

# Decomposition Without Regret (Poster)

Weixin Zhang<sup>1</sup>, Cristina David<sup>1</sup> and Meng Wang<sup>1</sup>

<sup>1</sup>University of Bristol, United Kingdom

## Abstract

Programming languages are embracing both functional and object-oriented paradigms. A key difference between the two paradigms is the way of achieving data abstraction. That is, how to organize data with associated operations. There are essential tradeoffs between functional and object-oriented decomposition regarding extensibility and expressiveness. Unfortunately, programmers are usually forced to select a particular decomposition style in the early stage of programming. Once the wrong design decision has been made, the price for switching to the other decomposition style could be rather high since pervasive manual refactoring is often needed.

In this talk, we show a bidirectional transformation system between functional and object-oriented decomposition. We formalize the core of the system in the **FOOD** calculus, which captures the essence of functional and object-oriented decomposition. We prove that the transformation preserves the type and semantics of the original program. We further implement **FOOD** in Scala as a translation tool called **COOK** and conduct several case studies to demonstrate the applicability and effectiveness of **COOK**.

## Keywords

Bidirectional program transformation, Functional decomposition, Object-oriented decomposition

Programming languages are embracing multiple paradigms, in particular functional and object-oriented paradigms. Modern languages are designed to support multi-paradigms. Well-known examples include OCaml, Swift, Rust, TypeScript, Scala, F#, and Kotlin. Meanwhile, mainstream object-oriented languages such as C++ and Java are gradually extended to support functional paradigms. When multiple paradigms are available within one programming language, a natural question arises: *which paradigm to choose when designing programs?*

A fundamental difference between functional and object-oriented paradigms is the way of achieving *data abstraction* [1, 2]. That is, how to organize data with associated operations. If we view a program as a matrix, data variants and operations are then the rows and columns of that matrix respectively. Object-oriented programming decomposes the program *by row* and is *operation first*: we first declare an interface that describes the operations supported by the data and then implement that interface with some classes. Conversely, functional programming decomposes the program *by column* and is *data first*: we first represent the data using an algebraic datatype and then define operations by pattern matching on that algebraic datatype.

There are important tradeoffs between functional and object-oriented decompositions in terms of extensibility and expressiveness. As acknowledged by the notorious Expression Problem [1, 3, 4], these two decomposition styles are complementary in terms of *extensibility*. Object-oriented decomposition makes it easy to extend data variants through defining new classes. On the other hand, functional decomposition makes it easy to add new operations on


---

STAF 2022 Workshop: Tenth International Workshop on Bidirectional Transformations (BX 2022)

✉ weixin.zhang@bristol.ac.uk (W. Zhang); cristina.david@bristol.ac.uk (C. David); meng.wang@bristol.ac.uk (M. Wang)



© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

expressions. Besides extensibility, object-oriented and functional decomposition have different expressive power. Object-oriented decomposition facilitates code reuse through inheritance and enables *interoperability* between different implementations of the same interface whereas functional decomposition allows inspection on the internal representation of data through (nested) pattern matching, simplifying abstract syntax tree transformations.

Unfortunately, programmers are forced to decide a decomposition style in the early stage of programming. A proper choice, however, requires predictions on the extensibility dimension and kinds of operations to model, which may not be feasible in practice. Once the wrong design decision was made, the price for switching to the other decomposition style could be rather high since pervasive manual refactoring is often needed.

A better way, however, allows programmers to choose a decomposition style for prototyping without regret. When the design choice becomes inappropriate, a tool automatically transforms their code into another style without affecting the semantics. Even at later stages, such a automatic translation tool could be used to make extensions of data variants or operations easier by momentarily switching the decomposition, adding the extension, and then transforming the program back to the original decomposition. Furthermore, studying the transformation between the two styles can provide a theoretical foundation for compiling multi-paradigm languages into single-paradigm ones. From an educational perspective, the tool can help novice programmers to understand both decomposition styles better.

To address this issue, we propose a bidirectional transformation between functional and object-oriented decomposition based on the observation that restricted forms of functional and object-oriented decomposition are *symmetric*. We formalize an automatic, type-directed transformation in the core calculus **FOOD**, which captures the essence of Functional and Object-Oriented Decomposition. We prove that the transformation preserves the type and semantics of the original program. We further implement **FOOD** in Scala as a translation tool called **COOK** and conduct several case studies to demonstrate the applicability of **COOK**. Interested readers may want to consult the full paper for more details [5].

## References

- [1] J. C. Reynolds, User defined types and procedural data structures as complementary approaches to data abstraction, in: D. Gries (Ed.), Programming Methodology, A Collection of Articles by IFIP WG2.3, Springer-Verlag, New York, 1978, pp. 309–317. Reprinted from S. A. Schuman (ed.), *New Advances in Algorithmic Languages 1975*, Inst. de Recherche d’Informatique et d’Automatique, Rocquencourt, 1975, pages 157-168. Also in taoop.
- [2] W. R. Cook, On Understanding Data Abstraction, Revisited, in: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA ’09, 2009, pp. 557–572. doi:10.1145/1639949.1640133.
- [3] W. R. Cook, Object-oriented programming versus abstract data types, in: Foundations of Object-Oriented Languages, Springer, 1991, pp. 151–178. doi:10.1007/BFb0019443.
- [4] P. Wadler, The Expression Problem, 1998. Note to Java Genericity mailing list.
- [5] W. Zhang, C. David, M. Wang, Decomposition without regret, 2022. URL: <https://arxiv.org/abs/2204.10411>. doi:10.48550/ARXIV.2204.10411.