# Not all extensions are equal

## Taxonomy of Haskell language extensions based on function and usage

**Julius Gvozdiovas**[1]
**Supervisor(s): Jesper Cockx**[1]**, Leonhard Applis**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Julius Gvozdiovas
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Leonhard Applis, Koen Langendoen

**Abstract**

Haskell programming language has a long history of extensions which extend and modify its syntax and semantics. They range from small quality-of-life syntax improvements, to complete overhauls of the type system. Such extensions are commonly implemented directly as a part of Glasgow Haskell Compiler (GHC) or as plugins for GHC through its plugin API. This paper looks at the present ecosystem of such language extensions, identifying the key categories into which extensions can be separated, based on how often and in which ways they are used, and their functionality.

We analysed which extensions are used in packages uploaded to Hackage, a central open-source Haskell archive. We further extracted the metadata about the packages, including the user-submitted tags and maintainer lists, to ascertain how and when are language extensions used.

The result of our research is a combination of several proposed potential taxonomies, that can be used by academics and practitioners alike.

# 1 Introduction

Haskell, named after logician Haskell Brooks Curry, has been a core programming language for developing and expanding the functional paradigm [13]. Haskell has a rich history of language extensions extending the original language and modifying its behaviour. Language extensions in Haskell serve the purpose of extending the base language of Haskell with additional optional features and constructs. Some of them provide syntactic sugar to reduce friction when developing Haskell applications, but do not change the overall capability of the language. Others enhance the language with a more flexible or stricter type system, metaprogramming or interfacing with other programming languages. Glasgow Haskell Compiler (GHC) serves as the de-facto standard compiler for Haskell, implementing a wide range of features, including language extensions [1]. While other other compilers exist [7], GHC stands out as the most prevalent and extensible.

Many language extensions have been included by default in the language editions that GHC supports [22]. Language editions `Haskell98` [14] and `Haskell2010` [16] are derived from the language standards of Haskell, whereas `GHC2021` and `GHC2024` serve as sets of commonly used language extensions that the wider community uses.

Haskell remains a core programming language in academia, with many papers proposing and implementing new extensions. Some extensions, have been originally proposed or described by academic papers. For example, `LinearTypes` was proposed by Bernardy et al. [1] and submitted as a proposal to GHC. Many extensions included in GHC also link the relevant papers. These include `TypeFamilyDependencies` [21], `TypeFamilies` [6, 5, 20], `StaticPointers` [8], `RecursiveDo` [9], `QuasiQuotes` [15], `QuantifiedConstraints` [2] and `PatternSynonyms` [18].

While there is research into individual extensions for Haskell, a research gap of a comprehensive, big-picture analysis of the language extension climate exists. At the moment, developers' usage of language extension is measured informally, e.g. using informal surveys [10]. Data about usage of language extensions is also a crucial part in determining the evolution of Haskell as a programming language. One of the key factors influencing which language extensions are enabled by default in language configurations such as `GHC2021` is

[1] For example, `https://ghcaniuse.damianfral.com/` lists the huge variety of language extensions that different versions of GHC implement

popularity [4]. This work aims to fill the gap by studying existing extensions, both incorporated in GHC and not incorporated, through the lens of their functionality and usage. A core contribution of this work is a taxonomy of Haskell language extensions.

The goal of this work is to classify and study not only built-in language extensions for Haskell, but also community-built extensions. A dataset, comprised of packages, language proposal implementations and GHC plugins is needed. Thus, RQ1 should locate such extensions:

**RQ1: What are the community built language extensions for Haskell?**

Informal overview of existing language extensions indicates that many of them perform greatly different functions in a Haskell program's code-base. We want to more formally assess these differences, culminating in a taxonomy that can be used for future research:

**RQ2: How can language extensions be classified into useful categories?**

Given a classification of language extensions, We want to take a practical look at their use in real-life Haskell libraries and programs. To this aim, We should measure the use of language extensions in Hackage packages:

**RQ3: How widespread is the use of language extensions in Haskell projects?**

With the data of language extension usage, We want to further examine it for useful patterns. We can measure and correlate what type of projects (using Hackage tags) tend to use what kind of extensions (using classification from RQ3):

**RQ4: What type of projects use which type of extensions?**

# 2 Methodology

Table 1: Research question methods and resulting outputs

| RQ | Research Question | Method | Result |
|---|---|---|---|
| RQ1 | What are the community built language extensions for Haskell? | Exploratory survey of dependencies of GHC and use of GHC plugins | List of Hackage packages of community-built language extensions |
| RQ2 | How can language extensions be classified into useful categories? | Examining existing categorisation, proposing new taxonomies | Set of proposed taxonomies |
| RQ3 | How widespread is the use of language extensions in Haskell projects? | Data mining packages from Hackage, detecting used extensions from Cabal files and pragmas in source code | Percentage of projects (and their files) using language extensions |
| RQ4 | What type of projects use which type of extensions? | Data mining packages and their tags from Hackage, clustering based on k-nearest neighbours | Tags grouped by their similar usage of extensions, extensions grouped by their similar usage |
| RQ5 | Do developers stick to language extensions they have used before? | Data mining packages and their maintainers from Hackage, correlating projects' extension usage with authors' prior extension usage | Correlation between past usage and present usage |

Table 1 shows the general overview of methods to answer each research question.

To answer **RQ1**, We will extract packages from Hackage which depend on `ghc` and extend GHC [23].

**RQ2** will be answered by proposing multiple possible taxonomies for classifying language extensions, noting down how well they apply. Then, in conjunction with the results from RQ4 and RQ5, We will propose which categorisation is most useful, especially in regards of developer's choosing which language extensions they wish to use.

To answer **RQ3**, We will download all projects on Hackage, and detect which language extensions they use.

**RQ4** will be answered using the Hackage data obtained from RQ4, and also additionally retrieving the Hackage tags of each project.

Finally, **RQ5** will be answered by analysing connections between maintainers and the projects they maintain, and the extensions those projects have.

## 2.1 What is a Language Extension?

In order to properly study language extensions, we must have a concrete definition. While GHC language extensions without a doubt count as such, more care has to be taken when considering extensions which have not been directly incorporated into GHC.

In order to find community-built language extensions, that is, language extensions which are not integrated into GHC, we needed a clear definition of what a language extension is. Thus, we chose to define **language extensions** as any software that satisfies the following properties:

1. It modifies syntax or semantics of Haskell code.
2. It's implementation fundamentally requires interfacing with the compiler, e.g. as a plugin.

By the nature that language extensions modify behaviour of Haskell, We expect them to import and depend on behaviour of GHC, most likely through the GHC plugin API.

Our method to answer **RQ1** is then to retrieve all packages in Hackage which depend on `ghc`, filter for those which use the GHC plugin interface. We further validate our findings by checking whether the GHC plugins are actually used.

## 2.2 Using Language Extensions

Language extensions are enabled in these ways:

1. In `.cabal` files, to be enabled for the entire project.
2. Using pragmas [25].
3. Using a language extension which implies an additional one.
4. Enabling a language edition, which then enables a collection of associated extensions.

For the purposes of this study, we looked only at usages where the developer explicitly enabled a given language extension, that is, using methods 1 and 2. Our goal was to examine how and when do developers make the decision to enable or disable an extension, thus we did not include implicit usages. At the same time, we tracked cases where developers chose to disable an extension which was enabled by the language edition they were using by default.

Haskell pragmas extend the usual comment syntax, and as such, usually do not impact the semantic meaning of a program. They can either be For this study, two specific, file-level pragmas are of note.

`LANGUAGE` pragma enables (or disables) language extensions that are directly integrated into GHC. For example, `OverloadedStrings` can be enabled for a given file by having `{-# LANGUAGE OverloadedStrings #-}` in its header. Multiple extensions can be enabled in a single invocation as well: `{-# LANGUAGE OverloadedStrings, CPP #-}`.

`OPTIONS_GHC` pragma allows for manipulation of GHC flags at a per-file level. Although not recommended by [25], GHC extensions also can be enabled by using command line options: `{-# LANGUAGE -XOverloadedStrings #-}`. More importantly, community-built extensions, implemented as GHC plugins, are usually enabled using the command line options. For example, Supermonads [3], which are implemented as a GHC plugin, can be used in a file by having the following pragma in the file-header. Note that such usage requires directly referencing the module where the plugin is located:
`{-# OPTIONS_GHC -fplugin Control.Super.Monad.Plugin #-}`.

Publicly available application `extensions`, developed by [19] is used to discern which extensions a given project uses. Since the aim is to study developer behaviour, only explicitly included extensions is counted. This means that transitively used extensions will not be counted. As an example, if a developer uses `FunctionalDependencies`, which implies `MultiParamTypeClasses`, which then implies `ConstrainedClassMethods`, but they only used the former explicitly, then only it will be counted. This will be then transformed into a list of projects, each annotated with the language extensions that they use. A quantitative analysis on this data will be performed, focusing on:

1. The portion of Haskell projects using any language extensions. This will be done by counting the number of projects using at least one language extension, and comparing it to the overall number of projects sampled.

2. Most popular language extensions used. To determine this, the number of projects using each language extension will be calculated.

3. Average number of language extensions used, by number of extensions used in each project, summing up and then dividing by the overall number of projects.

To answer **RQ3**, we sampled extension usage in two ways:

1. Usage in individual files. This allowed us to gain insight on whether certain extensions are used only in one-off situations.

2. Usage anywhere in the project (either in `.cabal` files or in individual source code files). This allowed us to reason about the overall, per-project usage of extensions.

## 2.3 Taxonomy creation and evaluation

In order to create a of Haskell language extensions, We began using the base categorisation that GHC documentation provides in [24]. As we are also considering cases where developers explicitly disabled extensions, we included cases such as `NoOverloadedStrings` in the categorisation as well. The resulting categorisation is **functionality-based**, as the exact feature set and area of effect was considered to derive the categories, as seen in Table 2.

We have devised additional ways to classify Haskell language extensions, based on the given observations:

[2]GHCi is GHC's interactive read-evaluate-print-loop environment

Table 2: Functionality-based categorisation, based off GHC documentation

| Category name | Functionality | #extensions |
|---|---|---|
| Strictness | Default strictness behaviour and strictness patterns | 3 |
| Bindings | Bindings and let-generalisation behaviour | 2 |
| Nonconformance | `NondecreasingIndentation` controls specifically how GHC default behaviour differs from the Haskell Report | 1 |
| Constraints | Additional type constraints | 3 |
| Deriving | Additional derivations | 8 |
| FFI | Foreign Function Interface | 7 |
| GHCi | `ExtendedDefaultRules` is enabled by default in GHCi [2] to not require the users to provide types when using the REPL | 1 |
| Import and Export | Behaviour of module and type imports and exports | 3 |
| Literals | Extends the allowed literals | 8 |
| Parallel and Concurrent | `StaticPointers` adds static pointer syntax which facilitates references which can be sent to other machines | 1 |
| Patterns | Additional pattern forms | 4 |
| Preprocessing | `CPP` allows using C pre-processor in Haskell files | 1 |
| Records | Records, fields, and how the are accessed | 15 |
| Safe Haskell | Signals that a module's types can be trusted | 3 |
| Syntax | Syntactic sugar and parsing behaviour | 19 |
| Template Haskell | Meta-programming | 3 |
| Type class | Type class system modifications | 11 |
| Type signatures | Modifies allowed type signatures | 7 |
| Types | Type system | 30 |
| Unboxed | Access to additional unboxed types | 4 |

1. Some extensions tend to be used only in a few files, while others tend to be enabled globally or used in majority of a project's files.

2. Many extensions implement functionality that the developer themselves could implement within the base language (such as deriving). At the same time, other extensions enable additional features that the base language cannot achieve.

## 2.4 Project Sampling

In order to attain an adequate selection of Haskell projects, I chose to sample from Hackage[3]. The initial research plan was to sample a small selection of projects, however, due to ease of sampling, all available Hackage projects were sampled.

[3] "Hackage is the Haskell community's central package archive of open source software." [11]

The package list is obtained using `cabal list --simple-output` CLI command, ignoring all but the latest version of the packages. Hackage API was directly polled for each package, fetching their tags by scrapping them from the HTML documents.

In order to obtain Hackage metadata about the packages, such as its tags and maintainers, Hackage API was polled for each package. In some cases, the API did not provide a JSON endpoint, meaning that the HTML endpoint had to be used and parsed.

# 3 Results

On 2024-06-23 we observed 314 packages with a direct dependency on `ghc`. However, we observed 48 projects using plugins with pragma `OPTIONS_GHC`.

17796 packages were fetched from Hackage, including their tags and maintainers.

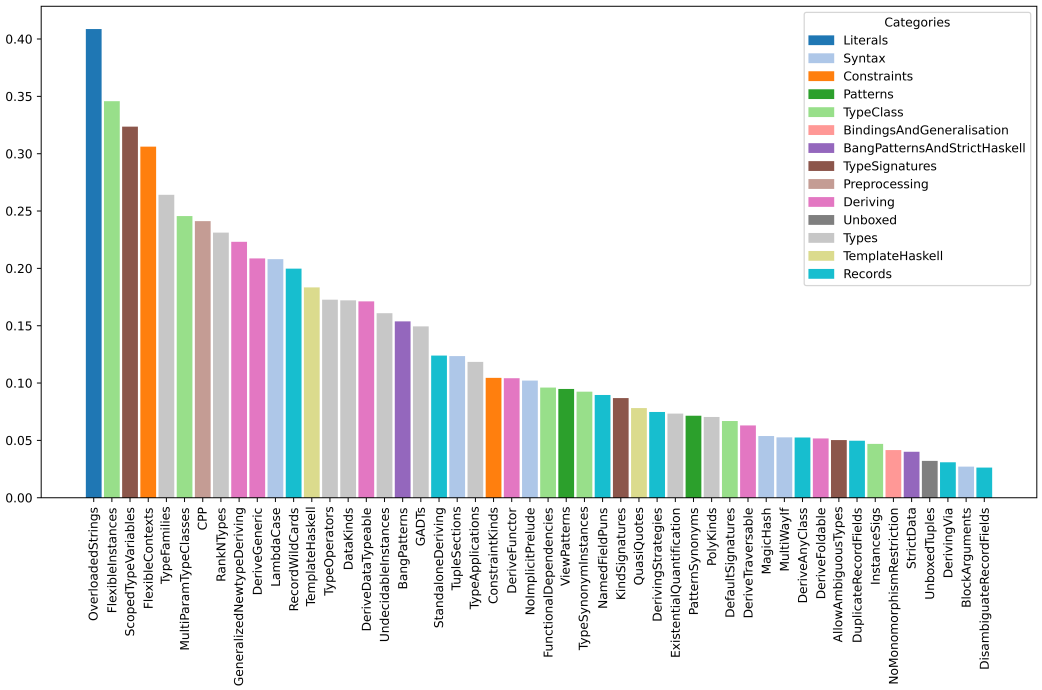


Figure 1: Top 50 most used extensions, by usage in all Hackage projects.

122 language extension pragmas were used in the projects. This counts not only direct extension usages, but also instances where developers turn off language extensions that are enabled by default by their language edition. Figure 1 illustrates most commonly used language extensions, limited to only top 30 for legibility (3 shows the number of projects each extension has been used in). Overall, 14399 projects ($\approx 80.911\%$ of all projects) used at least one extension. Figure 2 shows the per-file usage of extensions. We observe that extensions, in general, tend to be used in majority of the project, if they are used at all.

A total of 1068 unique tags were scrapped from Hackage. Of those, 224 have more than 10 usages, 49 have more than 100, and 25 have more than 300. Table 4 displays the extension and tag usage between those 25 tags and 50 most used language extensions.

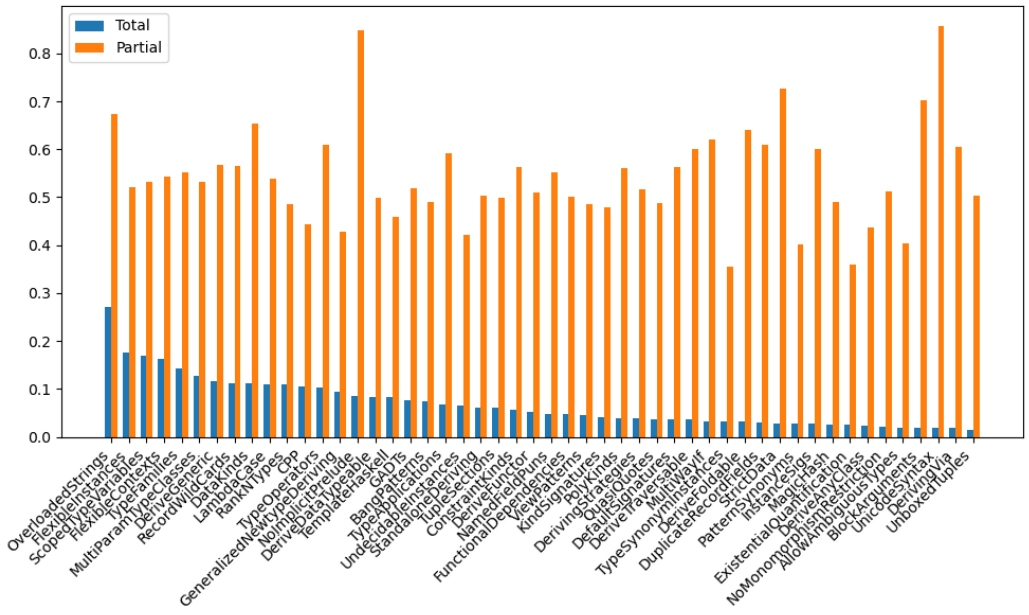


Figure 2: Usage of GHC extensions in individual files (thus not considering `.cabal` files). Total usage indicates the average portion of files using the extension from all projects, where as partial usage indicates the average portion of files using the extensions only in projects where the extension is used in at least one file. Top 50, by total usage, is shown for brevity.

Figure 3 illustrates a specific subsection of data. Observe that "data" and "language" tags tend to use similar extensions, but differ from "network".

Our analysis on whether developers tend to use language extension in-between projects seems to show that developer extension re-use seems to correlate with the language extension overall usage.

# 4 Responsible Research

There is a number of **ethical considerations** to be made in regards to this research. Collecting source code data and linking it to the developers who made it can be a privacy concern. For this reason, all maintainer names have been replaced by anonymized versions.

This research aims to be **reproducible** by ensuring the use of open-source data (all packages uploaded to Hackage must be under an open source license [12]), and by publishing both the source code of this research, and the reproducibility package with the archived dataset that was used to produce the results. The reproducability package is published at `https://data.4tu.nl/private_datasets/XXsPSQwBkJM8aCQZFiyPwXHZdLJPQ-iDfpXQ_xtf1Ro`.

I have also sought to explicitly avoid introducing undue bias (e.g. due to HARKing) by planning out the research process and the hypotheses I sought to verify or reject. I have been careful to only derive abundantly obvious conclusions from the data, and not to speculate upon the results without a solid basis.

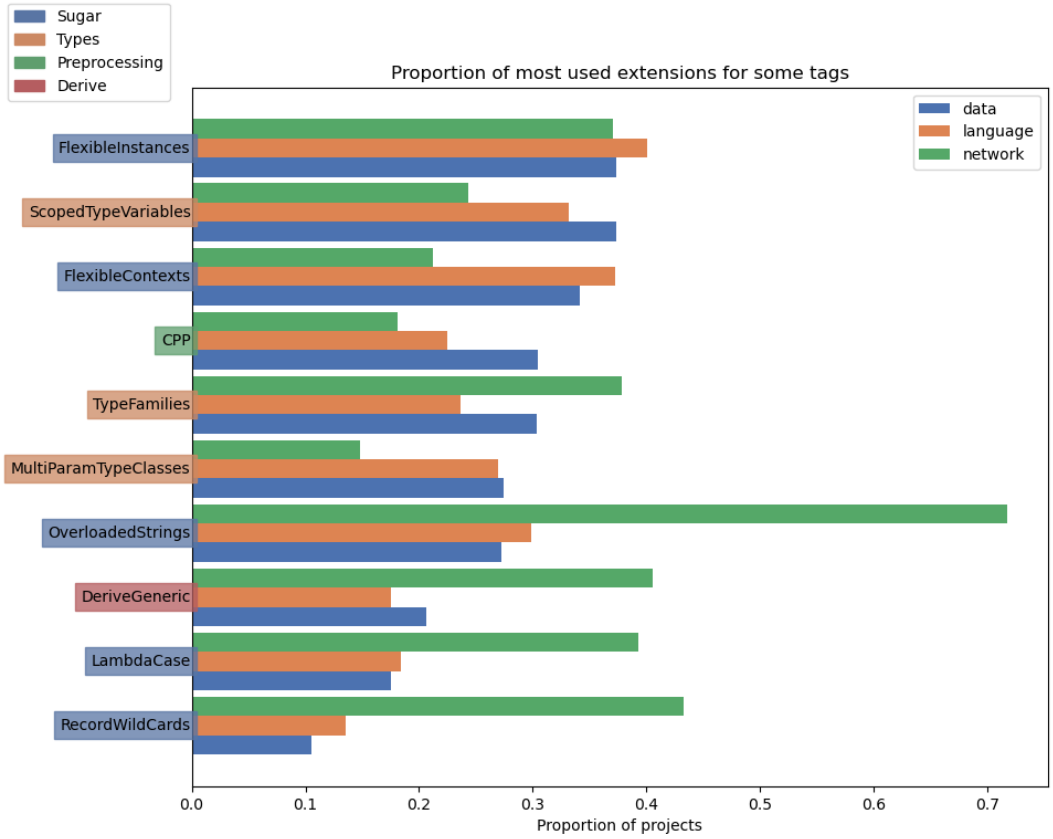


Figure 3: Usage of extensions, based on handpicked tags "data", "language" and "network". Extensions are shown such that top 5 most used extensions for each tag are represented (overlapping union).

# 5 Discussion

We observe a small number of packages depending on `ghc`, many of which are bound to be false positives - depending on the `ghc` package, but not strictly being extensions to Haskell. Even more so, with only 48 projects using GHC plugins, compared to 14399 projects using GHC built-in extensions, we see a great disparity between officially supported and community extensions.

While development on language extensions is one of the core ways functional languages have evolved over the years [13], it seems that the only path towards more widespread adoption is through integrating into GHC.

A notable extension is the `CPP` extension, alone occupying the entire domain of pre-processing Haskell files. With nearly 25% of projects using it, despite many C pre-processor's flaws criticism, Haskell developers seem to be leaning towards "love" in the love-hate relationship that developers have with it [17].

With more than 40% of Haskell packages using `OverloadedStrings`, we have shown that there is great demand for it, despite it being a quality-of-life extension.

In our results, we observe that extensions, when used, tend to be used in the majority of the project that they are used in. While we did not test any hypotheses regarding this, we can infer a few possibilities to explain this result:

-

# 6 Conclusions and Future Work

In this work, we sought to investigate community-built language extensions and GHC built-in language extensions, and found that, despite community building language extensions, they are not using them.

Our core contribution is the analysis of the overall ecosystem of Haskell language extensions. With our work, we sought to shed some light on how developers decide to choose which extensions they work on, and to provide guidance for future developers.

Given that this work sampled only projects from Hackage, other open-source platforms could be sampled in future work, such as GitHub or other forces. Our research so far has only looked at the present-day usage of extensions, in the latest versions of packages. Further work could expand upon this by studying the evolution of their usage, based on package versions.

The study of how and why language extensions are used in Haskell could be studied not only quantitatively, but also qualitatively. Future work should investigate individual developer sentiments about the usefulness of language extensions and their pitfalls. Openly available resources, such as Haskell Wiki, already advocate careful use of the optional language features [26].

# References


[1] Jean-Philippe Bernardy et al. "Linear Haskell: practical linearity in a higher-order polymorphic language". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158093.

[2] Gert-Jan Bottu et al. "Quantified class constraints". In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. Haskell 2017. Oxford, UK: Association for Computing Machinery, 2017, pp. 148–161. ISBN: 9781450351829. DOI: 10.1145/3122955.3122967.

[3] Jan Bracker and Henrik Nilsson. "Supermonads: one notion to bind them all". In: *SIGPLAN Not.* 51.12 (Sept. 2016), pp. 158–169. ISSN: 0362-1340. DOI: 10.1145/3241625.2976012.

[4] Joachim Breitner. *GHC 2021 Proposal*. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0380-ghc2021.rst. [Accessed 11-06-2024]. 2021.

[5] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. "Associated type synonyms". In: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. ICFP '05. Tallinn, Estonia: Association for Computing Machinery, 2005, pp. 241–253. ISBN: 1595930647. DOI: 10.1145/1086365.1086397.

[6] Manuel M. T. Chakravarty et al. "Associated types with class". In: *SIGPLAN Not.* 40.1 (Jan. 2005), pp. 1–13. ISSN: 0362-1340. DOI: 10.1145/1047659.1040306.

[7] Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. "The architecture of the Utrecht Haskell compiler". In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*. Haskell '09. Edinburgh, Scotland: Association for Computing Machinery, 2009, pp. 93–104. ISBN: 9781605585086. DOI: 10.1145/1596638.1596650. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/1596638.1596650.

[8] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. "Towards Haskell in the cloud". In: *SIGPLAN Not.* 46.12 (Sept. 2011), pp. 118–129. ISSN: 0362-1340. DOI: 10.1145/2096148.2034690.

[9] Levent Erkök and John Launchbury. "A recursive do for Haskell". In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell '02. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2002, pp. 29–37. ISBN: 1581136056. DOI: 10.1145/581690.581693.

[10] Taylor Fausak. *State of Haskell Survey Results*. https://taylor.fausak.me/2022/11/18/haskell-survey-results. [Accessed 11-06-2024]. 2022.

[11] Hackage. *Introduction*. https://hackage.haskell.org/. [Accessed 11-06-2024].

[12] Hackage. *Uploading packages and package candidates*. https://hackage.haskell.org/upload. [Accessed 11-06-2024].

[13] Paul Hudak et al. "A history of Haskell: being lazy with class". In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. 2007, pp. 12–1.

[14] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[15] Geoffrey Mainland. "Why it's nice to be quoted: quasiquoting for haskell". In: *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*. Haskell '07. Freiburg, Germany: Association for Computing Machinery, 2007, pp. 73–82. ISBN: 9781595936745. DOI: 10.1145/1291201.1291211.

[16] Simon Marlow et al. "Haskell 2010 language report". In: (2010).

[17] Flávio Medeiros et al. "The Love/Hate Relationship with the C Preprocessor: An Interview Study". In: *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Ed. by John Tang Boyland. Vol. 37. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015, pp. 495–518. ISBN: 978-3-939897-86-6. DOI: 10.4230/LIPIcs.ECOOP.2015.495.

[18] Matthew Pickering et al. "Pattern synonyms". In: *SIGPLAN Not.* 51.12 (Sept. 2016), pp. 80–91. ISSN: 0362-1340. DOI: 10.1145/3241625.2976013.

[19] Veronika Romashkina and Dmitrii Kovanikov. *extensions*. https://hackage.haskell.org/package/extensions. [Accessed 11-06-2024]. 2022.

[20] Tom Schrijvers et al. "Type checking with open type functions". In: *SIGPLAN Not.* 43.9 (Sept. 2008), pp. 51–62. ISSN: 0362-1340. DOI: 10.1145/1411203.1411215.

[21] Jan Stolarek, Simon Peyton Jones, and Richard A. Eisenberg. "Injective type families for Haskell". In: *SIGPLAN Not.* 50.12 (Aug. 2015), pp. 118–128. ISSN: 0362-1340. DOI: 10.1145/2887747.2804314.

[22] GHC Team. *Controlling editions and extensions*. https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/control.html. [Accessed 23-06-2024]. 2023.

[23] GHC Team. *Extending and using GHC as a Library*. https://downloads.haskell.org/ghc/latest/docs/users_guide/extending_ghc.html. [Accessed 23-06-2024]. 2023.

[24] GHC Team. *Language extensions*. https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts.html. [Accessed 23-06-2024]. 2023.

[25] GHC Team. *Pragmas*. https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html. [Accessed 23-06-2024]. 2023.

[26] Haskell Wiki. *Use of language extensions*. https://wiki.haskell.org/Use_of_language_extensions. [Accessed 11-06-2024]. 2021.

Table 3: Usage of GHC extensions

| Extension | # | Extension | # | Extension | # |
|---|---|---|---|---|---|
| OverloadedStrings | 7270 | AllowAmbiguousTypes | 886 | OverloadedRecordDot | 134 |
| FlexibleInstances | 6150 | DuplicateRecordFields | 876 | RebindableSyntax | 132 |
| ScopedTypeVariables | 5755 | InstanceSigs | 827 | ExtendedDefaultRules | 130 |
| FlexibleContexts | 5444 | NoMonomorphismRestriction | 731 | ImpredicativeTypes | 123 |
| TypeFamilies | 4696 | StrictData | 705 | Strict | 118 |
| MultiParamTypeClasses | 4366 | UnboxedTuples | 563 | MonadComprehensions | 110 |
| CPP | 4287 | DerivingVia | 541 | UnliftedFFITypes | 94 |
| RankNTypes | 4109 | BlockArguments | 474 | NegativeLiterals | 92 |
| GeneralizedNewtypeDeriving | 3966 | DisambiguateRecordFields | 459 | Unsafe | 89 |
| DeriveGeneric | 3708 | Trustworthy | 438 | GADTSyntax | 67 |
| LambdaCase | 3696 | OverloadedLists | 402 | CApiFFI | 67 |
| RecordWildCards | 3549 | PackageImports | 399 | ConstrainedClassMethods | 66 |
| TemplateHaskell | 3257 | UnicodeSyntax | 398 | PostfixOperators | 57 |
| TypeOperators | 3067 | PartialTypeSignatures | 381 | NumDecimals | 53 |
| DataKinds | 3057 | EmptyCase | 363 | EmptyDataDeriving | 44 |
| DeriveDataTypeable | 3040 | Safe | 338 | UnboxedSums | 38 |
| UndecidableInstances | 2857 | OverlappingInstances | 315 | NondecreasingIndentation | 37 |
| BangPatterns | 2730 | ApplicativeDo | 304 | NamedWildCards | 31 |
| GADTs | 2652 | Arrows | 266 | GHCForeignImportPrim | 26 |
| StandaloneDeriving | 2199 | ParallelListComp | 255 | HexFloatLiterals | 25 |
| TupleSections | 2191 | OverloadedLabels | 254 | InterruptibleFFI | 24 |
| TypeApplications | 2100 | RoleAnnotations | 253 | JavaScriptFFI | 23 |
| ConstraintKinds | 1853 | LiberalTypeSynonyms | 247 | AutoDeriveTypeable | 21 |
| DeriveFunctor | 1847 | TypeFamilyDependencies | 243 | UnliftedNewtypes | 19 |
| NoImplicitPrelude | 1810 | DeriveLift | 240 | TransformListComp | 16 |
| FunctionalDependencies | 1702 | QuantifiedConstraints | 238 | StaticPointers | 15 |
| ViewPatterns | 1680 | ImportQualifiedPost | 226 | LinearTypes | 12 |
| TypeSynonymInstances | 1638 | RecursiveDo | 222 | TypeAbstractions | 9 |
| NamedFieldPuns | 1586 | BinaryLiterals | 211 | QualifiedDo | 8 |
| KindSignatures | 1538 | NumericUnderscores | 208 | NoPatternGuards | 8 |
| QuasiQuotes | 1383 | TypeInType | 192 | NoForeignFunctionInterface | 6 |
| DerivingStrategies | 1322 | MonoLocalBinds | 185 | NullaryTypeClasses | 5 |
| ExistentialQuantification | 1297 | ImplicitParams | 180 | UnliftedDatatypes | 4 |
| PatternSynonyms | 1264 | ExplicitForAll | 177 | ParallelArrays | 3 |
| PolyKinds | 1245 | ExplicitNamespaces | 173 | LexicalNegation | 3 |
| DefaultSignatures | 1182 | NoStarIsType | 167 | FieldSelectors | 3 |
| DeriveTraversable | 1113 | UndecidableSuperClasses | 159 | TypeData | 2 |
| MagicHash | 949 | TemplateHaskellQuotes | 145 | OverloadedRecordUpdate | 1 |
| MultiWayIf | 928 | StandaloneKindSignatures | 142 | NoDatatypeContexts | 1 |
| DeriveAnyClass | 926 | IncoherentInstances | 142 | NoCUSKs | 1 |
| DeriveFoldable | 911 | ExtendedLiterals | 1 | | |

Table 4: Cooccurrence between project tags and used extensions. Only the 50 most commonly used extensions, and tags used at least 300 times, are shown.

| Extension | library | bsd3 | program | mit | data | web | network | deprecated | text | gpl | development | control | system | language | math | graphics | database | apache | mpl | unclassified | testing | aws | cloud | data-structures | public-domain |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OverloadedStrings | 4475 | 2485 | 1321 | 1247 | 448 | 1118 | 560 | 260 | 361 | 275 | 280 | 70 | 207 | 138 | 37 | 98 | 268 | 240 | 80 | 138 | 115 | 16 | 133 | 43 | 47 |
| FlexibleInstances | 3895 | 2334 | 640 | 827 | 614 | 604 | 298 | 252 | 226 | 203 | 156 | 260 | 109 | 186 | 170 | 89 | 181 | 137 | 44 | 107 | 125 | 4 | 120 | 122 | 55 |
| ScopedTypeVariables | 3617 | 2215 | 735 | 831 | 597 | 591 | 213 | 188 | 181 | 190 | 177 | 182 | 131 | 140 | 150 | 98 | 165 | 166 | 65 | 105 | 145 | 6 | 16 | 104 | 29 |
| FlexibleContexts | 3441 | 2060 | 647 | 830 | 541 | 560 | 178 | 241 | 190 | 181 | 144 | 208 | 86 | 172 | 178 | 110 | 164 | 143 | 45 | 106 | 112 | 5 | 11 | 92 | 50 |
| TypeFamilies | 2792 | 1588 | 365 | 643 | 516 | 478 | 223 | 147 | 105 | 129 | 97 | 191 | 73 | 102 | 128 | 67 | 126 | 121 | 44 | 67 | 80 | 3 | 119 | 92 | 22 |
| MultiParamTypeClasses | 2724 | 1647 | 431 | 607 | 457 | 467 | 119 | 182 | 111 | 149 | 88 | 242 | 78 | 115 | 124 | 63 | 130 | 97 | 32 | 71 | 77 | 2 | 6 | 92 | 41 |
| CPP | 2691 | 1796 | 504 | 545 | 476 | 383 | 156 | 150 | 201 | 111 | 182 | 133 | 184 | 122 | 114 | 73 | 110 | 117 | 31 | 33 | 114 | 6 | 5 | 76 | 38 |
| RankNTypes | 2577 | 1566 | 413 | 575 | 461 | 372 | 136 | 151 | 125 | 121 | 118 | 222 | 94 | 91 | 95 | 56 | 118 | 114 | 43 | 78 | 73 | 6 | 7 | 84 | 24 |
| GeneralizedNewtypeDeriving | 2258 | 1320 | 437 | 556 | 314 | 369 | 156 | 140 | 129 | 115 | 95 | 119 | 79 | 92 | 84 | 50 | 144 | 101 | 54 | 71 | 72 | 5 | 5 | 53 | 23 |
| DeriveGeneric | 2112 | 1081 | 461 | 544 | 332 | 374 | 256 | 94 | 114 | 110 | 99 | 56 | 63 | 77 | 50 | 38 | 108 | 125 | 48 | 75 | 60 | 8 | 126 | 49 | 12 |
| LambdaCase | 2115 | 1013 | 500 | 569 | 296 | 268 | 244 | 129 | 121 | 120 | 139 | 90 | 77 | 73 | 31 | 50 | 103 | 161 | 69 | 90 | 70 | 6 | 118 | 40 | 9 |
| RecordWildCards | 1953 | 995 | 567 | 547 | 168 | 380 | 272 | 92 | 105 | 106 | 150 | 39 | 90 | 51 | 25 | 56 | 109 | 114 | 54 | 81 | 69 | 6 | 124 | 25 | 13 |
| TemplateHaskell | 1979 | 1142 | 497 | 516 | 291 | 364 | 127 | 121 | 130 | 133 | 110 | 78 | 72 | 111 | 45 | 48 | 108 | 93 | 26 | 51 | 51 | 10 | 8 | 35 | 28 |
| TypeOperators | 1947 | 1076 | 241 | 407 | 369 | 325 | 184 | 96 | 74 | 94 | 68 | 126 | 33 | 67 | 78 | 32 | 60 | 103 | 31 | 48 | 74 | 2 | 118 | 61 | 7 |
| DataKinds | 1982 | 1015 | 313 | 471 | 343 | 345 | 198 | 88 | 68 | 84 | 75 | 98 | 40 | 59 | 62 | 29 | 91 | 136 | 40 | 68 | 63 | 2 | 120 | 45 | 6 |
| DeriveDataTypeable | 1822 | 1052 | 384 | 440 | 278 | 255 | 243 | 141 | 113 | 100 | 108 | 66 | 70 | 110 | 51 | 28 | 96 | 50 | 15 | 44 | 56 | 4 | 115 | 41 | 20 |
| UndecidableInstances | 1819 | 1135 | 203 | 345 | 366 | 268 | 69 | 105 | 81 | 75 | 47 | 195 | 49 | 79 | 86 | 35 | 86 | 73 | 30 | 37 | 43 | 1 | 4 | 70 | 24 |
| BangPatterns | 1610 | 975 | 312 | 417 | 324 | 146 | 80 | 91 | 82 | 77 | 80 | 61 | 66 | 84 | 111 | 48 | 75 | 67 | 23 | 52 | 26 | 2 | 1 | 79 | 15 |
| GADTs | 1649 | 904 | 248 | 431 | 289 | 257 | 78 | 95 | 65 | 89 | 80 | 134 | 43 | 70 | 60 | 24 | 80 | 66 | 21 | 54 | 54 | 1 | 2 | 48 | 17 |
| StandaloneDeriving | 1315 | 711 | 226 | 347 | 226 | 153 | 57 | 96 | 70 | 64 | 56 | 79 | 43 | 65 | 51 | 18 | 85 | 66 | 48 | 61 | 39 | 2 | 2 | 45 | 16 |
| TupleSections | 1286 | 711 | 333 | 348 | 149 | 210 | 82 | 64 | 79 | 74 | 85 | 64 | 49 | 59 | 23 | 28 | 67 | 72 | 46 | 47 | 30 | 3 | 6 | 39 | 8 |
| TypeApplications | 1330 | 672 | 242 | 355 | 245 | 163 | 57 | 43 | 55 | 54 | 57 | 65 | 32 | 31 | 41 | 18 | 56 | 112 | 55 | 49 | 73 | 3 | 4 | 38 | 4 |
| DeriveFunctor | 1127 | 635 | 174 | 304 | 189 | 147 | 35 | 93 | 63 | 63 | 39 | 64 | 24 | 64 | 42 | 24 | 53 | 50 | 20 | 73 | 29 | 4 | 2 | 44 | 3 |
| ConstraintKinds | 1135 | 564 | 150 | 336 | 191 | 164 | 42 | 85 | 43 | 52 | 33 | 78 | 19 | 34 | 40 | 20 | 61 | 60 | 43 | 67 | 38 | 4 | 3 | 36 | 6 |
| NoImplicitPrelude | 964 | 401 | 154 | 314 | 136 | 95 | 148 | 80 | 37 | 39 | 43 | 48 | 44 | 51 | 43 | 7 | 29 | 55 | 9 | 29 | 18 | 8 | 116 | 15 | 8 |
| FunctionalDependencies | 1047 | 589 | 132 | 254 | 178 | 144 | 50 | 70 | 45 | 40 | 34 | 113 | 36 | 40 | 32 | 23 | 53 | 42 | 20 | 36 | 13 | 1 | 2 | 32 | 11 |
| ViewPatterns | 948 | 519 | 260 | 226 | 145 | 138 | 42 | 49 | 55 | 56 | 105 | 31 | 33 | 44 | 40 | 23 | 30 | 86 | 42 | 31 | 27 | 2 | 3 | 28 | 7 |
| TypeSynonymInstances | 920 | 610 | 206 | 187 | 107 | 157 | 51 | 73 | 91 | 50 | 48 | 35 | 27 | 63 | 32 | 30 | 42 | 16 | 6 | 11 | 29 | 2 | 2 | 26 | 8 |
| NamedFieldPuns | 761 | 411 | 238 | 164 | 62 | 132 | 58 | 51 | 32 | 61 | 80 | 21 | 32 | 25 | 6 | 9 | 50 | 71 | 28 | 36 | 35 | 3 | 2 | 12 | 2 |
| KindSignatures | 953 | 580 | 135 | 173 | 199 | 160 | 36 | 39 | 34 | 40 | 27 | 61 | 29 | 40 | 43 | 15 | 33 | 38 | 37 | 26 | 20 | 1 | 4 | 34 | 3 |
| QuasiQuotes | 862 | 427 | 240 | 335 | 83 | 243 | 49 | 48 | 68 | 35 | 38 | 23 | 22 | 39 | 8 | 14 | 77 | 35 | 12 | 29 | 31 | 2 | 2 | 10 | 6 |
| DerivingStrategies | 614 | 295 | 123 | 152 | 110 | 83 | 34 | 14 | 19 | 21 | 32 | 12 | 19 | 14 | 11 | 12 | 47 | 59 | 44 | 25 | 24 | 1 | 0 | 19 | 0 |
| ExistentialQuantification | 735 | 450 | 156 | 150 | 89 | 111 | 32 | 66 | 37 | 35 | 42 | 55 | 35 | 31 | 13 | 20 | 54 | 36 | 5 | 13 | 25 | 1 | 1 | 19 | 9 |
| PatternSynonyms | 545 | 303 | 80 | 103 | 122 | 33 | 22 | 14 | 26 | 23 | 44 | 27 | 14 | 28 | 20 | 25 | 15 | 50 | 17 | 12 | 11 | 2 | 1 | 21 | 1 |
| PolyKinds | 793 | 484 | 77 | 156 | 169 | 120 | 29 | 69 | 26 | 33 | 17 | 65 | 11 | 28 | 28 | 13 | 23 | 43 | 23 | 30 | 23 | 1 | 2 | 28 | 2 |
| DefaultSignatures | 741 | 344 | 80 | 228 | 161 | 91 | 26 | 43 | 35 | 35 | 18 | 40 | 17 | 27 | 29 | 11 | 46 | 42 | 13 | 28 | 20 | 1 | 0 | 23 | 2 |
| DeriveTraversable | 670 | 347 | 85 | 200 | 136 | 66 | 16 | 67 | 34 | 31 | 16 | 32 | 12 | 31 | 29 | 13 | 35 | 33 | 8 | 49 | 14 | 4 | 1 | 28 | 1 |
| MagicHash | 587 | 338 | 46 | 170 | 180 | 31 | 20 | 35 | 32 | 10 | 25 | 26 | 16 | 21 | 31 | 13 | 19 | 25 | 6 | 18 | 6 | 1 | 0 | 36 | 1 |
| MultiWayIf | 537 | 231 | 117 | 198 | 84 | 51 | 27 | 42 | 29 | 29 | 42 | 15 | 26 | 17 | 20 | 11 | 33 | 38 | 15 | 24 | 11 | 1 | 0 | 11 | 2 |
| DeriveAnyClass | 526 | 257 | 141 | 138 | 108 | 85 | 39 | 15 | 24 | 22 | 40 | 12 | 16 | 23 | 6 | 12 | 28 | 55 | 16 | 18 | 11 | 1 | 1 | 14 | 3 |
| DeriveFoldable | 552 | 283 | 66 | 175 | 109 | 55 | 14 | 63 | 27 | 29 | 12 | 26 | 8 | 31 | 24 | 8 | 25 | 24 | 9 | 45 | 11 | 4 | 1 | 23 | 0 |
| AllowAmbiguousTypes | 545 | 254 | 66 | 148 | 113 | 59 | 18 | 11 | 20 | 31 | 13 | 39 | 9 | 19 | 13 | 8 | 19 | 46 | 17 | 18 | 35 | 0 | 0 | 15 | 1 |
| DuplicateRecordFields | 320 | 152 | 76 | 77 | 67 | 55 | 21 | 14 | 5 | 13 | 31 | 2 | 6 | 9 | 1 | 4 | 11 | 36 | 4 | 8 | 3 | 0 | 1 | 6 | 1 |
| InstanceSigs | 503 | 254 | 69 | 108 | 98 | 76 | 16 | 54 | 22 | 26 | 17 | 24 | 10 | 9 | 18 | 7 | 18 | 37 | 38 | 34 | 13 | 1 | 1 | 15 | 2 |
| NoMonomorphismRestriction | 403 | 214 | 86 | 145 | 50 | 56 | 15 | 32 | 21 | 36 | 18 | 37 | 11 | 16 | 13 | 13 | 19 | 9 | 2 | 16 | 6 | 0 | 1 | 12 | 3 |
| StrictData | 209 | 114 | 53 | 51 | 27 | 38 | 19 | 6 | 11 | 18 | 9 | 3 | 4 | 6 | 2 | 8 | 20 | 13 | 9 | 10 | 4 | 0 | 1 | 1 | 0 |
| UnboxedTuples | 343 | 185 | 23 | 118 | 106 | 18 | 9 | 13 | 14 | 6 | 19 | 18 | 11 | 3 | 17 | 4 | 13 | 12 | 0 | 13 | 4 | 1 | 0 | 19 | 1 |
| DerivingVia | 320 | 137 | 39 | 74 | 64 | 31 | 11 | 10 | 13 | 8 | 10 | 17 | 12 | 6 | 9 | 2 | 22 | 41 | 14 | 14 | 13 | 0 | 0 | 14 | 0 |
| BlockArguments | 288 | 133 | 60 | 72 | 36 | 19 | 12 | 3 | 22 | 8 | 11 | 16 | 12 | 5 | 5 | 11 | 13 | 34 | 12 | 15 | 2 | 0 | 1 | 3 | 0 |
| DisambiguateRecordFields | 73 | 26 | 17 | 18 | 6 | 4 | 9 | 4 | 3 | 5 | 8 | 0 | 1 | 2 | 2 | 1 | 8 | 9 | 2 | 2 | 2 | 0 | 0 | 0 | 0 |