Six Years of FUNAR

Functional Training for Software Architects

Michael Sperber
sperber@deinprogramm.de
Active Group GmbH
Tübingen, Germany

Abstract

Since 2019, the International Software Architecture Qualification board has featured a three-day curriculum for Functional Software Architecture. We have taught more than 30 trainings based on this curriculum, mostly to audiences with little or no exposure to functional programming. This paper reports on our experience, and how content and delivery of the training has evolved over the past four years.

CCS Concepts: • **Computer systems organization → Architectures**; • **Software and its engineering → Software architectures**; **Functional languages**.

Keywords: functional programming, software architecture, education

ACM Reference Format:
Michael Sperber. 2025. Six Years of FUNAR: Functional Training for Software Architects. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Software Architecture (FUNARCH '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3759163.3760428

1 Introduction

In late 2017, Active Group ("we"), a small consultancy in southern Germany, started developing a curriculum on "Functional Software Architecture" to use in professional training, which is now part of the Advanced Level certification program of the International Software Architecture Qualification Board (iSAQB) under the acronym *FUNAR* [13].

We have since taught more than 30 iterations of this training as three-day courses, with a one-day prequel course on functional programming. This paper reports on our journey so far, the curriculum design process, and our experience teaching it to diverse audiences.

FUNARCH '25, Singapore, Singapore

ACM ISBN 979-8-4007-2146-5/25/10
https://doi.org/10.1145/3759163.3760428

Overview. Section 2 provides some background on the history leading up to the creation of FUNAR. Section 3 describes the design and rationale of the original curriculum. Section 4 describes the design of our concrete training. Section 5 describes our experience teaching the training and the evolution it has gone through. Section 6 briefly touches upon the curriculum revision made on the basis of the experience gained. Section 7 concludes.

2 Background

We have used functional programming in almost all of our projects since 2010. Since 2012, we had also been developing a professional training business. Initially, this resulted in only sporadic bookings for in-house courses. In the winter of 2017/2018, we determined that we needed help marketing our training offering. We contacted the International Software Architecture Qualification Board (iSAQB), a volunteer-driven association based in Germany whose goal is to further education on software architecture—techniques, tools, and processes that help with managing large software projects.

The iSAQB offers a single, widely taught "Foundation Level" curriculum with basic tenets of software architecture [14], as well as an "Advanced Level" certification program with a variety of different trainings on subjects such as Domain-Driven Design, embedded systems, cloud architecture, and microservice architectures.

Together with a few iSAQB members—notably Stefan Tilkov and Eberhard Wolff—we explored the possibility of adding a curriculum on Functional Software Architecture. We also enlisted the help of Nicole Rauch to develop the curriculum, as we had little experience engaging with the software architecture community at large, and lacked a common terminological basis.

3 Curriculum Design

When we started to draft the curriculum, we realized that there was no coherent, written-up discipline for functional programming in the large, even though many companies had successfully deployed large functionally programmed software systems. Consequently, we wrote up the functional programming techniques most relevant to our daily work, with a focus on those that differ from object-oriented methodologies. The first draft had the following sections, aiming for a length of three days (common for ISAQB curricula):

Introduction to functional programming We assumed that programming would play a significant role in the training. Also, we wanted to make the course accessible to software architects who have not had any experience in functional programming.

Immutable data and state In our experience developing software functionally, immutability was the largest pragmatic difference in our daily work compared to object-oriented projects.

Modeling with Combinators Combinator libraries are a folk tenet of functional programing and play a role in many of our projects. They are however largely unknown in object-oriented programming circles—despite the fact that combinators play fairly well with object-oriented architecture.

Higher-Level Abstractions We wanted to cover algebraic concepts such as semigroups, monoids, functors, and monads, to help structure domain models as well as design higher-level abstractions.

Modularization Developers with an object-oriented background use classes and interfaces, and the combination of classes and behavior to organize their code. To address this background, the curriculum draft included a section on functional modularization techniques, noting that functional languages offer a wide variety of mechanisms to support them.

Domain-Specific Languages Many functional software projects include domain-specific languages, so it was natural to include this in the curriculum as an architectural technique.

This draft generated feedback from the ISAQB, specifically from Eberhard Wolff. It became clear that the headings (except maybe for "immutable data") were largely incomprehensible to an audience of object-oriented developers. Eberhard encouraged us to think of architectural techniques in terms of the requirements they help fulfill. Moreover, he suggested separating "programming" from "architecture". The latter point proved difficult, as the FP literature usually expresses architectural techniques in terms of code. We decided to elide the introduction to functional programming, making functional programming skills a prerequisite for the training. (More on the practical consequences of this later.)

It proved beneficial to view functional architecture through the lens of "regular" software architecture, and we ended up following the outline of a popular book on the subject [2]. The resulting structure was as follows:

System structure This section covered the basic tenets of structuring functional software systems: functions and values, composition, types, and modules.

Technologies This section provided a guide to the salient properties that distinguish functional languages from each other, such as typed vs. untyped, strict vs. lazy evaluation, and proper tail calls.

Implementation of functional requirements This section contained the material on combinator libraries and DSLs. Moreover, it referenced Domain-Driven Design (DDD), an important collection of techniques for domain modeling and organizing large projects. (More on this in section 5.3.)

Implementation of non-functional requirements This section covered parallelism, distribution, and persistence. We included event sourcing [9] even though it is not an originally functional concept, as its focus on immutability is a better match for functional architecture than the traditional data warehouse.

Architectural patterns This section collected the material on algebraic concepts. We also added more explicit mention of functional data structures, and the ensuing usage patterns, as well as the Model-View-Update pattern for UI development [5].

Example for functional software architecture This is a mandatory section in iSAQB curricula, and requires studying an example of functional architecture as part of the training.

This curriculum—for three six-hour days of training—was approved by the iSAQB and released in 2018, and has been available for use by training providers since then. The iSAQB eventually assigned Michael Sperber and Lars Hupel as curators.

4 Training Design

In 2019, we designed a concrete training. In order to attract software architects with no prior functional-programming exposure, we prefixed the three-day training with a one-day introduction to functional programming.

We had previously taught professional two-day and three-day trainings on functional programming. In those trainings, we had focused on systematic program construction via design recipes [7], which we had developed into an intro course to programming for universities as part of the DeinProgramm effort [21, 22]. In addition to the pedagogy, this approach uses specifically designed teaching languages that come with the Racket system [4] as well as the DrRacket development environment [8].

When we started teaching software developers, we assumed that the didactic ideas from DeinProgramm would carry over, but that we could use a "professional" language directly instead of the teaching languages. After all, the teaching languages shield students from syntactic warts and poor error messages, both of which are familiar to developers. We discovered however that using the teaching languages and DrRacket was more effective. Essentially, our functional programming training is now a condensed version of the intro course, and switches to a different language on the second or third day. For FUNAR, we further condensed this introduction into a single day, which contains systematic

data modeling using the design recipes, programming with lists, and higher-order abstractions.

To conduct the architecture training proper, we chose to switch to a second language. We chose Haskell, to show the variety and range of design decisions in functional languages—typed, lazy, indentation-sensitive rich syntax as opposed to untyped, strict, Lisp-syntax Racket.

Designing the training, we wanted to address a question that had come up frequently at developer conferences when introducing functional programming: "but how do you organize an entire system this way". We decided to spend a large part of the training putting together a complete application, including a web frontend. This would also cover the "example" section of the curriculum.

We settled on expanding the Hearts card game that Peter Thiemann and I had written up as an example for functional architecture [24]. Hearts requires modest data analysis, has many rules and also a moderately elaborate workflow.

To the original code, we added an event- and command-based interface. We also added monadic code to express the game logic (processing commands into events) and the player logic (processing events into commands). More on that in section 5.2.

Besides the simple console-based synchronous version of the game, we also wrote a concurrent version. This covered items from the "non-functional requirements" section of the curriculum. We also hoped that we would be able to cross-reference Domain-Driven Design with the Hearts application—see section 5.3.

Moreover, we implemented a web-service interface, and a simple frontend written in Elm [5] to cover the Model-View-Update pattern. We chose Elm because it is instantly readable for folks who know Haskell.

Seperately from the Hearts example, we added a section on combinator modeling with a simplified version of the LexiFi DSL for financial contracts [19]. We later added a variant based on Brent Yogey's diagrams library [27]. We also developed a semi-systematic process for developing combinator libraries, using it in exercises. (Ask for simple examples, decompose, remove redundancies, abstract, search for semigroups and monoids.)

Here is the rough schedule of the training:

day		topic
0		intro to FP
1	morning	intro to Haskell
	afternoon	monoids, functors
2	morning	Hearts: domain model
		Hearts: events and commands
	afternoon	intro to monads
		Hearts: game monad
3	morning	Hearts: macroarchitecture
3	afternoon	Hearts: frontend
		combinator modeling

We also designed a floating section for the "technology" part of the curriculum, presenting different industrially viable functional languages, along with code examples and various ways of classifying them.

Moreover, we prepared optional sections on modularization and lazy evaluation using John Hughes's game-search example [11], and a case study based on XMonad [26], which involves extensive use of property-based testing.

We also designed a set of exercises to be completed during the training, including:

- dealing with algebraic concepts, such as constructing monoids and functors for simple data types,
- constructing combinator models,
- data analysis for the Hearts game,
- design of event and command datatypes for Hearts,
- event processing functions for Hearts,
- writing a semantics for financial contracts.

5 Six Years of FUNAR

Starting in the summer of 2019, we started offering FUNAR training, and have taught more than 30 iterations, both open and in-house. The audience has been quite varied, including ambitious software developers, experienced software architects, enterprise architects, and managers—from a wide range of industries, among them software consultancies, manufacturing, transportation, automotive, banking, medical device software, health care, and insurance. Most, but not all attendees reported being actively interested in functional architecture, some reported being mainly after the credit towards an iSAQB Advanced certification. Quite a few attendees had contact with functional programming during their university studies. A small minority have used functional programming in their professional setting.

Attendees are able to follow the training, and succesfully solve the exercises (sometimes in groups), regardless of background. However, attendees have consistently reported that the training is exhausting, as the material is far outside their usual experience in software development. This has required walking a fine line between covering a significant amount of material and ensuring that the attendees can actually follow.

Moreover, a few specific issues with the training have consistently come up. The rest of the section describes those issues and how we have responded and revised the training.

5.1 Where is the Architecture?

For the first half-dozen or so iterations, the most consistent issue was this: At the end of the training, attendees reported that they did not really understand how to develop functional *architecture*. This was after having gone through a complete application, with explanations and exercises on the way, with every opportunity to ask questions or request further elaboration. Conversations with the attendees revealed two reasons for this shortcoming:

- The iSAQB curricula in general make a point of distinguishing between programming and architectural activities. During the training, we had been programming throughout—which was not an architectural activity in attendees' minds.
- We never planned a macroarchitectural structure for the project, but instead proceeded bottom-up from highly reusable domain logic to finally assembling the pieces into macroarchitecture at the very end. (In fact, by the end of the training, we had presented two or three macroarchitectures for the game.) This is in marked contrast to the attendees' experience, which usually starts from a macroarchitectural pattern such as hexagonal architecture [3], often supported by a specific framework such as Spring Boot or Quarkus.[1]

What this observation also revealed was that we never consciously thought about a specifically functional macroarchitectural activity. The macroarchitecture presented in the training follows the functional core/imperative shell pattern [1], but the imperative shell was a pattern that emerged from assembling the purely functional parts of the domain model into a working application. The macroarchitecture can also be explained in terms of the hexagonal-architecture pattern. Doing this still did not resolve the attendees' issue, however, which was more about the architectural activity than the resulting static structure of the system.

Nowadays, we emphasize two points early and often:

- We point out the architectural significance of data modeling throughout, starting with the functional-programming prequel course. (Design recipes are driven by data modeling.) It reappears throughout, prominently when modeling with combinators, using monads (representing processes as data), and performing functional validation (representing both the validators and the outcome as data). This is important as data modeling currently does not play a significant role in traditional software architecture [20].
- We explain our architectural activity as "late architecture" or "architecture avoidance", i.e. developing components decoupled from their surroundings and thus reusable in different macroarchitectural contexts. We stress this throughout the development of the Hearts example and discuss with attendees what architectural activities they would have undertaken at the respective stage of development. This helps resolve the issue to the attendees' satisfaction. It did raise amongst ourselves the question over whether this idea of "late architecture" is actually a frequently used architectural process among functional programmers, and, if so, whether the functional community has explained this idea adequately.

[1] A hexagonal architecture separates the domain logic of a software system from communication with the outside world. Moreover, this communication is strictly through abstract interfaces ("ports") that can have different implementations ("adapters").

5.2 Effects

A major issue in functional programming is how to deal with effects: We designed our training under the assumption that, in functional programming, we would generally try to avoid using effects and, if we could not, restrict and document their use. We had chosen to discuss effects in the context of player strategies: A player might combine a variety of different effects into a strategy. The game must combine the strategies of all players—where each one potentially uses a different set of effects.

Our first iteration implemented this idea using monad transformers [15] that allow assembling an effect stack from individual effects. However, using monad transformers in a modular fashion—where different parts of a program access different parts of the transformer stack—requires abstracting all monadic code over the concrete monad. This in turn is difficult to manage with higher-order code that passes monadic actions in data structures. We also found no satisfactory didactic approach to explain this idea to attendees, who consistently expressed confusion over the approach, and dismay at the high overhead this required.

The alternative approach recommended by some industrial Haskell folks—"just use I/O"—seemed, while pragmatic, unsatisfactorily coarse-grained after 25+ years of research into doing effects in Haskell. (Similarly for the `ReaderT` pattern.) We investigated effects libraries, and settled on Polysemy [17], which allowed us express the player effects and compose them in a modular fashion. This change somewhat alleviated the confusion among attendees. However, the type-level programming Polysemy made practical exercises using it out of reach. Consequently, we refactored again. As part of the training, we had always introduced monads via a free-monad construction for simple database programs. We used the same approach to construct a "Game monad", where the relevant operations come naturally out of a description of gameplay. The reference implementation looks like this:

```
data Game a =
   PlayValid Player Card (Bool -> Game a)
 | RoundOver (Maybe (Trick, Player) -> Game a)
 | PlayerAfter Player (Player -> Game a)
 | GameOver (Maybe Player -> Game a)
 | RecordEvent GameEvent (() -> Game a)
 | Return a
```

The `PlayValid` checks whether a certain move is valid within the game. `RoundOver` checks whether the trick is full and the current round is over, returning the trick and the player who needs to pick it up. `PlayerAfter` determines whose turn it is after a named player. Finally, `GameOver` returns the winner if the game is over. `Return` is the `return` of the

monad. `RecordEvent` uses the event datatype from the previous event exercise to announce that the game has progressed.

Attendees can construct this monad as an exercise, along with the description of gameplay using the monad, and the command-processing code. This solved our didactic problems, and had pleasant consequences: The concept of expressing domain operations as monad operations, rather than going directly to standard monads like state and reader is consistent with Domain-Driven Design and coincides with a talk Andres Löh gave at BOB 2023 about designing effects [16]. This approach also fits well with a hexagonal architecture: Domain workflows can be formulated using the monad, but connecting it to the outside world—analogous to the idea of connecting ports to the outside world via adapters—happens via interpretation functions in the imperative shell. Moreover, the approach is usable even in languages without the type-level capabilities needed to implement effects systems, and is thus of more direct practical value for attendees.

5.3 Domain-Driven Design

Domain-Driven Design (DDD) [6] is an architectural methodology known to many attendees of the FUNAR training. DDD's most important concepts are *bounded context* and *ubiquitous language*: "Bounded context" refers to the idea that a complex software system can be subdivided into parts mostly independent from each other. Each bounded context can establish its own models even for entities shared with other contexts. Within each bounded context, DDD insists on establishing a common vocabulary among users, domain experts, and developers, the "ubiquitous language".

Prima facie, these principles should go well with functional programming, and several books explain domain-driven design in functional terms [10, 25]. Moreover, DDD's modeling practices focus on modeling *processes* whereas functional programming has a rich history of modeling the data manipulated by the processes as well modeling the procceses themselves as data with monads and similar abstractions. Still, Domain-Driven Design has evolved independently from functional architecture. Thus, "functional DDD" is quite different from typical functional architecture.

Some differences are superficial: significant activities in domain-driven design are about identifying up-front (potentially mutable) entities and distinguishing them from (purely descriptive) value objects. This difference plays no significant role in functional architecture as domain objects should all be immutable.

Other differences are more fundamental: Current DDD practice focusses on modeling "close to the domain", generally not straying beyond requirements as formulated by domain experts. This is quite different from functional design, which often abstracts early, in particular in connection with combinator libraries. Attendees often report a disorienting feeling as they work on the exercises involving abstraction. Moreover, the design of combinator libraries—through the process of abstraction and decomposition—often produces building blocks that have no natural names in the ubiquitous language of the domain experts.

Furthermore, many DDD practitioners use Behavior-Driven Development (BDD) [18], actively avoiding data modeling. In functional design, data modeling and behavior design go hand-in-hand, as rich data types enable designing appropriate function signatures. Also, the Test-Driven Development methodology underlying BDD is quite different from the design recipes. This makes it difficult to explain effective functional programming in terms of the techniques familiar to DDD/BDD practitioners, as the underlying activities have very little in common [23].

These issues are an ongoing concern for the FUNAR training, and make it difficult to connect functional architecture to the experience of the attendees. Each has much to offer the other, however, and much works needs be to done on integrating DDD and functional architecture [20].

5.4 Optional Material

We have covered a variety of optional topics in FUNAR trainings, among them AUTOSAR, blockchains, prototyping using functional programming, and DSL design. We also turned the JSON deserialization of the Hearts game into an optional section, covering validation-as-parsing and applicative functors.

5.5 More Languages

At the request of a customer, we developed a Scala-based version of the training. The techniques effective in Haskell translate straightforwardly into Scala, making the Scala-based training very similar to the Haskell version. Other variants are possible—OCaml and F# should be straightforward. A Racket version would be quite different, shifting the focus away from types and possibly treating effects differently.

Occasionally, we demonstrate using the principles taught in the training in more traditional languages such as Java or C#, which have gained substantial functional features, especially in the realm of data modeling. Attendees have reported that the treatment in the training opened their eyes as to how these features were intended to be used.

6 Curriculum Revision

In 2023, the FUNAR curators revised the curriculum. The main goals were to reduce the amount of material to actually fit in three days and to better align the curriculum with a proven didactic flow.

Moreover, the iSAQB had adopted an Advanced curriculum on Domain-Specific Languages [12], so the DSL part of the curriculum could be removed.

We removed the section on "non-functional requirements" as we had realized that functional architecture generally proceeds starting with the functional requirements, whereas the

non-functional requirements play a larger role in traditional software architecture. This makes functional architecture a good match for typical requirements-gathering processes, which also start with the functional requirements [20].

Moreover, to better match the curriculum with the activities familiar to architects and to shape expectations about macroarchitecture, "functional requirements" and "architectural patterns" were re-organized into the new sections "functional modeling" and "functional macro architecture".

7 Conclusion

The software architecture and functional programming communities can benefit from each other. Training software architects in functional architecture is one step in this endeavor: Teaching it for six years has shown that we can introduce the most important tenets of functional programming and functional architecture to OO-trained architects.

Iterating the training has relied on feedback from the participants, as well as the results of the in-class exercises. While the feedback has been consistently positive, the impact of the training on the participants' work is unclear, and merits evaluation. This issue (raised by a reviewer) is not restricted to FUNAR, and we will explore evaluation at the association level.

The work on FUNAR has made it clear that most of the knowledge of functional programming in the large is not comprehensively written up, but is instead scattered among research papers and folklore. Moreover, functional architects would do well understanding Domain-Driven Design, applying its lessons in a functional context, and exploring ways of integrating the two disciplines.

Acknowledgments

I thank fellow FUNAR trainers Bianca Lutz, Johannes Maier, Markus Schlegel and Marco Schneider for the discussions that helped shape and improve our training, as well as the reviewers for their helpful comments.

References

[1] Gary Bernhardt. 2012. Functional Core, Imperative Shell. https://www.destroyallsoftware.com/screencasts/catalog/functional-core-imperative-shell.

[2] Simon Brown. 2012. *Software Architecture for Developers: Volume 1 – Technical leadership and the balance with agility.* Leanpub.

[3] Alistair Cockburn. 2005. Hexagonal Architecture. https://alistair.cockburn.us/hexagonal-architecture/.

[4] Marcus Crestani and Michael Sperber. 2010. Experience Report: Growing Programming Languages for Beginning Students. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. ACM, 229–234. doi:10.1145/1863543.1863576

[5] Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, 411–422. doi:10.1145/2491956.2462161

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

[7] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs* (second ed.). MIT Press.

[8] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: a programming environment for Scheme. *Journal of Functional Programming* 12, 2 (2002), 159––182. doi:10.1017/S0956796801004208

[9] Martin Fowler. 2005. Event Sourcing. https://martinfowler.com/eaaDev/EventSourcing.html. Blog post.

[10] Debasish Ghosh. 2016. *Functional and Reactive Domain Modeling.* Manning.

[11] John Hughes. 1989. Why Functional Programming Matters. *Computer Journal* 32, 2 (1989), 98–107.

[12] iSAQB e.V. 2023. Domain-Specific Languages. https://github.com/isaqb-org/curriculum-dsl. Release 2023.1-rev1.

[13] iSAQB e.V. 2023. Functional Software Architecture. https://github.com/isaqb-org/curriculum-funar. Release 2023.1-rev0.

[14] iSAQB e.V. 2025. iSAQB Foundation Level Curriculum. https://github.com/isaqb-org/curriculum-foundation. Release 2025.1-rev2.

[15] Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. ACM, 333–343. doi:10.1145/199448.199528

[16] Andres Löh. 2023. Structuring effectful programs. https://bobkonf.de/2023/loeh.html. Talk at BOB 2023.

[17] Sandy Maguire. 2021. Porting to Polysemy. https://reasonablypolymorphic.com/blog/porting-to-polysemy/. Blog post.

[18] Dan North. 2006. Introducing BDD. *Better Software* (March 2006). Blog post. https://dannorth.net/introducing-bdd/.

[19] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. 2000. Composing Contracts: An Adventure in Financial Engineering (Functional Pearl). In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, 280–292. doi:10.1145/351240.351267

[20] Michael Sperber. 2025. Things We Never Told Anyone About Functional Programming. In *Proceedings of the 25th International Symposium on Trends in Functional Programming*, Jeremy Gibbons (Ed.). Springer, Heidelberg.

[21] Michael Sperber and Marcus Crestani. 2012. Form over Function: Teaching Beginners How to Construct Programs. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming* (Copenhagen, Denmark) *(Scheme '12)*. ACM, 81–89. doi:10.1145/2661103.2661113

[22] Michael Sperber and Herbert Klaeren. 2023. *Schreibe Dein Programm!* Tübingen University Press.

[23] Michael Sperber and Henning Schwentner. 2023. DDD and FP Can't Be Friends—Yet. https://www.youtube.com/watch?v=kWbALI5Ik0A. Talk at Domain-Driven Design Europe 2023.

[24] Michael Sperber and Peter Thiemann. 2019. Funktionale Softwarearchitektur. *Java Magazin* 9 (Sept. 2019).

[25] Scott Wlaschin. 2018. *Domain Modeling Made Functional.* O'Reilly.

[26] XMonad [n. d.]. XMonad: a Tiling Window Manager Written in Haskell. https://xmonad.org/.

[27] Brent A. Yorgey. 2012. Monoids: Theme and Variations (Functional Pearl). In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark) *(Haskell '12)*. ACM, 105–116. doi:10.1145/2364506.2364520

Received 2025-06-16; accepted 2025-07-21; revised 16 June 2025