# Stretching the Glasgow Haskell Compiler

## Nourishing GHC with Domain-Driven Design


Jeffrey M. Young
jeffrey.young@iohk.io
IOG
USA

Sylvain Henry
sylvain.henry@iohk.io
IOG
France

John Ericson
john.ericson@obsidian.systems
Obsidian Systems
USA



## Abstract

Over the last decade Haskell has been productized; transitioning from a research language to an industrial strength language ready for large-scale systems. However, the literature on architecting such systems with a pure functional language is scarce. In this paper we contribute to that discourse, by using a large-scale system: the Glasgow Haskell Compiler (GHC), as a guide to more maintainable, flexible and effective, pure functional architectures. We describe, from experience, how GHC *as a system*, violates the desirable properties that make pure functional programming attractive: immutability, modularity, and composability. With these violations identified, we provide guidance for other functional system architectures; drawing heavily on Domain-Driven Design. We write from an engineering perspective, with the hope that our experience may provide insight into best practices for other pure functional software architects.



***CCS Concepts:*** • **Software and its engineering → Layered systems**; **Abstraction, modeling and modularity**.

***Keywords:*** functional systems, modularity, supple design

**ACM Reference Format:**

Jeffrey M. Young, Sylvain Henry, and John Ericson. 2023. Stretching the Glasgow Haskell Compiler: Nourishing GHC with Domain-Driven Design. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Software Architecture (FUNARCH '23), September 8, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3609025.3609476




*FUNARCH '23, September 8, 2023, Seattle, WA, USA*



ACM ISBN 979-8-4007-0297-6/23/09…$15.00
https://doi.org/10.1145/3609025.3609476


## 1 Introduction

In the keynote of last year's Haskell symposium[1], there were two interesting statements; both of which are true: "We need to build real tools and not just little demos", and "Haskell code does bitrot faster than SML code". Imagine we rise to this challenge and begin work on a real tool for modern Haskell. We would run head first into a significant issue: modern Haskell is whatever the Glorious Glasgow Haskell Compiler supports.

GHC distinguishes what it supports from the Haskell 2010 specification [Marlow 2010] by guarding new features behind feature flags called *language extensions*. But this is little help to the tool writer: language extensions are ubiquitous in modern Haskell, so any tool that supports modern Haskell must handle them as much as it handles the base language. This is so much work that almost every project decides to use the GHC API: a library called ghc-lib[2]. This is the case for: documentation generators (haddock) and example testers (doctest), REPLs (ghci), linters (hlint), IDEs (haskell-language-server), cross-compilers (GHCJS, Asterius, Eta), etc.

Unfortunately, ghc-lib's usability is poor because its design is *organic*. Over two decades, new features have been force-fitted into an existing architecture without a proper redesign. The resulting ghc-lib is difficult to understand, difficult to use, hard to modify, and buggy. This is not speculation, rather, most third-party tools (including every cross-compiler) that has depended on ghc-lib have historically been doomed to bitrot: Eta[3] lasted until GHC-7.10.3, and GHCJS and Asterius lasted to GHC-8.10.7. Tools that have survived, have survived by being pinned to the GHC repository, such as ghci and haddock. A notable exception is the Haskell Language Server (HLS), which has survived by sheer force of will driven by a group of maintainers; some of whom are also GHC developers.

The essential problem is this: in the last decade there has been a proliferation of new clients that depend on ghc-lib

[1] Haskell'22 https://youtu.be/2PGUt_dcHX0

[2] Ghc-lib is the bulk of GHC's code. It is the library used by the GHC program to compile Haskell code.

[3] "The GHC API was used initially, but the rigidity of the API forced me to inline the entire GHC frontend into the GHCVM [Eta] codebase."—abstract of Rahul Muttineni's HIW2016 talk

with vastly different use cases. This has expanded the *requirement distance* beyond what the monolithic, anti-modular architecture of ghc-lib can sustain. For example, consider the requirement distance between the GHC program and an IDE tool. The GHC program executes the whole compilation pipeline (from parsing to code generation), while on the other end of the spectrum, an IDE executes only the frontend (parser, type-checker) possibly after every user keystroke in the code editor. These two use cases have opposite concerns (one-shot *vs* long-running session, latency impact, etc.).

To put it simply: all tools in the Haskell ecosystem are incentivized to use ghc-lib, ghc-lib couples the tool to GHC development; GHC, being a vehicle for research, rapidly changes, thus every tool begins to bitrot. Therefore the burden of maintenance and barrier to entry increases for a real tool compared to a proof of concept and compared to other language ecosystems.

We believe, from our experience working on GHC, that the *design and architecture* of ghc-lib is a primary reason for these *ecosystem* issues. Our core idea is that a more modular design, based in the principles of Domain-Driven Design (DDD) [Evans 2003] will yield a more robust, usable, maintainable ghc-lib and consequently, a healthier Haskell ecosystem. Our contributions are:

- We present design flaws in ghc-lib (Section 2).
- We present recommendations based on principles from DDD and adapt these ideas to large-scale systems written in a pure functional language. (Section 3)

This paper is a consolidated conference version of our widely circulated white paper [Henry et al. 2022]. While the white paper focused on understanding the organic genesis of ghc-lib's design issues, this paper is focused on contributing to the software architecture discourse using GHC as an example and lodestone.

## 2 Anti-Modular Anti-Patterns in GHC

In this section, we explore the design issue that limit ghc-lib. Throughout the paper, we use the words "subsystem" and "component" interchangeably.

### 2.1 Problem 1: Shotgun Parsing

Shotgun parsing is a anti-pattern defined in [Momot et al. 2016]:

> Shotgun parsing is a programming anti-pattern whereby parsing and input-validating code is mixed with and spread across processing code—throwing a cloud of checks at the input, and hoping, without any systematic justification, that one or another would catch all the "bad" cases.

**Listing 1.** Example of shotgun parsing and the validate, don't parse anti-pattern in GHC. The function validates its argument in a guard which calls the predicate `isHoleModule`. The validation result is not captured at the type level.

```
loadInterface :: SDoc -> Module -> WhereFrom
  -> IfM lcl (MaybeErr SDoc ModIface)
loadInterface doc_str modul from
  | isHoleModule modul
  = do hsc_env <- getTopEnv
       let home_unit = hsc_home_unit hsc_env
       loadInterface doc_str
         (mkHomeModule home_unit
                       (moduleName modul))
         from
  | otherwise = ...
```

A system prone to shotgun parsing relies on runtime checks, not compile time checks, for correctness. Such a system fails to utilize types to model the domain and check the implementation's correctness. This degrades the value of types in the implementation, and requires each subsystem to be able to validate, which couples the processing subsystem to the input validation subsystem.

GHC exhibits a high degree of shotgun parsing. In Haskell, this anti-pattern manifests through excessive use of guard patterns. For example, consider the function `loadInterface` shown in Listing 1.

`loadInterface` first checks if its input is a hole module via a guard which calls `isHoleModule`. If the input is a hole module, then `loadInterface` enters a special case: it constructs a home unit and enters a recursive call. This code mixes input-validation (checking the module) with processing code (loading the interface file). Furthermore, the validation result is not captured at the type level, so if another function uses this `Module` *after* `loadInterface` then it must redundantly perform the hole check.

In GHC, shotgun parsing organically grew as the aggregation of numerous decisions distributed over numerous projects over several years. This particular case originates from Backpack [Yang 2017]. Since the introduction of Backpack, package components (libraries, executables) may have module holes that can be instantiated with other modules. This feature requires that the module system can determine which modules are holed, and which are whole. Rather than separate these distinct cases into separate constructors or types, which would require altering a significant portion of the module system, the implementation avoided a redesign and settled on adding a special "hole" value to the `Module` type. This allows any `Module` to be checked against the privileged value to determine if the `Module` is holed or not.

The result of this design is that representation of real modules and module holes differs, but the types do not. So the name and type `Module` in the *implementation* remained

while the *concept* changed because the path of least resistance was not redesigning with a proper domain analysis, even though the domain changed.

The key takeaway is this: One cannot avoid typing the domain of their system. Either types will be aids in the craft of software and will be represented in the type level of the implementation, or, the types will manifest via shotgun parsing as redundant, ad-hoc runtime checks that couple subsystems together.

### 2.2 Problem 2: Layering Violations

A layering violation is a breach in the abstraction boundary between one or more subsystems. For any two subsystems, layering violations occur when subsystem B relies on components in subsystem A that are not exposed by A's API. Layering violations are the nucleation sites of brittle software; they drive coupling, violate the separation of concerns, and grow bitrot.

GHC's command line flag handling is a classic example of a layering violation. After parsing, command-line flags are represented as a 191 field record called `DynFlags`. `DynFlags`'s purpose should be to pass information along to the compiler's driver (in ghc-lib), so that *the driver* can coordinate the relevant information to specific subsystems. However, `DynFlags` pervades ghc-lib as well as the GHC program, thereby creating a layering violation.

In the GHC code base, the `DynFlags` layering violation is expressed by `DynFlags` being an input to functions in subsystems that are unrelated to the functions' purpose. For example, consider the function to create an `Int#` literal in Core syntax (present up to GHC 8.10):

```
-- | Creates a 'Literal' of type @Int#@
mkLitInt :: DynFlags -> Integer -> Literal
```

`DynFlags` is an argument because the size of an `Int#` depends on the target architecture (32-bit, 64-bit), and GHC learns about the target architecture by reading its settings file whose contents is stored in the `DynFlags` record[4]. Thus, any ghc-lib client wanting to manipulate Core syntax must provide a `DynFlags` value, all 191 fields.

To understand the effect of this layering violation, consider the perspective of a caller of `mkLitInt` that resides in another subsystem. It is unreasonable for the caller to know which `DynFlags` fields `mkLitInt` requires, and if the caller had this information then that itself would violate the abstraction boundary. So the caller of `mkLitInt` must input `DynFlags` only to pass it to `mkLitInt`; thereby proliferating the layering violation.

This is not theoretical; before our work on modularizing GHC, `DynFlags` had proliferated into the Cmm and the ASM generators [Henry et al. 2022] and into GHC's parser. This made testing these subsystems difficult, each test required a `DynFlags` record, but such a record could change the system behavior in unforeseen ways.

This design limits clients. They must either provide a `DynFlags` record, or understand the inner workings of unrelated components. Both options increase coupling and bitrot by binding client code to GHC releases.

We can see this binding in HLS. GHC exhibits a layering violation in its error messages. The violation is that GHC does not expose a structured representation of errors. Instead, it couples a domain element, a type error, with a means of presentation, text, and exposes the text. This forces clients that wish to compute on errors, like HLS, to implement a parser. Which is predictably brittle and binds the client code to a specific GHC release. For example, GHC-9.4 exhibited a bug[5] that subtly changed error messages. This in turn broke a code action[6] in HLS because the error message could not be parsed. At time of writing there is a partial fix[7] but the author laments that a proper fix is requires changes in GHC.

The key takeaway is layering violations have two effects: They alter the abstraction boundaries between subsystems by distributing the information a subsystem requires, to another, unrelated subsystem. And worse, they are infectious; to avoid the information burden the most general type that works (such as `DynFlags`) will be used, which spreads the layering violation throughout the system and eventually to interfacing systems.

## 3 GHC with Domain-Driven Design

In this section, we describe the principles that guide our efforts. We utilize the ideas of DDD that we believe are relevant to pure functional programming[8].

### 3.1 Design Principle: Supple Design

We begin with the guiding principle behind our work: *supple design*. A design is supple if it is flexible, inviting and easy to change [Evans 2003] (pp. 243–245):

> "When software doesn't have a clean design, developers dread even looking at the existing mess, much less making a change that could aggravate the tangle or break something through an unforeseen dependency. In any but the smallest systems, this fragility places a ceiling on the richness of behavior it is feasible to build. It stops refactoring and iterative refinement… To have a project accelerate as development proceeds—rather than get weighed down by its own legacy—demands a design

[4] This is separately a violation of separation of concerns.

[5] https://gitlab.haskell.org/ghc/ghc/-/issues/22130

[6] https://github.com/haskell/haskell-language-server/issues/3473

[7] https://github.com/haskell/haskell-language-server/pull/3712

[8] This excludes already applied principles like immutability.

> that is a pleasure to work with, inviting to change. A supple design."

As system architects we should strive to achieve supple designs and architectures. The benefits are not just phenomenological, but also pragmatic. Non-supple designs resist change and therefore resist experimentation. Resisting experimentation is costly. A system that is hard to experiment on is hard to improve, hard to redesign, *is hard to learn from.* Thus, not only does the existing system petrify, but the body of knowledge to be gained from the existing system is reduced.

Fortunately, Haskell incentivizes supple design by encouraging architects to think deeply about the problem domain, and implement a precise and accurate model of the domain in the type system. A precise model makes obvious a correct implementation from an incorrect one. So an implementation with a precise model can discern more easily if a given change maintains correctness, thereby making the system more supple.

Consider again the example of Backpack and its use of shotgun parsing. We can now clarify that shotgun parsing is a bad design because it creates a less supple architecture. In subsection 2.1 we argued that shotgun parsing degrades the value of types in the implementation. When critical checks and concepts are in the type system the type checker can express exactly where alterations need to occur, making the system easier to change. Whereas with shotgun parsing, it is left to the developer to understand where changes must occur.

***Recommended concrete action.*** Our recommended action is to *embrace the domain modeling that Haskell encourages. A domain that is well modeled is a domain that is better understood and better implemented.*

### 3.2 Design Principle: Use a Ubiquitous Language

*Ubiquitous language* is the principle of using *precise domain* terminology *consistently* in code, in documentation, and in speech. In Haskell, we extend this principle to require that domain terminology be represented at the type-level in the implementation.[9] By doing this, function signatures have intention revealing interfaces and become a tool of thought [Iverson 2007] because they denote meaning and make statements about the domain.

Consider again ghc-lib. The concept of a "package" changed from "a set of modules whose code objects are bundled into a single library" to something more ill-defined: A Cabal package may contain several library components and each of them can be compiled into different units[10]; modules of a unit are bundled into a single library. Modern GHC deals with units instead of packages but the old language still pervades ghc-lib.

Using ubiquitous language would make obvious that GHC's user interface is ambiguous. For example, "package-qualified imports" and `-package-name` may refer to different units[11], and that some cabal-install's "projects" could perhaps be better subsumed by a hierarchical package component namespace.

***Recommended concrete action.*** Our recommended action is *to make naming precise, meaningful in the domain, consistent, documented, and checked by the type system* by adding new datatypes and refactoring as needed. Furthermore, include a glossary of terms and concepts that serves as a single source of truth for the language used to construct and refer to the system.

### 3.3 Design Principle: Layering

An important DDD principle is to divide complex software into conceptual layers, each of which is composed of distinct components or subsystems.[12]

> "The essential principle is that any element of a layer depends only on other elements in the same layer or on elements of the layer 'beneath' it. Communication upward must pass through some indirect mechanism" [Evans 2003] p. 69

We mapping the four standard layers to GHC as follows; with the presentation layer at the surface, thereby depending on each other layer:

- **Presentation Layer**: The user interface; this layer is responsible for interaction with the user or with another system. In GHC, this is code in GHC-the-program, ghci, HLS, or other clients of ghc-lib.
- **Application Layer**: This layer defines the jobs the system will execute, and directs the domain components to work out these problems. In GHC, this layer is the driver code in ghc-lib that constructs and executes build plans, does recompilation avoidance, etc.

[9] This idea is similar to type-driven design.

[10] depending on compilation options, dependencies, etc.

[11] This is well known; the GHC user guide even states that this feature leads to brittle dependencies. Our point is that these problems would have been obvious earlier with a ubiquitous language.

[12] This idea is also present in non-OOP literature. For example, the venerable SICP [Abelson et al. 1996] uses the terminology *stratified design* p. 140:

> "This is the approach of stratified design, the notion that a complex system should be structured as a sequence of levels that are described using a sequence of languages. Each level is constructed by combining parts that are regarded as primitive at that level, and the parts constructed at each level are used as primitives at the next level. The language used at each level of a stratified design has primitives, means of combination, and means of abstraction appropriate to that level of detail."

- **Domain Layer**: *This layer is the heart of business software.* It is responsible for representing concepts, rules, and information about the business. State that reflects the business situation is controlled and used here, even through the details of storing it are delegated to the infrastructure layer. In GHC, this code is the subject of research papers. It represents concepts, information, and rules about compiling Haskell programs: the compilation pipeline, different IRs (Haskell syntax, Core, STG, Cmm, ByteCode) and their transformers.
- **Infrastructure Layer**: The layer that provides generic technical capabilities that support the higher layers. In GHC, this is code supporting services: logging, dealing with file-systems, reporting errors and panics.

Layers should be further decomposed into subsystems which are composed of modules in a module hierarchy. For example, in GHC, modules in the `GHC.Stg.*` directory implement the STG syntax and its related code, and modules in `GHC.StgToCmm.*` directory implement the STG to C-- pass. Similarly, modules composing the Application Layer are confined in `GHC.Driver.*` hierarchy.

***Recommended concrete action.*** Our recommended action is to *use a layered architecture and represent the architecture in the module hierarchy.* This makes clear which components belong to which layer and makes layering violations obvious. For example, one would easily observe if the domain layer imports the presentation layer via `import UI.GHCi`[13].

### 3.4 Design Principle: Componentization

We only cover componentization of the infrastructure layer, we refer interested readers to the white paper [Henry et al. 2022] for other layers. Componentization of the infrastructure layer organizes the layer into datatypes that represent services the layer provides. A consumer then inputs the services they require. For example, componentizing ghc-lib's infrastructure layer changed ghc-lib's linker function signature from Listing 2 to Listing 3.

**Listing 2.** Non-componentized infrastructure bundles each service as fields in a record called `DynFlags`.

```
link :: DynFlags → ...
```

**Listing 3.** Componentized; services are not bundled into a record such as `DynFlags` .

```
link :: Logger → TmpFs → UnitEnv ...
```

Passing services to a function can look cumbersome: why not pass a single record, `SubSysEnv`? In our experience, this is how coupling begins. The sequence is:

1. Services for are bundled into records as `SubSysEnv`.
2. Functions which input `SubSysEnv` are part of this subsystem and thus have high coherence.
3. The code base evolves by adding new functions; some functions require more than what is in `SubSysEnv`, and some require services that exist in `SubSysEnv`.
4. This creates an incentive to alter `SubSysEnv`, or make a new `SubSysEnv`. Adding more records appears redundant, so the path of least resistance is taken. The existing `SubSysEnv` is expanded with fields required for the new functions. This grows `SubSysEnv`'s surface area, possibly past the original subsystem.
5. Now that `SubSysEnv` has grown, there are two effects: Coherence reduces because fewer functions use all of `SubSysEnv`. The incentive to pass `SubSysEnv` is stronger because `SubSysEnv`'s functionality has expanded.
6. A vicious loop occurs; more services and data are added into `SubSysEnv` because its surface area is so large, additions to it are almost always the path of least resistance. But `SubSysEnv` is large precisely because of the many services and data it provides.

This is exactly how `DynFlags` grew and proliferated in GHC and interfacing systems.

Componentization's value is in what it enables. In DDD, Componentization (and as we'll see in the next section, Component-wise configuration) enable *bounded contexts*. A bounded context is a part of the system where terms, definitions, and rules are applied in an internally consistent way. Componentization improves the internal consistency because different clients of a function can input different service implementations without changing unrelated services. For example `link` may input `SpecialLogger` *without* changing `TmpFs`. In contrast, the record style would silently manipulate or create a `DynFlags`. Likewise, Type signatures explicitly indicating the services a function uses. If a service is not passed to a function, then we can be certain it is not used. As a consequence, layering violations become obvious. Should a violation to occur, then an improper service must have been conspicuously passed to a client function.

***Recommended concrete action.*** Our recommended action is to *componentize services by representing components in the type system, explicitly pass each component and avoid passing records of functions.*

### 3.5 Design Principle: Component-Wise Configuration

Component-wise configuration is the idea that each subsystem should be treated as an external program: it should have its own bounded context, a distinct set of related concepts it is concerned with, and its interface must be well-defined with a subsystem-specific set of options. Furthermore, the

[13] In GHC, this module is actually `GHCi.UI`, which places the component before the layer. We prefer a layer-first naming convention and have written the example thusly

**Listing 4.** Cmm subsystem configuration.

```
data CmmConfig = CmmConfig
{ cmmProfile             :: !Profile
, cmmDoLinting           :: !Bool
, cmmOptSink             :: !Bool
...
```

**Listing 5.** Cmm handshake in the Application Layer.

```
initCmmConfig :: DynFlags -> CmmConfig
initCmmConfig dflags = CmmConfig
  { cmmProfile    = targetProfile         dflags
  , cmmDoLinting  = gopt Opt_DoCmmLinting dflags
  , cmmOptSink    = gopt Opt_CmmSink      dflags
  ...
```

set of options should be constructed by an upper layer and supplied to the component.[14]

As an example, consider the `Cmm` component. In the GHC pipeline, ghc-lib feeds the output from the `StgToCmm` component to downstream components: `CmmToAsm`, `CmmToLlvm`, or `CmmToWasm`. The downstream component is selected by a flag passed into the system, and represented in the presentation layer configuration, `DynFlags`.

We define a configuration datatype, shown in Listing 4, and a handshake function, shown in Listing 5 in the Application Layer (`GHC.Driver.Config.Cmm`). Then we use the handshake function at the interface between the `Cmm` component and the application layer to create and pass the `CmmConfig` to the `Cmm` component. This isolates the component so its only aware of its own configuration.[15]

Notice that the handshake function consumes the configuration of an *upper layer*: `DynFlags`, to produce the configuration for the `Cmm` component: `CmmConfig`, in the *lower layer*. Thus configuration information flows downward from the presentation layer to the domain layer.

Similarly, notice that the handshake function resides in the application layer and not in the component. This is purposeful; if the handshake function existed in the `Cmm` component, then the `Cmm` component would know about `DynFlags`. But the `Cmm` component shouldn't know `DynFlags` exists because `DynFlags` is the purview of the application layer only. Thus, the configuration input to the `Cmm` component is exactly `CmmConfig` and nothing else.

[14]This may seem counter to subsection 3.4. The key difference is that these records are to strictly limited to the component because the component is treated as an external program. Thus step 4 of the coupling sequence should never occur.

[15]No rocket science here: We don't expect C compilers to support being passed all of GHC's options and to know how to compile C files produced by GHC. Similarly we shouldn't expect ghc-lib's "internal compilers" (such as `CmmToAsm`) to support this either.

***Recommended concrete action.*** Our recommended action is to *treat components as if they are external programs by defining component-specific records and handshake functions to isolate the component.* The handshake function defines an abstraction barrier, enables "parse, don't validate" [King 2019] and improves suppleness.

## 4 Conclusion

We believe that pure functional languages are well suited to crafting supple large-scale systems. Isolating and tracking side-effects in the type system is a killer-feature for such systems. However, as we have demonstrated via GHC, and GHC's secondary effects on the Haskell ecosystem, if one is not careful, the system can lose the desirable properties that pure functional languages provide. Modularity is lost by layering violations and shotgun parsing, Immutability is lost when large ubiquitous records, like `DynFlags`, must be created ad-hoc with dummy values just for a particular subsystem, and finally composability is lost as a result of losing modularity and immutability.

To be succinct, the implementation language alone will not save us, but does allow us to save ourselves. We advocate for a design philosophy rooted in supple design. We must take up the mantle, change the status quo, diagnose the rigid aspects of our systems, and craft a healthier ecosystem. As system architects we must strive for a supple modular design, utilize the type system to learn the domain, decompose our systems into layers. We have only rigid uninviting systems to lose and supple systems to gain.

## References


[1] Harold Abelson, Gerald Jay Sussman, and with Julie Sussman. 1996. *Structure and Interpretation of Computer Programs* (2nd editon ed.). MIT Press/McGraw-Hill, Cambridge.

[2] Eric Evans. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software.*

[3] Sylvain Henry, John Ericsson, and Jeffrey M. Young. 2022. Modularizing GHC. https://hsyl20.fr/home/files/papers/2022-ghc-modularity.pdf.

[4] Kenneth E. Iverson. 2007. *Notation as a Tool of Thought.* Association for Computing Machinery, New York, NY, USA, 1979. https://doi.org/10.1145/1283920.1283935

[5] Alexis King. 2019. Parse, don't validate. https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/.

[6] Simon Marlow. 2010. Haskell 2010 Report. http://www.haskell.org/definition/haskell2010.pdf.

[7] Falcon Momot, Sergey Bratus, Sven M. Hallberg, and Meredith L. Patterson. 2016. The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them. In *2016 IEEE Cybersecurity Development (SecDev).* 45–52. https://doi.org/10.1109/SecDev.2016.019

[8] Edward Z. Yang. 2017. *Backpack: Towards Practical Mix-In Linking in Haskell.* Ph. D. Dissertation. Stanford University. https://github.com/ezyang/thesis.


Received 2023-06-01; accepted 2023-06-28