The Monad.Reader/Issue2 /EternalCompatibilityInTheory

From HaskellWiki

< The Monad.Reader | Issue2

The ECT Module Versioning Protocol

by Sven Moritz Hallberg for The Monad.Reader Issue Two
Date(2005-04-30T18:39:55Z)

Abstract. This article proposes a protocol (to be followed by the programmer) for isolating versions of Haskell modules, eliminating a large source of bit-rot in depending programs. The protocol consists of few simple rules. No language extensions are required and the scheme is immediately usable. If followed, the result is *Eternal Compatibility in Theory*.

Because of its simplicity, ECT is not restricted to Haskell.

Contents

1 Introduction

Every programmer knows the value of modularity. In particular, our languages have got *module systems* to support it, by decoupling code. However, much of the decoupling is

lost again when the flux of interfaces over time is considered.

As a program module evolves, functions and other elements are added to, removed from, and changed in its interface. It is clear that programs importing the module (its *dependants*) will not be compatible with all versions. At least, each program is compatible with one version, the one the author originally used, and usually a few ones before and after that. But if a program is not continuously updated, with time, chances rise dramatically that one of its dependencies as installed on a given host system will be incompatible. Alas, the program cannot be used. This effect comprises a major source of *bit rot*.

To avoid such a situation, the programmer importing a module has several options, all of which are unsatisfactory. Let a module P import A, a *dependency*.

1. Avoid the dependency altogether.
    1. Duplicate the functionality. If the functionality provided by A is simple, it might be possible to just replicate it with own code. This of course somewhat counters the purpose of creating module A in the first place. On a large scale, it results in many functions being duplicated by different individuals.
    2. Use standard functions instead. Instead of A, use a similar module from the standard library. Of course, the standard library does not provide everything. Also, even the standard library can become incompatible over time, though it usually takes longer.
2. Internalize the dependency. A "local" copy of A is packaged and distributed with P. This scenario relies on the host Haskell system to prefer the bundled definition of A to any others it might have available. Because that is the usual behaviour, this strategy generally puts the programmer on the safe side but has the disadvantage of locking out bug-fixes and other (compatible) improvements to the original module A. It also increases the overall size of the package for P so is not desirable for many (or) large dependencies.
3. Continuously update P to stay compatible. While continuous updates are certainly desirable, everyone's capacity is limited and in reality, no program can stay up-to-date forever. Also, this strategy always assumes the latest version of A to be available at the host. Updating a library to the latest version runs the risk of breaking other dependants requiring the older version.

Notice that option 1.2 is quite attractive. The standard library on a host is often reasonably up-to-date (though not always!) and, most importantly, widely available. As a result of these points, a large standard library is usually desired and welcomed. But that also brings large issues with it. A standard library should fit everyone, which is impossible to achieve and very hard to approximate well. Incompatible changes (improvements!) are very hard, if not impossible, to introduce in a standard library, since they will likely break many dependants, *especially* if the library is large.

So, in summary, we actually see two orthogonal problems when importing an external module:

1. *Availability:* Will A be available at the host site?
2. *Compatibility:* Will the version of A available at the host be compatible with P?

Solving issue 1, the easy and wide-spread distribution of Haskell libraries, is the combined goal of the Cabal (http://haskell.org/cabal/) and Hackage (http://haskell.org/cabal/hackage/) projects. Note, however, that Cabal does not solve issue 2, as will be discussed below.

As mentioned in the abstract, the contribution of this article is a protocol, dubbed *ECT*, by which it is possible for the programmer of the *dependency*, `A`, to solve, for his users, and with respect to `A`, issue 2. I.e. the complete elimination of concerns that `P` might break because of incompatibilities with `A`.

2 Solving the Problem by Removing the Problem

Recall the issue at hand. Interface changes break dependants. As elaborated above, the importer can do nothing against that, and in fact, neither can the dependency author. *Interface changes break dependants.*

Thence the solution is painfully obvious: *Stop interface changes.*

In ECT, the module `A` is split into separate submodules `A_1`, ..., `A_`*n* at version boundaries. The numbers are in chronological order,where *n* denotes the latest version and contains the current implementation. Whenever the interface changes in any way, the number is incremented, creating a new module. The original module `A` is retained as a short-hand (think re-export) for "the latest version".

If the importer `P` uses one of the specific, numbered modules, compatibility is guaranteed, since (by assumption) these modules have *static* interfaces. But because `A` is still present, using the number is not required. It would, after all, be quite tiresome to look up a version number for each module one wants to import.

In fact, `A` can be used to notify the programmer of the number to use via a deprecation pragma. `A` would look like the following.

```
module A
    {-# DEPRECATED "This is actually A_5." #-}
    (module A_5)
    where
import A_5
```

So the user, after first blindly importing module `A`, can conveniently add the number after compilation, then knowing that `A_5` is the actual version the program was written against.

The last remark above illustrates an important point. In any scheme to ensure compatibility of modules with their imports, there *must* be, for each module for each import, a specification of which interface version the module is referring to. And precisely that is what importing one of the numbered modules means.

"*I'm importing module* `A`*, referring to version 5 of its interface.*"

3 The Protocol

When first thinking through what my idea of module version management would be, I found I had only very vague ideas of what I actually meant by "module", "version", etc.. So let me start the formal description of the ECT versioning protocol by introducing some **Definitions.**

Module definition.
: A piece of code defining a Haskell module, as described in the Haskell Report, is called a *module definition*. Each module definition has an associated *module identifier*, e.g. `System.Console.Cmdline.Pesco` (which may be implicit in the case of Main).

Revision.
: Let *x* be a module definition and *X* be its associated module identifier. We call *x* a *revision* of *X*.

Module.
: In plain English, the module named *X* is defined as the set of all revisions of *X*. More formally, let *x* be a revision of *X*, *y* be a revision of *Y*, and call *x* and *y related* iff *X*=*Y*. We then call the equivalence classes of this relation *modules* and their associated identifiers - unique within the class - the modules' *names*.

Interface.
: The set of entities (functions, classes, instances, etc.) exported from a module is called its *interface*.

Compatibility.
: Two revisions are called *(mutually) compatible*, if their interfaces are equal and all exported entities have the same semantics.

Version.
: The equivalence classes of the compatibility relation, restricted to a single module *X*, are called *versions* of *X*.

Now, say you are implementing a module `A`. To fully conform to the protocol with respect to `A`, simply ensure the following invariants.

Let all versions of `A` be numbered from 1 to *n*.

1. For each *i*=1,...,*n* there exists a module `A_`*i* whose only version is equal to version *i* of `A`.
2. Version *n* contains the latest revision of `A`.
3. Module `A` is deprecated by means of a DEPRECATED pragma, with a message that `A_`*n* should be imported instead.

So for example, say you are writing `A` from scratch. Starting with version number 1, implement the first revision as `A_1`.

```
module A_1 where
```

```
foo = "foo"
```

Define module `A` to re-export `A_1` and include a DEPRECATED pragma that at the same time

1. reminds the user to explicify her import and
2. informs her of the current version number.

```
module A
    {-# DEPRECATED "Import A_1." #-}
    (module A_1)
    where
import A_1
```

When you modify the definition of `A_1`, consider any changes to its interface, including type signatures (of course) and semantics. If *anything* has changed, i.e. added, removed, or modified, a new version has formed, so increment the version number and move this revision to a new module, i.e. `A_2`. Change every occurrence of `A_1` in the definition of `A` to `A_2`. Either re-implement `A_1` in terms of `A_2` or keep the old revision in place.

If there were no changes whatsoever, keep the new definition a revision of `A_1`.

Repeat the above for every new revision.

Ideally, the versions of `A` will form a chain of adapters and re-exports, where every version is implemented in terms of a later one, `A_`*n* containing the actual codebase. But of course it is also possible to completely leave some version's old implementation in place, if the next version differs very significantly, for instance.

Now consider the case of switching an existing module, which has already gone through several versions, to ECT. To attain "full compliance", as per the above definition, every old version of the module would have to be identified and split into its own module.

Certainly, it is desirable to allow weaker invariants than given above, to allow, for instance all the pre-ECT versions of a module `A` to be grouped in `A_0` or similar. Or, a module owner might want to keep `A` locked on the last pre-ECT version to remain backwards-compatible. Then `A_0` could take the role of the "latest version" short-cut module.

In the end, the only property essential to the compatibility guarantees to be facilitated by ECT is the following.

> Let all versions of `A` be numbered from 1 to *n*. For each *i*=1,...,n, if there exists a module `A_`*i*, it has only one version which is equal to version *i* of `A`.

Any module satisfying the above shall be called *weakly conforming* to the ECT versioning scheme.

3.1 Bugs, Behaviour, Semantics

The above explanations have ignored the very likely possibility of program bugs. The common definition of a *bug* is a programmer error resulting in undesired behaviour of the program.

The question - in light of the earlier definition of compatibility - is, whether this unintended behaviour can or should be considered as part of the program's semantics. If a bug in some revision of a module is part of its semantics, the bug-fixed revision would be considered part of a new version and prompt the creation of a separate module. Clearly, this is usually undesirable.

To answer this question in detail would largely be a philosophical undertaking on *intent*, *meaning*, *collective vs. individual perception* etc.. So, for this discussion, let the following guidelines suffice.

An actual problem only arises out of a bug if some user mistakes the unintended behaviour for intended semantics and uses it, effectively creating a *dependency on the bug*. Then, if the bug is fixed in a later revision, that user's program will break.

Luckily, it will often be clear that a bug is not part of the intended semantics of an interface. If it is indeed safe to assume that no users will accidentally mistake the bug for a feature, it is also safe to leave the version numbering as it is.

If, however, it is unclear whether any importers could come to rely on the erroneous behaviour, there is also a simple solution: split the buggy module. So far, we have assumed module version numbers to be integers. When splitting a module due to a bug-fix, also "split" its version number into two rationals "close" to it. Let me explain by means of an example. If a dubious bug is found in version 4, create two new modules with versions 3.9 and 4.1 (add/subtract 0.1). Since the decimal point cannot be used in the module name, substitute an underscore yielding `A_3_9` and `A_4_1`.

Put the original module definition, with the bug, into the lower module, `A_3_9`. This makes it clear that the buggy version is semantically "before" what version 4 should have been. Likewise, the bug-fixed version goes into the higher module, `A_4_1`.

Now deprecate the old module `A_4` with a warning about the bug and let it re-export `A_3_9` to preserve compatibility.

It's good the rationals are dense, for thus we can split each module an arbitrary number of times. For instance, if another bug was discovered later in `A_3_9`, split its rational version number 3.9 again at the last digit: 3.9 is moved to 3.89, 3.9 itself becomes a deprecated re-export of 3.89, and 3.91 contains the bug-fix.

Please note that this fractional numbering has no relation to any second- or third-level version components in say, the package containing these modules. For instance, if `A` is released in a corresponding package `a`, `a-4.2` could refer to the second revision of the fourth release, consisting, using the example from above, of `A` and `A_1` through `A_4`, as well

as `A_3_9` and `A_4_1` (first revision), and `A_3_89` and `A_3_91` (second revision).

Finally, let me give an example for an appropriate bug deprecation message.

```
{-# DEPRECATED
    "A_4 is buggy: The foo function returns "foo" \
        \which should have been "Foo".\n\
    \Import A_4_1 for the correct behaviour or \
        \A_3_9 if you rely on the bug."
#-}
```

Notice that the user is informed of

1. how to determine whether she is relying on the bug and
2. which module to import in each case.

3.2 Tests

Consider the following likely situation. You are the author of module `A` which is currently at version 1 and you have a bunch of unit tests for `A_1`. You create a new version, `A_2`, which is backwards-compatible, i.e. the interface of `A_1` is a subset of `A_2`'s and the semantics of this subset are the same for both modules. In this case, any tests for `A_1` are also good for `A_2`, and, if you replace `A_1` with a partial re-export of `A_2`, it would be sufficient to run only the `A_2` tests in order to test both versions. Also, if any tests for the `A_1` subset of `A_2` were added to `A_2`, they would automatically apply to `A_1` as well.

In general, tests for functions that are re-exports can probably move up to the actual implementation safely. In the hypothetical case that the re-export is changed later, it just has to be remembered to pull down copies of the tests again.

For instance, if module `A_1` is defined as

```
module A_1
    (foo, bar)
    where
import A_2 (foo)
bar = "barr"
```

and `A_2` contains

```
module A_2
    (foo, bar, baz)
    where
foo = "foo"
bar = "bar"
baz = "baz"
```

the tests for `A_1` would consist only of tests for `bar` while the tests for `A_2` would contain tests for all three functions, `foo`, `bar`, `baz`.

3.3 Documentation

Similar remarks as for tests apply to documentation. If you use an autogenerator like Haddock, everything will be fine automatically: Re-exports inherit their documentation from the original. If you maintain documentation separately, the documentation must be split by hand. But keep in mind that nothing forces you to expend more work than you would without following ECT.

I.e. please consider the ECT protocol even if you do not want to spend time on the old documentation (or tests, for that matter)!

This concludes the discussion of the developer's side of the protocol.

3.4 On the User's Side

Luckily, no extra rules at all apply to the user of an ECT module. The user does not need to know about ECT at all. She can just import the unnumbered module as ever. Then, if she notices the deprecation message, and hopefully follows it, it is just all the better.

Of course, everyone will quickly get the hang of it, so in general, the procedure is as follows:

1. Import the unnumbered module.
2. Compile.
3. Observe the deprecation warnings and change all imports to include the given version number.
4. *Relax.* You are safe. ;)

4 Relation to the Cabal

The Cabal also includes provisions for versioning and dependency tracking. Keep in mind, however, that it is concerned with *packages* instead of single modules. The mechanism for dependency tracking consists of listing the dependencies (packages) in the package description, along with "version annotations", for instance `foo > 1.2` would mean "depends on package foo, any version later than 1.2 is compatible".

I shall demonstrate that the ECT protocol does not compete with Cabal's functionality, but, together with what the Cabal and Hackage offer, forms a necessary part of a very valuable whole, i.e. the solution to both problems presented in the introduction, availability and compatibility.

Let me first argue that Cabal (at least in its current form) is not suited to solve the problem of dependency tracking, i.e. compatibility. I will contrast that with the alternate scenario, ECT ensuring compatibility, and conclude that Cabal's dependency system in

fact fills an important need in that case.

4.1 Could Cabal Be Used Instead?

idea: dependency-tracking at the package level
problem: unknown when compatibility ends, e.g. 1.2 <= foo < 2.0
compatibility information is available only at the dependency!
importer knows exactly one version that is compatible.

This scenario implies dependency-tracking at the level of packages. As already described, a Cabal package can list a number of other packages as dependencies. However, there is a problem: A package author may know the *first* compatible revision of a dependency, but he has no way to know the last. For example, if I depend on a package `foo` and know that `foo-1.2` is compatible, I can specify `foo >= 1.2` as a dependency. If however, say, `foo-2.0` turns out not to be backwards-compatible, I should have written something like `1.2 <= foo < 2.0`, which I could not have known at the time.

In general, the compatibility information is available on the side of the dependency, but not at the site of import! The importer knows a priori only one version that is compatible, and that is the one he used.

A possible solution would be to establish a convention that Cabal packages are assumed backwards-compatible until the next major version, or similar. This scenario gives rise to a number of other problems, though.

1. It forces an interpretation onto package versions which might not be accurate. Many software packages are numbered according to largely arbitrary schemes.
2. Assuming the convention in question, consider the following consequence. It would be impossible for a single package to use two different major versions of the same dependency. A standard example would be a program to convert data from the old version into something suitable for the new.
3. Still assuming the convention in question, There are the following two alternative consequences:
    1. New major versions are released infrequently, slowing down interface evolution. For example, the programmer would hesitate to change an oddly-named function. Contrast this quickly with the ECT alternative: Freely change the function name (or anything else!) in the next revision, give it a new number, replace the old version by a trivial adapter, and you are done without having broken any of your dependants.
    2. New major versions are released frequently, to facilitate rapid interface evolution. This possible world would quickly become populated with many (to be assumed) incompatible versions of a package, many of which would probably find their way onto a host system. In the end, a host would for every package contain many different versions, as required for different dependants.

In summary, it should be clear that relying on Cabal to provide any kind of compatibility guarantees would be unpleasant at least. Luckily, there is a perfect alternative.

4.2 Cabal Provides, ECT Guarantees

but: cabal versions establish chronological ordering.
=> dependency-tracking with ECT, cabal to ensure availability
compatibility info encoded by dependency through re-exports and adapters
import specifies the one version known to be compatible and the point in *time* (i.e. package version!) it is available!

There is one thing that *is* safe to assume about package version numbers: They reflect the chronological order of the releases. Now note that if a package follows the ECT protocol, every one of its releases is backwards-compatible and a dependency of the form `foo >= 1.2` becomes meaningful!

The compatibility information is encoded where it originates, in the dependency itself, by means of the "back issues" of a module, be they implemented as simple re-exports, adapters, or the original old code. The import specifies only what is known there, that is the version the program was originally written against. And if Cabal is used, the dependant package can specify from which package release (i.e. point in time!) on the required version(s) are available, which can then be fetched automatically -- look for next issue's article on cabal-get!

A package can also seamlessly use multiple versions of the same module, just by depending on a sufficiently late release.

Finally, package authors have complete freedom over how they chose their package version numbers, provided they keep them in ascending chronological order. ;)

5 Conclusion and Call for Adoption

I have introduced a simple and unintrusive protocol that, when followed by a module implementor, can, as the name implies, theoretically guarantee eternal compatibility to his users. This eliminates a huge source of bit-rot. Also, the user himself needs to know next to nothing about the protocol.

ECT works with plain Haskell 98, relying only on basic features of the module system. It coexists synergetically with Cabal. In fact, when both are used together, they enable the formation of a richly interconnected network of Haskell modules, maximizing code reuse among them and enabling their rapid evolution.

In the long run, I envision a world where I can stumble across some module on the Net, immediately import it in my code, and, on the stroke of a single command, have my system locate, fetch, compile, and install that module, dropping me at an interactive prompt with the new import available! All of the above *without* having to worry about whether or how to bundle that module with my code, prompted by questions of its availability and compatibility on my users' hosts.

Standing next to Cabal, Hackage, and cabal-get, the ECT protocol is my contribution to the above, so here is my plea:

If you are a module implementor, please consider using the ECT versioning

scheme, for *Eternal Compatibility in Theory*!

I think it would be great if we could solve the problem of "creeping incompatibility", whether by my idea or otherwise. Please let me know of your experience should you try it out!

6 A Real-World Example

I've modified my own command line parsing module to use the ECT scheme. As of this writing (2005-05-02) the latest proper release (package version 1.1) contains only interface version 1 but 2 is practically finished, just waiting for the last documentation updates. Look in the public darcs repository for the latest version.

You can find both the release package, seperate documentation, and the darcs repository at

```
http://www.scannedinavian.org/~pesco/
```

under the title "System.Console.Cmdline.Pesco".

7 Acknowledgements

Though I had spent some thought about compatibility for myself, I owe thanks to Alexander Jacobson who, with a message to the Haskell mailing list, originally caused me to eventually turn those thoughts into something concrete.

Subsequent discussion with Lemmih, musasabi, and probably others I've forgotten on #haskell and the mailing list provided important enlightenment, especially that no extensions to the language would be required.

I also owe inspiration to the designers of GNU libtool, who use some similar ideas for the dependency managment of shared libraries.

Finally, I was only lately reminded by Eike Scholz (cptchaos on #h) that Bjarne Stroustrup actually described a very similar scheme to mine in "The C++ Programming Language", which I had completely forgotten about. Thanks Bjarne, I'm sure that chapter left some subconcious seeds. P