Refactor program with HLint suggestions

Matthew Pickering

Industrial strength refactoring tools are something that the Haskell community as a whole has desired for a number of years. As a result a number of projects have spawned to provide varying levels of abstraction to the refactoring process. Recently, `ghc-exactprint` promises to provide a robust foundation for refactoring. This summer I propose to use `ghc-exactprint` and HaRe to add a `--refactor` flag to HLint which will automatically apply relevant suggestions.

`ghc-exactprint`

> I'd really like HLint to have an "automatically replace" flag, but I want to do it preserving whitespace and style, which is quite hard.
>
> Neil Mitchell

Whilst the eye-catching end-goal of any refactoring tool is the transformation another key tenet of refactoring is that the changes don't affect any irrelevant parts of the program. It follows that if no transformation is applied then one should expect *no* changes to the source file. This is the challenge which `ghc-exactprint` attempts to solve.

`ghc-exactprint` works in two phases. The first transforms the GHC AST such that all absolute positions are replaced with relative positions. The second printing phase performs the inverse transformation and prints the desired output. Now to perform transformations, you only need to specify where your elements should be located relative to each other and the preceding element. This makes performing transformations whilst retaining layout much easier than ever before.

API Annotations

This approach is only feasible now[1] as before GHC 7.10 the parser discarded all information about the location of keywords. This made tools such as

[1]Previously HaRe used an extremely brittle mechanism which performed transformations on the AST but printed using the token stream produced by the lexer. This meant that performing any transformation it was necessary to update both. (see Li 2006, 57 - 59)

haskell-src-exts necessary if you wanted to precisely print a source file. Now, whilst parsing, the GHC parser records the information of these keywords in a map indexed by the annotation type (AnnKeywordId) and the SrcSpan of the element it came from. This information can then be used to recreate the original document with sufficient care.

Using this new and powerful machinery writing robust mechanical transformations is much easier that before.

HLint

HLint is a very widely used linting tool. It suggests a wide array of refactorings, most of which are trivial for a programmer to perform. The README on the HLint homepage talks briefly about the possibility of refactoring.

> If you want to automatically apply suggestions, … , there are a number of reasons that HLint itself doesn't have an option to automatically apply suggestions:
>
> The underlying Haskell parser library makes it hard to modify the code, then print it similarly to the original. Sometimes multiple transformations may apply. After applying one transformation, others that were otherwise suggested may become inappropriate. I am intending to develop such a feature, but the above reasons mean it is likely to take some time.

I will firstly address the feasibility of using HaRe to perform these refactorings and then the possibility that refactorings may interfere with each other.

Flavours of refactoring

I categorise the types of refactorings found in HLint into three different categories. A full classification is provided in Appendix A.

(1) Direct substitutions - these are listed in a source file and then a naive search of the AST is performed to check if there are any matches. Below is a simple example from the default definitions.

```
error = putStrLn (show x) ==> print x
^       ^                     ^
| Severity                    |
        | Expression to replace
                              | Expression to insert
```

(2) Simple transformations - Other simple transformations are achieved by traversing over the HSE AST. The simplest is the one which checks for a redundant dollar sign.[2]

```
[msg x y | InfixApp _ a d b <- [x], opExp d ~= "$"
          ,let y = App an a b, not $ needBracket 0 y a
          , not $ needBracket 1 y b]
```

(3) Complex type sensitive transformations - The most complicated transformation in HLint is the check for duplicated expressions. This is performed by maintaining a map of previously seen expressions and checking whether the current expression appears in the map. Details can be seen in `src/Hint/Duplicate.hs`.

I predict that automating both (1) and (2) are easy to achieve with further development of `ghc-exactprint` and HaRe. It has already been proved that a combination of these programs can be used to perform these kinds of simple substitutions.

(3) requires more care to avoid name clashes but would be achievable using a type-aware refactoring. Additionally, HLint does not suggest how to avoid the code duplication (as it does not know any semantic information) which means that implementing this suggestion would require additional machinery only found in HaRe.[3]

Conflicting Suggestions

There are certain situations where HLint suggests refactorings which would conflict with each other. Consider the following simple example[4]

```
example = f $ (x y)
```

which generates the following warnings.

```
example.hs:1:11: Warning: Redundant bracket
Found:
  f $ (x y)
Why not:
  f $ x y

example.hs:1:11: Warning: Redundant $
Found:
```

[2] A dollar sign (`$`) is redundant when both arguments do not require parentheses.
[3] Such as a fresh supply of variable names and the ability to check for name capture.
[4] As suggested by Neil Mitchell.

```
  f $ (x y)
Why not:
  f (x y)

2 suggestions
```

Clearly as the suggestions overlap each other, choosing to apply either one first results in the other becoming invalid but both reduce to an equivalent normal form.

One simple way to deal with this is to detect when two suggestions are overlapping. If so, then it can only be safe to perform one of the transformations. In general it would seem sensible to prioritise *errors* rather than the *warnings* but it would be possible explore different heuristics for this rare occurrence. Another possibility would be to provide an interface for the user to decide which suggestion they would like to apply.

Connecting HLint and HaRe

So far we have discussed how it is both possible to perform source transformations which preserve layout with `ghc-exactprint` and also the possibility of applying HLint suggestions. The final piece of the puzzle is how to link the two together.

Depending directly on HaRe

These transformations could easily be achieved by depending directly on HaRe and calling API methods in order to perform the necessary transformations. This approach would perhaps be simple but is undesirable due to HaRe's dependency on the GHC API. Said dependency is rather undesirable as it ties your users to a particular version of GHC. For a project in as wide use as HLint, this is unacceptable.

Intermediate specification format

Much like `pandoc-types` separates document synthesis from document generation. Perhaps the cleanest solution would be to separate refactoring specification from transformation. Designing a DSL for specifying refactorings without a dependency on HaRe would mean that external libraries could specify transformations before invoking HaRe and piping in their desired transformation. A typical invocation might then be as follows. This could be folded inside HLint as a system call.

```
hlint --refactor | ghc-hare
```

An obvious problem with this approach is how to specify Haskell AST elements to use in the transformations. For variable renaming, one need only provide a new name for the variable but say we want to inject more complicated expressions the problem becomes more difficult. A solution is to use the `parseExpression` endpoint exposed by GHC 7.10[5] and rely on your user to be able to at least give you a valid string representation of the construct they wanted to insert.

For example, if a user wanted to replace `concat . map f` with `concatMap` then the program may look as follows.

(1) The types of refactorings allowed are specified in a separate module.

```haskell
-- In a separate library, not depending on GHC.
data Refactor = Replace SrcSpan String | ...
```

(2) The refactoring is specified inside HLint.

```haskell
-- In HLint
findReplacement :: ... -> Refactor
findReplacement expr = Replace sspan "concatMap"
  where
    sspan :: GHC.SrcSpan
    sspan = findReplacementLocation expr
```

(3) The refactoring is interpreted inside HaRe.

```haskell
-- In HaRe
replaceExpr :: GHC.SrcSpan -> GHC.LHsExpr -> ...

refactor :: Refactor -> ...
refactor (Replace span expr) =
  let parsedExpr = parseExpression expr
    in replaceExpr span parsedExpr
```

Designing the intermediate library will be initially difficult. It isn't clear which refactoring operations should be atomic and even then there are clear overheads to performing composite refactorings without informed optimisation. For that reason I propose a very naive intermediate layer which is only intended to work between HLint and HaRe. Constructors should directly correspond to refactorings offered by HaRe as to simulate calling the API without causing a dependency on GHC.

[5] A full list of newly exposed endpoints is: `parseModule`, `parseImport`, `parseStatement`, `parseDeclaration`, `parseExpression`, `parseTypeSignature`, `parseFullStmt`, `parseStmt`, `parseIdentifier`, `parseType` and `parseHeader`.

Higher-level DSL

Coming full circle, after HaRe, Thompson and Li went on to design a second refactoring tool, this time for Erlang. The community uptake was much greater and as a result they designed such a DSL for scripting refactorings. (Li and Thompson 2012) If time permitted it would be productive to extend my proposed specification format to a more fully featured DSL which enabled users to specify their refactorings. Using Li and Thompson's work could provide a useful starting point. This design work is most likely outside the scope of this project.

Longer term vision

Propagating API changes

Similar to the idea that Roman Cheplyaka suggested at HIW 2012. With the capability to read a serialisation format, library authors could distribute a changes file which would then be used by HaRe to perform the necessary changes.

Name resolution would be an important and difficult challenge to overcome. Under the influence of `NoImplicitPrelude`, HLint already provides erroneous suggestions as the substitution mechanism performs syntactic rather than semantic matching. I think that extending HLints matching mechanism to perform semantic matching would be difficult. `haskell-src-exts` doesn't immediately resolve names, it would be necessary to use `haskell-names` to provide name resolution which in turn relies on the interface files generated by `hs-gen-interface`. These kinds of changes can lead you down a rabbit hole so are not something I plan to pursue in detail this summer.

Caching a loaded module

Applying many small disjoint operations individually could get very expensive as the source file is reloaded by HaRe on each separate transformation. A better approach would be to inspect each transformation and apply disjoint transformations without reloading the file.

Updated vim and Emacs bindings

Most Haskell refactoring tools provide bindings to both vim and Emacs, it is also important that HaRe provides such bindings. Once the HaRe API stabilises in the next year this is something I could provide towards the tail end of the summer.

Promotion

It is well-known that the secret to the success of any library is a mixture of solving a cool problem and a suitable level of promotion. HaRe has been around for a number of years but has not gained very much traction. Once it is easy to write your own transformations then I would look to write a series of blog posts demonstrating how easy Haskell program synthesis can be.[6]

[6] or maybe just how you can now simultaneously change you names of all your cost centres at once. The reader can decide which she finds more appealing.

About Me

Since successfully completing a project working on Pandoc last year I have remained involved in the Haskell community. I have continued to contribute to Pandoc through triaging bug reports and contributing infrastructure patches.

Over the winter I completed an internship at the social media startup Borders where I worked on writing slack-api and contributing to internal admin tools.

I am already intimately familiar with `ghc-exactprint` after recently completing a significant refactoring of the core machinery. This work has made me well aware of how fiddly these kind of foundational tools can be! Something to be conscious of when committing a summer to a project.

Timeline

Due to university exams, I plan to start a month late and finish a month late. This arrangement worked well last summer.

Start	End	Activity
25th May	20th June	Inactive due to university exams.
20th June	20th July	Work on `ghc-exactprint` and HaRe to provide robust transformations for simple refactorings like those found in HLint.
20th July	20th August	Design a simple intermediate library which can be used to bridge between HLint and HaRe.
20th August	20th September	Fine tune the intermediate library with experience gained in the first two months.

Deliverables

1. Solidify the relationship between HaRe and `ghc-exactprint` by providing primitive refactorings in `ghc-exactprint`.
2. A standalone refactoring DSL (provided by a separate library) based on expression replacements.
3. Modifying HLint to produce suggestions in this format.
4. Modify HaRe to understand this DSL and perform the replacements.

Appendix A - HLint refactorings classified

Kind	Module	Type	Comments
Substituting malformed pragmas	`Comment.hs`	(2)	Pragmas and comments are treated differently in the AST so it may be a little bit fiddly to transform one from the other.
Removing redundant parentheses/dollars	`Bracket.hs`	(2)	This should be straightforward.
Duplicated code blocks	`Duplicate.hs`	(3)	As previously discussed this will be difficult to get right without additional work to HaRe.
Remove unused language extensions	`Extensions.hs`	(2)	
Combining import declarations	`Import.hs`	(2)	Can be viewed as a deletion followed by an insertion.
Replace lambda functions with common library definitions	`Lambda.hs`	(2)/(1)	Very much the same flavour as straight substitution.
Replace expressions built with list constructors with sugar	`List.hs`	(2)	Care needed to correctly manage annotations.
Replace recursive functions with higher-order functions	`ListRec.hs`	(2)	Best to deal with by insertion and deletion rather than trying very hard to maintain formatting.
Direct substitutions given by a file.	`Match.hs`	(1)	Direct substitution.
Replace monadic expressions with more idiomatic counterparts.	`Monad.hs`	(2)	Direct substitution.

Kind	Module	Type	Comments
Check for camel case variable name	`Naming.hs`	(3)	Renaming requires knowledge of types but HLint currently doesn't suggest names which will clash with definitions in the module.
Checks for `OPTIONS` and `LANGUAGE` pragmas which should be expressed differently.	`Pragma.hs`	(2)	Almost direct substitution.
Structural refactorings	`Structural.hs`	(2)	
Checks to see if every usage of unsafePerformIO has a `{-# NOINLINE #-}` pragma	`Unsafe.hs`	(2)	Care will be needed to make sure to attach the pragma to the correct location.

References

Li, Huiqing. 2006. "Refactoring Haskell Programs." PhD thesis. http://kar.kent.ac.uk/14425/.

Li, Huiqing, and Simon Thompson. 2012. "A domain-specific language for scripting refactorings in Erlang." *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7212 LNCS: 501–15. doi:10.1007/978-3-642-28872-2_34.