Working with Source Plugins

Matthew Pickering
University of Bristol
United Kingdom

Nicolas Wu
Imperial College London
United Kingdom

Boldizsár Németh
Eötvös Loránd University
Hungary

Abstract

A modern compiler calculates and constructs a large amount of information about the programs it compiles. Tooling authors want to take advantage of this information in order to extend the compiler in interesting ways. Source plugins are a mechanism implemented in the Glasgow Haskell Compiler (GHC) which allow inspection and modification of programs as they pass through the compilation pipeline.

This paper is about how to write source plugins. Due to their nature—they are ways to extend the compiler—at least basic knowledge about how the compiler works is critical to designing and implementing a robust and therefore successful plugin. The goal of the paper is to equip would-be plugin authors with inspiration about what kinds of plugins they should write and most importantly with the basic techniques which should be used in order to write them.

CCS Concepts • **Software and its engineering → Functional languages**; *Source code generation*; *Preprocessors*.

Keywords tools, plugins, metaprogramming

ACM Reference Format:
Matthew Pickering, Nicolas Wu, and Boldizsár Németh. 2019. Working with Source Plugins. In *Proceedings of the 12th ACM SIGPLAN International Haskell Symposium (Haskell '19), August 22–23, 2019, Berlin, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3331545.3342599

1 Introduction

Many modern compilers are fantastic and extensible beasts: the trick is to find appropriate extension points for plugins in these systems. Plugins allow users to extend their functionality with domain-specific transformation and analysis passes that would not be appropriate in the main release.

Source plugins make GHC extensible by allowing users an easy way to modify and inspect the phases of compilation for source programs. Source plugins can affect the parser, renamer, type checker, and many other phases.

The type of a plugin *graphModules* that extracts information about the dependency structure of a module is:

```
graphModules :: [CommandLineOption]
  → ModSummary → TcGblEnv → TcM TcGblEnv
```

An invocation of *graphModules opts m tc* takes a list of options, summary information about a module, and the result of type checking the module as an argument. With this the plugin produces a potentially modified type checked module. After type checking is finished the plugin is invoked and then compilation proceeds as before. This way the user can add program elements, modify existing ones, or extract some information about a module as it is being compiled.

Plugins are packaged in a module and enabled with a command line flag. A plugin exports an identifier called *plugin* :: *Plugin*, that interacts with the rest of the system.

```
module GraphModules (plugin) where

plugin :: Plugin
plugin = defaultPlugin
  {typecheckResultAction = graphModules}
```

When the user invokes the *GraphModules* plugin it will output a dot graph of the dependency structure of their module. This is achieved by using the `-fplugin` flag to pass the module name as a parameter to GHC.

The purpose of this paper is to provide a road-map for the plugin system, and the main contributions are:

1. a description of the full interface of extension points, detailing their role in the compilation phases (Section 2),
2. a demonstration of the main tasks involved in plugin development through three worked examples (Section 3),
3. an explanation of how the plugin system can be used within the context of the rest of the system (Section 4),

This is a pragmatic paper and so in addition to these contributions we include tips and hints to writing plugins (Section 5), an overview of some interesting plugins that have been written by the community (Section 6), and a discussion of how plugins interact with other language features (Section 7).

We also include a comparison of source plugins with other metaprogramming tools implemented in GHC and similar compilers (Section 8). The plugin system interacts with much of GHC's pipeline, and is the culmination of over a decade's worth of effort from numerous contributors (Section 9).

Haskell '19, August 22–23, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6813-1/19/08…$15.00
https://doi.org/10.1145/3331545.3342599

```
parsedResultAction :: [CommandLineOption] → ModSummary → HsParsedModule → Hsc HsParsedModule          ①
renamedResultAction :: [CommandLineOption] → TcGblEnv → HsGroup GhcRn → TcM (TcGblEnv, HsGroup GhcRn) ②
typeCheckResultAction :: [CommandLineOption] → ModSummary → TcGblEnv → TcM TcGblEnv                   ③
spliceRunAction :: [CommandLineOption] → LHsExpr GhcTc → TcM (LHsExpr GhcTc)                          ④
interfaceLoadAction :: ∀lcl.[CommandLineOption] → ModIface → IfM lcl ModIface                         ⑤
```

Figure 1. Plugin extension point interface

2 Plugin Interface

Source plugins intercept different parts of the GHC compilation pipeline, allowing plugin authors to access information at almost every significant stage. Each part of the pipeline that can be intercepted is called an extension point.

A plugin is an ordinary Haskell module which exports an identifier called *plugin* :: *Plugin*. The name of the plugin is the name of the module. The value *plugin* is a record containing functions that define the different plugin actions (Figure 1).

A plugin is usually defined by overriding the *defaultPlugin* which contains identity transformations for all the passes. The source plugin mechanism extends the existing plugins mechanism with new extension points.

```
plugin :: Plugin
plugin = defaultPlugin
  {parsedResultAction = parserPlugin}

parserPlugin :: [CommandLineOption]
             → ModSummary
             → HsParsedModule
             → Hsc HsParsedModule
```

When a plugin is loaded, the compiler executes the relevant plugin action at the relevant point of compilation.

Command line options can be passed to the plugin using the `-fplugin-opt` flag. The optoins that are passed are made available to the plugin in the [*CommandLineOption*] argument where a *CommandLineOption* is just a *String*.

2.1 Extension Points

There are five new extension points which make up the API for source plugins. These are integrated into the GHC compilation pipeline [10]. These extension points correspond to an interface of five functions (Figure 1), with labels corresponding to those in the diagram of the pipeline (Figure 2).

The compilation pipeline has phases that will parse, rename, type check and desugar code before it is passed on to a lower level. Additionally, if the code contains templates, splicing of code may also occur. These phases work with various representations of the source code as they pass through the compilation pipeline: *LHsExpr GhcPs*, *LHsExpr GhcRn*, *LHsExpr GhcTc*, and *Exp*, before finally producing *CoreExpr* values, which are passed into the various extension points.

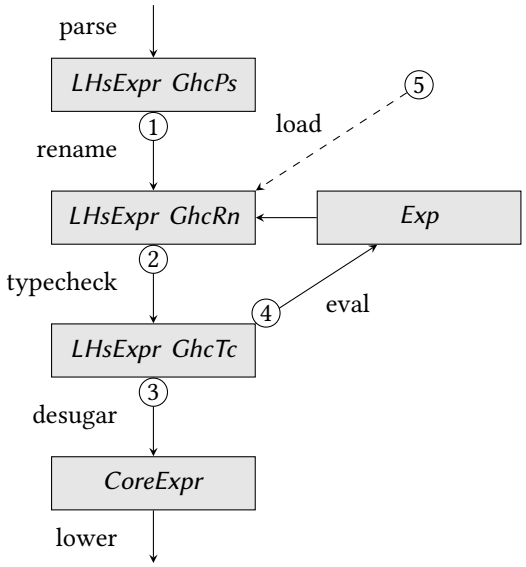

Figure 2. GHC compilation pipeline with extension points

① **Parser Plugin** The plugin can be used to inspect and modify the result of the parser. It runs in the *Hsc* monad which is limited to emitting simple errors and accessing the global environment.

② **Renamer Plugin** A renamer plugin runs after the names in each group of declarations have been resolved. Renaming and type checking are interleaved with running splices so the source program is split up into groups delimited by splices before they are renamed. After each group has been renamed the user can inspect and modify the group.

③ **Type Checker Plugin** The plugin can be used to inspect and modify the result of the type checker. This is run as the last action in the source compilation pipeline before the source language is desugared.

④ **Splice Plugin** A splice plugin is run on an expression contained in a splice before it is evaluated. The implementation should be similar to a type checker plugin but a type checker plugin runs after splicing is completed so it is too late to modify any expressions contained in a splice.

⑤ **Interface Plugin** When an import statement is resolved the interface file for that module is loaded. This plugin runs after it is loaded so that it can inspect the interface files that are being brought into scope as they are requested.

2.2 Controlling Recompilation

During the implementation of source plugins it was noticed that the recompilation check would always recompile a module if a plugin was enabled for that module. Many plugins are pure, which means that unless their input changes then the output is the same.

The solution is to introduce a new extension point to a plugin which allows the plugin author to specify how a plugin should affect recompilation.

```
pluginRecompile :: [CommandLineOption]
  → IO PluginRecompile
```

This function will be run during the recompilation check which happens at the start of every module compilation. It returns a value of the *PluginRecompile* data type.

```
data PluginRecompile = ForceRecompile
                     | NoForceRecompile
                     | MaybeRecompile Fingerprint
```

There are three different ways to specify how a plugin affects recompilation.

NoForceRecompile It doesn't contribute anything to the recompilation check. We will only recompile a module if we would normally recompile it.

ForceRecompile The module the plugin is enabled for should always be recompiled. For example, the plugin reads an externally changing source.

MaybeRecompile Computes a *Fingerprint* which we add to the recompilation check to decide whether we should recompile.

2.2.1 Library Functions

The *Plugins* interface provides some library functions for common configurations. The *impurePlugin* function returns the constant *ForceRecompile* result. This is the default for the field since it is a conservative choice that also happens to maintain backwards compatibility.

```
impurePlugin :: [CommandLineOpts]
  → IO PluginRecompile
impurePlugin _ = return ForceRecompile
```

The *purePlugin* function is useful for static analysis tools which don't modify the source program at all and just output information. Other plugins which modify the source program in a predictable manner such as the *GraphModules* plugin should also be marked as pure.

```
purePlugin :: [CommandLineOpts] → IO PluginRecompile
purePlugin _ = return NoForceRecompile
```

If you have some options which affect the output of the plugin then you might want to use the *flagRecompile* option which causes recompilation if any of the plugin flags change.

```
flagRecompile :: [CommandLineOption]
  → IO PluginRecompile
flagRecompile =
  return · MaybeRecompile · fingerprintFingerprints
    · map fingerprintString · sort
```

It is sometimes necessary to be overly conservative when specifying recompilation behaviour. For example, you can't decide on a per-module basis whether to recompile or not. Perhaps the interface could be extended with this information if users found it necessary.

3 Plugin Development

Working with the plugin interface requires an understanding of some of the internal GHC data types that represent various parts of the system. This section details the main points of interest for the development of three plugins: haskell-indexer (Section 3.1), idioms-plugin (Section 3.2), and assert-explainer (Section 3.3). These have been chosen to showcase different parts of the source plugin system.

3.1 haskell-indexer

Source plugins are useful for performing analysis of source programs. The haskell-indexer[1] plugin analyses a source file and emits information about the document's structure in the kythe[2] indexing format, which is used to provide language-agnostic code tools. This involves information about a symbol's definition, its uses and other intra-project references. The haskell-indexer plugin is an example of a plugin which doesn't modify the user's program, making it an easy first example to understand.

The haskell-indexer plugin is an example that uses a type checker plugin ③ . These plugins run after the end of type checking and have access to the final internal state of the type checker. This means that analysis plugins can inspect fully type checked bindings and other information the type checker produces.

The extension point for haskell-indexer is a type checker plugin called *indexModule*:

```
indexModule :: [CommandLineOption]
            → ModSummary
            → TcGblEnv
            → TcM TcGblEnv
```

The idea behind the plugin is to inspect the contents of *TcGblEnv*, the internal state of the type checker, summarise it, and then output the result in the kythe index format. This is repeated for each module. The kythe entries files can then be combined to produce indexing information for an entire project or an entire ecosystem.

[1] https://github.com/google/haskell-indexer
[2] https://kythe.io

The haskell-indexer plugin takes several arguments which control its output, for example, where to output the indices and which package is currently being indexed.

The *ModSummary* and *TcGblEnv* provide information about the state of the module that is currently being compiled. Most analysis plugins will inspect these data structures in order to learn information about the current module to report back to the user.

ModSummary The *ModSummary* contains meta information about the module such as the name of the module, the location of the module's source code, information about its interface files and so on. The most important field in the *ModSummary* for plugin authors is the *ms_mod* field which contains a *Module*. A *Module* contains the unit identifier for the package the module belongs to as well as the name of the current module.

```
data Module = Module { moduleUnitId :: UnitId
                     , moduleName :: ModuleName }
```

A plugin author should use this information to inform themselves of the module their plugin is currently processing if it is important for their analysis.

TcGblEnv All the information specific to the contents of the module is present in *TcGblEnv*. The data type is quite large so an author should consult GHC's documentation in order to familiarise themselves with its contents, but some important fields will be highlighted here.

The bindings for a module are located in the *tcg_binds* field. Their type is *LHsBinds GhcTc*, which indicates that the bindings are fully type checked. This field only contains bindings (both function and value declarations); other declaration types are present in other fields. For example, class instances are found in *tcg_insts*, pattern synonym declarations in *tcg_patsyns*, and type family instances in *tcg_fam_insts*.

The source structure of these different types of declarations is lost after type checking. The internal representation of a pattern synonym that remains after type checking (*PatSyn*) contains no information about the source position or structure of the pattern synonym.

The renamed source can be retained for inspection in a type checker plugin if a command line flag is enabled. The *keepRenamedSource* plugin (defined in *Plugins*) is a renamer plugin which turns on this flag. When it is enabled the renamed source structures can be found in the *tcg_rn_decls* field.

In short, a plugin author should inspect the contents of the *TcGblEnv* before starting to write their plugin as they may find that the information they want to summarise already present and can be reused.

The implementation of the plugin inspects the command line options, as well as *ModSummary* and *TcGblEnv* before outputting the indices. The unchanged *TcGblEnv* is returned by the plugin so that the compilation continues as normal for the rest of the pipeline.

3.2 Idioms Plugin

The idioms-plugin[3] is a source plugin which implements idiom brackets [11]. This is an example of a plugin that modifies the user's program. Modifying a program is more complicated than analysing a program because correct syntax has to be constructed.

An idiom bracket is a context where whitespace should be interpreted as applicative application rather than normal function application. The "syntax" for an idiom bracket is a singleton list surrounded by parentheses.

```
> ([(+) (Just 1) (Just 2)])
Just 3
```

The source plugin identifies occurrences of this pattern and rewrites the contained expression by the rules of idiom brackets, so the above code becomes:

```
pure (+) ⟨*⟩ Just 1 ⟨*⟩ Just 2
```

What about singleton lists? They still work correctly, even without special logic in the implementation. ([5]) is transformed to *pure* 5 which equals [5].

The idioms-plugin is a parser plugin ①. Parser plugins run immediately after parsing and therefore have to implement the following interface:

```
pluginImpl :: ModSummary
           → HsParsedModule
           → Hsc HsParsedModule
```

A *HsParsedModule* contains the parsed syntax tree for the user's program.

The plugin works by first traversing this syntax tree in order to find the occurrences of singleton lists surrounded by brackets. It is convenient to implement these traversals using SYB [9] because all the data types defined by GHC define *Data* instances:

```
transform dflags =
  SYB.everywhereM (SYB.mkM transform') where ...
```

Once a list surrounded by parentheses is identified, it is checked to see whether the list is immediately contained within brackets. The transformation is not applied if there are any spaces between the two. This is to give the illusion that ([and]) are individual syntactic entities.

```
inside :: SrcSpan → SrcSpan → Bool
inside (RealSrcSpan a) (RealSrcSpan b) = and
  [ srcSpanStartLine a == srcSpanStartLine b
  , srcSpanEndLine a == srcSpanEndLine b
  , srcSpanStartCol a + 1 == srcSpanStartCol b
```

[3] https://github.com/phadej/idioms-plugins

```
    , srcSpanEndCol a == srcSpanEndCol b + 1]
  inside _ _ = False
```

Finally, now the correct locations have been identified, the plugin constructs the necessary syntax to perform the transformation. There are many combinators included in GHC for building expressions. A good place to start looking is the *HsUtils* module.

For a parser plugin you need to construct a *LHsExpr GhcPs* that is a syntax tree which contains unresolved references to variables. These are called *RdrName*s. The plugin constructs *RdrName*s for *pure* and ⟨∗⟩ before manipulating the user-written program in order to insert the combinators.

```
pureRdrName, appRdrName :: RdrName
pureRdrName = mkRdrUnqual (mkVarOcc "pure")
appRdrName  = mkRdrUnqual (mkVarOcc "<*>")
```

This transformation works like the `RebindableSyntax` extension where the *pure* and ⟨∗⟩ are resolved to whatever definition of *pure* and ⟨∗⟩ the module user has in scope. In Section 5.4 we'll discuss about how to use precise references.

Finally, the expression is reconstituted by applying *pure* to the first argument and then combined together using ⟨∗⟩. The combinator *nlHsApps* creates an application of a *RdrName* to a list of arguments.

```
transformExpr :: LHsExpr GhcPs
              → [LHsExpr GhcPs]
              → LHsExpr GhcPs
transformExpr f xs = foldl app puref xs where
  puref = nlHsApps pureRdrName [f]
  app fe e = nlHsApps appRdrName [fe, e]
```

Implementing a transformation like this using a different parser to the one implemented in GHC would be very frustrating due to inevitable differences in the behaviour of the parsers and complexity in modifying source files.

3.3 assert-explainer

The assert-explainer[4] plugin rewrites an assertion to provide additional information to the user when the assertion fails.

```
assert (length xs == 4)
```

An *assert* is a special function introduced by the plugin which marks the body should be manipulated by the plugin. The plugin modifies the expression to also print out the values of all subexpressions if the assertion fails.

```
- Assertion failed!
    length xs == 4 /= True (at Test.hs:18:12-25)

  I found the following sub-expressions:
    - length xs = 3
    - xs = [1,2,3]
```

[4] https://github.com/ocharles/assert-explainer

Knowing further information about the assertion failure can make debugging much easier.

assert-explainer is a type checker plugin ③ which means that it is implemented using the same interface as haskell-indexer. However, assert-explainer synthesises syntax. Generating syntax in a type checker plugin is more complicated than a parser plugin because the syntax has to be explicitly typed and all evidence that the type checker produces has to be filled in.

Writing a type checker plugin has other advantages though, the compiler knows a lot more information about the program so code can be generated in a more sophisticated manner. For example, notice in assert-explainer that despite *length* being a subexpression of *length xs*, its value is not printed. During the process of code generation the constraint solver is consulted in order to verify that each subexpression can be printed.

Constructing terms In order to insert or modify a binding when writing a type checker plugin, it is necessary to create a type checked term. A type checked expression is of type *LHsExpr GhcTc* but constructing these directly can be difficult as in particular you have to understand how the constraint solver will generate evidence. It is easier to construct an untyped representation of the term before passing it to the normal type checking functions in order to create the type checked expression.

In fact, there are several different representations of terms to choose from, and it is worth understanding them:

LHsExpr GhcTc A type checked expression is difficult to construct because the implicit evidence must be filled in.

LHsExpr GhcRn A renamed term has all references to names resolved. It can then be type checked but the plugin author must take care to make sure that their term is correctly and unambiguously typed. Failure to do so will result in an error when the plugin is run.

LHsExpr GhcPs A parsed term is very easy to construct as it most closely resembles the languages source syntax. References to names are not fully determined until the plugin is run.

Exp Using Template Haskell quotations to construct terms is a good compromise between ease and specificity. Terms constructed in this way can still fail to type check but the references of free variables are fixed. It is also convenient to use quotations to construct terms as then you do not have to use combinators to build the representations.

As an example, we will construct the term *print* () using the *Exp* method. The term is first quoted using the quotation brackets. At this point the reference to *print* is fixed so that it refers to the same *print* as in scope in the module where the plugin is defined.

```
mkNewExprTh :: TcM (LHsExpr GhcTc)
mkNewExprTh = do
  th_expr ← liftQ [|print ()|]
  ps_expr ← case convertToHsExpr noSpan th_expr of
    Left _err → error "Bad expression"
    Right res → return res
  io_tycon ← tcLookupTyCon ioTyConName
  let exp_type = mkTyConApp io_tycon [unitTy]
  renameExpr ps_expr >>= typecheckExpr exp_type
liftQ :: Q a → TcM a
liftQ = liftIO . runQ
```

After the quotation is run, the result is a *LHsExpr GhcPs* which has to be renamed and type checked. In order to help the type checker the result type of the whole expression is synthesised, in this case *IO* (). By fixing a monomorphic type as the result of the expression there are less likely to be ambiguities during the type checking process. However, errors can still occur if you try to type check something which is type incorrect! It is up to the plugin author to construct a correct term and handle errors which arise from calling *TcM* actions. If they do not catch and handle the errors then they are displayed directly to the user.

If the generated term is static, not dependent on the current program context, then a good option is to bypass syntax generation entirely by defining the function in an external module and calling it by creating a reference to the variable. Creating references to variables is discussed in Section 5.4.

Conversing with the constraint solver In this plugin, only subexpressions which have a type that is an instance of *Show* should be displayed. In order to achieve this the constraint solver has to be asked whether the type satisfies the constraint. This can be achieved from within a type checker plugin.

The *getDictionaryBindings* function asks the constraint solver if a constraint is satisfiable. There are two inputs, a variable which will store the evidence information and the constraint we want to solve (e.g. *Show* ()).

```
getDictionaryBindings ::
  Var → Type → TcM (WantedConstraints, EvBindMap)
getDictionaryBindings dict_var dictTy = do
  loc ← getCtLocM (GivenOrigin UnkSkol) Nothing
  let nonC = mkNonCanonical CtWanted
    { ctev_pred = dictTy
    , ctev_nosh = WDeriv
    , ctev_dest = EvVarDest dict_var
    , ctev_loc = loc}
    wCs = mkSimpleWC [cc_ev nonC]
  runTcS (solveWanteds wCs)
```

getDictionaryBindings then constructs a *CtWanted* constraint [20] which is then asked to be solved by *solveWanteds*. The return value of the whole function is the set of unsolved constraints, which should hopefully be empty, and the evidence generated by solving the constraints. This information can be used to work out whether the constraint was soluable or not.

4 Using Plugins

Now that the interface (Section 2) and the development of plugins (Section 3) have been described, it is time to look at how plugins can be invoked.

4.1 Invoking a Plugin

A plugin is invoked by passing a command line options to the compiler.

`-fplugin <module>` Load the specified module as a plugin and run it when compiling. Multiple plugins can be enabled at the same time by passing multiple `-fplugin` options. The plugins are run in the order of the arguments.

`-fplugin-opt <string>` Pass these options to the plugin when it is run. Multiple `-fplugin-opt` options can be passed. The value of each `-fplugin-opt` option is prefixed with the name of the plugin receiving that option.

`-plugin-package` Use this package name in order to find the plugin specified by `-fplugin` and any dependencies.

`-plugin-package-id` Use this package identifier (which is a name pinned to a specific version) in order to find the plugin specified by `-fplugin`.

`-hide-all-plugin-packages` By default, all packages that are imported with `-fpackage` are available for a plugin to use. This flag means that they can only use dependencies specified by the package name with `-fplugin-package` or by its identifier with `-fplugin-package-id`.

The interface is quite low level but users are not expected to call these options themselves, just like they are not expected to pass other configuration flags to GHC themselves. It is the domain of build tools to take a user's high level specification for a build and translate it into these primitives.

4.2 Nix Interface

None of the dedicated build tools for Haskell projects provide first class support for adding and invoking plugins. However, we have implemented a user interface for the Nix ecosystem which uses the general purpose Nix package manager [5] to build Haskell projects.[5] This is detailed here since it helps to pinpoint the way in which plugins interact with the rest of the system.

The Nix package manager already has comprehensive support for building Haskell packages.[6] The extension allows existing build instructions to be modified in order to also enable plugins to be run during the package's compilation.

[5] https://github.com/mpickering/haskell-nix-plugin
[6] https://nixos.org/nixpkgs/manual/#users-guide-to-the-haskell-infrastructure

The behaviour of how a plugin can be applied to a module is specified as a Nix attribute set which is a collection of the following named fields:

pluginPackage The name of the Nix package in which the plugin is defined.

pluginName The name of the module the plugin is defined in. This is also used as the name of the output for the plugin, as well as the parameter that names the plugin passed to GHC.

pluginOpts A function which takes two arguments and produces a list of options to pass to the plugin. The first argument is a directory where the plugin can write its output to. The second argument is the package that the plugin is applied to.

There is a special output directory created for the plugin to use. This directory is typically passed to the plugin as an argument and it uses it to write intermediate results.

The package is also passed to this function so that the plugin options can depend on metainformation about the package. For example, you might want to pass the current package name to a plugin. Some of this information is also available from inspecting the compiler state using the plugin but it is more straightforward to use the well-structured information by inspecting the Haskell package.

pluginDepends Any additional system dependencies that the plugin requires in addition to the normal Haskell dependencies.

initPhase A shell script which runs before the module is compiled. This is used for doing any initialisation that the plugin requires to run successfully such as making directories or initialising a database.

finalPhase A shell script which runs after the package has been compiled. The individual outputs of running the plugin on each module can be collated and rendered into a format suitable for the user to consume.

As an example that uses most of these fields, here is the specification of the graphmod plugin, that corresponds to the *graphModules* type checker plugin (Section 1):

```
graphmod =
  { pluginPackage = hp.graphmod-plugin;
    pluginName    = "GraphMod";
    pluginOpts    = ({path,pkg}: ["${path}/output"]);
    pluginDepends = [ nixpkgs.graphviz ];
    finalPhase    = {path,pkg}:
      ''graphmod-plugin --indir ${path}/output >
                        ${path}/out.dot
        cat ${path}/out.dot | tred | dot -Tpdf >
                        ${path}/modules.pdf
      ''; } ;
```

The graphmod plugin is defined in code that has been bundled into the `hp.graphmod-plugin` package, and the module of the plugin definition is `GraphMod`.

Recall that the definition of *graphModules* takes a value *opts* :: [*CommandLineOption*] as a parameter. This list is populated with the information from the application of the `pluginOpts` field. In this example the field should contain the path where diagrams are output, and so `pluginOpts` provides `["${path}/output"]` as the argument.

The plugin uses `graphviz` to render its output, and so this is added as a dependency in the `pluginDepends` field.

Finally, once the package has been compiled the script contained in the `finalPhase` field is executed. In this case the graph is be rendered using `tred` and `dot`, which are provided by the `graphviz` package.

The `graphmod` package is now ready to be applied to a particular package. For instance, here is the invocation required to trace out the dependencies of the either package:

```
eitherWithPlugin =
  addPlugin graphmod haskellPackages.either
```

By building `eitherWithPlugin`, the generated graph for the either Haskell package will be contained in an attribute called `eitherWithPlugin.GraphMod`.

Using a whole system package manager such as Nix makes it easy to configure a plugin that interacts with both system tools as well as Haskell packages. The combination is a common feature of plugins that perform code analysis.

5 Tips and Tricks

In our experience of implementing plugins we've also had to solve lots of other small problems. In this section we catalogue a selection of other small problems and their solutions.

5.1 Combining Together Old and New Syntax

Combining existing subexpressions in order to create a bigger program is not straightforward to achieve directly. If you make new syntax using quotation brackets in the manner described in Section 3.3 then you will still want to insert *LHsExpr GhcTc* expressions into the syntax tree.

The easiest way to achieve this is to construct a function which can be then applied to the existing expressions. For each place where you want to insert a subexpression create a new variable, abstract over all these occurences and then finally apply the expression after conversion to the *LHsExpr GhcTc* arguments.

For example, consider a plugin which replaces $2 * e$ with $e + e$. Suppose that after syntax analysis e :: *LHsExpr GhcTc*, first construct a closed Template Haskell term $[\![\lambda e' \to e' + e']\!]$, convert the representation to a *LHsExpr GhcTc* and finally apply the function to e. The resulting term is $(\lambda e' \to e' + e')\ e$ which is equivalent to $e + e$.

5.2 Finding the Type of an Expression

When writing a type checker plugin it is common to want to know the type of a certain subexpression. At the time of writing, the type can not be computed directly from the *LHsExpr GhcTc*. However, by observing that the type of an

expression should remain invariant after desugaring the type can be computed from the desugared term.

```
getType :: LHsExpr GhcTc → TcM (Maybe Type)
getType = do
  hs_env ← getTopEnv
  (_, mbe) ← liftIO (deSugarExpr hs_env e)
  return (exprType ⟨$⟩ mbe)
```

5.3 Reporting Errors

In assert-explainer we also want to warn a user, at compile-time, if there are no sub-expressions at all which can be displayed. This indicates that they won't get any useful diagnostics when the assertion fails.

The normal *TcM* functions can be used to display the error to the user in a way uniform to other compiler errors.

For example, by using the functions *setSrcSpan*, *addErrCtxt*, and *failWithTc*, we can create errors which will get reported in the same way as a normal compiler error when the plugin has finished executing.

```
makeError :: LHsExpr GhcTc → TcM ()
makeError (L l e) =
  setSrcSpan l $
    addErrCtxt (text "In" ⟨+⟩ ppr e) $
      failWithTc "Error!"
```

5.4 Specifying Names

When specifying what names we mean, it is important to understand the context in which different name mechanisms work in. Renamer functions which operate in the *TcM* monad will be executed when the plugin runs, in the context of the user's program so it will only find names that the user has brought into scope in that module. In order to properly persist names the resolution of the name should be determined in the module the plugin is defined in.

For example, constructing the *OccName* for the identifier *f* if you use the *lookupTopBndrRn* function in order to resolve its name in a source plugin then this function will be run when the plugin is executed and the lookup will only succeed if the user has an identifier named *f* in global scope.

On the other hand, if the Template Haskell name quotation mechanism is used by referring to the name as *'f* and then converting it to a *Name* using a combination of *vName* and *isExact_maybe*, then the *Name* will refer to an *f* that is currently in scope in the module defining the plugin. This persists names unambiguously to the generated syntax.

The idioms-plugin could have used the quotation mechanism to fix references to the definitions of *pure* and ⟨∗⟩ defined in the `Control.Applicative` module.

5.5 Communicating Between Phases

Plugins operate one phase at a time and one module at a time. The API provides no functions to natively propagate state between the phases. Each plugin must decide how it serialises and communicates information between its invocations. Plugins can communicate between each phase by using a global mutable reference to store state and between invocations by writing interface files.

Communication by mutable variables If two different phases need to communicate between themselves, for example, a renamer plugin communicating with a type checker plugin, then a mutable variable such as an *IORef* can be used to store information between the phases.

A global *IORef* can be created by using *unsafePerformIO*. It is important to mark the variable with a `NOINLINE` pragma so that the variable isn't inlined and duplicated.

```
var :: IORef Int
var = unsafePerformIO $ newIORef 0
{-# NOINLINE var #-}
```

The variable will only be initialised once when the plugins are first loaded into the session. Information written to the variable is persisted across the phases so changes to the variable made by a renamer plugin are available in a type checker plugin.

Communication by serialisation In the common situation that a plugin needs to communicate between different modules, the plugin author is expected to serialise the information which needs to be communicated to some form of persistent storage.

A common design is for the plugin to take the output directory as an argument and create a serialisation of its work to place into the directory. The module name and unit identifier can be used to make a unique filename which can be accessed by another invocation of the plugin or after the plugin has finished processing all the modules.

This design fits into how GHC supports separate compilation by writing interface files to communication information about already compiled modules.

6 Other Plugin Examples

The plugin system provides a comprehensive means of interacting with GHC, and so there is a huge variety of possibilities for different plugins.

In order to motivate source plugins we will first describe some plugins which have already been implemented but without going into too many technical details.

6.1 lift-plugin

The purpose of the lift-plugin[7] is to augment how the *Lift* type class from `template-haskell` works.

[7] https://github.com/mpickering/lift-plugin

```
class Lift a where
  lift :: a → Q Exp
```

The *lift* :: $a \rightarrow Q\ Exp$ function is used to turn a value into its Template Haskell representation. *lift* is implemented for base types such as *Int*, *Bool*, () and so on, as well as compound types such as tuples and lists. Any value which can be represented in the Template Haskell AST by copying can be made an instance of *Lift*. The notable exception to *Lift* is that functions are not liftable. This means that at runtime there is no general method to convert a function to its representation.

When using quotations, top-level definitions can be persisted due to their top-level nature. $[\![id]\!]$ is the representation of *id*, which stores the path to the top-level position where *id* is defined.

The goal of the lift-plugin is to remove this restriction for top-level functions so that if *lift* is directly applied to a top-level function then it will work as though there was a *Lift* instance for functions. For example, *lift id* should result in the representation for *id*.

It is substantially harder to implement plugins which need to change how a program is type checked. The only mechanism which can be used to recovered from type errors is a constraint solver plugin. Unsolved constraints can be intercepted and solved in this plugin but with the current interface recovering from normal type errors is impossible. A type checker plugin will only be run if the program is already type correct.

This means that if the plugin should fix a type error then the type error must be manifested as a failure to solve a constraint and then that should be fixed in the constraint solver. However, when solving these constraints not enough contextual information about how they arose is available to implement this plugin. In particular, the fact the *Lift* constraint arose from an application to *id*. Therefore, the constraint solver delays the decision about solving before a type checker plugin actually provides the evidence and finally a core plugin verifies that no unsolved constraints remain in the program. Type errors are reported in a natural manner to users by being persisted from the constraint solver plugin (Section 7.3) to the core plugin using a global variable.

6.2 smuggler

The smuggler[8] plugin automatically rewrites a user's import list to add missing imports. Analysis is performed as a type checker plugin to find the minimal set of imports, the source file is then rewritten in-place to replace the import list with the computed set. This means that when used in conjunction with automatic reloading that referencing a new definition will automatically add the import to your file.

This level of self-modification raises some questions about recompilation avoidance and the safety of writing over a user's source file. When the plugin modifies the user's source file then next time the module is compiled then the recompilation checker will observe that the source file has changed and reinvoke the plugin. However, the plugin has to observe that it doesn't need to modify the file again as if it writes to the file then the modification time of the file will change and GHC's own recompilation check will recompile the file causing an infinite loop.

The plugin authors took their own steps to avoid this problem by recording a hash of the input file and not processing the file if it was unchanged. Another option would be to modify GHC's recompilation checker to itself be based on a hash of the file's contents rather than the modification time.

6.3 Haskell by Contracts

Design by Contract [12] is an approach to designing software. It uses preconditions, postconditions and invariants to verify the interactions between software components. Contracts can be introduced into functional programming by performing certain checks before and after evaluating certain definitions [7].

Directly inserting the checks into the program code leads to cluttering the business logic of the application, mixing specification and implementation. A better way is to add these to the program representation after the program has been analysed by the compiler.

The contracts can be added as annotations to the components they are specifying. GHC annotations can be accessed during the compile process. It is possible to add the checks using Template Haskell, but compiler plugins have complete freedom when operating on the program representation. The transformation can be performed after either the parse or type check phase, each having different advantages and disadvantages. In the parse phase modifying the syntax tree is easier, as it is simpler, but in the later stages with more information it is easier to interpret the annotations that control the transformation. Previous proposals [3, 21] to integrate contracts into Haskell could have been prototyped using source plugins.

6.4 what-it-do

what-it-do[9] is another debugging plugin implemented by Ollie Charles. It takes a do-expression and rewrites each line to trace the result of evaluating the line. In a similar way to the assert-explainer plugin described in Section 3 it interacts with the constraint solver to check whether each value can be rendered or not.

[8] https://github.com/kowainik/smuggler

[9] https://github.com/ocharles/what-it-do

6.5 kleene-type

Oleg Grenrus also implemented a parser plugin in the style of the idioms-plugin (Section 3.2) for constructing heterogenous lists.[10] The plugin steals the syntax for a nested singleton list and converts it to use the heterogenous cons operator. For example: `[['a', 1, True, ()]] :: HList [Char, Int, Bool, ()]`

The plugin is used in conjunction with a constraint solver plugin in order to implement support for specifying types (of heterogenous sequences) by using regular expressions.

6.6 detect-unquantified-tyvars

Thomas Winant demonstrates that plugins are useful for implementing custom warnings in the compiler. The detect-unqualified-tyvars plugin[11] raises a warning if any type variable is implicitly quantified. By using GHC's own error reporting functions the error messages integrate seamlessly.

7 Interactions

There are already a lot of different ways to extend GHC. In this section we will give an overview of these different methods, how they compare to source plugins and why you might want to use one rather than another.

7.1 Hooks

The hooks interface is a similar mechanism to modify parts of the GHC compilation pipeline. Using hooks differs from source plugins in two significant ways:

- Hooks have to be defined using the GHC API.
- Hooks replace an entire phase rather than running in addition to the normal compiler pipeline.

Source plugins are enabled by the command line but in order to use hooks one must write an executable which replicates the frontend of the compiler. In practice history has showed us that writing programs which emulate GHC's frontend is surprisingly hard to get right.

A hook replaces an entire phase, which is more powerful than a source plugin which operates in addition to normal compilation. This makes a hook harder to write as you often want to keep the core functionality of a program the same but with one small difference. When writing hooks it is necessary to carefully inspect the original definition to make sure that you reimplement the functionality in a faithful manner.

The hooks interface is somewhat ad-hoc as it was motivated by the needs of GHCJS, which modifies parts of the backend of the compiler to generate Javascript. This means that the interface is focused on specific parts of the backend such as how linking is performed, how Template Haskell splices are run, and how foreign imports are interpreted.

[10] https://github.com/phadej/kleene-type
[11] https://github.com/mrBliss/detect-unquantified-tyvars

7.2 Safe Haskell

Safe Haskell [19] is a mechanism which aims to warn users about potentially unsafe interactions by using unsafe language features. A module is either marked as *Safe*, *Unsafe* or *Trustworthy*.

Unfortunately, Safe Haskell and source plugins do not always play nicely. A source plugin can modify the user's source file in any way the plugin author desires, so this means that in order to be conservative any module compiled using a source plugin should be marked *Unsafe*. There are many source plugins that act in a completely safe manner but due to the unrestricted nature there is no way to analyse whether a plugin does anything to compromise safety.

This restriction causes practical problems when trying to run a plugin on dependencies which declare Safe Haskell properties. For instance, a module which is marked as *Safe* can't import any *Unsafe* modules. Running a plugin on the entire dependency tree marks are modules as *Unsafe* which means that the Safe Haskell check fails and compilation of the dependency fails. This issue also arises when other consumers (such as Haddock) inspect and render the Safe Haskell mode of a module.

The solution which we have implemented to this problem is to allow the invoker of the plugin to specify an additional command line flag which turns off the Safe Haskell check for a module.

`-fno-safe-haskell` Ignore any declared safety property of a module

The flag was preferred to allowing plugin authors to declare the safety of a module themselves because Safe Haskell is designed to be conservative.

7.3 Constraint Solver Plugins

A constraint solver plugin [8] allows developers to augment the constraint solver with custom solving logic. A plugin is asked if it can solve any residual constraints leftover after the normal constraint solving process has finished. The plugin can solve these constraints by providing evidence or emit further constraints to be solved.

Constraint solver plugins are the only plugin mechanism which allows type errors to be recovered from. This means that if you want to make a program type check which wouldn't otherwise the failure needs to be in the form of a insoluable constraint. This constraint can then be intercepted in the constraint solver plugin and solved rather than reporting the error to the user.

Other plugin phases such as renamer plugins or a parser plugin can pre-emptively fix type errors but in order to do this they must implement their own type checking algorithm. A more common method is to introduce terms in a renamer plugin which will certainly cause a type error which can be recovered from in the constraint solver plugin.

7.4 Core Plugins

Source plugins are implemented using the same machinery as core plugins. The difference is that the source plugins operate during the frontend of the compiler whilst the representation still closely resembles the source language.

Core plugins operate on the intermediate representation after the source program is desugared. Core plugins can implement new optimisation passes or other analysis of core programs but reflecting these changes back to the user is challenging as the provenance of a core expression is not precise and lost by program transformation.

Core plugins have not seen wide-spread adoption in the community despite being implemented in 2011. To my knowledge there are no commonly used plugins which implement domain specific optimisation passes or the like. The most commonly used core plugin is inspection-testing [2] which verifies properties about the compilation of a program.

Source plugins are easier to conceptualise than a core plugin as ordinary users are familar with the source language and source language features. Core plugins on the other hand rely on knowing how the optimiser works and specifics of the internal representation.

7.5 Frontend Plugins

Frontend plugins[12] are intended to be used by tool authors in order to replicate the precise session that GHC uses to compile a module.

The idea is that the frontend plugin replaces the whole compiler apart from the flag parsing and target loading logic. From there, a plugin author can do whatever they like with the environment to perform their analysis. The design has three problems which source plugins attempt to resolve.

1. The plugin author has to reimplement much of the compiler pipeline if they want to type check a module.
2. It is not possible to use multiple frontend plugins together with each other.
3. The plugin has to be run seperately to the normal compilation process so every module is compiled twice, once for normal compilation and once for the plugin.

It is common for a tooling author to want to extract something specific from the compiler but with a frontend plugin you have to replicate all the compilation logic yourself by copying functions defined in the compiler. The advantage of a source plugin is that you are presented with the type checked module and then just have to perform the analysis to find out the information you need.

Projects which require a great deal of control over the compilation process such as GHCJS[13] and Asterius[14] are suitable for frontend plugins but for general tooling they are not appropiate.

8 Related Work

In this section we reflect how source plugins are different to other metaprogramming facilities implemented in GHC and other methods of compiler extension in other typed functional programming languages.

8.1 Template Haskell

Template Haskell [18] provides a more convenient and hygienic API for compositionally generating programs. Syntax can be created by quoting Haskell terms and then inspected and modified by manipulating abstract syntax trees.

In comparision to source plugins, inspecting the definitions of everything in a module is not possible. Syntax is usually generated using combinators, precise control over the final generated code is hard as the syntax tree corresponds quite closely to source programs. Limited type-directed synthesis is available by calling special functions implemented in the *Q* monad, for example, the type of a variable can be consulted and whether a type satisfies certain instances. Finally, invoking Template Haskell is quite syntactically restrictive. There is no means to abstract over the splice operator.

At the other end of the spectrum, Typed Template Haskell ensures a hygienic program generation, although this comes at the expense of introspection.

8.2 Deriving Mechanisms

Type class deriving is a convenient and automatic metaprogramming facility where type class definitions can be mechanically derived from the structure of a data type.

There are a number of different deriving strategies which are built into the compiler. For example the *stock* deriving strategy uses logic built into the compiler to derive classes such as *Eq* and *Ord*. *anyclass* uses default implementations, *via* dervives a class via a representationally equal data type which already implements the class [1] and so on. One part missing from the deriving arsenal is a way for users to take precise control over how a type class should be derived with their own strategy.

A new type class deriving strategy could be implemented with a source plugin with these steps:

1. Define a new class that you want to derive by using a class definition.
2. In a renamer plugin remove the class from the deriving clause of the data type.
3. In a type checker plugin generate a class definition due to your deriving strategy.

This implementation strategy relies on the knowledge that type class instances are derived after type checking. This is necessary because the class definition must be well formed before the instances can be derived from it. Removing the

[12] https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/extending_ghc.html#frontend-plugins
[13] https://github.com/ghcjs/ghcjs
[14] https://github.com/tweag/asterius

class from the deriving clause means that GHC will not try to derive the class itself and raise an error. This means the decision about whether to derive a class needs to be communicated using one of the techniques discussed in Section 5.5.

8.3 Elaborator Reflection

Elaborator reflection [4] is the name given to a family of techniques which expose the internal implementation of the compiler to the user so they can generate terms in a manner similar to the compiler. Elaborator reflection is implemented in Idris and Agda.

In Idris, users write functions of type *Elab* () which describe how to construct a term. For example, the *mkId* :: *Elab* () script will construct an identity function taking into account its context. Syntax is also added to the language which allows elaboration scripts to be run.

```
idNat :: Nat → Nat
idNat = %runElab mkId
```

The construct *%runElab* is similar in operation to the splice. Its result is evaluated and the resulting term inserted into the program.

The interface that GHC implementers use to construct terms is not similar to the method implemented in elaborator reflection. It is a low-level combinator based approach. We conjecture that a tactics based approach could be implemented as a library and run inside the *TcM* monad to provide a similar interface.

8.4 Scala Plugins

The most closely related plugin mechanism is the Scala plugin system. They operate under the same principle as source plugins. Users define transformation passes which work on the compilers internal representation.

One interesting feature is that the user specifies when a plugin should run relative to other compiler phases [14]. This yields a set of constraints which is linearised by the compiler in order to find the correct order to run the plugins. This approach is only possible because the internal AST representation in scala is untyped. All plugins have a uniform type but rely on additional information being present after certain phases of the compilation.

It's interesting to see what kinds of plugins scala authors have implemented as many of them could also work for Haskell programs. Several linting tools such as WartRemover[15] and Scalafix[16] are implemented as plugins. This is in contrast to Haskell where linters have been implemented using libraries such as haskell-src-exts. Analysis tools to detect module cycles[17] and a projects dependency structure[18] are also popular. More experimental plugins extend the language with syntax for type lambdas[19] and mutual recursion[20].

Dotty (Scala 3) will also support compiler plugins.

8.5 C# Roslyn Analyzers

The Roslyn compiler for C# has a sophisticated plugin mechanism which makes implementing custom diagnostic passes straightforward. The motivation and implemenentation is geared around reporting the diagnostics back to the user rather than external analysis or modification. The extension points are a lot more fine grained, most internal functions can be extended in the definition of an analyzer.[21][22]

Developers can include custom analyzers in their library definitions which makes the integration and utilisation of plugins transparent to end users.

9 Historical Remarks

The authors have learnt from many others about how to write different styles of plugins over the years so it would be amiss to neglect to mention the contributions of others to this cause. The development of plugins was started by Max Bolingbroke and Austin Seipp in 2008. The first version of GHC to support core plugins was 7.2.1[23] which was released in 2011. Since then the adoption of core plugins has been quite slow but there are several notable examples. Elliott [6] implements automatic syntax overloading, Breitner [2] checks properties of compiled programs and Izbicki[24] integrates Herbie [16] into GHC. Gundry [8] fulfilled the OutsideIn(X) [20] prophecy by allowing users to write plugins to extend the constraint solver. Diatchki[25] and Baaij[26] implement solvers to solve type-level aritemetic. Baaij has also written useful tutorials about constructing constraint solver plugins[27][28]. Frisby[29] implemented row types and Otwani and Eisenberg [15] solve a constraint DSL using Z3.

The idea for source plugins has been floated several times, initially by Edsko de Vries[30] and then completed five years

[15] http://www.wartremover.org/
[16] https://scalacenter.github.io/scalafix/
[17] https://github.com/lihaoyi/acyclic
[18] https://github.com/lightbend/scala-sculpt
[19] https://github.com/typelevel/kind-projector
[20] https://github.com/wheaties/TwoTails
[21] https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/tutorials/how-to-write-csharp-analyzer-code-fix
[22] https://docs.microsoft.com/en-us/visualstudio/extensibility/getting-started-with-roslyn-analyzers?view=vs-2017
[23] https://downloads.haskell.org/~ghc/7.2.1/docs/html/users_guide/release-7-2-1.html
[24] https://github.com/mikeizbicki/HerbiePlugin
[25] https://github.com/yav/type-nat-solver
[26] https://github.com/clash-lang/ghc-typelits-extra
[27] https://qbaylogic.com/blog/2016/08/10/solving-knownnat-constraints-plugin.html
[28] https://qbaylogic.com/blog/2016/08/17/solving-knownnat-custom-operations.html
[29] https://github.com/nfrisby/coxswain
[30] https://mail.haskell.org/pipermail/ghc-devs/2013-June/001377.html

later by Németh [13]. Pickering [17] made further improvements to the recompilation API; then implemented by Pickering but with significant help from Baaij. Charles[31][32], Kovanikov and Romashkina[33], and Grenrus[34][35] have been early adopters and their feedback in developing source plugins has led to further improvements and ideas for this paper.

10 Conclusion

Source plugins are an extremely powerful and flexible mechanism for interacting with GHC. The goal of this paper is to make this work accessible to the masses by providing a detailed overview of the system and its possibilities.

The interface for working with source plugins allows programmers to interact with the GHC compilation pipeline through a series of extension points, and modify results as necessary. This plugin system provides a comprehensive means of interacting with the compiler, making it possible to write a wide variety of plugins for different purposes.

Acknowledgments

We thank Csongor Kiss and Jamie Willis for comments on earlier drafts. This work has been supported by EPSRC grant number EP/S028129/1 on "SCOPE: Scoped Contextual Operations and Effects".

References

[1] Baldur Blöndal, Andres Löh, and Ryan Scott. 2018. Deriving Via: or, How to Turn Hand-Written Instances into an Anti-Pattern. *SIGPLAN Not.* 53, 7 (Sept. 2018), 55–67. https://doi.org/10.1145/3299711.3242746

[2] Joachim Breitner. 2018. A Promise Checked is a Promise Kept: Inspection Testing. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell (Haskell 2018)*. ACM, New York, NY, USA, 14–25. https://doi.org/10.1145/3242744.3242748

[3] Olaf Chitil. 2012. Practical Typed Lazy Contracts. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 67–76. https://doi.org/10.1145/2364527.2364539

[4] David Christiansen and Edwin Brady. 2016. Elaborator Reflection: Extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 284–297. https://doi.org/10.1145/2951913.2951932

[5] Eelco Dolstra. 2006. *The purely functional software deployment model.* Utrecht University.

[6] Conal Elliott. 2017. Compiling to Categories. *Proc. ACM Program. Lang.* 1, ICFP, Article 27 (Aug. 2017), 27 pages. https://doi.org/10.1145/3110271

[7] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-order Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/581478.581484

[8] Adam Gundry. 2015. A Typechecker Plugin for Units of Measure: Domain-specific Constraint Solving in GHC Haskell. *SIGPLAN Not.* 50, 12 (Aug. 2015), 11–22. https://doi.org/10.1145/2887747.2804305

[9] Ralf Lämmel and Simon Peyton Jones. 2003. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI '03)*. ACM, New York, NY, USA, 26–37. https://doi.org/10.1145/604174.604179

[10] Simon Marlow and Simon Peyton Jones. 2012. The Glasgow Haskell Compiler. In *The Architecture of Open Source Applications*, Amy Brown and Greg Wilson (Eds.). Chapter 5. https://www.aosabook.org/en/ghc.html

[11] Conor McBride and Ross Paterson. 2008. Applicative Programming with Effects. *Journal of Functional Programming* 18, 1 (2008), 1âĂŞ13. https://doi.org/10.1017/S0956796807006326

[12] Bertrand Meyer. 1992. *Eiffel: The Language.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[13] Boldizsár Németh. 2018. Source Plugins. GHC proposal. https://github.com/ghc-proposals/ghc-proposals/pull/209

[14] Anders Bach Nielsen. 2008. Scala Compiler Phase and Plug-In Initialization for Scala 2.8. https://www.scala-lang.org/old/sid/2

[15] Divesh Otwani and Richard A. Eisenberg. 2018. The Thoralf Plugin: For Your Fancy Type Needs. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell (Haskell 2018)*. ACM, New York, NY, USA, 106–118. https://doi.org/10.1145/3242744.3242754

[16] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/2737924.2737959

[17] Matthew Pickering. 2018. Refining the Plugin Recompilation API. GHC Proposal. https://github.com/ghc-proposals/ghc-proposals/pull/108

[18] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691

[19] David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. 2012. Safe Haskell. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA, 137–148. https://doi.org/10.1145/2364506.2364524

[20] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X) Modular Type Inference with Local Assumptions. *J. Funct. Program.* 21, 4-5 (Sept. 2011), 333–412. https://doi.org/10.1017/S0956796811000098

[21] Dana N. Xu, Simon Peyton Jones, and Koen Claessen. 2009. Static Contract Checking for Haskell. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 41–52. https://doi.org/10.1145/1480881.1480889

[31] https://github.com/ocharles/assert-explainer
[32] https://github.com/ocharles/what-it-do
[33] https://github.com/kowainik/smuggler
[34] https://github.com/phadej/idioms-plugins
[35] https://github.com/phadej/kleene-type