, it will be inlined into each of the other mutual fixpoints that needs it with a `let`-binding; additionally, a top-level Coq definition is generated for each `let`-bound function that simply calls into the predefined recursive functions.

This facility is useful when translating groups of mutually recursive functions that contain "preprocessing" or "postprocessing" functions, where the group is otherwise structurally recursive. These functions are not "truly" mutual recursive, as they just hand along values of the type being recursed, and so if Coq could only see through them, everything would work fine. And indeed, as `let`-bindings, Coq can see through them.

As an example, consider the following pair of mutually recursive data types, which represent a `Forest` of nonempty `Trees`. Each `Branch` of a `Tree` holds an extra boolean flag, which we can extract with `isOK`. In Haskell:

```
data Forest a = Empty
              | WithTree (Tree a) (Forest a)

data Tree a = Branch Bool a (Forest a)

isOK :: Tree a -> Bool
isOK (Branch ok _ _) = ok
```

And in cleaned-up Coq:

```
Inductive Forest a : Type
  := Empty    : Forest a
  |  WithTree : Tree a -> Forest a -> Forest a
with Tree a : Type
  := Branch : bool -> a -> Forest a -> Tree a.

Arguments Empty    {_}.
Arguments WithTree {_} _ _.
Arguments Branch   {_} _ _ _.

Definition isOK {a} : Tree a -> bool :=
  fun '(Branch ok _ _) => ok.
```

Now we can define a pair of mapping functions that only apply a function inside subtrees where the boolean flag is true. The Haskell code is simple:

```
mapForest :: (a -> a) -> Forest a -> Forest a
mapForest f Empty           = Empty
mapForest f (WithTree t ts) = WithTree (mapTree f t) (mapForest f ts)

mapTree :: (a -> a) -> Tree a -> Tree a
mapTree f t | isOK t    = mapOKTree f t
            | otherwise = t

mapOKTree :: (a -> a) -> Tree a -> Tree a
mapOKTree f (Branch ok x ts) = Branch ok (f x) (mapForest f ts)
```

However, the (cleaned-up) Coq translation fails:

```
Fail Fixpoint mapForest {a} (f : a -> a) (ts0 : Forest a) {struct ts0} : Forest a␣
↪:=
  match ts0 with
  | Empty         => Empty
  | WithTree t ts => WithTree (mapTree f t) (mapForest f ts)
```

(continues on next page)

(continued from previous page)

```
  end
with mapTree {a} (f : a -> a) (t : Tree a) {struct t} : Tree a :=
  if isOK t
  then mapOKTree f t
  else t
with mapOKTree {a} (f : a -> a) (t : Tree a) {struct t} : Tree a :=
  match t with
  | Branch ok x ts => Branch ok (f x) (mapForest f ts)
  end.
```

The issue is that `mapTree` calls `mapOKTree` on the *same* term, and not a subterm. But this just a preprocessing/postprocessing split – there's nothing *actually* recursive going on.

But with

```
inline mutual mapOKTree
```

we instead get working Coq code (again, cleaned up):

```
Fixpoint mapForest {a} (f : a -> a) (ts0 : Forest a) {struct ts0} : Forest a :=
  match ts0 with
  | Empty         => Empty
  | WithTree t ts => WithTree (mapTree f t) (mapForest f ts)
  end
with mapTree {a} (f : a -> a) (t : Tree a) {struct t} : Tree a :=
  let mapOKTree {a} (f : a -> a) (t : Tree a) : Tree a :=
        match t with
        | Branch ok x ts => Branch ok (f x) (mapForest f ts)
        end in
  if isOK t
  then mapOKTree f t
  else t.

Definition mapOKTree {a} (f : a -> a) (t : Tree a) : Tree a :=
  match t with
  | Branch ok x ts => Branch ok (f x) (mapForest f ts)
  end.
```

This is the idea. However, to be completely fair, we never produce `Fixpoint` commands; both in the failing case and in the successful case, we generate `fix` terms. In this example, this looks like (reindented)

```
Definition mapForest {a} : (a -> a) -> Forest a -> Forest a :=
  fix mapTree f t :=
    let mapOKTree arg_0__ arg_1__ :=
          match arg_0__, arg_1__ with
          | f, Branch ok x ts => Branch ok (f x) (mapForest f ts)
          end in
    if isOK t : bool
    then mapOKTree f t
    else t
  with mapForest arg_0__ arg_1__ :=
    match arg_0__, arg_1__ with
    | f, Empty => Empty
    | f, WithTree t ts => WithTree (mapTree f t) (mapForest f ts)
    end
  for mapForest.
```

(continues on next page)

(continued from previous page)

```
Definition mapOKTree {a} : (a -> a) -> Tree a -> Tree a :=
  fun arg_0__ arg_1__ =>
    match arg_0__, arg_1__ with
    | f, Branch ok x ts => Branch ok (f x) (mapForest f ts)
    end.

Definition mapTree {a} : (a -> a) -> Tree a -> Tree a :=
  fix mapTree f t :=
    let mapOKTree arg_0__ arg_1__ :=
          match arg_0__, arg_1__ with
          | f, Branch ok x ts => Branch ok (f x) (mapForest f ts)
          end in
    if isOK t : bool
    then mapOKTree f t
    else t
  with mapForest arg_0__ arg_1__ :=
    match arg_0__, arg_1__ with
    | f, Empty => Empty
    | f, WithTree t ts => WithTree (mapTree f t) (mapForest f ts)
    end
  for mapTree.
```

3.8 Invariant Edit

3.8.1 `add invariant` – Specify an invariant that a datatype should satisfy

NOTE: Currently only applies to datatypes with a single constructor!

Format:

add invariant {

module = *module*,

qid = *qualified_name*,

tyVars = [*type_variables*],

constructor = *qualified_name*,

useSigmaType = [*names*],

invariant = *coq_definition*

}

Note: Currently, you cannot leave any of the parameters out, and they must appear in the order above.

Meaning of each parameter:

- `module` denotes the name of the module to which the invariant applies.
- `qid` is a qualified name that denotes the type to which the invariant applies.
- `tyVars` is a list of the type variable arguments of the type passed to `qid`. For example, if the type passed to `qid` was declared in Haskell as

```
data Foo a b where
...
```

then `tyVars` should be set to `[a, b]`.

- `constructor` is a qualified name referring to the constructor of the type to which the invariant applies.
- `useSigmaType` is a list of **unqualified** names of the definitions that should preserve the invariant. The `Program` keyword will be added in front of these definitions, so that Coq generates the needed proof obligations (see below under "Effect"). You must supply the proofs (see the edits file of the extended example below).
- `invariant` is a Coq definition encoding the invariant. Note that the type of the definition passed to `invariant` should be a function type, where the argument type is the name of the type passed to `qid`, prefixed by "Raw", and the return type is `Type`. For instance, if the type passed to `qid` is `MyModule.Foo`, the type of the definition passed to `invariant` should be `MyModule.RawFoo -> Type`.

Effect:

Inserts Coq code to support a datatype invariant for the type passed as `qid`, which we will henceforth refer to here as `<ty_qid>`.

In particular, creates a sigma type that pairs a value of type `<ty_qid>` with a proof that it satisfies the invariant. In the translated Coq file, the type `<ty_qid>` will now denote this sigma type. The original type (which was denoted by `<ty_qid>`) will be renamed by prefixing "Raw" in front of the original unqualified name (e.g. `MyModule.Foo` becomes `MyModule.RawFoo`).

Also creates notation that acts as a constructor for this new sigma type. Internally, this notation uses `existT`. Note that the last argument to `existT` is the proof part of the sigma type. Thus, wherever this notation is used to construct a value of the sigma type, the corresponding proof must be provided. Coq's Program mode automatically unwarps sigma types and generates the necessary proof obligations (see below).

The original constructor will be renamed by adding the suffix `_Raw`.

The definitions specified in `useSigmaType` will use the sigma type instead of the original type. Thus, we need a way to provide the proof part of the sigma type. To this end, hs-to-coq will re-define these definitions using the `Program` keyword. As a result, when Coq encounters these definitions, it will enter Program mode. `Program` mode automatically applies or unwraps sigma types, which may generate proof obligations for the user.

You **MUST** use an *obligations edit* to supply the proofs; see the example below.

Example:

Suppose we create a Haskell module representing a counter, as shown below. We say that a Counter is *valid* if its internal integer is non-negative.

Below is the Haskell code:

```
module Counter(Counter, zeroCounter, inc, dec, isZero) where

data Counter = MkC Int
    deriving Show

zeroCounter :: Counter
zeroCounter = MkC 0

inc :: Counter -> Counter
inc (MkC x) = MkC (x+1)

dec :: Counter -> Counter
dec (MkC x) = if x > 0 then MkC (x - 1) else MkC 0

isZero :: Counter -> Bool
```

(continues on next page)

(continued from previous page)

```
isZero (MkC x) = x == 0

valid :: Counter -> Bool
valid (MkC x) = x >= 0
```

Notice that any `Counter` created using the public API has the property that it is non-negative: `zeroCounter` is non-negative, and given a non-negative `Counter`, the public functions defined on `Counter` type preserve the fact that the counter is non-negative.

In Coq, we can formalize the invariant that the counter is non-negative with the help of an `invariant edit`.

Here is the edits file:

```
add invariant {
  module = Counter,
  qid = Counter.Counter,
  tyVars = [],
  constructor = MkC,
  useSigmaType = [Counter.zeroCounter, Counter.inc, Counter.dec],
  invariant = Definition Counter.NonNegInv : (Counter.RawCounter -> Type)
    := fun x => valid x = true.
  }

  promote Counter.valid  # promote valid from the term-level to the type-level

  obligations Counter.zeroCounter admit
  obligations Counter.inc admit
  obligations Counter.dec admit
```

Note that we have used a `promote` edit to lift the definition of `valid` from the term level to the type level. We need to do this because `NonNegInv`, which is in the type level, references `valid`. We have also provided (trivial) proof obligations for the definitions in the list passed to `useSigmaType`.

Here is the Coq output (relevant parts, cleaned up and with added comments):

```
(* Converted type declarations: *)

(* The original Counter datatype, now renamed to RawCounter.
   Notice that the constructor MkC has been renamed to MkC_Raw. *)
Inductive RawCounter : Type := | MkC_Raw : GHC.Num.Int -> RawCounter.

(* The definition of validity. Note that this came directly
   from the Haskell source. It is not suitable as an invariant,
   because the return type is not Type. *)
Definition valid : RawCounter -> bool :=
  fun '(MkC_Raw x) => x GHC.Base.>= #0.

(* The invariant. Notice the input has type RawCounter. *)
Definition NonNegInv : RawCounter -> Type :=
  fun x => valid x = true.

(* The sigma type. *)
Definition Counter : Type :=
  { x : RawCounter & NonNegInv x }.

(* The "constructor" for the sigma type. The last argument
   to @existT is the proof part of the sigma type. *)
```

(continues on next page)

(continued from previous page)

```
Local Notation MkC y :=
  (@existT _ _ (MkC_Raw y) _).

(* Converted value declarations: *)

Program Definition zeroCounter : Counter :=
          MkC #0.
Admit Obligations.

Program Definition inc : Counter -> Counter :=
          fun '(MkC x) => MkC (x GHC.Num.+ #1).
Admit Obligations.

Program Definition dec : Counter -> Counter :=
          fun '(MkC x) =>
            if Bool.Sumbool.sumbool_of_bool (x GHC.Base.> #0)
            then MkC (x GHC.Num.- #1)
            else MkC #0.
Admit Obligations.

Definition isZero : RawCounter -> bool :=
  fun '(MkC_Raw x) => x GHC.Base.== #0.
```

Notice that some renaming has occurred: `Counter` has been renamed to `RawCounter` and `MkC` has been renamed `MkC_Raw`.

Note also that `valid` has been promoted to the type level. This is necessary, because `NonNegInv` refers to `valid`.

Finally, note the use of `Program` before the definitions of `zeroCounter`, `inc`, and `dec`, and observe that these definitions use `Counter` and `MkC`, while `isZero` uses `RawCounter` and `MkC_Raw`.

3.9 Meta-edits

3.9.1 `in` edit - restrict scope of an edit to a single definition

Format:

in *qualified_name edit*

Effect:

This is a meta-edit: The given edit is only applied during the translation of the given definition. This is most useful to rename or rewrite only within a specific function, or to give termination arguments to local functions.

While all edits are allowed, not all edits are useful when localized.

Examples:

```
in SrcLoc.Ord__RealSrcLoc_op_zl__ rewrite forall, SrcLoc.Ord__RealSrcLoc_compare␣
↪= GHC.Base.compare
in Util.exactLog2 termination pow2 deferred
```

3.9.2 `except in` edit - exclude definitions from an edit

Format:

except in *qualified_names edit*

Effect:

This is essentially the opposite of an `in` edit. The given edit is applied as normal, except during the translation of the given definition(s). This is most useful to rename or rewrite a definition everywhere except within one or more functions. In addition, it can be used when there is a local function with the same name in multiple definitions, in order to give termination arguments to all occurrences of that local function except those in the specified definitions.

As with `in` edits, while all edits are allowed, for many edits it doesn't make sense to apply `except in`.

Examples:

```
except in SrcLoc.Ord__RealSrcLoc_op_zl__ rewrite forall, SrcLoc.Ord__RealSrcLoc_
↪compare = GHC.Base.compare
except in Util.exactLog2 termination pow2 deferred
except in ModuleName.def_1 skip case pattern _

# Multiple qualified names are separated with commas
except in ModuleName.def_1, ModuleName.def_2 rename type GHC.Types.[] = list
```

Caveats:

This edit does not currently work with data type definitions.

3.10 Deprecated edits

3.10.1 `add scope`

Format:

add scope *scope* **for** *place qualified_name*

Effect:

Examples:

3.10.2 `type synonym`

Format:

type synonym *name* **:->** *name*

Effect:

Examples:

CHAPTER 4

Identifiers and Notation

Most Haskell names can be reused as Coq names (fully qualified). However, due to differences in parsing and keywords, `hs-to-coq` must sometimes modify the generated identifiers.

4.1 Coq keywords

The following Coq keywords are automatically translated with an extra _ following them:

```
Set Type Prop fun fix forall return mod match as
cons pair nil for is with left right exists
```

4.2 Operators

Coq does not allow the definition of identifiers composed with punctuation.

To name these identifiers, `hs-to-coq` uses GHC's z-encoding to give textual names to operators. These textual operator names are preceded by `op_` and followed by two underscores.

For example, the Haskell identifier == is translated to the Coq identifier `GHC.Base.op_zeze__`

4.3 Notation

Nevertheless, we want users to be able to use the operator syntax, (e.g. ==) in the output. Therefore, `hs-to-coq` defines appropriate Coq syntax for it:

```
Notation "'_==_'" := (op_zeze__).
Infix "==" := (op_zeze__) (no associativity, at level 70).
```

This works fine within the given module (here: `GHC.Base`), and within every module that imports `GHC.Base`. But in general, `hs-to-coq` does not *import* modules (it only *requires* them, using `Require GHC.Base.`), so the notation is not visible.

Therefore, the translated module will contain a `Notations` submodule at the end:

```
Module Notations.
Infix "GHC.Base.==" := (op_zeze__) (no associativity, at level 70).
Notation "'_GHC.Base.==_'" := (op_zeze__).
End Notations.
```

When a module needs to be required, and any of the imported names look like operators, then `hs-to-coq` imports the the contained `Notations` module:

```
Require GHC.Base.
Import GHC.Base.Notation.
```

This brings the "qualified operators" into scope.

In manually written modules (e.g. those containing the proofs), the user can choose to follow that example, and use all names and operators qualified, or they can `Require Import GHC.Base`, and use all names and operators unqualified, and ignore the `Notations` module.

In the rare case that you need to define operators in the preamble, then you have to manually add the `Notations` as shown here. You put the definition of the syntax for the qualified in a nested module `ManualNotation` and use the `manual notation` edit to make `hs-to-coq` include the `ManualNotation` module in the generated `Notation` module.

4.4 String Notation

Since Coq 8.9, string syntax notations are now available only when using `Import` of libraries and not merely `Require`. The Coq files generated by `hs-to-coq` will no longer offer these notations by default, either, but can be enabled by importing `GHC.Base.Notation`, `GHC.Err`, or the standard Coq libraries. `hs-to-coq` cunrrently cannot automatically import these notations even if the translated code uses them.

CHAPTER 5

Interface files

When translating a module *Foo* that uses a type class or algebraic data type from another file *Bar*, then *hs-to-coq* needs to know some information about these types. Therefore, when it creates *Bar.v*, it also writes an *interface file Bar.h2ci*. This interface file is loaded on demand during the translation of *Foo*, when and if it needs the information about *Bar*.

Some notes about interface file:

- You need to pass *–iface-dir foo/* to make *hs-to-coq* search for interface files in *foo/*. This flag can be used multiple times. Usually, you will at least passt *–iface-dir path/to/base –iface-dir output/* where *output/* is the argument to *-o*.
- When it cannot find an interface file, *hs-to-coq* complains loudly (but still produces output). It is expected that the user will fix the problem, either by processing the dependent-upon file first, or by skipping the offending declarations.
- *hs-to-coq* can help with figuring out the right dependency order. If you pass *–dependency-dir deps* to it, it will create a file *deps/Foo.mk* after processing module *Foo*. This will, in *Makefile* syntax, list all read interface files as dependencies of *Foo.v*, ensuring that from now on all files are built in the right order.
- Skipping instances prevents hs-to-coq from trying to load the interface files of the class'es module.
- Coq types as well as information about the type classes *Eq* and *Ord* are hard-coded in *src/lib/HsToCoq/ConvertHaskell/BuiltIn.hs*.
- When you have a manual file that defines types or type classes, you may have to create a faux interface files. Simply create a text file that is an valid empty yaml file (e.g. '{}').

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index

Symbols

-iface-dir <dir>
    command line option, 5
-permissive
    command line option, 6
-strict
    command line option, 6
-P
    command line option, 6
-S
    command line option, 6
-e <editfile>
    command line option, 5
-m <midamble-file>
    command line option, 6
-o <dir>
    command line option, 5
-p <preamble-file>
    command line option, 6

A

add scope, edit, 30
add, edit, 15
alias module, edit, 17
axiomatize definition, edit, 14
axiomatize module, edit, 13
axiomatize original module name, edit, 13

C

class kinds, edit, 19
coinductive, edit, 21
collapse let, edit, 18
command line option
    -iface-dir <dir>, 5
    -permissive, 6
    -strict, 6
    -P, 6
    -S, 6
    -e <editfile>, 5
    -m <midamble-file>, 6
    -o <dir>, 5
    -p <preamble-file>, 6
corecursive, termination argument, 22

D

data kinds, edit, 18
deferred, termination argument, 23
delete unused type variables, edit, 19

E

except in, edit, 30

I

import, edit, 15
in, edit, 29
inline mutual, edit, 23
invariant, edit, 26

M

manual notation, edit, 21
measure, termination argument, 22

O

obligations, edit, 23
order, edit, 20

P

polykinds, edit, 19
polyrec, edit, 21
promote, edit, 20

R

redefine, edit, 17
rename module, edit, 16
rename type, edit, 16
rename value, edit, 16
rewrite, edit, 17

S

set type, edit, 21
skip case pattern, edit, 12
skip class edit, 10
skip constructor, edit, 10
skip equation, edit, 11
skip method, edit, 11
skip module, edit, 13
skip, edit, 10
struct, termination argument, 22

T

termination, edit, 22
type synonym, edit, 30

U

unaxiomatize definition, edit, 14

W

wf, termination argument, 22