HAL8 Workshop:
A Practical Approach to GHC Language Extensions

Matthias Fischmann and Sönke Hahn

{*mf,sh*}*@zerobuzz.net*

May 2, 2013

Abstract

GHC language extensions are a mechanism for extending the language standard in a controlled way, where every feature can be switched on and off individually with compiler pragmas. In this workshop, we delve into the plethora of extensions and explore their purpose, usefulness, robustness.

1 Introduction

Programming languages evolve over time. New syntax is introduced, new semantic features are added and old features are phased out. One approach to deal with these changes in a programming language is to have a versioned language standard that evolves over time. A different approach that has emerged in the Haskell community are so-called language extensions.

A language extension is a modification of the Haskell language itself. An extension can add new syntactic constructs and change the semantics of Haskell. In most cases, extensions are downwards compatible, i.e. the resulting language is a superset of standard Haskell.

The concept of extensions is not to be confused with macro systems (see Section 4.1 Language Extensions vs. Macros for a comparison of the two approaches.)

Most extensions are disabled by default and have to be turned on individually. There are three ways of enabling / disabling an extension.

- On the command line, an extension `Foo` can be enabled (disabled) with `ghc -XFoo` (`ghc -XNoFoo`).
- In cabal files:

```
executable foo
  ...
  extensions: MultiParamTypeClasses NoTypeSynonymInstances
  ...
```

- In the source code with language pragmas:

```
{-# LANGUAGE MultiParamTypeClasses   #-}
{-# LANGUAGE NoTypeSynonymInstances  #-}

module Main where ...
```

Generally, we recommend to use language pragmas as it makes module files self-contained and therefore reusable in other contexts.

A list of all supported extensions can be produced with

```
ghc --supported-extensions
```

The GHC manual has documentation on most extensions.[1] The documentation is sometimes incomplete, slightly out of date and lacks a clear overview about which extension is documented where. So sometimes additional internet research seems necessary.

This workshop and this paper attempt a *practical* approach to GHC's extensions, emphasizing the things most interesting to the software developer as opposed to the language designer.

It is structured as follows: in the first half, we present a list of selected language extensions including short examples of their usage. We also talk about general advantages and disadvantages of the language extension mechanism. In the second half, we present programming challenges and solve them in small groups using different (or no) language extensions and compare the results.

The code examples and latest version of this paper can be retrieved from darcsden with

```
darcs get http://darcsden.com/mf/hal8_ghc_extensions
```

2 A Classification of GHC Language Extensions

Language extensions differ widely along a range of categories, e.g.

category	description
topic	type (no classes), type (classes), syntactic sugar, or other
usefulness	was an extension worth implementing? (1..6)
difficulty	how easy is it to understand and use an extension? (1..6)
risks	how badly can things go wrong? (1..6)
exports	are importing modules affected? (yes/no)
ghc release	how long has this been around?

In the workshop, we are going to classify the discussed extensions with these categories based on our understanding of the matter and (highly subjective) best guesses.

3 A Selection of Individual Extensions

In the following, we present a selection of extensions. The order is more or less random and motivated, if at all, by entertainment value. In the workshop, we may pick a different selection of extensions depending on the preferences of the audience, but we will follow this structure when introducing them.

MultiParamTypeClasses

Allows to define and instantiate type classes with more than one type parameter. Type classes with multiple parameters can be thought of as relations on types rather than sets of types.

Example: the string-conversions package provides the function

```
cs :: ConvertibleStrings a b => a -> b
```

that takes a string of type `a` and produces a string of type `b`, given an instance has been implemented for `a` and `b`.

OverloadedStrings

Changes the type of string literals from `String` to `IsString a => a`. This is useful when working with values of type `Text` or `ByteString`. It is very similar to how number literals are typed.

[1] `http://www.haskell.org/ghc/docs/latest/html/users_guide/ghc-language-features.html`

NamedFieldPuns

Allows to shorten record pattern bindings by binding field values to identifiers with the same name as existing record fields. See also RecordWildCards.

Instead of `Person{age = age}`, where the left-hand side is the record field and the right-hand side is the bound identifier, you can write `Person{age}` to get the same behaviour.

RecordWildCards

Allows to shorten record pattern bindings by replacing the list of field names with a wild card symbol. See also NamedFieldPuns.

Instead of

```
Person{name = name, age = age, balance = balance, rating = rating}
```

simply write

```
Person{..}
```

This is related to, but much more controversial than NamedFieldPuns because it extends the name space without making explicit which identifiers are added. If you read a function that matches its arguments with wild cards, you need to know the data types involved and the record fields it introduces in order to understand the body of the function. If another identifier with the same name as a record field is used in the context of the function, this can be very confusing.

Worse, if a new record field is added to a data type constructor, the new implicitly bound identifier might shadow another previously used identifier, thereby changing the semantics of the function.

ScopedTypeVariables

Introduces lexically scoped type variables using `forall`. In Haskell 2010, type variables are always implicitly universally quantified, i.e. two type variables in two nested definitions are different variables, whether they have the same name or not. As a consequence, if you define a local function `f` in a where clause of the definition of a function `g`, you cannot refer to the type variables from the signature of `g` in the signature of `f`.

This extension lets you bind type variables to the `forall` quantifier so that they are visible in its entire lexical scope. It also allows to assign types to left-hand sides of monadic binds in do expressions (`<-`).

CPP

Switches on the C pre-processor. This is useful for conditional compilation.

Derive*

Haskell 2010 allows deriving instances for `Eq`, `Ord`, `Enum`, `Bounded`, `Show` and `Read`. There are multiple extensions that allow deriving instances for other classes: DeriveDataTypeable (for classes `Typeable` and `Data`), DeriveFunctor, DeriveTraverseable, DeriveFoldable and DeriveGeneric. (For DeriveGeneric see also DefaultSignatures.)

DefaultSignatures

When writing a type class Haskell 2010 allows to give default implementations which have to have the same types as the implemented class methods. This extension allows to declare default implementations with (explicitly stated) differing types, e.g. with added class constraints. These default implementations will be used when not overwritten in the instance and when this can be done type-safely (e.g. when the class constraints are fullfilled).

This is often used in conjunction with DeriveGeneric: the class `Generic` allows to inspect the data type. This allows to write complex default implementations and can remove the necessity to implement boilerplate instances manually.

StandaloneDeriving

Allows to derive instances after the data type has been declared. This is useful if derivation is allowed, but an upstream library lacks a needed derived instance for a data type it provides. All data constructors must be in scope (exposed by the library).

Caveat: same as for other orphan instances, i.e. the derived instance might clash with other (derived or implemented) instances for the same class and types.

ViewPatterns

Enables a new syntax in pattern matching that allows to apply a function to an argument (before possibly doing further pattern matching).

PatternGuards

Allows to write arbitrary boolean expressions on the left hand side of definitions in addition to pattern matching. This extension is enabled by default since Haskell 2010.

TypeHoles

Allows to write an underscore (`_`) to produce a hole. A hole is an expression of type `a` and can therefore be used anywhere. It differs from `undefined` in two ways:

- During compilation GHC will output the type it expects in the place of the hole.
- Using a hole at runtime will produce a nice error message including the source location of the hole.

TypeHoles are useful during development when types are still in flux. They open the way for a more demand-driven development style.

ForeignFunctionInterface

Allows to import entities from or export them to other languages. Mostly the C calling convention is used for this.

TemplateHaskell

Allows compile-time meta-programming (running Haskell functions on the syntax tree).

GeneralizedNewtypeDeriving

A type declared using `newtype` is always a wrapper around one single inner type. GeneralizedNewtypeDeriving allows to derive any class that is instantiated for the inner type for the outer type.

GADTs

Allows to explicitly state types of data constructors. These types can be more constrained than the types that would have been implicitly created using traditional data type declarations. These additional constraints include

type expressions like `(a, b)`, and not only variables like `c`, as well as additional class constraints. Selectors have to be implemented manually.

4 Discussion

Any way of changing the definition of a programming language is not without risks and disadvantages. This is also true for GHC language extensions:

- Compatibility: code that makes use of a language extension supported by GHC does not work on other compilers any more (unless they keep pace with GHC).
- Stability: the compiler grows bigger, and thus buggier, and some bugs may trigger even if no language extension is enabled. This trend may be amplified by the fact that writing a language extensions is more light-weight than changing the language standard, and thus the extensions are less mature and less well documented.
- The choice which language extensions to use is theoretically up to the individual Haskell user. In practice however, using certain libraries from hackage often forces the use of certain extensions that are enabled in these libraries.

But there are obvious and strong benefits. The extension mechanism can be viewed as a second-level language standard. The separation of the standardization into two levels has several effects:

+ New research can be implemented and released much easier and faster than it could be with any standardization efforts, no matter how innovation-friendly the release cycles are.
+ The community can have debates about which changes are beneficial and which should be dropped, and can base these debates on solid evidence.
+ This takes pressure off the standardization comittee: Standard Haskell can be released with 12-year gaps in between releases, arguably without getting out of date. Further, all changes that make it into the standard have been tested in production software for years.

4.1 Language Extensions vs. Macros

Do not confuse language extension mechanisms with a macro system like TemplateHaskell, CPP, or Lisp macros. Macros let you perform transformations on either the source code (CPP) or the parse tree (TemplateHaskell), while language extensions can do their magic in any phase of the compiler, be it parsing, name resolution, type checking or desugaring.

Macros can be released in the form of a library, which is a big plus because it keeps people who use a new feature compatible with those who don't. Even though in practice, transforming source code before the parser gets to take a look (CPP) can lead to outrageous errors and incomprehensible error reporting, more advanced macro systems allow for well-written, easy-to-understand, robust code.

In Lisp, there is very little you cannot do with macros. This is because Lisp has no static type system and virtually no syntax (S-expressions are essentially the parse tree itself). Haskell has both.

Even though TemplateHaskell (the closest relative to Lisp macros) has its use cases, it is less powerful than changing the language directly: TemplateHaskell can transform the parse tree of a Haskell program, but it can only do that explicitly (with newly introduced parenthesis/bracket syntax). What it cannot do at all is type system extensions like MultiParamTypeClasses, or introduction of genuinely new primitive concepts like FFI.

References

[GHC] The GHC User's Guide
`http://www.haskell.org/ghc/docs/latest/html/users_guide/index.html`

[HR2010] The Haskell 2010 Report
`http://www.haskell.org/onlinereport/haskell2010/`