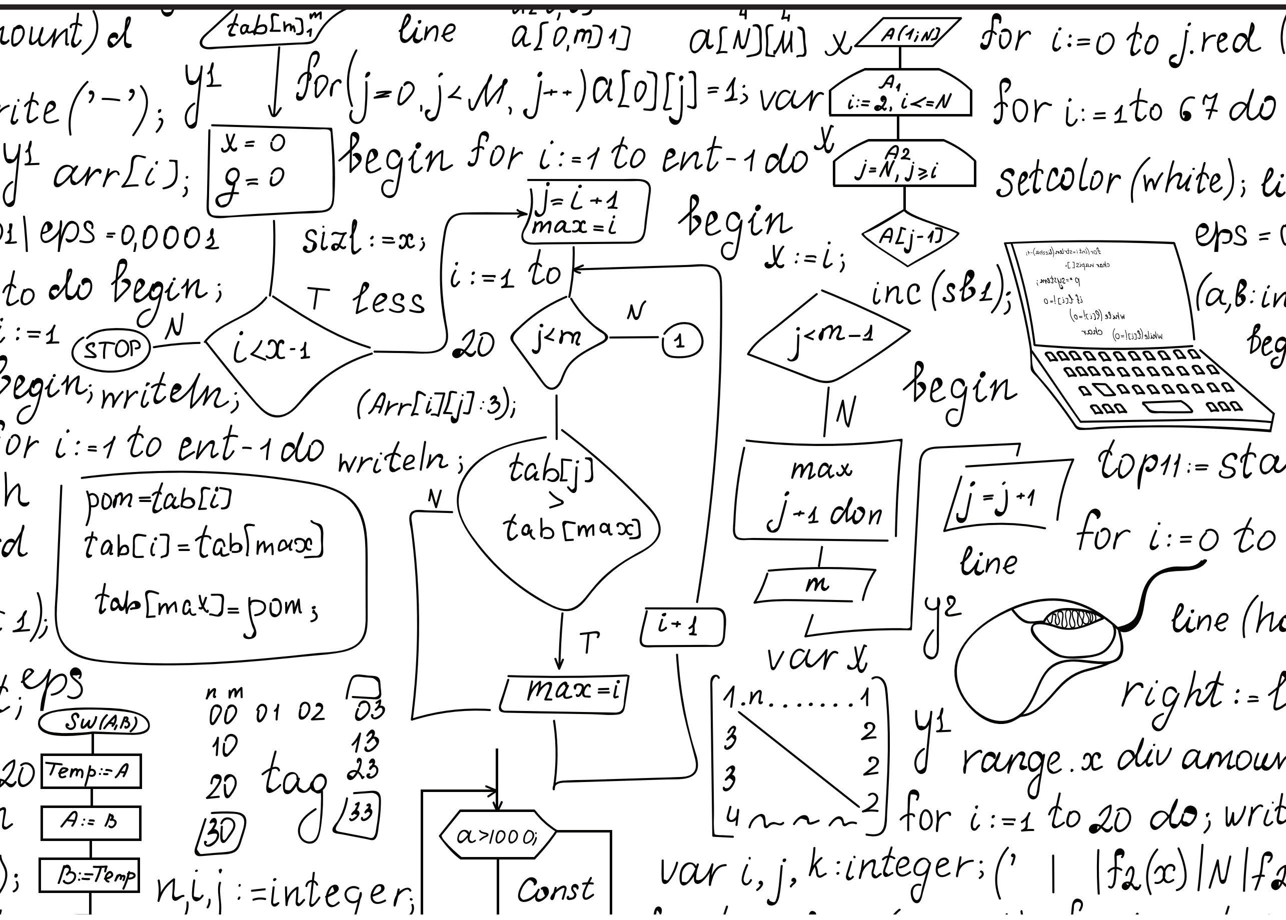
1862

RIGA TECHNICAL UNIVERSITY

Ruslan Batdalov

DEVELOPMENT OF AN OBJECT-ORIENTED TYPE SYSTEM USING DESIGN PATTERN METHODOLOGY

Doctoral Thesis



RTU Press
Riga 2024

RIGA TECHNICAL UNIVERSITY
Faculty of Computer Science, Information Technology and Energy

Ruslan Batdalov
Doctoral Student of the Study Programme “Computer Science and Information Technology”

DEVELOPMENT OF AN OBJECT-ORIENTED TYPE SYSTEM USING DESIGN PATTERN METHODOLOGY

Doctoral thesis

Scientific supervisor
Dr.sc.ing., professor
OKSANA NIKIFOROVA

Riga 2024

ANOTĀCIJA

Projektēšanas šabloni ir nepieciešami jebkurā netriviālā programmatūras projektējumā, bet to izmantošana bieži ir saistīta ar pārmērīgu sarežģītību. Uz šabloniem balstītu risinājumu sarežģītība var būt raksturīga un atbilstoša problēmai, bet citos gadījumos tā kļūst pārāk augsta. Šādos gadījumos projektēšanas šabloni rada sarežģītību, nevis to savalda. Šis darbs izpeta vienu no iespējamajiem iemesliem tam, proti programmēšanas valodu nepietiekamo izteiksmību. Mazākas izteiksmības dēļ programmēšanas valodās ir grūti attēlot domu konstrukcijas, kuras kodē šabloni.

Risinājums piedāvāts šajā darbā ir vispārinot tipiskas programmēšanas valodu konstrukcijas. Vispārinājums tiek panākts, aprakstot pašas šādas konstrukcijas kā šablonus. Turpmāka formalizācija, izmantojot tipu teorijas rīkus, piešķir šiem šabloniem nepieciešamo stingru formu. Autors piedāvā šādu šablonu kopu un to formalizāciju elementārām datu salikšanas un skaitļošanas konstrukcijām.

Šādu izmaiņu ietekme tiek analizēta, izmantojot MIX iedomātā datora emulatora piemēru. Šis paraugprojekts parādīja, ka pat tik izteiksmīga valoda kā Scala noteiktās situācijās nav ērta. Šajā darbā tiek apspriests, kā šī projekta īstenošana un attīstība varētu būt vienkāršāka, ja aprakstītie šabloni būtu atbalstīti.

Promocijas darbs ietver ievadu, sešas nodaļas un noslēgumu. Tajā ir 148 lappuses neieskaitot pielikumus, 23 attēli, trīs tabulas, 128 bibliogrāfijas avoti un divi pielikumi.

ABSTRACT

Design patterns are essential to any non-trivial software design, but their usage is often associated with excessive complexity. The complexity of patterns-based solutions may be inherent and appropriate to the problem, but it becomes too high in other cases. In these cases, design patterns create complexity instead of handling it. The present work studies one of the possible reasons for that, the insufficient expressiveness of programming languages. Due to lower expressiveness, programming languages struggle to represent the mental constructs that patterns encode.

The solution proposed in the present work is generalising typical programming language constructs. The generalisation is achieved by describing such constructs themselves as patterns. Subsequent formalisation using a type-theoretical toolset gives these patterns a necessary strict form. The author proposes a set of such patterns and their formalisation for elemental data composition and computation constructs.

The effect of such changes is analysed on the example of an emulator of the MIX imaginary computer. This example project demonstrated that even such an expressive language as Scala is inconvenient in certain situations. The present work discusses how the implementation and evolution of this project could be more straightforward if the described patterns were supported.

The thesis includes an introduction, six chapters, and a conclusion. It contains 148 pages, not including appendices, 23 figures, three tables, 128 bibliography sources, and two appendices.

TABLE OF CONTENTS

INTRODUCTION

Design patterns are extensively used in software engineering and development. Patterns provide standard solutions to recurring problems, document proven software design and architecture, identify higher-level abstractions, and facilitate communicating design by providing common vocabulary (Buschmann et al., 2013). Frank Buschmann et al. pointed out that any non-trivial design inevitably involves many patterns, consciously or otherwise (Buschmann et al., 2007b).

However, patterns-based design solutions have their cost, which is increased complexity. One should distinguish between inherent complexity, necessary to solve a problem, and excessive complexity, related to implementing a more complicated solution than needed (Batdalov, 2016). Unfortunately, indiscriminate application of design patterns tends to produce excessively complicated solutions, against which Erich Gamma et al. warned in their seminal work that impeded the wide usage of design patterns (Gamma et al., 1995).

The practice of producing unnecessary complex solutions cooled down the initial enthusiasm about design patterns significantly. In the first book of the 'Pattern-oriented Software Architecture' series, Frank Buschmann et al. stated that patterns help developers in managing software complexity (Buschmann et al., 2013), but in the fifth one admitted that many design failures had been caused by the unnecessary and accidental architectural complexity, despite (or even due to) intentional and explicit applying design patterns (Buschmann et al., 2007b). Peter Sommerlad, a co-author of the same series, went even further and bluntly argued that design patterns are bad for software design because of this undue complexity (Sommerlad, 2007).

The author of the present work proposed a hypothesis that the excessive complexity of patterns-based solutions is partially caused by insufficient expressiveness of the existing programming languages (Batdalov, 2016). A programming language's expressiveness (expressive power) measures the breadth of ideas that programmers can express using the language (Leitão and Proença, 2014). In other words, expressiveness determines how easy it is to write a known solution in a particular language. Thus, if a solution is described in terms of patterns, the expressiveness determines the implementation complexity of the patterns comprising the solution. However, there is a gap between patterns, representing mental constructs in which programmers reason about their programs, and programming language facilities, which have historically evolved from underlying technical means. This gap creates the problem of insufficient expressiveness.

Obviously, developers can only hope to close the mentioned gap partially, as humans can always invent new higher and higher-level abstractions. However, design patterns provide a direction in which programming languages could evolve to efficiently represent the same mental constructs as patterns. It would make the implementation of patterns-based design solutions more straightforward.

The implementation difficulty is only one of many sources of complexity. A patterns-based design solution may be unnecessarily complicated, for example, due to redundant flexibility or using inappropriate abstractions. The complexity may also be inherent to the problem. However, addressing at least the implementation difficulty is a reasonable way to tackle the overall complexity problem.

The present work studies how the expressiveness of programming languages can be improved to make it closer to the mental constructs represented by design patterns. An actual programming language based on these principles is outside the scope of the work. However, the work describes a theoretical ground for such a language, a system of types that need language-level support. The type system uses formalisms of the type-theoretical F_ω calculus (Pierce, 2002), providing support for universal and existential types sufficient for the goals of the present work. The described types are to represent the general cases of various concepts in which programmers reason about their programs.

An essential aspect of design complexity is the evolution of an existing system over time. It is usually relatively easy to implement a pattern-based solution from the beginning. However, introducing a pattern into an existing system is complicated, and eliminating a pattern may be much more challenging than introducing it (Sommerlad, 2007). Representing similar concepts with similar constructs in a programming language may facilitate these processes. For the goals of the present work, the described types should cover cases as general as possible so that substitution of one subtype or implementation with another one in the course of program evolution is easy.

In order to achieve the necessary generalisation, the considered types are first described as patterns. Comparison of solutions in different programming languages and identifying all components of a pattern allows finding the general case of various similar solutions. Only then the described patterns are formalised using type-theoretical apparatus.

Topicality of the Subject

New programming languages emerge at a high pace. Some relatively recent languages that apply novel approaches and have already gained high popularity include Scala (Odersky et al., 2023), Kotlin (Jet, 2023), Go (Goo, 2023), TypeScript (Mic, 2023), and Rust (Rus, 2023). Even creators of Java admitted the widespread desire to have 'the next great language' (Gosling et al., 2015). Besides, many research languages are created to test new approaches, concepts, and features before introducing them in mainstream languages. This situation demonstrates that the search for conceptual improvements in the programming languages field is going on permanently.

Older programming languages do not stay fixed, either. Such languages as C++ and Java, as well as others, undergo significant changes between versions, borrowing concepts from other languages and introducing new ones. The strive for higher expressiveness is a significant driver of changes for both new and existing programming languages (Batdalov, 2017).

The topic of design complexity has not been discussed recently in the patterns community as much as it was before. However, the problem is rather accepted than solved. The patterns community mainly concentrates on discovering new patterns, whereas conceptual revisions of the field have not appeared for a long time. The problem of excessive complexity of patterns-based solutions, admitted before, still exists.

The present work supports the programming languages' movement towards higher expressiveness and partially addresses the problem of patterns-based solutions' complexity. It also partially systematises the ongoing processes in programming languages' evolution as the described types often generalise recent novelties. Thus, the present work aligns with the programming languages evolution and design patterns' current needs.

Doctoral Thesis's Goal

The present work aims to develop a system of types generalising existing programming languages' constructs to the abstraction level of mental constructs typical in reasoning about programs, thus increasing their expressiveness compared to existing languages.

Doctoral Thesis's Tasks

In order to achieve the thesis's goal, the following tasks were defined:

1. Analyse programming languages' evolution trends and previously described difficulties in design patterns' implementation.
2. Formulate requirements for the type system based on the mentioned trends and difficulties.
3. Describe common patterns of data composition and basic computation primitives.
4. Formalise type-theoretical constructs representing the described patterns.
5. Develop an example project illustrating the need for more expressive constructs.
6. Validate that the higher expressiveness of the proposed types would simplify the implementation of the example project and its evolution.

Research Object

The object of the doctoral thesis study is abstract concepts representing programming language-level constructs.

Research Subject

The subject of the doctoral thesis study is common patterns generalising basic programming-language level constructs and their type-theoretical formalisation.

Research Methods

The following methods were applied in the doctoral thesis:

1. Comparative analysis of programming languages to identify similar language-level constructs and describe their general forms.
2. Type-theoretical formalisation of the identified constructs using formalisms of the F_ω calculus.
3. Thought experiment on applicability of the developed theoretical constructs in a practical project.

Scientific Novelty

1. Concepts representing typical programming language-level constructs are described as design patterns.
2. Using the language and structure of design patterns allowed generalising the mentioned constructs.
3. The identified patterns are formalised using the type-theoretical apparatus.

Practical Significance of the Study

The validation results demonstrate that the language-level features proposed in the study would simplify practical software development and evolution in certain situations. The methodology developed in the course of the present study can be used for other language and library features in the future.

Research Results Approbation

The results of the doctoral thesis have been reflected in eight publications in international and recognised by Latvian Council of Science journals and proceedings:

1. Ruslan Batdalov. Inheritance and class structure. In Pavel P. Oleynik, editor, *Proceedings of the First International Scientific-Practical Conference Object Systems — 2010*, pages 92–95, 2010. URL `https://cyberleninka.ru/article/n/inheritance-and-class-structure/pdf`
2. Ruslan Batdalov. Is there a need for a programming language adapted for implementation of design patterns? In *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP '16)*, pages 34:1–34:3. Association for Computing Machinery, 2016. ISBN 978-1-4503-4074-8. doi: 10.1145/3011784.3011822
    - Indexing: Scopus.
3. Ruslan Batdalov and Oksana Nikiforova. Towards easier implementation of design patterns. In *Proceedings of the Eleventh International Conference on Software Engineering Advances (ICSEA 2016)*, pages 123–128. IARIA, 2016

- Personal contribution: related work analysis, development of the approach, and description of the proposed features.

4. Ruslan Batdalov, Oksana Nikiforova, and Adrian Giurca. Extensible model for comparison of expressiveness of object-oriented programming languages. *Applied Computer Systems*, 20(1):27–35, 2016. doi: 10.1515/acss-2016-0012
   - Personal contribution: related work analysis, development of the comparison model, and performing the comparison.
   - Indexing: Web of Science.
5. Ruslan Batdalov and Oksana Ņikiforova. Implementation of a MIX emulator: A case study of the Scala programming language facilities. *Applied Computer Systems*, 22 (1):47–53, 2017. doi: 10.1515/acss-2017-0017
   - Personal contribution: related work analysis, emulator design and implementation, and Scala features analysis.
   - Indexing: Web of Science.
6. Ruslan Batdalov and Oksana Nikiforova. Three patterns of data type composition in programming languages. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs (EuroPLoP '18)*, pages 32:1–32:8. Association for Computing Machinery, 2018. ISBN 978-1-4503-6387-7. doi: 10.1145/3282308.3282341
   - Personal contribution: related work analysis and description of the proposed patterns.
   - Indexing: Scopus, Web of Science.
7. Ruslan Batdalov and Oksana Nikiforova. Elementary structural data composition patterns. In *Proceedings of the 24th European Conference on Pattern Languages of Programs (EuroPLoP '19)*, pages 26:1–26:13. Association for Computing Machinery, 2019. ISBN 978-1-4503-6206-1. doi: 10.1145/3361149.3361175
   - Personal contribution: related work analysis and description of the proposed patterns.
   - Indexing: Scopus, Web of Science.
8. Ruslan Batdalov and Oksana Nikiforova. Patterns for assignment and passing objects between contexts in programming languages. In *Proceedings of the 26th European Conference on Pattern Languages of Programs (EuroPLoP '21)*, pages 4:1–4:9. Association for Computing Machinery, 2021. ISBN 978-1-4503-8997-6. doi: 10.1145/3489449.3489975
   - Personal contribution: related work analysis and description of the proposed patterns.
   - Indexing: Scopus, Web of Science.

The main doctoral thesis results were presented at eight international scientific conferences:

1. The First International Scientific-Practical Conference Object Systems — 2010, May 10–12, 2010 — Rostov-on-Don, Russia, "Inheritance and class structure."
2. EuroPLoP '16 — The 21st European Conference on Pattern Languages of Programs, July 6–10, 2016 — Irsee, Germany, "Is there a need for a programming language adapted for implementation of design patterns?"
3. ICSEA 2016 — The Eleventh International Conference on Software Engineering Advances, August 21–25, 2016 — Rome, Italy, "Towards easier implementation of design patterns."
4. Riga Technical University 57th International Scientific Conference, October 13–16, 2016 — Riga, Latvia, "Extensible model for comparison of expressiveness of object-oriented programming languages."
5. Riga Technical University 58th International Scientific Conference, October 12–15, 2017 — Riga, Latvia, "Implementation of a MIX emulator: A case study of the Scala programming language facilities."
6. EuroPLoP '18 — The 23rd European Conference on Pattern Languages of Programs, July 4–8, 2018 — Irsee, Germany, "Three patterns of data type composition in programming languages."
7. EuroPLoP '19 — The 24th European Conference on Pattern Languages of Programs, July 3–7, 2019 — Irsee, Germany, "Elementary structural data composition patterns."
8. EuroPLoP '21 — The 26th European Conference on Pattern Languages of Programs, July 7–11, 2021 — Graz, Austria, "Patterns for assignment and passing objects between contexts in programming languages."

Theses Submitted for Defense

1. Design patterns methodology is applicable to generalising language-level constructs in the form of patterns.
2. The described patterns are formalisable as types (which potentially allows making them programming languages constructs) and can facilitate development of a software system.

Structure of the Thesis

The doctoral thesis is structured as follows. Section 1 describes the patterns and type-theoretical context of the work, as well as presents the research methodology. Section 2 describes the evolution trends of programming languages and identifies related requirements to the developed type system. Section 3 describes patterns of data composition and their type-theoretical formalisation. Section 4 describes computation patterns, except for assignment, and their type-theoretical formalisation. Section 5 describes assignment patterns and their type-theoretical formalisation. Section 6 demonstrates the applicability of the developed type system to a software development project. The conclusion summarises the work and describes future research directions. Appendix 1 contains definitions of the terms used in the work. Appendix 2 contains the list of described patterns with examples of their usage in various programming languages.

1. BACKGROUND AND MOTIVATION

The present work studies the relationship between design patterns and elementary constructs of programming languages. Design patterns describe solutions repeatedly used in different systems written in different languages (Gamma et al., 1995). Therefore, design patterns allow unveiling the general case of a particular problem and a solution for it. The author, in his master's thesis, argued that existing programming languages often support only special cases of a particular pattern, which causes difficulties when a slightly different solution is needed (Batdalov, 2017). During the European Conference on Pattern Languages of Programs (EuroPLoP), a focus group discussion acknowledged that insufficient language expressiveness is a common problem of design patterns' implementation in the existing programming languages (Batdalov, 2016). In this situation, describing the general case as a pattern can help understand what facilities a programming language should have.

However, design patterns themselves lack formality required to describe programming languages' semantics. Type theory studies programming language types, which allow defining the semantics strictly (Pierce, 2002). Therefore, the generalisations described in the language of patterns should then be formalised in the language of type theory.

This section describes the design patterns approach, the general concepts of type theory, related studies, and how these approaches are applied to achieve the present work's goals.

1.1. Design Patterns

Design patterns are widely used in software design. Their primary role is to provide standard solutions for frequently arising design problems. These solutions may be used as building blocks in composing a software system, which reduces the amount of design efforts. Many common patterns are described in the literature in the nearly ready-to-use form and can be applied in various systems. Examples of widely used patterns include Façade, Factory Method, Iterator, Visitor (Gamma et al., 1995), Broker, Publisher-Subscriber (Buschmann et al., 2013), and many others.

However, design patterns are not just software design components. They form a language to communicate between developers. Design solutions may be described in the form of patterns, which makes them, in a sense, a modelling language (even though not a very formal one). The literature on design patterns significantly facilitates the task of communication too.

Historical Reference

Design patterns approach originates from civil architecture. In 1977, Christopher Alexander et al. proposed using patterns as a common language for architects, expressing widely used architectural solutions (Alexander et al., 1977). They explained the essence of a pattern in the following way: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" (Alexander et al., 1977). Although Alexander's book is devoted to patterns in civil architecture, the given explanation is not bound to this domain. The patterns approach can be used in other areas, including software design.

Software design patterns were popularised by the seminal book by Erich Gamma et al. (Gamma et al., 1995). The authors of this book, collectively known in the patterns community as the 'Gang-of-Four', explained how Christopher Alexander's approach can be applied to software architecture and described 23 design patterns, among which are such widely used patterns as Abstract Factory, Builder, Façade, Iterator, etc. (Gamma et al., 1995). This book became a classical and often referenced work on design patterns.

Other prominent works concerning design patterns include the book series 'Pattern-oriented software architecture' (Buschmann et al., 2007a,b, 2013; Kircher and Jain, 2013; Schmidt et al., 2013), 'Software patterns' by James O. Coplien (Coplien, 1996), 'Patterns of enterprise application architecture' by Martin Fowler (Fowler, 2012), and others. Primarily, these sources describe various design patterns used in software design. Proceedings of yearly conferences on the pattern languages of programs (PLoP, EuroPLoP, AsianPLoP, etc.) contain descriptions of many other design patterns and pattern languages (sets of interrelated patterns that are used together in the same domain). In 2007, Grady Booch wrote that he had catalogued nearly 2000 patterns described in the literature (Booch, 2007). By now, this number is obviously much higher.

The degree of systematisation of the huge amount of known patterns is rather low. Grady Booch's catalogue has not been published by now, and the pace at which new pattern descriptions appear significantly hinders the task of composing a systematic catalogue. Most books and scientific papers on design patterns simply describe a number of discovered patterns. However, there are some common points shared by most members of the patterns community, which can be a basis for patterns systematisation. These points are covered in the next section.

Pattern Concept, Structure and Classification

Researchers use different definitions of a design pattern, however, these definitions generally bear similar ideas. Erich Gamma et al. stated that patterns are 'descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context' (Gamma et al., 1995). Frank Buschmann et al. proposed another but similar definition: 'A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate' (Buschmann et al., 2013). Both definitions emphasise such main attributes of a design pattern as a context, a problem and a solution for the problem.

The pattern format used by Erich Gamma et al. is aligned with the given definition. The description format includes the following components (Gamma et al., 1995):

- pattern name, which reflects its essence and is used to reference the pattern, and classification,
- intent, which describes the purpose of the pattern,
- 'also know as' or other used names of the pattern,
- motivation, an example of the problem and its solution by means of applying the pattern,
- applicability, which describes the situations where the pattern can be applied,
- structure, a graphical representation of the classes participating in solution and their interaction,
- participants, i.e. classes and objects that compose the solution, as well as description of their responsibilities,
- collaborations of the participants in order to solve the problem,
- consequences, which are benefits and trade-offs related to applying the pattern,
- implementation, which does not prescribe a particular way of implementing the pattern but discusses common techniques and pitfalls to consider,
- sample code, which illustrates the pattern implementation,
- known uses, i.e., examples of the pattern in real systems,
- related patterns, mentioning the patterns that serve similar purposes and describing their similarities and differences.

In this format, the intent and the applicability sections describe the context and the problem, whereas the structure, the participants, the collaborations, the consequences, the implementation and the sample code sections give the solution. Variations of this format are commonly used to describe design patterns in other sources as well. For example, Frank Buschmann et al. used a format, in which the context and the problem are described explicitly in separate sections (Buschmann et al., 2013).

There are different approaches to pattern classification. Erich Gamma et al. classified the described design patterns along two dimensions: purpose and scope. With respect to the purpose, they distinguished creational (object creation), structural (composition of classes or objects) and behavioural (interaction of classes and objects) patterns. With respect to the scope, they classified the patterns into class patterns and object patterns depending on the relationships that are in the focus, inheritance relationships and object relationships correspondingly (Gamma et al., 1995). For example, both the Factory Method and the Abstract Factory patterns are creational since they deal with object creation (Gamma et al., 1995). However, the former is classified as creational class pattern as it defers the creation from an abstract class to its subclass, whereas the latter is a creational object pattern as the creation is deferred to another object (Gamma et al., 1995).

Frank Buschmann et al. used another classification, related to the place of a pattern in a system design. They distinguished architectural patterns, which represent fundamental structure schemata for software systems, design patterns, which act at the level of subsystems or components and their relationships, and idioms, which cover low-level implementation details and are typically specific o a particular programming language (Buschmann et al., 2013). Architectural patterns include such examples as Broker and Layers, design patterns: Proxy and Command Processor (Buschmann et al., 2013). An example idiom that Frank Buschmann et al. mentioned is Counted Pointer in C++ (Buschmann et al., 2013).

Another classification criterion is pattern applicability domain. Although many patterns are universal and applicable to various software systems, some are rather related to particular domains. This criterion is often used to choose a set of patterns for a book, for example, 'Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects' by Schmidt et al. (2013) and 'Patterns for Parallel Software Design' by Ortega-Arjona (2010).

Patterns Usage

The primary goal of pattern application, which Erich Gamma et al. placed in the subtitle of their book, is reusability (Gamma et al., 1995). It has at least two aspects: not only patterns themselves are reusable pieces of design, but also a design built with patterns is supposed to be reusable and easily adaptable to different conditions and situations (Gamma et al., 1995).

Frank Buschmann et al. identified the following roles of patterns in software architecture (Buschmann et al., 2007b):

- Patterns document existing best practices built on tried and tested design experience.
- Patterns identify and specify abstractions that are above the level of single objects, classes and components.
- Patterns provide a common vocabulary and shared understanding for design concepts.
- Patterns are a means of documenting software architectures.
- Patterns support the construction of software with well-defined properties.
- Patterns capture experience in a form that can be independent of specific project details and constraints, implementation paradigm, and often even programming language.

Two important aspects of pattern usage should be discussed with respect to the mentioned pattern roles. First, using pattern in these roles may be deliberate, but does not always have to. Grady Booch stated that every development culture tends to converge on a set of architectural patterns over time (Booch, 2007). According to Frank Buschmann et al., any non-trivial design uses a lot of patterns, *whether consciously or otherwise* (Buschmann et al., 2007b). That means that patterns represent constructs of thinking which are not always realised when used (as Mr. Jourdain in Molière's play did not realise that he spoke prose). Similarly, researchers of patterns usually speak about pattern *discovery*, not inventing (Buschmann et al., 2007b, 2013). Inventing a new pattern would lead to describing an unverified solution, whereas necessity of proven solutions is always emphasised in patterns literature, starting from Erich Gamma et al. (Gamma et al., 1995) and Frank Buschmann et al. (Buschmann et al., 2013). This consideration also means that patterns exist in real software systems even before they are first described. Pattern description is useful in avoiding common problems and implementation mistakes, but it is not strictly necessary for applying the patterns.

Second, although a pattern represents a ready-to-use solution, programmers may not be able to modularise this solution as a software component. Patterns componentisation is a kind of the Holy Grail in patterns literature. Frank Buschmann et al. showed impossibility to componentise the Observer pattern and explained it by stating that the solution given by a pattern is not passively generic, but generative (Buschmann et al., 2007b). Frederik Løkke showed out that, even in such an expressive language as Scala, not all design patterns are componentisable (Løkke, 2009). A pattern gives instructions how to solve a particular problem in a particular context, but does not necessarily provide a 'one-size-fits-all' software module that solves it.

There are also known problems related to patterns application. Frank Buschmann et al. mentioned the following common traps and pitfalls (Buschmann et al., 2007b):

- developers' temptation to turn all software development activities and artifacts into patterns;
- describing design solutions that are not reusable or proven as patterns;
- developers' unwillingness to deviate from the pattern description given in the literature even when it is needed (the pattern is considered as a fixed and unchangeable solution);
- describing coding guidelines that do not contain a solution to a problem as patterns;
- application of the wrong pattern due to a limited or misunderstood pattern vocabulary, which does not allow to make the proper choice;
- belief that mechanical application of patterns ensures good architecture in all cases, even in complex ones;
- lack of creativity when described patterns are applied to a system that requires novel approaches;

- too high expectations from design patterns;
- impossibility of complete automation of pattern usage;
- non-componentisability of many patterns;
- using patterns instead of refactoring or vice versa, whereas both should be used in their time.

These problems made Frank Buschmann et al. admit in the fifth volume of the 'Pattern-oriented Software Architecture' series that many systems in which patterns were used intentionally and explicitly ended up with unnecessarily complex architecture (Buschmann et al., 2007b), whereas they stated in the first volume that one of the primary goals of design patterns is managing software complexity (Buschmann et al., 2013).

Certainly, the problems of unnecessary complexity and patterns overuse, as well as other mentioned problems, do exist. However, the benefits of patterns are significant too and often outweigh the disadvantages provided that the potential problems are considered and treated correctly.

1.2. Type Theory

Type theory as a field of computer science studies type systems existing in programming languages. Type theory provides a formal ground for reasoning about properties of languages and programs that are defined by the used type system. As a result, it serves as a basis for the calculi used to prove formal statements about programming languages. There exist various calculi suitable for analysis of different languages depending on the programming paradigm and other properties.

In the practical sense, type-theoretical results are generally concerned with program safety. The essence of the concept of a type is related to prevention of using data in an inappropriate way (either at the compile time or at the run time). Therefore, the type system of a language plays an important role in protection of the abstractions defined by the language.

However, type theory is not only applicable to analysis of the existing language safety. Being a formal mathematical theory and providing a well-developed framework of concepts and relationships, it allows finding new approaches in language design by means of generalising existing ones and other ways of theoretical reasoning. Therefore, type-theoretical analysis of a language is useful not only in ensuring program safety, but also in development of the language expressive power.

Types and Type Systems

The central concept of type theory is a data type or simply a type. Even though developers and researchers constantly deal with types, such as integers, strings, arrays, maps, classes defined for a specific purpose (e.g., a regular expression parser), and others, the general notion of a type is not easy to define. Yet in 1976, David Parnas et al.

identified five different approaches how types were defined (sometimes implicitly) in the literature (Parnas et al., 1976):

Syntactic approach A type is the information about a variable given in its declaration. This approach considers types to be arbitrary labels without presuming any semantics. As a result, it hardly allows informative reasoning about programs.

Value space approach A type is a set of possible values.

Behavioural approach A type is defined by a value space and a set of operations on the elements of the value space.

Representation approach A type is defined by the way in which it is represented in terms of more primitive types. This approach requires some primitive types, which cannot be reduced further, to be defined by hardware or the compiler.

Representation plus behaviour A type is defined by its representation and the set of operations.

Obviously, all these approaches capture important characteristics of types, which are actually applied in programming languages. However, none of them is full enough to be used as the primary definition. For example, the syntactic approach captures how types are assigned to variable, but gives little information about what this assignment means. Moreover, dynamically typed languages do not require a type to be assigned to a variable at all. The representation and representation plus behaviour approaches capture how compound data types (such as collections, classes, etc.) are built in programming languages. However, abstract data types (e.g., a queue) are only defined by their behaviour and not associated with any particular implementation (Pierce, 2002). Thus, abstract data types cannot be explained within the representational approaches. The value space and behaviour approaches, on the contrary, describe a type from the point of view of its usage, but ignore how the data type is implemented.

In order to have a definition that is not bound to specific aspects of types only, David Parnas et al. proposed to treat types as classes of variables. Specifically, a type is a group of equivalence classes (called modes) on variables in the sense that any value that can be stored in a variable of a given mode can be stored in another variable of the same mode, and substitution of a variable for another variable of the same mode will not cause a compile-time error (Parnas et al., 1976). By now, this definition is outdated too because it does not take into account the run-time type system. However, the idea to define types as classes of interchangeable variables and expressions (so that the exact semantics of these classes is defined according to the scope of the study) is applied by modern type theory as well. Thus, Benjamin Pierce defines a type system as ‘a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute’ (Pierce, 2002).

There are a number of important points in the definition of a type system (Pierce, 2002):

- Application to programs. Type theory of programming languages is a part of the general type theory, a logical, mathematical and philosophical discipline developed to overcome the foundational crisis of mathematics. Type theory for programs applies the approaches originating from the general type theory to a specific domain of programs and programming languages.

- A type system is based on classification of language phrases. This classification is an approximation of the run-time behaviour that can be done at the compile time (at least, for statically typed languages). For example, a compiler may not know which particular value will be stored in a variable, but if the variable has an integer type, it is known that the string representation of a value can be found using the inverse of Horner's rule. The classification is typically defined compositionally, i.e., the type of an expression only depends on the types of its subexpressions.

- Conservativeness. A type system can prove the absence of some bad behaviour, but cannot prove its presence. For example, in most cases, expression `if <test expression> then 5 else <error>` will be rejected even if `<test expression>` always evaluates to true.

- Limitedness to certain types of errors. For example, typical type systems can ensure that both sides of an arithmetic expression are numbers, but not that a divisor is not zero. The choice of particular behaviours against which a type system protects is a language choice. Statically typed languages tend to protect against a wider range of errors at the expense of restricting programmers' freedom, but dynamically typed ones prefer allow more and discover errors at the run time. In general, there is a tension between conservativeness and expressiveness in programming languages.

- Tractability. Type checking is defined formally, and thus it can be performed automatically during compilation.

Traditionally, type theory primarily works with the types known at the compile time (statically assigned). The compile-time types are associated with identifiers (variables, constants, fields, etc.) and restrict a set of values that can be assigned to an identifier. In many cases, the compile-time type is sufficient to resolve all operations on an identifier (choose a particular implementation that applies a specific interpretation of the binary data). Then there is no need to store the type information at the run time (erasure (Pierce, 2002)). However, some advanced features, such as subtype polymorphism, require preserving type information at the run time. For example, a variable of the compile-time type `AbstractMap<String, Integer>` may hold a value of the run-time type `HashMap<String, Integer>` and call appropriate methods to access an entry in the map.

Unlike compile-time types, run-time types are associated with particular pieces of data instead of identifiers. Moreover, some languages (dynamically typed ones) do not support compile-time types at all. In these languages, any value can be assigned to any identifier, and all operations are polymorphic and resolved on the basis of the run-time type. Some checks studied in type theory (such as compile-time error detection) require compile-time types, but others (such as language safety, see below) are supported in dynamically typed languages as well.

Type systems provide the following benefits (Pierce, 2002):

Detecting errors early Type errors are detected automatically and (in the case of static typing) before the program runs and the errors can badly influence its behaviour.

Help in maintenance and refactoring (in the case of static typing). If the type of a variable or a definition of a type is changed, all usages usually become invalid. As a result, it is often sufficient to fix arising compile errors instead of manual looking for all places where a type or a variable is used.

Abstraction Types usually correspond to abstractions in the data model of a program. By providing means to create types that capture essential features of data, type systems assist in reasoning about programs at the abstract level.

Documentation (in statically typed programming languages) Type declarations clarify the goal of a variable and thus facilitate reading programs. Unlike comments, type declarations cannot become outdated.

Language safety All languages provide some abstractions of machine-level implementation, and a safe language is one that protects its own abstractions. For example, if a language supports arrays, one may expect operations on an array only mutate the array itself and not other data past its end. Actually, static typing is not sufficient for language safety. For example, statically typed C and C++ allow many unsafe operations (such as writing beyond the boundaries of an array, unchecked type casts, etc.). As mentioned above, type systems only protect against certain types of errors since they operate with broad classes of values instead of values themselves. Language safety usually relies on run-time type checking, which has all necessary information. In the example of writing past the end of an array, the actual size of an array is known at the run time, but not at the compile time. As a result, the run-time environment can protect against crossing array boundaries, but the compiler cannot. Static typing can assist in language safety (e.g., in Java), but not ensure it. Interestingly, virtually all dynamically typed languages are safe in the sense of the definition above. They have to check correct typing at the run time anyway, so it is just natural to check correct usage of all their abstractions.

Efficiency When type of a variable is known at the compile time, the choice of implementation of an operation can be performed at the compile time as well without time-consuming run-time type checks.

Security Type checking can be used to check that external data (e.g., parameters of a remote call) are consistent with a program's expectations.

Base Types

A language usually has a number of types that are known to its compiler in advance. These types are known as base, primitive or atomic types. Examples include Booleans, numbers, characters, etc. The particular set of base types depend on a language. For example, in Java, a character is a base type and a string is a compound type backed by an array of characters (Gosling et al., 2023), whereas in JavaScript, strings are considered a base type and a character is simply a string of length 1 (ECMA-262).

Base types play an important role in the representational interpretation of a type, mentioned above. The representation approach requires all types except for base types to be reducible to other types, whereas the base types themselves are considered as is.

In principle, type theoretical inference can be performed without base types at all (e.g., in the simply typed lambda-calculus (Pierce, 2002)). However, the features of real programming languages are better reflected with base types since the languages have such types that they do not decompose any more. Type theory does not give base types any interpretation, they are simply added to the list of types and used as building blocks in creation of compound types (Pierce, 2002). Thus, a type-theoretical calculus does not depend on a particular set of base types supported in a language.

A specific kind of base types consists of singleton types, i.e. types that have a single value (e.g., the unit or void type). However, not all singleton types are primitive, therefore, they are considered below in a separate subsection.

Compound Types

The most interesting part of type theory begins when a programmer can define his or her own types. That requires type theory to be able to derive the features of a type from the way how it is constructed without knowing the type in advance.

Compound types are structures built from other types (either base types or other compound types). For example, an pair of two integers is a compound type. Type theory allows describing various compound types with operations that they support (e.g., taking the first component of a pair), but the particular set of compound types depends on the language (e.g., a record is a common compound data type in statically typed languages, but is rarely supported in dynamically typed languages). However, there are a number of common composition mechanisms: references/pointers (often supported only implicitly for safety reasons), number-indexed compounds (arrays, tuples, lists, etc.), name-indexed compounds with a fixed set of keys (records), name-indexed compounds with a variable set of keys (associative arrays), sets, variant types (general variants, the optional type, enumerations) (Batdalov, 2017). Composition may be recursive (e.g., a list is either empty or a node followed by a list). These composition mechanisms allow creation of complex data types from simpler ones.

An important theoretical issue related to compound data types is the difference between nominal and structural type systems. Type theory primarily considers structural type systems, in which a type is determined by its structure, the type name is only an abbreviation, and structurally equivalent types are the same type independently on their names (AbdelGawad, 2017). However, most programming languages typically apply nominal type systems, in which the type name is a part of its definition, and two structurally equivalent types with different names are different (AbdelGawad, 2017). The advantages of nominal type systems include availability of run-time type information, natural support of recursive types, easy check of subtyping, and language-level differentiation between structurally compatible, but semantically different types (Pierce, 2002). However, some advanced features (e.g., generics) are hard to implement in nominal type systems, and programming languages with these features rather use hybrids of nominal and structural ones (Pierce, 2002).

There are different approaches to solving this problem. Moez AbdelGawad argued for describing the 'missing' features in terms of nominal type systems in order to bring type theory closer to what really exists in programming languages (AbdelGawad, 2017). However, this work has not been completed yet. Another approach is taken in TypeScript, whose compile-time type system is mostly structural. However, the type system of TypeScript still has nominal features in order to support such object-oriented constructs as private fields (Mic, 2023). Moreover, the run-time type information is also nominal since the run-time environment of TypeScript is shared with JavaScript. Therefore, the ability to describe a nominal type system in the language of type theory would be useful for TypeScript too.

Subtype Relationship

The subtype relationship is a reflexive transitive relationship between types that semantically means that any value of a subtype is also a value of its supertype (Pierce, 2002). Liskov substitution principle gives a formal definition of subtyping: type A is a subtype of type B (`A <: B`) if any correct program that uses a value of type B will remain correct if a value of type A is substituted for the value of type B (Martin and Martin, 2006). Subtyping for name-indexed compounds (records and associative arrays) can be performed in two different ways (Pierce, 2002):

- breadth subtyping: adding new fields to the compound type,
- depth subtyping: subtyping a field of the compound type.

Generics (Parameterised Types)

Generics are generators (templates) of types that produce a type when given some parameters, which may be other types or rarely values. For example, a list may be a generic, whose parameter is the type of values in the list (e.g., a list of integers). In type theory, this mechanism is called parametric polymorphism (Pierce, 2002).

The most difficult problem of generics from both the theoretical and the practical point of view is variance, i.e. subtyping relationships between different parameterisations of the same generic. Generic `T` is called covariant if it follows the subtyping relationship for its parameter (i.e., if `A <: B` then `T A <: T B`), contravariant if it reverses this relationship (i.e., if `A <: B` then `T B <: T A`), and invariant if neither is true (i.e., only if `A <: B` and

`B <: A` then `T A <: T B` and `T B <: T A`) (Pierce, 2002). Variance of the same generic with respect to different parameters may differ.

There are two main approaches to defining variance of generics. One is based on the observation that the functional type is contravariant with respect to its argument types and covariant with respect to its return type. Therefore, any type parameter may be marked as covariant or contravariant, covariant type parameters may only be used as return types of the methods of the class, and contravariant type parameters may only be used as argument types. The parameters that are not marked (invariant) can be used both in arguments and return types. This approach is called declaration-site variance and is implemented, for example, in Scala (Odersky et al., 2023), C# (ECMA-334) and Kotlin (Jet, 2023). Another approach allows declaration of methods using generic types, whose parameters are not known in advance. Type system only proves that such a type exists (existential types (Pierce, 2002)). Actual parameters of these generics are calculated when a method is used. This approach is called usage-site variance and implemented in Java (Gosling et al., 2023). In Scala, existential types were supported since version 2.6 (Odersky, 2007) but dropped in version 3 (LAM, 2023).

It should be noted that violation of variance rules may lead to serious consequences. For example, standard arrays in Java were designed as covariant (Gosling et al., 2023). It means that if `A <: B`, then `A[] <: B[]`. In turn, that means that it is possible to have a variable of type `B[]` that holds a value of type `A[]`. Then it is possible to put a value of type `B` in the array of type `A[]`, which should not be allowed. Java controls this at the run time, which is highly inefficient (Pierce, 2002). Covariant arrays violate Liskov substitution principle, so arrays have to be invariant. In Scala (Odersky et al., 2023) and Kotlin (Jet, 2023) arrays are invariant indeed, even though they are implemented using covariant Java arrays.

Singleton Types

Singleton types are the types that have a single value. Examples of such types include:

- The unit type (called void in such languages as C, C++ and Java). It is a special type that indicates absence of a meaningful value. It is primarily used with functions that have side effects (Pierce, 2002).
- Literal types in Scala (Odersky et al., 2023) and TypeScript (Mic, 2023). Variables of these types can only hold a single value defined by a literal. In particular, these types are useful for defining enumerations (which are unions of literal types). Literal types can also be used as parameters of generics if a language only supports types as parameters and not values.
- Objects in Scala (Odersky et al., 2023). These are compound objects (not primitive values as the first two cases), which are used instead of static fields and methods. They implement the Singleton design pattern (Gamma et al., 1995).

In some cases, singleton types may not require any space to hold a value: the type definition and the type information (compile-time or run-time) are sufficient to restore it. However, it depends on the implementation, for example, literal types in TypeScript are simply variables of the corresponding type (e.g., a string or a number) that hold the corresponding value.

Specifics of Object-Oriented Languages

The ways to create types studied in type theory are mostly applied in all programming languages independently on programming paradigm. However, there are a number of considerations that are specific to object-oriented languages and to how the general principles apply to them. In particular:

- An object-oriented programming language has to have a way to represent such types as classes and interfaces. The data in classes are represented by a compound type, but the particular used compound type depends on the language. Statically typed languages tend to use records as the basis for classes, but dynamically typed ones prefer associative arrays. Operations on classes (methods) are usually polymorphic (virtual) and resolved on the basis of the run-time type information. Sometimes, methods are not polymorphic (e.g., non-virtual calls in C++) and are resolved on the basis of the compile-time type. Interfaces may either be pure declarations or contain some behaviour, but not data (in the latter case the operations are resolved in the same way as for classes).

- Subtyping is usually associated with inheritance and interface implementations. Even though some researchers argued that inheritance is not subtyping (Cook et al., 1989), programming languages usually allow using a value of a subclass instead of a value of a superclass, i.e., treat subclasses as subtypes.

- In terms of type theory, inheritance represents breadth subtyping. Depth subtyping is not typically supported in object-oriented languages. Therefore, inheritance does not face the variance problems that exist for generics. However, depth subtyping could be useful in implementation of design patterns, therefore, borrowing the approaches designed for generics may make sense (Batdalov, 2010; Batdalov and Nikiforova, 2016).

- The concept of encapsulation is compatible with a nominal type system only because structurally equivalent classes cannot be considered the same class.

Type-Theoretical Calculi

After supported types have been defined, semantics should be assigned to them. There are three main approaches to define formal semantics in type theory (Pierce, 2002):

- Operational semantics specifies the behaviour of a programming language by defining an abstract machine so that a program defines transition between the states of the machine. The meaning of a term of the language is the final state of the machine if the term itself is the initial state.

- Denotational semantics establishes a correspondence (interpretation) between the terms of a language and mathematical objects.
- Axiomatic semantics defines a programming language as a set of logical laws. The meaning of a term is what can be proved using these laws.

Defined semantics establishes a correspondence between type systems and logic, called Curry-Howard correspondence. Types correspond to logical propositions, language terms correspond to proposition proofs, and computation corresponds to proof simplification (Pierce, 2002). Thus, type checking is formal reasoning about programs. Type theory has developed formal criteria for various desired properties of programs that can be proved (e.g., type system safety requires progress, i.e., any well-typed term either is a value can take a step according to the evaluation rules, and preservation, i.e., if a well-typed term takes a step of evaluation, the resulting term is well-typed too (Pierce, 2002)).

Types and their semantics define a calculus, which makes possible reasoning about programs and type systems. A particular calculus is dependent on the used types. For example, Martin Abadi and Luka Cardelli proposed a general calculus for object-oriented programming languages (Abadi and Cardelli, 2012). Its extension Featherweight Java models Java type system in particular (Igarashi et al., 2001). So, a calculus should be adjusted to a particular language or type system.

When a calculus is defined, it allows proving formally such properties of a type system as safety, possibility of type erasure, etc. However, its role is not restricted by such proofs. For example, Scala's feature of defining an explicit self type of a class (the type of this pointer, which may differ from the class itself in Scala) originated as a technical measure for the needs of the calculus used to reason about Scala type system (Odersky et al., 2006). However, this feature happened to be useful in real programming too. Similarly, literal types in Scala were proposed because singleton types had existed in the language semantics before, but were not supported by the syntax (Leontiev et al., 2019). Again, the formal description of the language gave an impetus to its development.

1.3. Related Work

A few aspects should be considered to achieve the goals of the present work. First, it is the difficulties in design patterns implementation, the primary motivation of the present work. The second issue is how to build a type system to avoid these difficulties. The present section analyses how these problems are treated in the literature. For simplicity, the type system part is split into structural and behavioural aspects as these questions can be considered separately.

Pattern Implementation Problem

It is generally accepted in the patterns community that one cannot represent patterns in code but only provide a particular implementation (Alexandrescu, 2001). Numerous books are devoted to implementing common design patterns in different languages (for example, in C# (Bishop, 2008) or Scala (Løkke, 2009)).

However, the volume of these books and the necessity to write a separate book for each language suggest that implementing design patterns is complicated. Indeed, the complexity of design patterns implementation has been recognised for a long time (Buschmann et al., 2007b). Moreover, even if one knows how to implement a pattern, introducing patterns into existing code is problematic because it often requires a lot of coordinated changes in different parts of the system. It is challenging to introduce a pattern and even more challenging to eliminate it (Sommerlad, 2007).

Several approaches to solving patterns implementation complexity have been proposed. For example, Frank Buschmann et al. tried to create configurable generic implementations for patterns covering their whole design space (Buschmann et al., 2007b). However, they quickly showed that it is impossible even in relatively simple cases (Buschmann et al., 2007b).

Some implementation approaches employ advanced language features. For example, Andrei Alexandrescu proposed generic implementations based on extensive use of C++ templates and template metaprogramming (Alexandrescu, 2001). However, usual drawbacks of template-based implementations are implementation complexity and being challenging to follow.

A similar approach is related to using aspect-oriented languages, which allow extraction of cross-cutting responsibilities from classes to separate modules (aspects) (Kiczales et al., 1997). Jan Hannemann and Gregor Kiczales described implementations of common design patterns in Java and AspectJ (Hannemann and Kiczales, 2002). Miguel P. Monteiro and João Gomes used another aspect-oriented language, Object Teams (Monteiro and Gomes, 2013). Pavol Bača and Valentino Vranić developed the idea and proposed replacing the commonly known object-oriented design patterns with aspect-oriented ones (Bača and Vranić, 2011). It seems that unrelated responsibilities separation is crucial in implementing patterns as it allows covering the most general case of patterns applicability. The role of this principle in the present work is discussed in Section 2.3. However, full-blown aspect-oriented languages are not necessary for pattern implementation. The present work considers approaches applicable in languages without aspect-oriented capabilities.

Another important direction of research is pattern decomposition. The enormous number of discovered patterns suggests that they may consist of basic building blocks (like the significant amount of discovered hadrons, which used to be considered elementary particles, led to the emergence of the concept of quarks). This idea is essential in the present work context because if patterns are challenging to implement, their building blocks may have more straightforward implementations.

Thus, Uwe Zdun and Paris Avgeriou tried to identify architectural primitives of patterns (Zdun and Avgeriou, 2008). These primitives are architectural by nature, so they are applied more to system design than implementation. Francesca Arcelli Fontana et al. described common design pattern micro-structures in order to facilitate pattern detection in existing systems (Fontana et al., 2011, 2013). The observation by Frank Buschmann et al. that design patterns are rather generative than generic (Buschmann et al., 2007b) refers to many of these micro-structures as their descriptions cover a wide range of possible solutions. Jan Bosch described design patterns as a composition of layers and delegations and proposed an implementation using the so-called Layered Object Model (Bosch, 1998). This approach catches and formalises an essential characteristic of patterns: they are often based on delegation. This idea is used in Section 4.3.

The functionality objectives discussed in Section 2 are formulated according to the same approach by explicitly requiring that pattern building blocks are identified at the abstraction level of programming language features (Batdalov and Nikiforova, 2016). However, this approach does not seem sufficient. Despite the mentioned attempts, a limited set of universal building blocks is unknown and does not seem feasible.

The implementation complexity problem can also be solved from the other end: by raising the programming languages' level of abstraction and including patterns into languages directly. Joseph Gil and David H. Lorenz described the gradual percolation of design patterns into programming languages (Gil and Lorenz, 1998). This process is also discussed in Section 2.5. However, this discussion is primarily observational because the distinction between patterns becoming and not becoming part of languages in unclear. Furthermore, there is no doubt that many patterns are too complicated and generative to be directly supported in programming languages ever.

Structural Aspects

Type theory knows multiple ways of building compound data types: e.g., pairs, tuples, records (including classes), lists, variants (including options and enumerations), and references (Pierce, 2002). Type theory describes the formal semantics of these compound types and their relationships. However, this set of primitives seems to be chosen from the low-level implementation perspective; it does not necessarily reflect the ideas behind the programs. For example, an associative array (a map) is not considered a data composition primitive. That is understandable because an associative array is reducible to other data composition primitives. However, a key-value correspondence (a map) is a rather primitive operation in reasoning about programs, as many algorithms involve such correspondence. Therefore, the present work aims to raise the level of abstraction in comparison with traditional type-theoretical studies. It is not necessary for type-theoretical soundness, but the goal is to describe compound types suitable for expressing how people think about programs.

The structural patterns described by Erich Gamma et al. (Gamma et al., 1995), on the contrary, represent a too high level of abstraction. Such patterns as Adapter, Proxy, or Façade hardly can be implemented directly at the language level as they describe specific cases of a complex interaction between multiple participants. Some languages, e.g., Python (Pyt, 2023) and TypeScript (Mic, 2023), support the Decorator pattern, but it is not a part of their core functionality. Thus, an intermediate level of abstraction is required, which could support the implementation of well-known patterns but not represent them directly.

A balance between low-level composition types in type-theoretical studies and high-level compound types in programming languages may be found in programming language-agnostic ways to describe data structures. For example, Unified Modeling Language (UML) describes such structural relationships as composition and aggregation and their cardinality (OMG, 2017). UML semantics is informal, but there have been some attempts to formalise it, such as Swinging Types (Padawitz, 2000), formal Description Logics (Berardi et al., 2005), and SysML extension formalisations (Bouabana-Tebibel et al., 2012). However, the present work does not use UML structural relationships directly as they are far from how one usually defines data composition in programming languages. Nevertheless, the set of compound types in the present work was designed to support, among other things, UML primitives.

Olaf Zimmermann et al. proposed the concept of data transfer representation as a counterpart of a data transfer object (Fowler, 2012), which does not depend on the particular programming language used (Zimmermann et al., 2017). They described several common patterns used to build data transfer representation (Zimmermann et al., 2017). Their work relates to the narrower field of message-based remote application program interfaces, but a similar approach is applicable in the present work. The present work considers general structural patterns common for building data transfer representation and the data type composition in programming languages.

The problem of data composition also exists for data workflow definition languages. However, the composition primitives used in such languages are different. For example, Johan Montagnat et al. describe only two recurring data composition patterns in workflow defintion languages: one-to-one composition and all-to-all composition (Montagnat et al., 2006). Data composition primitives in workflow definition languages are complementary to those in programming languages as the former define dynamic data composition, whereas the latter define static data composition. According to Peter Kelly, the core difference is that workflow definition languages usually treat data as opaque (Kelly, 2011). At the same time, Peter Kelly pointed to the limitedness of this approach and argued for the inclusion of data manipulation facilities from programming languages to workflow definition languages (Kelly, 2011). The present work does not cover data workflow definition languages, so the composition primitives specific to them are outside the scope of the work. However, the static primitives described in the present work are applicable in workflow definition languages that support data manipulation.

Apart from how to build a compound data structure, the question of what such a structure is is also important. As mentioned in Section 1.2, type theory distinguishes nominal and structural type systems. In structural ones, a type is defined by its structure, and its name is only an alias. In contrast, in nominal systems, the name is a part of a type's identity, and two structurally equivalent types are not necessarily the same (Pierce, 2002).

Structural type systems are typical for SQL (ISO/IEC 9075-2:2023) and other query languages, such as SQL/SDA (Lin and Huang, 2001), GraphQL (He and Singh, 2010), or PGQL (van Rest et al., 2016). In these languages, a query returning a particular data structure is always compatible with another query returning structurally equivalent data (independently of their semantics). However, general-purpose programming languages more frequently use nominal or at least hybrid type systems (Pierce, 2002). It makes practical sense: for example, a complex number can be represented either with a real and an imaginary part or with an absolute value and an argument. Both representations contain two real numbers and are thus structurally equivalent, but they are not interchangeable in computations on complex numbers. Riley Moher et al. argue for using semantically rich types to ensure type safety (Moher et al., 2022). In general-purpose programming languages, it is ensured by the nominality of a type system and encapsulation.

Encapsulation is another related concept, which targets minimising interdependencies among separately-written modules by defining strict external interfaces (Snyder, 1986). Strictly speaking, encapsulation does not require information hiding (Schärli et al., 2004). However, it is usually associated with hiding low-level details, removing direct mutating access, and providing a high-level interface to them. Programming languages may have different encapsulation policies, which define the access rights to internal details (Schärli et al., 2004; Schärli et al., 2004). In particular, Janina Voigt et al. emphasised the difference between module encapsulation, when the minimal unit of encapsulation is a module (or a class in object-oriented languages), and object encapsulation, when internal details of an object may be hidden even from other objects of the same class (Voigt et al., 2010). Module encapsulation is more frequent than object encapsulation in widely-used programming languages, though the latter is used too.

The present work aims to combine these concepts in a simple but sufficiently generic model. For this purpose, the present work treats encapsulation as a consequence of the type system nominality. A nominal type necessarily has an internal data structure but hides it from the outside world. The present work does not specify how encapsulation policies can be defined. It is only assumed that the minimal encapsulation unit is a specific object, and access rights may be defined as necessary.

Behavioural Aspects

The behaviour of programs written in a programming language, i.e., how they are evaluated, is determined by the semantics of the language. As mentioned in Section 1.2, there are three main approaches to formalising semantics (Pierce, 2002):

- Operational semantics describes behaviour using an abstract machine.
- Denotational semantics interprets each term as a mathematical object, e.g., a number or a function.
- Axiomatic semantics defines a language as a set of laws for reasoning about program behaviour. Operational and denotational semantics operate such laws too, but they derive the laws from the abstract machine behaviour or terms' interpretation correspondingly. Axiomatic semantics, on the contrary, puts these laws into the foundation.

Historically, denotational and axiomatic semantics were associated with mathematical soundness and strictness, whereas operational semantics was considered weak and primarily suitable for only quick and dirty definitions (Pierce, 2002). On the other hand, operational semantics more easily operates such 'non-mathematical' concepts as concurrency and procedures (Pierce, 2002). For simplicity, the present work uses the operational style.

The primary behavioural abstraction in type theory is a lambda expression. It represents a function that can be defined once and then applied to arguments calculated in other parts of a program. The concept of a lambda expression originates from Alonzo Church's attempt to formalise the concept of an algorithm, reducing computation to function definition and application (Church, 1936). Church's lambda-calculus is not the only possible option but one of the most popular ways to formalise programs' behaviour (Pierce, 2002). In the ultimate (or pure) form, lambda expressions can represent anything: not only functions but also arbitrary data, like Booleans or numbers (Pierce, 2002). However, type systems that include base types instead (using lambda expressions only to represent behaviour) are closer to existing programming languages.

Another powerful abstraction, especially popular in functional programming studies, is a monad. Monads allow representing such impure features as state, exceptions, and continuations in a purely functional environment (Wadler, 1992b). Since a purely functional system is easier to model formally, encapsulating impure features in monads is helpful in formal studies. According to Philip Wadler's definition, a monad is defined by (Wadler, 1992a):

1. an operator M on types, which is applied to type x to create a new datatype Mx (e.g., if we consider lists of integers, x is the integer type, M is the generic list type, and Mx is the type of lists of integers),

2. function *map* of type $(x \to y) \to (Mx \to My)$, which accepts a transformation from type x to type y and converts it into a transformation from type Mx to type My (e.g., if we have a transformation of integers to strings, a list of integers can be transformed to a list of strings by applying the transformation to each element and collecting them in a list),

3. function *unit* of type $x \to Mx$, which creates a value of type Mx from a value of type x (e.g., for lists, unit creates a list that contains a single element), and

4. function *join* of type $M(Mx) \to Mx$, which takes a value of type $M(Mx)$ and 'flattens' it, producing a value of type Mx (e.g., transforms a list of lists of integers to a list of integers by taking all elements of the inner lists and collecting them in a list).

The operator and functions must satisfy the following laws (called monadic laws) (Wadler, 1992a):

1. $map(id) = id$, where id is the identity function $id(x) = x$ (e.g., if we apply the identity function to each element of a list, the result will be the same list),

2. $map(g \cdot f) = map(g) \cdot map(f)$, where $g \cdot f$ is function composition $(g \cdot f)x = g(fx)$ (e.g., if we apply composition of two functions to each element of a list, the result

will be the same as if we apply the first function, collect the values in a list and then apply the second function to each element of the new list),

3. $map(f) \cdot unit = unit \cdot f$ (e.g., if we turn an element into a single-element list and then apply a transformation to each element of the list, the result will be the same as if we first apply the transformation to an element and then convert it to a single-element list),

4. $map(f) \cdot join = join \cdot map(map(f))$ (e.g., if we flatten a list and then apply a transformation to each element, the result will be the same as if we first apply the transformation to each element of the inner list, and then flatten the result),

5. $join \cdot unit = id$ (e.g., if we take a list $[1, 2, 3]$, transform it into a single-element list of lists $[[1, 2, 3]]$, and then flatten the second list, we will get the initial list $[1, 2, 3]$),

6. $join \cdot map(unit) = id$ (e.g., if we take a list $[1, 2, 3]$, transform each element in a single-element list, thus getting $[[1], [2], [3]]$, and then flatten the second list, we will get the initial list $[1, 2, 3]$),

7. $join \cdot join = join \cdot map(join)$ (e.g., if we flatten a list of lists of lists twice, we will get the same result as if we first flatten each inner list and then flatten the list itself).

Another commonly used definition of a monad uses function *bind* instead of *map* and *join*, as well as other monadic laws, but these definitions are equivalent (Wadler, 1992b).

In procedural languages, behaviour mainly involves changing the name-value association (assignment). For this purpose, type theory describes such types as `Ref` (a reference, an abstraction of a location that contains a modifiable value), `Source` (a reference that can only be used for retrieving a value), and `Sink` (a reference that can only be used for assigning a value) (Pierce, 2002). Similarly to the structural aspects discussed above, the present work considers abstractions of higher level, closer to the language-level assignment operations. They are focused on the purpose of applying the low-level mechanisms known from type theory.

It is well known that the assignment operator in programming languages can have completely different meanings. For example, the Structural Operational Semantics (SOS) framework distinguishes binding and assignment (Mosses, 2006). The assignment assumes value copying (for primitive types only), whereas the binding makes different names refer to the same value (referential assignment). The Abstract State Machine semantics describes the same operations in terms of changes in the state of a computational device (Mosses, 2006).

The authors of the Anzen programming language fought assignment ambiguity by introducing three different assignment operators (Racordon and Buchs, 2019). This way, they wanted to make the assignment semantics explicit and avoid confusing usage of the same `=` operator in different senses, typical for many programming languages. In particular, the three assignment operators in Anzen correspond to the deep copy assignment, the shallow copy assignment, and the referential assignment.

The assignment may be constrained by type modifiers, such as the `const` keyword in C++. Type modifiers are a specific kind of subtyping relationship (Foster et al., 1999). For example, a reference to a mutable object may be passed instead of a reference to an immutable object because the receiver will not attempt to modify the object. Type modifiers often prohibit certain operations but can also bear different semantics (Carlson and Wyk, 2019). Prohibition of assignment operations is a typical use case of type modifiers.

The ultimate form of assignment constraints is the single static assignment (SSA) form: each symbol is assigned only once in the program (Cytron et al., 1991). Although it sounds purely functional, compilers of imperative languages (for example, LLVM-based (Lattner and Adve, 2004)) may use the SSA form as an intermediate representation.

The variability of the assignment semantics means that the meaning of assignment constraints is also different. For example, one can forbid changing the state of the object referenced by a variable or rebinding to another object.

When immutability guarantees are required, the name-value association should be unchangeable, implying the unchangeability of both the name-object association and the object state. However, it is difficult to achieve in most object-oriented languages (C++ being an exception). They usually allow declaring a name as a constant. However, for reference types (i.e., for all non-primitive types in such languages as Java (Gosling et al., 2023) and C# (ECMA-334)), it only prohibits changing the object the name references. The state of the object still can change. If a language does not provide means to forbid object state changes (as C++ does), such guarantees can only be provided by the object itself (or the class of the object as the allowed operations are defined there). Immutable objects are those whose state cannot be changed after creation (Potanin et al., 2013). Immutability guarantees are useful in ensuring invariants, concurrency (as immutable objects are thread-safe), security (Haack et al., 2007), and distributed architecture (Perry, 2020).

However, immutable data types do not fit well into the object-oriented paradigm. In object-oriented programming, subclasses usually introduce new fields and methods. Therefore, even if a class does not contain mutating methods, they can be introduced in a subclass. Prohibiting inheritance is only a partial solution because it will also prohibit adding non-mutating methods. Therefore, without language-level support for immutable types, it is not easy to guarantee that a particular type is immutable. Moreover, object-oriented language type systems usually do not even track which fields and variables a particular method mutates.

Another problem with immutable data types in object-oriented programming languages is the heavy load on the memory allocator and the garbage collector. Since every 'modification' of an immutable object creates another object and the intermediate steps of computation are usually not needed anymore, objects are created and deleted constantly. Typical general-purpose memory allocators are hardly suitable for frequently creating and deleting multiple small objects (Alexandrescu, 2001).

Probably due to the mentioned problems, immutability guarantees support at the language level is scarce even in such functional-capable languages like Scala (Odersky et al., 2023), Swift (App, 2023), and Kotlin (Jet, 2023). The standard libraries often contain immutable data types that they cannot be made mutable even on inheritance, but only the classes themselves ensure this restriction. Implementing another class with this property requires manually implementing the restriction on mutation. Moreover, due to the run-time environment's restrictions, one cannot strictly guarantee even this immutability notion. For example, even immutable objects in Scala change their state during creation (when a constructor is executing) (Odersky and Spoon, 2023). As a result, they are not safe in the concurrent environment because, in JVM, to which Scala compiles, objects may be accessible to other objects even before their construction has finished (Gosling et al., 2023).

1.4. Research Methodology

The present work is methodologically based on design patterns and type theory. The role of design patterns is twofold. On the one hand, decomposition of well-known design patterns shows that commonly used programming languages lack certain features that would make implementing the patterns more straightforward (Batdalov and Nikiforova, 2016). The programming languages lack expressiveness to implement the ideas communicated by design patterns shortly and concisely (Batdalov, 2017). Several such features, among other ones proposed by the author, are discussed in Section 2. In a sense, known design patterns are a source of ideas for possible directions of programming languages' development. This approach is in line with one of the current programming languages' evolution trends, discussed in Section 2.5.

On the other hand, the developed type system features are themselves described in the form of patterns. It allows analysing the solutions concerning the forces affecting it and describing the general case of solutions used in different languages. In this sense, the patterns approach is the core methodology of the present work: the type system features are not chosen arbitrarily but systematically described and generalised as much as possible by treating them as patterns. These pattern format descriptions are given in Sections 3–5. The suitability of the proposed constructs for the overall goals of the work (i.e., facilitating the implementation of design patterns) is verified in Section 6.

Type theory is used for formalising the proposed patterns as language-level constructs. The patterns are described as types with corresponding semantics and operations. Therefore, each pattern description in this work has an additional section 'Formalisation'. The described patterns and other, traditionally used, types comprise a complete type system. This type system is the main result of the present work.

2. TRENDS AND OBJECTIVES

The author, in his master's thesis, identified the following trends in the evolution of object-oriented programming languages (Batdalov, 2017):

- convergence between static and dynamic programming languages;
- convergence between object-oriented and functional languages, including using advanced type theory notions in object-oriented languages;
- lessening traditional restrictions on multiple inheritance and access to data members (data encapsulation);
- turning operators from built-in features of a language to class-defined methods;
- emergence of constructs corresponding to common design patterns.

The author also proposed a few potential language-level features in line with these evolution trends (Batdalov, 2017):

- Generalisation and interface extraction.
- Predicate and depth subclassing.
- Default implementation.
- Chameleon objects.
- Extended initialisation.
- Object interaction styles.
- Processing state management.

This section considers the objectives of the considered type system that follow from these trends and features. They are primarily discussed in the context of statically typed languages. In dynamically typed ones, they are less needed because dynamic languages have more opportunities to componentise and re-implement behaviour (Batdalov, 2016). In a sense, the proposed features would increase the flexibility of static languages towards what dynamic ones provide, but without violation of the typing rules (as discussed in Section 2.1).

2.1. Static and Dynamic Programming Languages Convergence

As discussed in Section 1.2, variables may be associated with types in either a static or a dynamic way, and programming languages typically tend to prefer one of these ways. Statically typed languages rely on compile-time type annotations, usually resolve functions applied to variables during compilation and erase type annotations as they are redundant at the run time. For example, suppose a compiler knows that a particular memory address holds an unsigned 32-bit integer. In that case, it can pass this address to a function that interprets data in this way and not to another function that interprets them as a four-ASCII character string (even though values of these two types may share the same binary representation).

On the other hand, dynamically typed languages typically do not require variables to have type annotations at the compile time. Instead, they keep type information at the run time and resolve applicable functions only when the functions are called. Thus, in the example above, the run-time environment must know how to interpret the given four bytes of data; the data themselves are insufficient.

However, this traditional distinction is getting blurred. First of all, giving type annotations to every variable is tedious. In many cases, the compiler can infer them from the context (the values assigned to a variable, the functions to which the variable's value is passed). Therefore, many statically typed programming languages allow omitting type annotations if the compiler can infer them automatically (e.g., `auto` in C++ (ISO/IEC 14882:2020), omitting type in Scala (Odersky et al., 2023) and TypeScript (Mic, 2023)). Type inference does not change the nature of statically typed languages (each variable still has a compile-time type). However, it removes the burden of annotating variables from the programmer. On the other hand, omitting type annotations can reduce code readability and should be used cautiously.

Second, statically typed programming languages often cannot afford to erase type information. Polymorphic functions are resolved at the run time and thus require keeping the type information until that moment (Pierce, 2002). C# took a logical next step and introduced dynamic classes, which behave like classes in dynamically typed languages (ECMA-334). If the run-time type information is already present, it is logical to perform polymorphic resolution always. Not only the choice of a specific method but also whether an object has a specific method or a property at all can be decided at the run time.

On the other hand, dynamically typed PHP allows type specification for the parameters and return types of functions since version 7.0 (PHP, 2023). The opportunity to specify function parameter types was also introduced in Python 3.12 (Pyt, 2023). Similarly, TypeScript is a whole language that supports compile-time type declarations for JavaScript variables and functions (Mic, 2023). TypeScript's motto, 'Typed JavaScript at any scale', discloses that type annotations are necessary for the maintainability of large programs. It seems that the absolute freedom of dynamically typed programming languages without type annotations makes it challenging to reason about programs.

Thus, even though statically and dynamically typed languages remain different in their fundamental principles, they gradually receive features that allow using the approaches usually associated with the other class. Dynamically typed languages give flexibility, statically typed languages give safety, and programmers need both. Table 2.1 summarises this mutual influence.

Since the present work does not propose a particular programming language nor its implementation, the choice between static and dynamic typing is not very significant. Type systems of static and dynamic programming languages are not as much different

Table 2.1
Examples of static and dynamic languages convergence

	Static	Dynamic
Type annotations	Traditionally: obligatory Exceptions: optional with type inference	Traditionally: not used Exceptions: allowed in function declarations in PHP and Python; allowed everywhere in TypeScript
Type erasure	Traditionally: used Exceptions: incompatible with certain types of polymorphism	Impossible

as the languages themselves. However, the trend to convergence suggests that some generalisation of these approaches would be beneficial. The present work assumes the following generalisation:

- Each symbol has a compile-time type, and each value has a run-time type.
- The run-time type of a value assigned to a symbol must be a subtype of its compile-time type.
- The type system contains the `Top` type, which is a supertype of any other type.
- Erasure of the run-time type information is allowed if the run-time type is known during compilation. For example, it is possible if the compile-time type does not have subtypes other than the type itself.
- Both choice of a method and detecting if a method exists can be performed polymorphically.
- If the run-time type information is erased, polymorphic methods are checked and chosen at the compile time.

The 'traditional' static typing approach in such languages as Java:

- does not have a common supertype for all types (only for object types), and
- supports the polymorphic choice of a method only (not checking if a method exists).

The 'traditional' dynamic typing approach is equivalent to the given generalisation if all symbols have `Top` as the compile-time type.

The relationship between the compile-time type and the run-time type is further discussed in Section 2.9.

2.2. Object-Oriented and Functional Languages Convergence

Functional programming is a programming paradigm based on strict assumptions, such as the absence of assignment, mutable data and iteration. Despite its restrictive environment, functional programming allows solving the same class of tasks as imperative programming and other programming paradigms. Functional programming languages have developed many high-level constructs that facilitate programming in this restrictive environment.

The popularity of functional programming has been increasing recently. As a result, object-oriented programming languages have acquired certain features that used to be strongly associated with functional programming only. Examples of such features are immutable data types and higher-order functions. The mainstream languages only borrow some functional features, but such languages as Scala and Swift fully incorporate the functional style. Object-oriented languages are not purely functional since they still support assignment and mutable data. However, the data types, algorithms and general approaches developed for functional programming languages can be applied in object-oriented ones as well. The benefits of this include mathematical soundness of functional constructs and their safety in a concurrent or a parallel environment.

Functional programming is based on function calls as the primary programming construct (Michaelson, 2011). Although functions are standard for imperative programming languages too, their role in functional programming is more important. Many operations for which imperative languages have alternative facilities are performed through function calls in functional programming languages. In particular, Greg Michaelson mentions the following primary differences between the functional and the imperative approach (Michaelson, 2011):

- In imperative languages, a single name may be associated with different values at different times (a variable). In functional languages, a name is only associated with one value.

- In imperative languages, a program consists of a sequence of commands, which most often change values associated with variables (assignment). In functional languages, a program is an expression that consists of a function call, which in turn calls other functions (function composition or nesting).

- In imperative languages, the result of program execution depends on the order of execution. Even the result of a function call in an imperative language may depend on the order in which the nested functions are called if these functions have side effects (e.g., a function changes the value of a global variable). In purely functional languages, nested functions cannot interfere with each other, and the result of a function call does not depend on the order in which these functions are called.

- In imperative languages, repetition of operations is performed using iteration, which repeatedly changes the same variables' values. In functional languages, repetition is performed employing recursion, which creates new names for each recursive call and assigns values to them only once.

- In imperative languages, a programmer can use assignment to change elements of a composite data structure (such as a record or an array). In functional languages, any operation on such data requires that a new data structure is created. This consideration influences the choice of appropriate data structures, e.g., in functional languages, lists are used more often than arrays. In general, functional programming languages prefer nested (recursively defined) data structures and process them with recursive function calls of the same structure.
- In functional languages, functions are treated as values that can be arguments of other functions (higher-order functions) or returned from a function call.

It can be seen that, in almost all these cases, facilities of functional languages are restricted in comparison with imperative ones. The only feature that is crucial for functional languages but is not always supported in imperative languages is treating functions as values (or, as often said, 'first-class citizens'). Other differences assume lack of a particular feature in functional languages (assignment, iteration or changing a composite data structure). Moreover, imperative languages do support the features that functional ones use instead of the lacking features (function calls, recursion, recursive data structures). Of course, this lack of functional languages features in does not mean that they allow solving fewer tasks than imperative ones. Functional languages are based on lambda-calculus, which was one of the first attempts to formalise the notion of an algorithm (Michaelson, 2011). Since all formalisations of an algorithm are proved to be equivalent, functional languages solve the same set of tasks as imperative ones. However, considering the lacking features, it is not clear why programming in the functional style instead of the imperative one makes sense at all.

Traditionally, benefits of functional programming are associated with mathematical soundness. Usual mathematical systems rarely deal with mutable state and order of operations. Instead, they work with generally valid truths that can be proved formally. Function calls on immutable data structures represent precisely this mode of operation. Furthermore, functional programming languages support advanced ways to construct data types, such as algebraic types (intersections and unions of types as value sets) and parametric polymorphism (Pierce, 2002). They reflect common mathematical concepts and thus make the language used to reason about programs closer to the language of mathematics. Theoretically, the validity of a functional program is a theorem that can be proved or invalidated.

Moreover, functional programming languages are typically equipped with a rich arsenal of immutable data structures and operations that can work in this 'restricted' environment. The data structures include lists, vectors, ranges, immutable sets, etc. Typical functional operations on immutable data include, for example:

- mapping (applying the same function to each element of a sequence),
- reducing (recursive calculation of a single value, e.g., the sum of a sequence),

- copy-change (creating a data structure that holds the same data as the source except for some changed parts).

These data structures and operations are operational counterparts of imperative ones:

- One can use lists and vectors instead of arrays.
- One can use mapping instead of data structure transformations in place.
- One can use reducing instead of some frequent iterative algorithms.
- One can use copy-change instead of assigning to an element of a data structure.

However, the 'typically functional' data structures and operations work in the functional programming paradigm's 'restricted' environment. Therefore, it is theoretically possible to prove mathematical theorems about them and use them to verify a program's correctness.

However, the actual provability of program correctness hardly meets the original expectations. Greg Michaelson points out that the lack of mutable state is itself a significant hindrance for reasoning about applications that do their work in time and that proving properties of functional programs depends on the highly compositional structure of a program (Michaelson, 2011). Proving program correctness is challenging, whether imperative or functional (Michaelson, 2011).

In practice, we cannot prove the absence of any errors in a reasonable time even for functional programs. We can only prove the absence of certain erroneous conditions, guaranteed by the type system of a language (Pierce, 2002). Furthermore, we prove each program's correctness separately, whereas a type system's guarantees are enough to prove once per language. Concerning the type system, functional languages do not have substantial benefits compared to imperative ones. Imperative languages also have their type systems, whose guarantees can be proved formally.

However, due to their initial interest in mathematical soundness, functional language creators pay much more attention to the type systems' guarantees and their formal proof. Type systems themselves are often described in functional languages, such as ML (Pierce, 2002). In a sense, functional languages have driven the development of the established type theory in general. Therefore, at least in this sense, functional languages are still associated with mathematical soundness and strictness.

Another benefit of functional languages is related to their independence of the order of execution. Thanks to this independence, functional programs are inherently safe in a concurrent environment. No locks or other synchronisation mechanisms are needed because there is no shared mutable state. Furthermore, for the same reason, functional programs can be easily parallelised to multiple workers with subsequent collection of the computation results. This feature is employed in such parallel computation systems as Google MapReduce (Dean and Ghemawat, 2008), Apache Hadoop (Apa, 2023a) and Apache Spark (Apa, 2023b).

Recursive immutable data structures also benefit from the opportunity to share data between different instances. For example, a list is defined as either an empty list or an element (head) followed by another list (tail). The tail itself is another list, which can have another name in a program. Even though the program works with two lists with different names (the whole list and its tail), the shared part of their data is stored in memory once. An example of such a structure is shown in Figure 2.1.

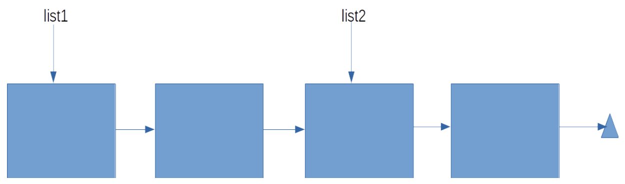


Figure 2.1. Example of shared data in immutable structures.

Therefore, the copy-change operation discussed above (e.g., adding an element to the beginning of a list) often does not even require to copy existing data. Instead, a data structure that contains new data and a reference to the old data structure is created. Since both data structures are immutable, sharing data is perfectly safe because it is impossible to change one of them and get unexpected side effects on the other one.

There are also some disadvantages caused by the restrictions of functional programming. First of all, relying on recursion instead of iteration may be inefficient. Each recursive call creates a frame in the program stack, which holds the state of local variables for each call. As a result, a recursive algorithm typically requires memory proportional to the number of calls, whereas the corresponding iterative algorithm may work in constant memory. For example, to find $n!$, a recursive algorithm has to create n stack frames, but a single accumulator is sufficient for an iterative one. In some cases, recursive algorithms may lead to a stack overflow in a situation that is easy for an iterative one. Repeated saving and restoring call contexts also requires extra time for recursive algorithms.

The technique known as tail recursion is often used to manage the problems of recursion inefficiency. Suppose the recursive call is the last operation in a function. In that case, a compiler may substitute the call with jumping to the beginning of the function without additional actions from a programmer (Pratt and Zelkowitz, 2001). However, the opportunity to apply this optimisation critically depends on whether the function is written in the tail-recursive form. For example, since $n!$ is defined as 1 for $n = 0$ and $n(n-1)!$ for $n > 0$, it is natural to define the factorial function as shown in Figure 2.2 (using Scala syntax). The example and its following variations are taken from the 'Functional programming principles in Scala' online course (Odersky, 2016).

```
def factorial(n: Int): Int =
  if (n == 0) 1
  else n * factorial(n - 1)
```

Figure 2.2. Natural (non-tail-recursive) factorial definition.

This form is not tail-recursive since the last operation in the computation is multiplication. The function may be re-written in the tail-recursive form, for example, as shown in Figure 2.3.

```
def factorial(n: Int): Int = {
  def _factorial(n: Int, acc: Int): Int =
    if (n == 0) acc
    else _factorial(n - 1, acc * n)

  factorial(n, 1)
}
```

Figure 2.3. Tail-recusive factorial definition.

The function can be optimised now. However, this form is more complicated and loses the recursive definition's main benefit, the natural mapping between the function definition in a program and the mathematical definition. Essentially, this form hides the iterative algorithm behind the recursive form. For simplification of such functions, functional languages usually support a rich set of common patterns of recursion. For example, in Scala, the factorial function can be defined as shown in Figure 2.4.

```
def factorial(n: Int): Int = (1 to n).product
```

Figure 2.4. Factorial defined as an integer range product.

There are also more general functions (combinators), which can be used even if the predefined patterns are insufficient. For example, the factorial function can also be written in the way shown in Figure 2.5. This form has the benefit that an arbitrary function can be used instead of multiplication to perform non-trivial combinations. The functions of this kind are implemented tail-recursively internally. Thus, they redeem the programmer from the complicated transformation of a function to the tail-recursive form. However, they still express the iterative logic of computation and not the mathematical definition. Thus, the hopes to use functional programming as a direct counterpart of mathematical constructs, are not fulfilled in this part either.

```
def factorial(n: Int): Int = (1 to n).foldLeft(_ * _)
```

Figure 2.5. Factorial defined using a general combinator.

Another disadvantage is caused by the fact that not all widely used data structures allow recursive definitions. For example, unlike lists, mentioned above, immutable sequences that support efficient random access to their elements (like arrays in imperative languages) are not easy to define recursively. Without a recursive definition, these data structures do not provide such benefits as data sharing and the absence of data copying on creating a modified data structure. Functional algorithms to process such data structures may be complicated because the recursive structure of an algorithm typically reflects the recursive structure of the data structure itself.

Linked data structures, such as lists, also suffer from inefficient memory access to non-contiguous data. Performance of modern computers critically depends on caching, which depends on the ability to put sequentially processed data in contiguous areas.

Random long jumps, typical for processing linked data structures, significantly deteriorate algorithms' performance.

However, some trade-offs are possible. For example, Scala's immutable counterpart for arrays is vectors, which are internally represented by trees with the branching factor of 32 (Odersky and Spoon, 2023). Vectors provide logarithmic access time and require to copy only a small fraction of data when a modified vector is created.

Despite the described problems, the features of functional programming languages are strongly associated with mathematical soundness and thread safety. They are designed to work in functional languages' restricted environment but can exist in the languages supporting such operations as assignment and iteration. Therefore, it is reasonable to speak not about imperative and functional languages but rather about imperative and functional programming styles. Under certain conditions, programming in the functional style is also possible in languages that are not purely functional.

The degree of support of the functional style in object-oriented programming languages is different. In general, two groups may be distinguished. One of them consists of traditional (mainstream) object-oriented languages, such as C++, Java and C#. They tend to get some features useful for programming in the functional style but not the whole set of functional capabilities. The other group consists of languages for which extended support of the functional programming style is one of primary design goals. Examples of such languages are Scala, Swift, and Kotlin. However, in both groups, the number of supported functional features is growing over time.

Thus, integrating functional facilities in object-oriented programming languages is one of the current trends in the evolution of object-oriented languages. In the context of this work, this means that the type system under design should integrate functional languages' features from the very beginning. The main features to consider at the language level are:

- functions as values (see Section 4.1) and
- algebraic data types (see Sections 3.6 and 3.8).

Other important language-level constructs are guaranteed immutability and parametric polymorphism, but they are not considered in the present work to keep the discussion scoped.

2.3. Lessening Traditional Object-Oriented Restrictions

Some concepts and constructs of object-oriented programming languages impose significant restrictions on their usage. The purpose of the restrictions is to support 'disciplined' usage of language features for the sake of safety or good object-oriented design. However, in some cases, the original restrictions tend to be relaxed over time. Usually, some restrictions continue to exist, but the opportunities of the corresponding constructs tend to grow. This process probably shows that the original, restricted constructs proved to be insufficient for programmers and required extension for better expressiveness. This section considers a few examples of such restrictions.

Interfaces as Restricted Classes

In Java, interfaces are the solution to the problems related to multiple inheritance. Similar concepts in other languages may be called mixins or traits. In Java, a class may inherit from only one class but implement any number of interfaces (Gosling et al., 2023). Therefore, interfaces describe 'additional' traits of classes. Some design methodologies recommend using classes for nouns and interfaces for adjectives appearing in a problem description (Larman, 2016).

Originally, Java interfaces were very restricted counterparts of classes, which were allowed to contain only declarations of public methods. However, since Java 8, interfaces may also define the so-called 'default' implementation of methods, used as a fallback if a class implementing an interface does not define or inherit another implementation (Gosling et al., 2023).

In TypeScript, the opportunities of classes and interfaces differ even less because interfaces can also define data members (Mic, 2023). The difference between classes and interfaces in TypeScript is related to nominal vs structural types. Interfaces are purely structural, whereas classes may have private members and non-trivial constructors. Therefore, structurally equivalent classes, unlike interfaces, may not be equivalent in the language; that is an element of a nominal type system.

In Scala, traits can also define private members. Only constructor parameters are disallowed in Scala traits (Odersky et al., 2023). Interestingly, Scala traits have more opportunities than Java interfaces, even though Scala source code is compiled into Java Virtual Machine (JVM) bytecode. Scala compiler creates additional classes when a trait cannot be compiled into an interface, thus making a trait a generator of Java classes instead of a binary object directly supported by JVM.

Thus, there is a tendency for extension of interfaces capabilities. The 'additional' traits of classes can be as complicated as the 'main' ones. Therefore, an overly restricted construct for expressing the 'additional' traits is not expressive enough and gets extended over time.

Evolution of Data Access Restrictions

Private and protected class members (the ones that can be accessed from within the class only and from the class and its descendants correspondingly) are vital in encapsulation. They play a crucial role in the open-closed principle of object-oriented design (software entities should be open for extension but closed for modification) (Martin and Martin, 2006). In particular, making all data members private and accessing them only through methods is considered a good style (except for classes that have intentionally open internal structure) (Martin, 2008). However, having done this, programmers often create getters and setters, methods that allow direct access to and modification of the data members. Even though creating getters and setters for every private data member should be avoided (Martin, 2008), this operation is so typical that it brought to life such constructs as properties in C# (ECMA-334) and `get` and `set` methods in TypeScript (Mic, 2023). Again, the original restrictions were too strict and required greater flexibility from the languages.

Pointers and References in C++ and Java

Being backwards compatible with C, C++ inherited the concept of a pointer, a memory address that provides indirect access to a value. However, C pointers are known to be error-prone and allow various unsafe operations. For example, a pointer can hold an arbitrary value, which can cause a segmentation fault if the program tries to access the value at this address. To address some pointers shortcomings, C++ introduced another indirect access type, a reference. References are different from pointers in the following aspects (ISO/IEC 14882:2020):

- a reference must always be initialised on creation by binding to an existing variable (so, it always refers to a valid and accessible memory location when created),
- a reference cannot be reassigned (it is always bound to the same memory location), and
- a const reference (a reference to a value that does not allow to modify the value) can be bound to a temporary value (e.g., the result of an expression). Pointers and non-const references can only address variables[1].

Most often, references can be viewed as non-optional pointers. References express additional safety restrictions and thus are safer than pointers. However, they are not entirely safe. For example, even though a reference must address a valid memory location on creation, this location can be freed later. An attempt to use such a reference will again cause a segmentation fault.

Furthermore, references differ from pointers in several relatively independent aspects. For example, it is impossible to get a non-optional but rebindable pointer with C++ built-in types only. Such a situation requires using pointers as a less restrictive type and losing the references' safety guarantees.

References also do not help with another typical pointers' problem, memory leak. If a memory chunk has been allocated, but all pointers and references to it have gone out of scope, the location cannot be accessed anymore. To partially address this problem, C++ Standard Library provides so-called smart pointers, `std::unique_ptr` and `std::shared_ptr` (ISO/IEC 14882:2020). They incorporate the concept of ownership, i.e., who is responsible for releasing memory when it is no longer used. However, smart pointers do not provide safety guarantees: they are optional and can point to already freed memory (e.g., it is possible to get a raw pointer from a smart pointer and apply the delete operator to that). Nevertheless, smart pointers are a vital step in ensuring the correctness and safety of programs.

Java standard library also provides more flexibility than the built-in types do. Garbage collection and the absence of explicit memory releasing in Java should ensure against

[1]Strictly speaking, pointers can address whatever, but the address-of operator `&`, required to bind a pointer to an existing value, cannot be applied to temporary values.

memory leaks and unsafe memory access. However, Java standard library contains such types as `SoftReference`, `WeakReference`, and `PhantomReference`, which have special garbage collection rules (e.g., the referenced objects may be garbage-collected even when accessible) (Ora, 2023). Again, the basic concepts encoded in built-in types are not expressive enough in more complicated situations.

Memory management is an important and complicated problem, and it is impossible to reduce it to one or two basic types. Programming languages tend to provide diverse solutions to this problem, which are more flexible than the built-in types and can be tailored to a specific situation.

Influence on the Considered Type System

The most important lesson to learn from the examples above is that it does not make sense to define two or more similar concepts that differ in more than one aspect. Such a design can cover only a limited number of allowed combinations, whereas other ones (such as interfaces with data members or non-optional rebindable pointers) cannot be expressed. The number of possible combinations grows exponentially, and we cannot define a separate concept for every combination. Andrei Alexandrescu wrote that each independent design decision or constraint should become a separate policy, and every combination of orthogonal policies should be supported (Alexandrescu, 2001). He said that about class design, but this idea applies to language-level constructs as well. The author tried to follow this principle in the considered type system as much as possible.

The principle described in the previous paragraph is rather a general guideline than specific decisions. However, the discussed specific examples affect the interpretation of inheritance (see Section 2.6) and encapsulation (see Section 4.3) applied in the present work.

2.4. Operators as Class-Defined Methods

This trend is loosely related to the present work because type systems have little to do with operators. In comparison with functions, operators are primarily syntactical constructs, affecting how the compiler builds the abstract syntax tree. Operators can always be substituted with appropriate function calls. Only the final functional transformation exists at the type system level, whether expressed as a function or an operator in the source code. Therefore, operators are only briefly discussed in this section.

Usually, programming languages have a fixed set of operators, and the language provides an implementation of operators for the built-in types. However, when it comes to the user-defined types, some languages allow programmers to define operators for these types (e.g., C++ (ISO/IEC 14882:2020)) and some do not (e.g., Java (Gosling et al., 2023)). If operators for user-defined types are not allowed, programmers should define corresponding methods instead and compose expressions of method calls. If allowed, programmers typically cannot change the priority and associativity of operators so that building an abstract syntax tree is unambiguous.

Scala has two novel features concerning user-defined operators. First, a programmer may define an arbitrary operator without limiting it to a predefined set (Odersky et al., 2023). From Scala's point of view, operators are simply methods with special names. An operator's priority and associativity are defined by the character sequence used for the operator (Odersky et al., 2023). Second, Scala allows the call by name strategy in operators (just like in other methods), which means that a programmer may define that an operand is evaluated only when needed (Odersky et al., 2023). The call by name strategy is not specific to Scala: the operands of conditional operators are not evaluated beforehand in virtually all languages that have conditional operators. Doing it the other way would significantly diminish the opportunities of conditional operators in checking error conditions (Pierce, 2002). However, other languages typically use the call by value strategy in all methods and user-defined operators (the operands are always evaluated before applying the operator).

The discussed trend can be generalised in the following way: there is nothing specific about built-in types. Every feature or trait that can be applied to a built-in type can be applied to user-defined types as well. It is one of the main reasons to introduce generalised functions (see Section 4.1).

2.5. Constructs Corresponding to Common Design Patterns

The relationship between design patterns and language constructs is complicated. Erich Gamma et al. said that what is a design pattern in one language is just a language construct in another one (Gamma et al., 1995). On the other hand, Frank Buschmann et al. distinguished architectural patterns, design patterns, and idioms (Buschmann et al., 2013). Only idioms are specific for particular programming languages, whereas architectural and design patterns express higher-level generative concepts, which can be *implemented* but not *represented* directly in the code. However, even with this caveat, it is clear that many programming languages have acquired features corresponding to common design patterns (or, in other words, features that reduce the implementation of a design pattern to a single operator). Examples of such features include:

- for-each loop, implementation of the Iterator design pattern (Gamma et al., 1995),
- objects in Scala (Odersky et al., 2023), implementation of the Singleton design pattern (Gamma et al., 1995),
- channels in Go (Goo, 2023), implementation of the Pipes and Filters architectural pattern (Buschmann et al., 2013), and
- events in C# (ECMA-334) and observables in Kotlin (Jet, 2023), implementation of the Publisher-Subscriber design pattern (Buschmann et al., 2013).

The present work follows the same trend, as discussed in Section 1.1. The functional objectives described below are inspired by the widely known design patterns to a great extent.

2.6. Generalisation and Interface Extraction

Inheritance in object-oriented programming languages is a special case of subtyping because any object of a subclass is considered an object of its superclass (Pierce, 2002). The subtyping interpretation has been criticised for a long time (Cook et al., 1989; Taivalsaari, 1996). Antero Taivalsaari proposed a broader interpretation of inheritance as an incremental modification in the presence of late-bound self-reference (Taivalsaari, 1996). However, most statically typed languages still assume the subtyping relationship between classes and subclasses.

The primary obligation the subtyping relationship imposes on the language and the programmer is Liskov Substitution Principle: subtypes must be substitutable for their base types (Martin and Martin, 2006). Therefore, a program that works correctly given an object of some class should also work correctly with an object of its subclass.

Robert C. Martin and Micah Martin mention that such practices as removal of some behaviour in a subclass (or cancellation in Taivalsaari's terminology (Taivalsaari, 1996)) violate Liskov Substitution Principle (Martin and Martin, 2006). However, despite this technical problem, removal of a behaviour represents a valid way of thinking: generalisation of known concepts (instead of specialisation represented by the traditional inheritance). For example, complex numbers were historically introduced later than real numbers, though real numbers are a special case of complex numbers and support other types of behaviour (e.g., are comparable).

Therefore, the actual problem is not that removing behaviour violates Liskov Substitution Principle but that traditional inheritance allows a specialisation but not a generalisation. If a class could be defined as a superclass of a previously defined class, removing behaviour would be natural and consistent with Liskov Substitution Principle.

Defining a class through its subclass apparently violates the standard rule that a class should be ignorant about its subclasses. In turn, it may cause violation of the open/-closed principle of object-oriented design: software entities should be open for extension but closed for modification (Martin and Martin, 2006). However, a type system may guarantee that the superclass has only the opportunities that an independently defined superclass could have (e.g., the superclass cannot reuse the implementation that uses private members of the subclass). Under this condition, the open/closed principle would not be violated.

Implementation of this proposal requires caution due to the complexity of the inheritance relationship. Typically, inheriting from a class means that:

- The subclass has the same interface (externally visible method declarations) as the superclass and may extend it (add new methods).
- The subclass has the same implementation (including non-externally visible methods) as the superclass and may override it.
- The data representation of the superclass became a part of the data representation of the subclass.

The fact that these operations are bundled together is one of the main reasons why Erich Gamma et al. recommended inheriting from abstract classes only and favouring object composition over class inheritance (Gamma et al., 1995). If object composition is applied, only the data representation becomes a part of the new class automatically, whereas the interface and the implementation do not. The implementation may be reused by calling methods of the member object. Reusing the interface requires copying its required parts into the new class. The latter requirement makes object composition more challenging to use when interface compatibility is needed.

According to the discussion in Section 2.3, the present work tries not to bundle independent things together. Therefore, the considered type system contains different relationships:

- inheritance of the interface,
- delegation of the implementation of the methods, and
- data composition.

Inheriting the interface is obligatory when a class is derived from another one. Without inheriting the interface, reusing the implementation of the methods does not make sense, and data composition can be used independently without marking a class as derived. Delegation and its relationship to data composition are discussed in Section 4.3.

In order to support both a specialisation and a generalisation, the following rules are used:

- A class derived from another class can be either a subtype or a supertype of the base class.
- A derived subclass inherits the interface of its superclass and may add new methods.
- A derived superclass inherits the interface of its subclass and may remove some methods.
- A derived class (independently on the direction) may delegate method calls to its base class or provide a new implementation.
- A derived subclass may include an instance of a superclass (equivalent to object composition) and delegate data access to it but does not have to. Instead, it can provide data accessors to its independent data structure so that all reused implementation of methods works correctly.
- A derived superclass must provide such data accessors (without reusing the base class data structure) that all reused implementation of methods works correctly.

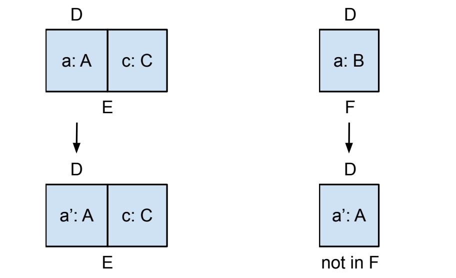
2.7. Predicate and Depth Subclassing

Most mainstream programming languages support only width subclassing, i.e., adding new members. Type theory also describes depth subclassing, i.e., subtyping the types of existing members (Pierce, 2002). For example, given that `B <: A`, `C` is an unrelated type, and class `D` has field `a` of type `A`:

- Class `E` that inherits from class `D` and adds field `c` of type `C` would be a width-subclass of `D`.

- Class `F` that inherits from class `D` and changes the type of `a` to `B` would be a depth-subclass of `D`.

The author previously showed that depth subclassing would be beneficial in implementing such design patterns as Template Method and Visitor (Batdalov and Nikiforova, 2016). However, depth subclassing support in programming languages is scarce. The main problem is probably using subclass references as mutable superclass references. An illustration of the problem is shown in Figure 2.6. If only width subclassing is used, it is possible to mutate the state of the inherited part, and the result will still be of type `E`. This fact allows using a reference to `E` as a reference to `D`. However, a reference to `F` cannot be used as a mutable reference to `D`. If some method of `D` changes the state of `a`, it may assign an arbitrary value of type `A`, and thus violate the restriction defined in `F`.



(a) Width subclassing (b) Depth subclassing

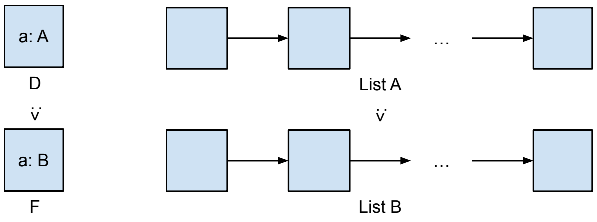
Figure 2.6. Mutable references using width and depth subclassing

The same challenge would appear in the case of a copy-change operation: creating a copy of an object with one field changed (see Section 5.3). However, using a reference to `F` only for reading would not cause problems. It is effectively the same problem as the variance of generics, discussed in Section 1.2. Indeed, depth subclassing is very similar to subtyping the type of elements held in a container (see Figure 2.7).

In terms of variance, the problem of depth subclassing can be described in the following way. Type theory distinguishes different types of references: `Source` that allows only reading a referenced value, `Sink` that allows only writing to a referenced storage, and `Ref` that allows both operations (Pierce, 2002). According to the variance rules (Pierce, 2002):

- `Source` is covariant: if `B <: A`, `Source B <: Source A` because one can read a value of type `A` from a reference of type `B`.

- `Sink` is contravariant: if `B <: A`, `Sink A <: Sink B` because one can write a value of type `B` to a reference of type `A` (with a possible data loss, though).



(a) Depth subclassing (b) Container element subtyping

Figure 2.7. Depth subclassing vs container element subtyping

- `Ref` is invariant: only if both `B <: A` and `A <: B`, `Ref A <: Ref B` and vice versa. The opportunity to read from and write to a reference requires the types to be equivalent in the subtyping relationship.

However, mainstream programming languages usually allow using mutable references covariantly. As discussed above, the existence of only width subclassing allows that. Supporting depth subclassing would require solving the variance problem for object references.

As discussed in Section 1.2, the two most popular solutions for the variance of generics are:

- Declaration site variance, in which covariant type parameters may serve only as return types, contravariant type parameters may serve only as argument types, and invariant type parameters may be used in both positions.

- Usage site variance, in which methods may be defined in terms of existential types, whose parameters are not known in advance, but the type system proves that such types exist.

It is not difficult to extend the declaration site approach to the case of depth subclassing, but such a solution would hardly support a simplification for width subclassing. The declaration site approach does not know anything about the relationships between particular classes and superclasses and cannot distinguish the cases when covariant usage of return types is safe. Therefore, the solution considered in the present work is based on the usage site variance. In particular:

- A method that can mutate the current object's state is treated as having an implicit extra return value, the object's state after the mutation.

- When a class acts as the return value type of its method (either explicit or implicit for mutating methods), it is treated as an existential type (i.e., the return value type may be different from the class itself).

- The actual type is inferred when such a method is called. The type inference considers, among other things, the way how a subclass was formed (width or depth subclassing).

- The variable that receives a return value must be of an appropriate type. In particular, it means that some methods defined in a superclass cannot be called on a variable of a subclass (but only if depth subclassing is involved).
- A necessary practical consequence is that if we have a variable of type `A`, its state may change to any other object of class `A`. It will be important in Section 2.9.

Depth subclassing can be generalised to predicate subtyping, in which a subtype may be narrowed using an arbitrary predicate (Batdalov and Nikiforova, 2016). For example, regular expressions are strings in a specific format. The same solution described above should work for predicate subtyping, so it makes sense to consider the general case.

2.8. Default Implementation and Execution Contexts

Usually, creating an object requires specifying a concrete class to instantiate. Even if the object is further used as an object of an abstract class (e.g., one only needs a map, no matter whether it is a hash table or a tree-based map), abstract classes are not instantiable directly. If a programmer always uses the same implementation for an abstract class but later decides to change the implementation, all instantiation statements should be changed.

The Abstract Factory pattern solves this problem by encapsulating object instantiation in a separate class, a factory of objects (Gamma et al., 1995). It is possible to define a factory for the abstract class and choose a specific implementation in the creation method body. The factory is often a singleton and usually allows changing the used implementation at the run time. Thus, the concrete implementation of an abstract class may be easily changed without touching the usage sites and even does not have to be known at the compile time.

An alternative approach to solving the default implementation problem is dependency injection, usually supported at the framework level (for example, in Spring Framework (Johnson et al., 2023)). However, dependency injection works in the object initialisation phase by providing necessary arguments to the constructor (Johnson et al., 2023). Therefore, it is not suitable when an object is created outside a constructor of another object.

In the presence of at least three different methods of object instantiation, traditional constructors, factories and dependency injection, one has to choose between them at the compile time. Changing this choice if one proves insufficient or inappropriate requires changing every instantiation statement again.

The considered type system uses abstract factories as the primary way of instantiating objects. Every class, even an abstract one, may have an associated factory singleton object, which takes care of creating objects of this class. The factory object may choose the default implementation according to its own logic (which does not prohibit instantiating another implementation directly when the default one is not suitable). It is essentially the approach used in Scala, where the companion object of a class may be responsible, among other things, for creating objects (Odersky et al., 2023). The main difference is in the scope of run-time changes, discussed further.

Assigning a default implementation to a class is a kind of binding, so its scope should be clearly defined (Batdalov and Nikiforova, 2016). On the one hand, the scope cannot be restricted to being local because changing the used implementation would require changing every client class. On the other hand, a non-local scope is a shared state, which should be carefully treated in a concurrent environment. Actually, it is a more general problem of the scope of singletons (as mentioned above, an abstract factory is usually a singleton). Most implementations of the Singleton pattern (Gamma et al., 1995) assume the global scope, which means that a singleton object is the same for all code that uses it and creates tight coupling between a singleton and its usages. The tight coupling contradicts the ultimate goal of design patterns and creates multiple problems in designs employing singletons (Sommerlad, 2007).

In the considered type system, all singletons (including abstract factories) belong to execution contexts. An execution context is a space of globally accessible names inherited when spawning a new thread. A thread may modify its execution context and the execution context of the child threads, but not the execution context of the parent thread. This solution should allow replacing singleton objects when required and eliminate tight coupling. Execution contexts are associated with executors (see Section 4.2).

2.9. Chameleon Objects

The State pattern allows an object to change its behaviour at the run time so that it appears to change its class (Gamma et al., 1995). The State pattern is reflected in the UML state machine diagram, although the UML assumes the behaviour of all states to be implemented in one class (OMG, 2017). However, placing unrelated behaviour in different classes is a better coding style (Martin, 2008). Therefore, the multi-class implementation of a state machine is considered here.

The essence of the State pattern is that a single object behaves as an object of different classes at different times. The original implementation of the pattern employs an extra level of indirection (the State class), which redirects calls to actual objects (the ConcreteState classes) (Gamma et al., 1995). The author proposed to support such behaviour directly at the language level by allowing objects to change their classes at the run time (Batdalov and Nikiforova, 2016).

There are two main problems to solve (Batdalov and Nikiforova, 2016):

- Run-time changes of an object type must be type-safe.
- Classes may differ in their data structure, and then changing the class involves data conversion.

The considered type system ensures type safety by assuming that the run-time type of an object may change only within the compile-time type of the variables that refer to it (see Section 2.7). In particular, it means that one cannot have mutable references of different types to the same object. It agrees with the traditional understanding of a reference as an invariant type (Pierce, 2002). A different view on reference variance in mainstream programming languages and the solution of this problem for the considered type system are discussed in Section 2.7.

Possible differences in the data structure are considered a responsibility of the class itself. Thus, to change the run-time type of an object, a method may either change the run-time type tag (if the data structure is shared) or replace the object completely.

The final rules are as follows:

- A method defined in a class may mutate the current object to belong to another class.
- Such a method must declare the type within which the current object's type may change.
- The object's type change is allowed only within the compile-time type of a variable referencing it (i.e., if a method declares a broader type change, it cannot be called).
- Different mutable references to the same object cannot have different compile-time types (see Section 2.7).

2.10. Extended Initialisation

The lifecycle of an object consists of several phases, e.g., initialisation, run time, and finalisation. The language may have different rules for different phases. For example, `final` fields in Java may be assigned only from the object constructor (Gosling et al., 2023). `const` fields in C++ may be assigned only in initialisation statements (but not in the constructor body) (ISO/IEC 14882:2020). As a result, all data necessary for object initialisation must be parameters of a single constructor.

Having many parameters of a single method is error-prone and undesirable (Martin, 2008). Creational patterns, such as Builder and Factory Method, overcome this problem (Gamma et al., 1995). However, they are not exceptional from the point of view of the language and do not allow modification of constant fields outside the constructor. Therefore, these patterns require one of two options:

- Make all fields non-constant even if one does not need to modify them after the initialisation.
- Make the pattern participants use a multi-parameter constructor, which is better than using it on the client side but still error-prone.

The author proposed to extend the initialisation phase so that it can consist of several operations instead of a single constructor (Batdalov and Nikiforova, 2016):

- Some methods are callable during the initialisation phase only (and thus allowed to modify constant fields).
- The methods that can be used at the object run time must respect the constantness of the fields.

Different behaviour during and after initialisation effectively means that the type of an object changes depending on the lifecycle phase. Therefore, the proposal may be implemented by combining the approaches described in Sections 2.7 and 2.9 according to the following principles:

- Each class has subclasses for different lifecycle phases. However, the considered type system does not prescribe a specific list of lifecycle phases.
- States may be nested; therefore, having lifecycle phase states does not forbid user-defined states for chameleon objects.
- Different states may have different conditions on their data. For example, a non-optional pointer may be null during initialisation but must have a non-null value at the run time.
- Different states may have different behaviour. For example, the initialisation phase state may define methods that modify fields that will eventually be constant.

2.11. Object Interaction Styles

Most often, objects interact with each other by synchronous function calls. It is the simplest and universal way of interaction. However, programming languages tend to introduce alternative ways of interaction, such as asynchronous calls in many languages and channels in Go (Batdalov, 2017). The alternative interactions styles are ultimately reducible to synchronous calls but allow expressing object interaction more naturally.

The set of currently supported interaction styles does not seem complete (Batdalov and Nikiforova, 2016). For example, another class may represent an external system or a component, e.g., when the Façade, Proxy (Gamma et al., 1995) or Broker (Buschmann et al., 2013) pattern is used. Component interaction styles are more diverse than programming languages currently support: synchronous request-response, asynchronous request-response, pipe&filter, broadcast, blackboard, publish-subscribe (Crnković et al., 2011). Many design patterns representing these styles are described: Observer (Gamma et al., 1995), Blackboard, Forwarder-Receiver, Master-Slave, Pipes and Filters, Proxy, Publisher-Subscriber (Buschmann et al., 2013). The author proposed to extend the support of different interaction styles to cover those cases.

However, the list of interaction styles is potentially open, which prohibits directly supporting all of them at the language level. Instead, the particular interaction styles should be defined as library classes and operators by combining language-level constructs. The supporting features in the considered type system are:

- generalised functions (see Section 4.1),
- executors (see Section 4.2),
- functions that return multiple values one by one (see Section 4.4), and
- unboxing assignment (see Section 5.5).

2.12. Processing State Management

Conventional functions in programming languages and type theory are solely behavioural; they contain only code and do not hold any state. However, it proves insufficient when computation is stateful, e.g., for captures or piece-by-piece value return (cf. Section 4.4). The Command pattern generalises the concept of a function so that it is a first-class object, allows inheritance, may store execution state, supports undo, redo and logging (Gamma et al., 1995). Effectively, it is an object that can behave like a function with a stored state.

Frank Buschmann et al. noticed that implementing all this functionality in a Command may be unreasonable and described the Command Processor pattern as partially responsible for the state management (Buschmann et al., 2013). A Command Processor is an abstraction of an executor, such as a thread, a thread pool, or a debugging environment.

The considered type system uses these abstractions as basic building blocks. The corresponding type system features are generalised functions (see Section 4.1) and executors (see Section 4.2).

3. DATA COMPOSITION

The present section describes the elementary patterns of data composition identified in the present study (Batdalov and Nikiforova, 2019). They are used as the structural features of the developed type system.

Data composition is used in programming languages to build compound data types (such as arrays and classes) from simpler ones (Pierce, 2002). Similar mechanisms are applied in data transfer representation languages (such as JSON, XML and network protocols) in order to compose data objects into a more complex one. However, in spite of the common goal, the particular mechanisms used to achieve it may differ. For example, there are different ways to convert an XML document into a programming language object since the constructs used in these languages do not have a one-to-one correspondence.

Data transfer representation is a representation that is agnostic to the programming languages and paradigms used to interpret the data (Zimmermann et al., 2017). Therefore, data transfer representation languages are a good source to find common patterns, independent of particular programming languages. This section describes composition primitives that are used both in programming languages and in data transfer representation. These primitives are based on the list of basic compound types proposed by the author for comparison of programming languages features (Batdalov, 2017; Batdalov et al., 2016) and later described in the form of patterns (Batdalov and Nikiforova, 2018, 2019). The primitives can be applied recursively (i.e., an element of a collection does not have to be an atomic value, but may be another compound type), which allow building arbitrarily complex structures.

An important dichotomy to consider is the distinction between declaration and usage of the data composition primitives. It is significant because some of the discussed aspects may be valid in one or another context only. For example, the size of a Java array is fixed, but the array declaration does not contain information about its size. It differs from XML, where an XML schema may declare a sequence of elements of a fixed size. Table 3.1 illustrates what the distinction between declaration and usage means for different languages. It should be noted that data composition primitives may be used without any declaration at all. For example, dynamic programming languages do not support compile-time types, and XML or JSON data may be transferred without a predefined schema of the data.

A correspondence between composition primitives of programming language objects and data transfer representation is necessary for conversion between them. Conversion of a data object in a programming language to its data transfer representation is called marshalling, and the opposite operation is unmarshalling (Zimmermann et al., 2017). It should be noted that unmarshalling generally cannot be performed without additional information. Since data transfer representation is programming language-agnostic, it cannot contain information about the data type to which the representation should be

Table 3.1
Declaration and usage

	Declaration	Usage
Programming languages	Compile-time types	Run-time types
DTR languages	XML Schema, Swagger	XML, JSON, network protocols

converted. Choosing an appropriate data type is the responsibility of a programmer who unmarshals data.

Table 3.2 summarises the proposed data composition primitives and the corresponding constructs in programming languages and data transfer representation languages.

Table 3.2
Patterns and language constructs

Pattern	Programming languages	DTR languages
Sequence	arrays, linked lists, tuples, stacks, queues	sequence of possibly repeating values[a]
Set	hash sets, tree sets, bit masks	sequence of unique values
Multiset	hash multisets, tree multisets[b]	sequence of possibly repeating values[a]
Map	hash maps, tree maps, records	sequence of key-value pairs with unique keys
Multimap	hash multimaps, tree multimaps[b]	sequence of key-value pairs with possibly repeating keys
Variant	general variants, enumerations, optional type	declaration of alternatives in a schema

[a] Sequences and multisets share the common data transfer representation (sequence of possibly repeating values), but give it different semantics.

[b] Multisets and multimaps are rarely supported in a programming language directly. If not supported, they may be implemented by applying the Map pattern as described in Sections 3.3 and 3.5.

3.1. Sequence

A general sequence is a collection that contains an ordered series of possibly repeating values. The values may be accessed sequentially or using a numerical index. Sequences include such common data types as arrays, lists and tuples.

Context

A piece of data contains a number of values arranged in a particular order. The processing logic involves accessing these values either sequentially or by their positions (indexes) in the given order.

Problem

The general problem addressed by the pattern is supporting processing multiple values if the processing logic primarily depends on the order of these values and the order is generally independent of the values themselves (e.g., the values are not necessarily processed in the ascending or descending order). Specific problems (special cases of the abstract general statement) include, for example:

- mapping a sequence to another sequence if the corresponding elements go in the same order (e.g., squaring each number in a sequence or calculating running totals),
- searching for a value in a sequence if its position is required for subsequent computations,
- sorting a sequence of values,
- transferring a number of values so that the recipient receives them in the same order, etc.

Forces

Particular order The values are arranged in a particular order, defined by the program logic. In the general case, the order of the values is independent of the values themselves.

Iteration The values can be accessed sequentially in the defined order.

Indexing A value can be accessed by an integer-valued index that corresponds to its position in the defined order.

Index expression The index used to access a value may be either a constant or the value of a more complex expression.

Common value type The values in a sequence may be of the same data type or not.

Resizing The number of values (the size of the collection) may be either fixed or able to change at the run time.

Size restriction The program logic may assume restrictions on the minimal and/or maximal size of the collection.

Solution

Programming Languages

Create a data type that supports storing several values given in a particular order (force *Particular order*), iterating over them (force *Iteration*) and accessing by an integer-valued index (forces *Indexing* and *Index expression*). The data type may also support adding values to or removing them from the collection (force *Resizing*). The type system may allow declaration of a common type for the values (force *Common value type*).

Forces *Indexing* and *Resizing* partially conflict between each other because there is a trade-off between efficient implementation of indexing and resizing. Most often, variations of two prototypical implementations are used:

- Contiguous storage (e.g., arrays) places all values or references to them sequentially in a single chunk of memory. Indexing is a basic operation, performed in constant time by adding an offset to the memory address of the whole collection. Iteration is performed by enumerating all allowed indices. Resizing a collection requires to copy data to another chunk of memory. When extra space is left as a reserve for values that may be added later, resizing can be performed in *amortised* constant time (but actual time of particular resize operations may be higher) (Cormen et al., 2001).

- Linked storage (e.g., linked lists) places values in different parts of memory together with links to neighbouring values (next, previous or both). Iteration is a basic operation, performed by following the links between collection elements. Indexing is performed by means of iteration until the n^{th} element is reached and requires linear time. Resizing may be performed in constant or linear time depending on the available links and the position in which an element is added.

The derived operations (iteration for contiguous storage and indexing for linked storage) do not have to be supported by the data type itself, but may be implemented by the data type user instead. However, it is typical for modern solutions to implement both ways to access elements.

If collection resizing is not required, the trade-off between indexing and resizing does not exist, and contiguous storage may be used as it provides faster indexing (fixed-size arrays). However, it does not mean that immutable data types necessarily use contiguous storage because creation of an immutable collection as a modification of another collection (e.g., creation of a list that consists of a given element and all elements of another list) causes similar problems to collection resizing. On the contrary, immutable data types tend to use the linked storage because of easier data sharing between collections and natural implementation of recursive processing.

If contiguous storage is used, forces *Indexing* and *Common value type* may conflict because efficient indexing requires the collection elements to occupy the same amount of memory. If the values are not of the same type or even of the same variable-size type, this property is not guaranteed. In many cases, this conflict is solved by storing references to values instead of values themselves. However, if it is impractical (e.g., for strings in a variable-size encoding, such as UTF-8), it is impossible to calculate the offset of a particular element and indexing can only be performed by means of iteration (similarly to linked storage).

Force *Size restriction* is not typically addressed in programming languages. Theoretically, for fixed-size sequences, size restrictions might be supported in the compile time type (i.e., the compile time type of a variable dictates the minimal and/or the maximal size of a collection that may be assigned to the variable), but for variable-size sequences, such restrictions have to be supported also in the data type itself (i.e., adding or removing an element is prohibited if it violates the restriction). However, neither of these features is common in programming languages.

Data Transfer Representation Languages

Represent a collection with a sequence of values (force *Particular order*). The schema may declare a common type of the values (force *Common value type*) and/or restrictions on the number of values (force *Size restriction*).

Marshalling and Unmarshalling

Marshalling an in-memory object is trivial: iterate over the values and the sequence and place the data transfer representation of each value in the resulting sequence in the same order. Data transfer should preserve the order of values. Unmarshalling is easy for variable-size data types: start with an empty object and add all values from the data transfer representation to it. If the data type is immutable, the values cannot be 'added' in the proper sense. Instead, for each value from the data transfer representation, another object, which contains the new value and shares the preceding values with the previous object, is created. Depending on the implementation, additional actions may be needed. For example, if a singly linked list only supports adding values to the beginning of the list, the result of adding a number of values will be reversed in comparison with the original sequence. Then the resulting list should be reversed before using.

Unmarshalling to a fixed-size but mutable object (such as Java array) is slightly more tricky since it is necessary to know the amount of data to create an object *before* adding values to it. Two possible options are adding information about the number of values in the data transfer representation before the first value (it would require extension of the data transfer representation language and adding the operation of retrieving the number of values to the sequence data types) and storing the values in a temporary variable-size buffer (such as `ArrayBuilder` in Java (Ora, 2023), implementing the Builder pattern (Gamma et al., 1995)) and subsequent conversion to the target object.

Consequences

Benefits

Natural operations Sequential iterations and index-based access naturally appear in many algorithms that process ordered sequences of values. The same is true for the additional operations typically implemented for the data types implementing the

Sequence pattern.

Multiple implementations There are many specific data types, which all represent data arranged as a sequence of values but differ in the relative efficiency of operations and in supported additional operations.

Common data transfer representation The same data transfer representation can be used to represent the data from different data types.

Order preservation The order in which the values are given is preserved when the program logic requires it.

Simple marshalling and unmarshalling Marshalling and unmarshalling can be implemented by simple single-pass algorithms. In typical implementations, marshalling and unmarshalling may be performed in linear time.

Liabilities

Performance considerations Different data types that implement the same pattern may have dramatically different performance features.

Dangerous programming to an interface As a result, programming to an interface instead of an implementation (the concept promoted by Erich Gamma et al. (Gamma et al., 1995)) is dangerous. If the underlying implementation is unknown, programming to an interface may lead to unpredictable results in terms of performance.

Limited support for size restrictions Restrictions on the size of a sequence are sometimes supported in data transfer representation languages (e.g., in XML Schema (W3C, 2012)), but are not reflected in programming languages. Size restrictions are used in software modelling (e.g., they are supported in UML (OMG, 2017)), so they are probably useful for reasoning about programs. However, the type systems of most common languages do not support such restrictions neither at the compile time not at the run time. As a result, programming languages do not protect programmers from violating the restrictions.

Size in data transfer representation As mentioned in the Solution section, unmarshalling to a fixed-size mutable object would be easier if the data transfer representation contained information about the size of the sequence. However, it would mean that different data transfer representations are used in different cases and, what is even worse, the requirements for the data transfer representation depend on the used data type. That contradicts the main idea of data transfer representation. Fortunately, the solution based on the Builder pattern is free from this disadvantage. The solution with size in data transfer representation can still be used for performance reasons, but it should be considered an optimisation for efficient processing specific data types, not the general solution.

Existing Languages

The prototypical example, existing in virtually all imperative programming languages, is an array. Depending on the language, array resizing at the run time (force *Resizing*) may be supported (as in JavaScript (ECMA-262)) or not (as in Java (Gosling et al., 2023) and C++ (ISO/IEC 14882:2020)).

Other examples include various implementations of the `List` interface in Java (Ora, 2023), `Seq` trait in Scala (É, 2023), classes `vector`, `list`, `deque`, `array` and `forward_list` in C++ STL (ISO/IEC 14882:2020), etc. They typically differ in the underlying implementation (array, singly linked list, double-linked list) and additional supported operations.

Tuples in Scala represent fixed-size sequences of values (which do not share a common type) too (Odersky et al., 2023). However, they do not support using an expression to calculate an index (force *Index expression*). A value can only be accessed by giving its position directly. Essentially, Scala tuples are closer to key-indexed collections (described below as the Map pattern) with the keys assigned automatically. Unlike in Scala, in Python, tuples are another example of the Sequence pattern, which differ from arrays only in tuples' immutability (Pyt, 2023).

In JSON, a sequence is represented as an array literal (ECMA-404). In XML, it can be represented as a series of elements.

In statically typed languages (such as Java (Gosling et al., 2023), C++ (ISO/IEC 14882:2020), Scala (Odersky et al., 2023)), the type system typically support declaration of the common type of values (force *Common value type*) by means of generic types. For example, a sequence of type `List<Integer>` can only store integer values.

Imposing restrictions on the collection size (force *Size restriction*) is typically not supported in programming languages. A rare exception is Pascal, where the size of an array is strictly determined by the indexing type (ISO/IEC 7185:1990). In other languages, the compile-time type of a variable typically contains no information about the size of the collection and adding and removing elements from a collection, if supported at all, is unrestricted. However, XML Schema allows defining a range of the number of elements, thus restricting the length of a sequence (W3C, 2012). As a result, if an in-memory sequence violates the restrictions defined in a schema, it will be discovered only during actual marshalling. Programming languages do not allow creating such a data structure that would be guaranteed to agree with a particular schema.

Related Patterns

Iteration over values in a sequence can be performed using Gang-of-Four's Iterator pattern (Gamma et al., 1995) or its generalisation, the Traversable Once pattern (see Section 4.4). The Iterator pattern assumes that the data are stored in a collection, so it is appropriate for iteration over values in an in-memory object. The Traversable Once pattern describes a more general case with no assumption about the source of the values, so it covers also the case of iteration over values in data transfer representation. The Traversable Once pattern also explicitly allows for asyncronous iteration, which may be the case, for example, in parsing XML data representation.

The Atomic parameter list pattern (Zimmermann et al., 2017) is a special case of the Sequence pattern if the only allowed values in a sequence are scalar (numbers, strings, Booleans). This special case is important when a data transfer representation language has limited capabilities in building arbitrarily complex data structures.

Using a variable-size buffer to collect values, which will be then saved in a fixed-size object, from a data transfer representation is an example of application of the Builder pattern (Gamma et al., 1995).

The Map pattern, described in Section 3.4, is a generalisation of the Sequence pattern for the case of non-integer or integer non-subsequent indices.

Formalisation

A sequence is a universal and an existential type at the same time. It is universal (generic) because it can hold values of different types and existential (abstract) as it declares requirements that can be satisfied by multiple implementations. The general sequence type can be formalised in the following way:

$$\begin{aligned} \mathit{Sequence} = \forall T.\{\exists S, \{ \\ &\mathit{SeqCreate} : \mathit{TraversableOnce}\ T \to S, \\ &\mathit{SeqIterate} : S \to \mathit{TraversableOnce}\ T, \\ &\mathit{SeqIterateAssignable} : \mathit{Ref}\ S \to \\ &\ \to \mathit{TraversableOnce}\ \mathit{AssignableOnce}\ T\}\} \end{aligned}$$

T is the type parameter, the type of values in a sequence. The universal quantifier applied to T allows storing values of different types (according to the rules of F_ω, the omitted domain of T is $\star$, the kind of proper types).

S is a concrete type that implements a sequence. Examples of such types include linked lists, arrays, and vectors. The used order of quantification ($\forall T.\exists S$ as opposed to $\exists S, \forall T$) allows unique implementations depending on the value type. For example, one may implement a sequence of Booleans using a vector of bits. The reverse order ($\exists S, \forall T$) is more typical in type-theoretical literature as it facilitates using implementation types (S) which are universal themselves. However, it does not support such type parameter-specific implementations.

SeqCreate, *SeqIterate*, and *SeqIterateAssignable* are three operations common for all sequences. *SeqCreate* traverses a `TraversableOnce` (see Section 4.4) and stores the values in a sequence. *SeqIterate* performs the opposite operation, generating a `TraversableOnce` that would iterate through the sequence in the same order. Depending on the implementation, iteration may be performed by incrementing indexes (for arrays), by following links (for linked lists), or by tree traversal (for tree-based structures). *SeqIterateAssignable* is similar to *SeqIterate* but provides assignable slots (see Section 4.5) instead of values. Depending on mutability, assigning to an assignable slot may change a sequence member or perform a copy-change operation.

The typing of *SeqIterateAssignable* deserves a separate discussion. We need a reference (*Ref S*) on the left side. If it were just *Sink* or *AssignableOnce*, the target sequence might not have elements yet, so we could not iterate over them. These types guarantee that a sequence can be assigned to them, but the target might not exist before the assignment. At the same time, the right side iterates over AssignableOnce. Even if we have a reference to the sequence, it does not necessarily decomposes into references to its elements. It is usually valid for mutable sequences, but a copy-change operation on an immutable sequence needs its restructuring, thus making the assignable slot unusable.

Other operations may be reduced to the basic ones. For example, mapping can be implemented as iterating over a sequence, performing a computation, and creating a new sequence from the calculated values.

Unlike sets (see Sections 3.2 and 3.3) and maps (see Sections 3.4 and 3.5), general sequences in the present work do not support resizing operations (such as adding a value at the front). Such operations usually exist, but particular operations depend on concrete types. Therefore, the given general definition omits them.

The definition also lacks index-based operations (e.g., take the n^{th} element). Theoretically, such operations are reducible to iteration (iterate n times and return an element unless the end has been reached). From the practical perspective, such reduction does not always make sense (e.g., an array supports random access faster than iteration). However, since not all data structures support efficient index-based access, making it a basic operation in the general definition does not make sense from both theoretical and practical points of view.

3.2. Set

A general set is a collection of unique values that do not have a specific order.

Context

A piece of data contains a number of unique values, which do not have a particular order. The processing logic may involve checking whether a particular value is in a set and iterating over the values. Since the values do not have any specific order, iteration may be performed either in an arbitrary order (irrelevant from the point of view of the program logic) or in the order of the values themselves (e.g., ascending or descending).

Problem

The general problem addressed by the pattern is supporting processing multiple values if the processing logic primarily depends on the values themselves. Specific example problems include:

- checking if a value is among some given ones,
- mapping values to other values if the order is not significant,
- transferring a number of values if their order does not have to be preserved, etc.

Forces

Value uniqueness Each value may appear in a set only once.

Checking for value presence The set can be checked whether it contains a particular value or not.

Checking for expression value The value to check may be either given as a constant or calculated as the value of a more complex expression.

Iteration The values can be enumerated in an arbitrary order (i.e., a particular order is not required by the program logic) or in an order determined by the values themselves (ascending or descending).

Order independence Processing logic depends neither on the order in which values are added to a set nor on the order in which the values are enumerated (unless enumeration in the ascending or descending order of values is required).

Value type The possible data types of values depend on the language and the used implementation. The values may be of the same data type or not.

Solution

Programming Languages

Create a data type that supports storing unique values and checking if a value is in the set (forces *Value uniqueness*, *Checking for value presence* and *Checking for expression value*) and iterating over values (force *Iteration*). If a set is implemented with a hash table, the order of iteration is undefined, but search trees allow iteration in the ascending or descending order.

Hash tables allow storing values of any data type on which a hash function is defined, and search trees require the data type to be ordered (force *Value type*). Both implementations work correctly only if the objects in a set are immutable. This problem is further discussed in the Solution section of the Map pattern (Section 3.4). The type system of a language may allow declaration of a particular data type of stored values.

Forces *Iteration* and *Order independence* partially conflict because it is difficult to ensure truly order-independent processing when iteration is performed in an unpredictable order. For example, it is natural to consider two sets equal if they contain the same elements. However, it is not enough to iterate over the elements and check them for equality (as we would do with sequences) since the same order of iteration of two sets is not guaranteed. Therefore, such operations as set equality should be carefully implemented to ensure order independence.

Data Transfer Representation Languages

Represent a collection with a sequence of values, in which each value occurs only once (force *Value uniqueness*). The schema may declare a common type of the values (force *Value type*).

A possible alternative could be to allow repetitions in the data transfer representations and simply ignore the second and further occurrences of the same value. However, this solution silently ignores possible mistakes in data and changes the semantics of the given sequence. Therefore, it is rather an antipattern.

Marshalling and Unmarshalling

Marshalling and unmarshalling are essentially the same as for the Sequence pattern. Marshalling requires iteration over the values and placing them in the resulting representation, and unmarshalling assumes parsing the data transfer representation and adding them to an empty set. Since the order of values is insignificant, data transfer does not require order preservation. For example, it is possible to put different chunks of data in different packets, send them in parallel and collect on the receiving side in an arbitrary order. However, in order to do it, it is necessary to distinguish whether a particular data transfer representation represents a Sequence or a Set. The data transfer representation itself does not contain such information, so some metainformation is needed.

Consequences

Benefits

Checking for uniqueness Typical implementations assure that each value is included in a set only once. However, they do not usually prohibit adding a value to a set the second time, this operation just has no effect.

Efficient checking for a value Checking whether a value is in a set can be performed much faster than with a sequence (depending on the implementation, in logarithmic or amortised constant time).

Iteration in the order of values Search trees allow iteration over the values in the order of values themselves (ascending or descending).

Simple marshalling and unmarshalling Marshalling and unmarshalling can be implemented by simple single-pass algorithms. Depending on the implementation, marshalling and unmarshalling can be performed in linear or linearithmic time.

Liabilities

Different requirements to values Different solution decisions with respect to the allowed values in different programming languages hinder data transfer between them. Data transfer representation can no longer be programming language-agnostic.

Indistinguishability from a sequence The data transfer representation of a set is generally indistinguishable from the one of a sequence. Presence of value repetitions hints that it cannot be a set, but if all values are unique, the data can represent either a sequence or a set. Additional context should be communicated to interpret the data correctly.

Existing Languages

Standard libraries of many languages contain a general interface for sets (e.g., `Set` in Java (Ora, 2023) and Scala (É, 2023), `ISet` in .NET Framework (ECMA-335)). Typical implementations include sets based on hash tables (e.g., `HashSet` in Java (Ora, 2023), Scala (É, 2023) and .NET Framework (ECMA-335), `unordered_set` in C++ (ISO/IEC 14882:2020)) and search trees (e.g., `TreeSet` in Java (Ora, 2023) and Scala (É, 2023), `SortedSet` in .NET Framework (ECMA-335), `set` in C++ (ISO/IEC 14882:2020)).

The languages in which a map is a basic primitive, such as JavaScript (ECMA-262) and Perl (Per, 2023), usually do not have a separate type for sets. Instead, sets are usually implemented by using maps (see the Map pattern in Section 3.4) and storing a trivial value (such as 1) for all keys. However, this solution hinders marshalling and unmarshalling since the data transfer representation of sets and maps with trivial values is different.

Data transfer representation languages, such as JSON and XML, do not have a separate primitive for sets. Instead, the same representation as for sequences is used. Interpreting data as a sequence or a set is the receiver's responsibility.

In statically typed languages, value type (force *Value type*) can be declared using generic types (e.g., `Set<Integer>`).

Related Patterns

Similarly to the Sequence pattern, iteration over values in a set can be performed using the Iterator pattern (Gamma et al., 1995) and/or the Traversable Once pattern (see Section 4.4).

The Multiset pattern, described in Section 3.3, is a generalisation of the Set pattern, where a single value may be included in a set more than once. The Map pattern, described in Section 3.4, is another generalisation of the Set pattern, in which a value in a set is considered an access key and additional data are assigned to each key.

Formalisation

The general set type can be formalised in the following way:

$$
\begin{aligned}
Set = \forall T.\{\exists S, \{ \\
&SetCreate : TraversableOnce\ T \to S, \\
&SetIterate : S \to TraversableOnce\ T, \\
&SetAdd : Ref\ S \to T \to Ref\ S, \\
&SetRemove : Ref\ S \to T \to Ref\ S\}\}
\end{aligned}
$$

T is the type parameter, the type of values in a set. S is a concrete type that implements a set.

The signatures of *SetCreate* and *SetIterate* are the same as for *SeqCreate* and *SeqIterate* correspondingly (see Section 3.1). However, their semantics is different: sequences are expected to keep the order of elements, whereas sets are not. Therefore, the order of iteration over a set depends on the implementation and may differ from the order of elements passed to *SetCreate*. This distinction is not encodable in types, so the signatures look the same. That is why prefixes (*Seq* and *Set*) are necessary (see Section 1.3).

SetAdd adds an element to a set, and *SetRemove* removes one. For immutable sets, they are copy-change operations according to their signatures. For mutable sets, these operations mutate a set in place (mutability is interpreted as assigning the value of a function to the source variable, see Section 2.2).

The given definition does not contain an operation to check whether an element is in a set. The reasons are the same as in the case of the take-n$^{\text{th}}$-element operation in a sequence (see Section 3.1): reducibility to iteration and the possibility of sets that do not have a more efficient implementation. Actually, sets without an efficient membership check are much rarer than sequences without random access but still possible.

Unlike sequences (see Section 3.1), sets do not associate assignable slots with their elements. In most cases, changing a value in a set will cause reshaping of the set, so it does not make sense to differentiate it from removing the old value and adding the new one.

3.3. Multiset

A general multiset is an unordered collection of values that may repeat. Multisets are an extension of the Set pattern, described in Section 3.2, and the descriptions of the two patterns have a lot in common. Differences from the Set pattern are *italicised.*

Context

A piece of data contains a number of values, which do not have a particular order. *The values may repeat.* The processing logic may involve checking whether a particular value is in a set and iterating over the values. Since the values do not have any specific order, iteration may be performed either in an arbitrary order (irrelevant from the point of view of the program logic) or in the order of the values themselves (e.g., ascending or descending).

Problem

The general problem addressed by the pattern is supporting processing multiple *possibly repeating* values if the processing logic primarily depends on the values themselves. *A sample specific problem is storing and transferring a set of available resources, in which resource identifiers may repeat, all resources with the same identifier are interchangeable, and acquiring or releasing a resource is reflected by removing an identifier from or adding it to the set.*

Forces

Value repetition *Each value may appear in a multiset multiple times.*

Checking for multiplicity The multiset can be checked whether it contains a particular value and, if yes, *how many times the value occurs.*

Checking for expression value The value to check may be either given as a constant or calculated as the value of a more complex expression.

Iteration The values can be enumerated in an arbitrary order (i.e., a particular order is not required by the program logic) or in an order determined by the values themselves (ascending or descending).

Order independence Processing logic depends neither on the order in which values are added to a set nor on the order in which the values are enumerated (unless enumeration in the ascending or descending order of values is required).

Value type The possible data types of values depend on the language and the used implementation. The values may be of the same data type or not.

Solution

Programming Languages

Apply the Map pattern (Section 3.4) and use the values of a multiset as keys and their multiplicities as values.

Forces *Iteration* and *Order independence* partially conflict because it is difficult to ensure truly order-independent processing when iteration is performed in an unpredictable order. For example, it is natural to consider two sets equal if they contain the same elements *and their multiplicities coincide.* However, it is not enough to iterate over the elements and check them for equality (as we would do with sequences) since the same order of iteration of two sets is not guaranteed. Therefore, such operations as set equality should be carefully implemented to ensure order independence.

Data Transfer Representation Languages

Represent a collection with a sequence of values without requiring them to be unique. The order of elements in the sequence is semantically insignificant.

Marshalling and Unmarshalling

Marshalling requires iteration over the values and placing them in the resulting representation, and unmarshalling assumes parsing the data transfer representation and adding them to an empty multiset. Since the order of values is insignificant, data transfer does not require order preservation. For example, it is possible to put different chunks of data in different packets, send them in parallel and collect on the receiving side in an arbitrary order. However, in order to do it, it is necessary to distinguish whether a particular data transfer representation represents a Sequence or a Multiset. The data transfer representation itself does not contain such information, so some metainformation is needed.

Consequences

Benefits

Efficient checking for a value Checking whether a value occurs in a multiset *and, if yes, how many times* can be performed much faster than with a sequence (depending on the implementation, in logarithmic or amortised constant time).

Iteration in the order of values Search trees allow iteration over the values in the order of values themselves (ascending or descending).

Simple marshalling and unmarshalling Marshalling and unmarshalling can be implemented by simple single-pass algorithms. Depending on the implementation, marshalling and unmarshalling can be performed in linear or linearithmic time.

Liabilities

Different requirements to values Different solution decisions with respect to the allowed values in different programming languages hinder data transfer between them. Data transfer representation can no longer be programming language-agnostic.

Indistinguishability from a sequence The data transfer representation of a set is generally indistinguishable from the one of a sequence. Presence of value repetitions hints that it cannot be a set, but if all values are unique, the data can represent either a sequence or a set. Additional context should be communicated to interpret the data correctly.

Existing Languages

Multisets are not usually supported in programming languages directly since they can be easily implemented with maps. An exception is C++ STL, which contains classes `multiset` *and* `unordered_multiset` *(ISO/IEC 14882:2020).*

In data transfer representation languages, the same representation as for sequences and sets can be used for multisets. Therefore, a separate primitive is not required, but additional context to interpret the data should be shared between the sender and the receiver.

Related Patterns

The Iterator pattern (Gamma et al., 1995) may be applied to iterate either over the values (each value may appear in the iteration multiple times) or over the value-multiplicity pairs (as if we simply iterated over the underlying map). However, traversing the corresponding data transfer representation can only give the first variant.

The Map pattern, described in Section 3.4, is a natural way to implement a multiset as described in the Solution section.

The Set pattern, described in Section 3.2, is a special case of a multiset, when each value can only appear once. The Multimap pattern, described in Section 3.5, is a generalisation of a multiset, in which a value in a set is considered an access key and additional data are assigned to each appearance of such a key.

Formalisation

The formal definition of a multiset is the same as for a set (see Section 3.2) except for different prefixes:

$$
\begin{aligned}
MultiSet = \forall T.\{\exists S, \{ \\
& MSetCreate : TraversableOnce\ T \rightarrow S, \\
& MSetIterate : S \rightarrow TraversableOnce\ T, \\
& MSetAdd : Ref\ S \rightarrow T \rightarrow Ref\ S, \\
& MSetRemove : Ref\ S \rightarrow T \rightarrow Ref\ S\}\}
\end{aligned}
$$

The difference is only semantical: the methods of a set assume and enforce values uniqueness, but a multiset may hold the same value multiple times.

As mentioned in the Solution section, implementations of a multiset are different from the ones of sets and are usually based on maps. However, the external interface hides the implementation details and makes multisets look like sets.

3.4. Map

A general map is a correspondence between keys and values, in which each key may occur only once. It can be represented as a set of key-value pairs where keys do not repeat. The keys may be predefined (as in records and classes) or not (as in associative arrays). There are no requirements for the values: they do not have to be unique, and efficient access to keys by values is not required.

Context

A piece of data contains a number of values, which do not have a particular order, but instead have access keys.

Problem

The general problem addressed by the pattern is supporting processing multiple key-value pairs with unique keys if the processing logic primarily depends on the keys. Specific example problems include:

- storing and transferring a dictionary that is used to decode keys used for internal representation of a value to labels visible to users (so that the labels can be changed easily without updating the internal representation),

- storing and transferring a complex object with multiple attributes (e.g., the login, the password, the first name and the last name of a user),
- storing a transferring a set of complex objects if one attribute (e.g., the login) serves as a key that represents the object during computation and the other attributes are accessed when necessary, etc.

Forces

Keys Each value has a corresponding key, which is unique within a particular collection.

Access by key A value can be accessed by its key.

Index expression The key used to access a value may be either given as a constant or calculated as the value of a more complex expression.

Iteration The values can be enumerated in an arbitrary order (i.e., a particular order is not required by the program logic) or in an order determined by the keys (but usually not in an order determined by the values).

Order independence Processing logic depends neither on the order in which keys and values are added to a map nor on the order in which the keys are enumerated (unless enumeration in the ascending or descending order of keys is required).

Key type The set of values that may serve as keys (the data type of keys) depends on the language and the used implementation. It may also be restricted on the declaration site.

Variable set of keys The set of keys may be either fixed or able to change at the run time.

Required keys The declaration site may declare some keys that must present in a map.

Extra keys The declaration site may either allow or prohibit presence of other keys in addition to the required keys.

Value type The values in a map may be of the same data type or not.

Different value types depending on keys If the values do not share the same data type, the declaration site may assign specific data types to the particular keys.

Solution

Programming Languages

Create a data type that supports storing and accessing a value using a key (forces *Keys*, *Access by key* and *Index expression*) and iterating over keys or key-value pairs (force *Iteration*).

The choice of the data type implementation is dictated by the trade-offs between forces *Variable set of keys*, *Access by key*, *Iteration*, *Key type* and *Different value types depending on keys*. Usually, variations of three prototypical implementations are used:

- Fixed-position contiguous storage (records) associates a fixed offset with each key and stores the corresponding value at this offset from the collection address. It is possible only if the set of keys is fixed and the value type associated with each key requires a fixed amount of memory (or a fixed-size reference to an actual variable-size value can be stored). This implementation provides guaranteed constant time of value access. Usually, only valid identifiers can be used as keys, however, it is rather a tradition and is not required by the implementation itself.

- Hash table storage is possible with keys of any type on which a hash function is defined and allows changing the set of keys at the run time. This implementation provides amortised constant data access and modification time.

- Search tree storage is possible with keys of any ordered type. This implementation provides logarithmic data access and modification time, but it is the only option available if iteration in the ascending or descending order of keys is required.

Hash table and search tree implementations work correctly only if keys are immutable. Since most languages do not have language-level guarantees that a particular data type is immutable, this requirement is difficult to enforce. Some languages (e.g., JavaScript (ECMA-262)) restrict the allowed keys to well-known immutable data types, such as numbers and strings. Other languages, such as Java (Ora, 2023), allow arbitrary objects, and their immutability is programmers' responsibility.

If hash table storage is used, forces *Iteration* and *Order independence* partially conflict similarly to the corresponding forces for the Set pattern. Logic that implies iteration over keys should take into account that the order is iteration is unpredictable. With other implementations, the order of iteration is typically predefined and can be relied.

The type system of a language may allow declaration of keys and values data types (forces *Key type* and *Value type*), value data types for particular keys (force *Different value types depending on keys*), the set of required keys (force *Required keys*) and whether other keys are allowed in addition to the required ones (force *Extra keys*).

Data Transfer Representation Languages

Represent a collection with a sequence of key-value pairs, in which each key occurs only once (force *Keys*). A possible modification of these scheme is to represent data with a sequence of values only if the rules how to find a key for a value are given in addition. For example, such rules may define the attribute of a complex object which serves as the key or that the key is the XML tag enclosing the value. The schema may declare the same restrictions that are valid on the declaration site for programming languages, as described in the previous paragraph (forces *Key type*, *Required keys*, *Extra keys*, *Value type* and *Different value types depending on keys*).

Marshalling and Unmarshalling

Marshalling requires iteration over the keys of a map, generating key-value pairs in the required syntax and storing them in the resulting representation. Unmarshalling assumes beginning with an empty (for associative arrays) or uninitialised (for records) map and assigning values to keys according to the data transfer representation. If the modified scheme, described in the previous paragraph is used, unmarshalling requires calculating keys from values. Similarly to the Set pattern, data transfer does not have to preserve the order of elements.

Consequences

Benefits

Wide range of keys In comparison with the Sequence pattern, the Map pattern is not restricted by a continuous range of integer-valued keys, but allows using string and other keys as well.

Semantic keys String keys may be used as memoisable identifiers, understandable for a programmer.

Efficient access Despite the wide range of allowed keys, typical implementations of the Map pattern ensure efficient access to the values associated with keys (constant or logarithmic time).

Iteration in the order of keys Search trees allow iteration in the order of map keys (ascending or descending).

Simple marshalling and unmarshalling Marshalling and unmarshalling can be implemented by simple single-pass algorithms. Depending on the implementation, marshalling and unmarshalling can be performed in linear or linearithmic time.

Liabilities

Different sets of allowed keys Decisions taken in different programming languages define different sets of allowed keys. As a resul, data transfer is difficult and not really programming language-agnostic.

Absence of equivalent in XML As mentioned below, XML does not have a standard way to represent maps unless all values are scalar. It hinders marshalling and unmarshalling and requires an additional context to be shared between the sender and the recipient.

Existing Languages

Records, which are maps with a fixed set of keys and sequential arrangement of values in memory, are widely used in procedural programming languages, such as C (ISO/IEC

9899:2018) and Pascal (ISO/IEC 7185:1990). In statically typed object-oriented languages, such as C++ (ISO/IEC 14882:2020), Java (Gosling et al., 2023) and C# (ECMA-334), it is common to use records as the basis for classes. Records typically do not support using expression to calculate a key (force *Access by key*) and iteration over keys or key-value pairs (force *Iteration*) except for reflection facilities.

Dynamically typed object-oriented languages, such as JavaScript (ECMA-262) and Python (Pyt, 2023), tend to represent classes with hash tables instead. In statically typed languages, hash tables usually supported by standard libraries (e.g., `HashMap` in Java (Ora, 2023) and Scala (É, 2023), `Dictionary` in .NET Framework (ECMA-335), `unordered_map` in C++ (ISO/IEC 14882:2020)).

Similarly, search trees are typically supported in standard libraries (e.g., `TreeMap` in Java (Ora, 2023) and Scala (É, 2023), `SortedDictionary` in .NET Framework (ECMA-335), `map` in C++ (ISO/IEC 14882:2020)).

Standard libraries often contain abstract interfaces for maps independently of their implementation (e.g., `Map` in Java (Ora, 2023) and Scala (É, 2023), `IDictionary` in .NET Framework (ECMA-335)). However, records (classes) in these languages do not implement these interfaces.

Tuples in Scala, as mentioned above, are close to records, but the keys are assigned automatically on the basis of the position of a value in a sequence (`_1`, `_2`, etc.).

In JSON, a map is represented as an object literal (ECMA-404). In XML, the set of attributes of a tag represent the same pattern. However, attributes may only have values of primitive types (W3C, 2006), and XML does not have a standard way to represent a map with complex values. Such solutions as using XML tag name or the value of a specified attribute as a key are possible, but require extra information about how to interpret the given data. Furthermore, only standard identifiers may be used as tag and attribute names in XML (W3C, 2006), which hinders mapping XML data to other languages, in which arbitrary strings and even other types may be used as keys.

In statically typed languages, the type system usually allow declaration of data types of keys and values. For maps with fixed keys (records), required keys (force *Required keys*) and the value type for each key (force *Different value types depending on keys*) are declared in the type definition. For maps with changing keys (hash tables and search trees), data types of keys and values (forces *Key type* and *Value type*) can be declared using generic types (e.g., `Map<String, User>`), but required keys and different value types depending on a key cannot. However, TypeScript supports defining required keys and value types for each key for hash tables (which are the basis for objects in JavaScript and TypeScript) (Mic, 2023).

A class definition typically establishes only required keys (force *Required keys*), but does not state that a particular object does not have other keys since extra keys may be introduced by inheritance. However, if inheritance is prohibited (e.g., for `final` classes in Java (Gosling et al., 2023)), it is guaranteed that only declared keys exist (force *Extra keys*). TypeScript does not support prohibiting additional keys either by inheritance or by direct adding of a key to an object. However, TypeScript has a restriction that an object literal cannot have non-declared keys (Mic, 2023).

Related Patterns

The Iterator pattern (Gamma et al., 1995) and its generalisation, the Traversable Once pattern (see Section 4.4) may be used to iterate over keys, values or key-value pairs in a data object or in its data transfer representation.

The Parameter tree pattern and the Parameter forest pattern (Zimmermann et al., 2017) describe complex structures built from maps and sequences.

The Sequence pattern, described in Section 3.1, is a special case of the Map pattern for the case of a continuous range of integer-valued access keys. However, implementations of the Sequence pattern are typically different for efficiency reasons. The Set pattern, described in Section 3.2, is another special case of the Map pattern, in which values are not used (all values have type `Unit` or `void`). The Map pattern also uses the Set pattern since the keys of a map form a set. The Multimap pattern, described in Section 3.5, is a generalisation of the Map pattern, where a single key may be used more than once.

Formalisation

The general map type can be formalised in the following way:

$$\begin{aligned}
Map = \forall K.\forall V.\{\exists M, \{ \quad & \\
& MapCreate : TraversableOnce\ (Pair\ K\ V) \to M, \\
& MapIterate : M \to TraversableOnce\ (Pair\ K\ V), \\
& MapIterateAssignable : Ref\ M \to \\
& \to TraversableOnce\ (Pair\ K\ (AssignableOnce\ V)), \\
& MapAdd : Ref\ M \to K \to V \to Ref\ M, \\
& MapRemove : Ref\ M \to K \to Ref\ M\}\}
\end{aligned}$$

K and V are the type parameters, the type of keys and values in a map correspondingly. S is a concrete type that implements a map.

$MapCreate$ creates a map from key-value pairs. $MapIterate$ performs the opposite operation. $MapIterateAssignable$ provides an assignable slot for a value but not for a key because changing a key effectively removes an old entry and adds a new one.

$MapAdd$ adds a key-value pair to a map, and $MapRemove$ removes a key together with the associated value. Similarly to sets (see Section 3.2), they are copy-change operations for immutable maps and mutating operations for mutable maps (mutability is interpreted as assigning the value of a function to the source variable, see Section 2.2).

The given definition does not contain the access by key operation. Like the take-n$^{\text{th}}$-element operation in a sequence (see Section 3.1) and the membership check operation in a set (see Section 3.2), this operation is reducible to iteration. Map implementations that do not have more efficient access by key than iteration are rare but exist.

3.5. Multimap

A general multimap is a correspondence between keys and values, in which keys may repeat. It is an extension of the Map pattern (Section 3.4). Differences from the Map pattern are *italicised.*

Context

A piece of data contains a number of values, which do not have a particular order, but instead have access keys. *The keys may repeat within a collection.*

Problem

The general problem addressed by the pattern is supporting processing multiple key-value pairs with *possibly repeating* keys if the processing logic primarily depends on the keys. *A sample specific problem is storing and transferring a set of objects that can be grouped with respect to one attribute (e.g., the code of a department for employees).*

Forces

Keys Each value has a corresponding key, which *may repeat* within a particular collection.

Access by key A value can be accessed by its key.

Index expression The key used to access a value may be either given as a constant or calculated as the value of a more complex expression.

Iteration The values can be enumerated in an arbitrary order (i.e., a particular order is not required by the program logic) or in an order determined by the keys (but usually not in an order determined by the values).

Order independence Processing logic depends neither on the order in which keys and values are added to a map nor on the order in which the keys are enumerated (unless enumeration in the ascending or descending order of keys is required).

Key type The set of values that may serve as keys (the data type of keys) depends on the language and the used implementation. It may also be restricted on the declaration site.

Variable set of keys The set of keys may change at the run time.

Value type The values in a map may be of the same data type or not.

Forces Required keys, Extra keys *and* Different value types depending on keys, *described for the Map pattern, as well as collections with fixed sets of keys, are not addressed by the known uses of the Multimap pattern.*

Solution

Programming Languages

Apply the Map pattern, but store a sequence or a set of values (possibly empty if necessary) instead of single values.

Data Transfer Representation Languages

Use a sequence of key-value pairs or a sequence of values with a known way to calculate keys (as in the Map pattern), but do not require uniqueness of keys. An alternative representation is applying the same approach as in the programming languages: use a sequence of key-value pairs, in which each value is a sequence of final values.

Marshalling and Unmarshalling

Marshalling requires iteration over the keys of a map, generating key-value pairs in the required syntax and storing them in the resulting representation. Unmarshalling assumes beginning with an empty (for associative arrays) or uninitialised (for records) map and assigning values to keys according to the data transfer representation. If the modified scheme, described in the previous paragraph is used, unmarshalling requires calculating keys from values. Similarly to the Set pattern, data transfer does not have to preserve the order of elements.

Consequences

Benefits

Wide range of keys In comparison with the Sequence pattern, the Map pattern is not restricted by a continuous range of integer-valued keys, but allows using string and other keys as well.

Semantic keys String keys may be used as memoisable identifiers, understandable for a programmer.

Efficient access Despite the wide range of allowed keys, typical implementations of the Map pattern ensure efficient access to the values associated with keys (constant or logarithmic time).

Iteration in the order of keys Search trees allow iteration in the order of map keys (ascending or descending).

Simple marshalling and unmarshalling Marshalling and unmarshalling can be implemented by simple single-pass algorithms. Depending on the implementation, marshalling and unmarshalling can be performed in linear or linearithmic time.

Liabilities

Different sets of allowed keys Decisions taken in different programming languages define different sets of allowed keys. As a resul, data transfer is difficult and not really programming language-agnostic.

Absence of equivalent in XML As mentioned below, XML does not have a standard way to represent maps unless all values are scalar. It hinders marshalling and unmarshalling and requires an additional context to be shared between the sender and the recipient.

Existing Languages

Programming languages typically do not support multimaps directly. They always can be implemented with a map, whose values are sequences of values in which we are interested. However, C++ STL does contain separate classes `multimap` *and* `unordered_multimap` *(ISO/IEC 14882:2020).*

JSON (ECMA-404) and XML (W3C, 2006) do not support duplicate keys in the object literal and attribute lists correspondingly. In JSON, the primary way to represent multimaps is using key-sequence pairs. In XML, key-sequence pairs are not representable with attributes (since attributes cannot have non-primitive values). Therefore, the representation with a sequence of values and computable keys should be used (as described in the Map pattern).

The representation with a sequence of key-value pairs and possible keys repetition is used in HTTP headers (Thomson and Benfield, 2022). Each HTTP header is a key-value pair, and multiple occurences of the same header name mean that all corresponding header values are associated with the header name.

Related Patterns

The Iterator pattern (Gamma et al., 1995) may be applied to iterate over the keys, the values or the key-value pairs in one of two orders: either each element appears as many times as the corresponding key is used or each key appears only once and the corresponding value is a collection of multimap values. Traversing the corresponding data transfer representation can only give the first variant.

The Map pattern, described in Section 3.4, is a special case of the Multimap pattern, when each key is used only once. The Multiset pattern, described in Section 3.3, is another special case of the Multimap pattern, in which values are not used (all values have type `Unit` *or* `void`*).*

Formalisation

The multimap type can be formalised in the following way:

$$
\begin{aligned}
MultiMap = \forall K.\forall V.\{\exists M, \{ \\
& MMapCreate : TraversableOnce\ (Pair\ K\ V) \to M, \\
& MMapIterateElements : M \to \\
& \to TraversableOnce\ (Pair\ K\ V), \\
& MMapIterateCollections : M \to \\
& \to TraversableOnce\ (Pair\ K\ (TraversableOnce\ V)), \\
& MMapIterateAssignable : Ref\ M \to TraversableOnce(\\
& Pair\ K\ (TraversableOnce\ (AssignableOnce\ V))), \\
& MMapAdd : Ref\ M \to K \to (TraversableOnce\ V) \to \\
& \to Ref\ M, \\
& MMapRemove : Ref\ M \to K \to Ref\ M\}\}
\end{aligned}
$$

As discussed in the Related patterns section, iteration may generate either individual values (MMapIterateElements) or collections (MMapIterateCollections). However, modifying operations (MMapIterateAssignable, MMapAdd, and MMapRemove) are defined only at the collection level. The reason is that modifying operations at the element level involve modifying collections in which these elements are stored and thus depend on the used collection.

3.6. Variant Type

A variant is a choice between two or more alternatives (e.g., a variable that can hold either an integer or a string value). Generally, variants make sense at the declaration site only since at the usage site the concrete alternative is already chosen. Important special cases of variants are options and enumerations.

Context

The set of values that may be assigned to an identifier in a program or data representation is the union of two or more already defined data types.

Problem

Support processing of a piece of data that can hold data of different types, and the processing logic depends on the type. Specific example problems include:

- storing in the same variable data of different types coming from an external system, which are further processed depending on their type,
- writing a polymorphic function that can process data of different types if the particular data type is known only at the run time,
- returning either the result of a successful action or an error condition if an error happened, etc.

Forces

Declaration site Typically, variants are required on the declaration site (the compile time type or data representation schema) because the usage site (the run time type or actual data transfer representation) contains a particular alternative, not a choice between alternatives.

Common operations If a variable has a variant type, only operations common for the alternatives may be called.

Unrelated implementations Implementations of the common operations may be completely unrelated.

Operations that are not common The alternative data types may support operations that are not common. They are unsafe to be called directly, but may be called if the data type is checked at the run time.

Solution

Programming Languages

Define a compile-time type of an identifier to be the union of previously defined types (force *Declaration site*). Ensure that only common operations of these data types are called directly at the compile time (force *Common operations*).

Concerning forces *Unrelated implementations* and *Operations that are not common*, at least two options are known: labelled (or tagged) and unlabelled (untagged) variants. An object of a labelled variant comprises a label indicating the run-time type of the object and an object of this run-time type (essentially, the mechanism used in object-oriented languages for virtual methods). An unlabelled variant is simply a union of underlying types as sets.

The difference between labelled and unlabelled variants is not only an implementation detail:

- Labelled variants can hold arbitrary types. Unlabelled variants are applicable only if the run-time type information is already available or the types in the union share binary representation. Otherwise, it is impossible to determine how to interpret the binary string in memory.

- If the underlying types intersect, a labelled variant may contain multiple values with the same underlying value. For example, if $A \cap B = C$ and $c \in C$, the variant type values (A, c) and (B, c) differ. In an unlabelled variant, the same underlying value always gives the same variant type value.

- A dedicated type label provides a natural way to write an algorithm that depends on the run-time type: creating a code branch for each possible label value. For unlabelled variants, such an algorithm can use the general means of run-time type checking. Both these techniques achieve the desired goal, but they are behaviourally different in which branch handles a value of the underlying types intersection. With a labelled variant, the type label determines the chosen branch (again, the same underlying value labelled with different types makes different values of the labelled variant type). Without the type label, the *first* branch covering the run-time value will be executed.

Data Transfer Representation Languages

Define allowed alternatives in the schema of data (force *Declaration site*).

Consequences

Benefits

Natural logic The variant data type naturally reflects the idea of union of sets, which is a basic thinking construct.

Avoiding excessively wide types The variant type allows avoiding a common antipattern of using a very wide data type, most values of which are not appropriate for the program logic, simply because a type that contains the appropriate values only cannot be defined. For example, if a variable can have one of several predefined values, it is better to use an enumeration, a special case of a variant, than a string.

Liabilities

Checking the run time type It is necessary to check the run time type of an object for every operation (either in the run time environment to choose the implementation of the common operations or in the program code to check applicability of the operations that are not commons).

Unsafe operations There is a risk that operations on the data type will be called in the unsafe manner if the type system does not contain sufficient checks (as in the case of the `null` value in Java).

Existing Languages

General variants are not common in programming languages. However, C++ Standard Template Library contains template `std::variant` (ISO/IEC 14882:2020), and Scala standard library contains class `Either` (É, 2023) that serve these purposes. TypeScript supports type operator `|` that allows defining general variant compile-time types too (Mic, 2023). C++ and Scala's variant types are labelled variants, whereas the TypeScript version is unlabelled. This distinction corresponds to the general preference of the former languages towards nominal types and TypeScript's preference towards structural types.

Special cases of variant types are supported more often. Optional type, which is either a value or its absence, is supported in many languages either directly (as `Option` in Scala (É, 2023)) or by allowing objects to have the `null` value.

In languages with exceptions (C++ (ISO/IEC 14882:2020), Java (Gosling et al., 2023), Scala (Odersky et al., 2023) etc.), a function implicitly returns a variant type: either a value of the declared return type or an exception. In Scala, this variant type exists also explicitly and is called `Try` (É, 2023).

Enumerations are unions of trivial types, each of which only contains one value (Pierce, 2002). Enumerations are supported at the language level in C++ (ISO/IEC 14882:2020), Java (Gosling et al., 2023), Scala 3 (LAM, 2023) (earlier only in the standard library), etc.

XML Schema allows declaration of alternatives that may be placed in a specific place of a document (W3C, 2012).

Related Patterns

The Optional value pattern (see Section 3.7) and the corresponding Maybe monad (Wadler, 1992a) are special cases of the Variant pattern when the choice is between a normal type and the special null type.

Formalisation

The labelled variant type can be formalised in the following way:

$$\begin{aligned}
LabelledVariant = \forall T_1, \dots, T_n . \{\exists S, \{\forall R . \{ \\
& LVarCreate_1 : T_1 \to S, \\
& \dots \\
& LVarCreate_n : T_n \to S, \\
& LVarApply : S \to (T_1 \to R) \to \\
& \dots \to (T_n \to R) \to R, \\
& LVarMutate : Ref\ S \to \\
& \to (AssignableOnce\ T_1 \to R) \to \\
& \dots \to (AssignableOnce\ T_n \to R) \to R\}\}\}
\end{aligned}$$

LVarCreate operations create an object of the variant type from an object of an underlying type. *LVarApply* applies one of the provided functions to the underlying value depending on its type. *LVarMutate* does the same with mutating functions.

The unlabelled variant type is simply the structural union of the underlying types without creating a new nominal type (and thus without separate functions):

$$UnlabelledVariant = T_1 \cup \dots \cup T_n$$

3.7. Optional Value

Context

An object may not be assigned with a particular value.

Problem

Support processing logic that can handle the case of the absence of the used data.

Forces

Normal usage is unsafe The declared variable type tempts to use the variable according to this type. If the variable may not hold a value, protection against using the variable without checking this condition is required.

Temporary absence of value Absence of a value may be a temporary condition, for example, when an object has not been fully initialised yet.

Error condition Absence of a value is also used to indicate an error condition. Although it is considered a bad style in programming languages that support exceptions (Martin, 2008), this use of optional values still exists.

Solution

Extend the set of values that a variable may hold with a special value (e.g., `null`, but it may be denoted differently in different languages) that indicates that the real value is absent. In a statically typed language, it is done by means of an algebraic type, a sum of the main type of the object and a type whose only value is `null`) (Pierce, 2002). In a dynamically typed language, existence of the `null` value is sufficient by itself since any variable may hold any supported value, including `null`.

If absence of a value is temporary during the object initialisation phase, a special type is not needed, but the compiler is responsible for checking that a particular value is assigned during initialisation.

Consequences

Benefits

Explicit absent value The pattern allows a programmer to explicitly mark a particular value as absent and inappropriate for typical usages.

Type safety The type system of the programming language may ensure correct processing of absent values (even though not all programming languages actually do so).

Liabilities

Constant checking a value for presence The opportunity of having an absent value requires to routinely analyse possible cases on every usage. A failure to do this (especially when the optional type is not controlled at the compile time) leads to harsh error conditions.

Unclear boundaries of initialisation The initialisation solution described above, which eliminates the necessity to declare a class member with an optional type, only works in the case of initialisation performed by class constructors. If a class lifecycle contains an initialisation phase implemented by a separate method, and a member receives its value only during this phase, the type of this method still should allow assigning an absent value.

Variations in semantics Inconsistent semantics of the optional type in different systems (see examples below) hinders interaction between them.

Existing Languages

In most statically typed object-oriented programming languages (e.g., in Java (Gosling et al., 2023), C# (ECMA-334), and TypeScript (Mic, 2023)) object and array variables can always have a special `null` value, which make their values optional. However, the type systems of the mentioned languages do not prevent using `null` as a normal object (`null` dereference), which causes errors at the run time (this decision is known as the 'billion-dollar mistake' (Hoare, 2009)). C++ does not have this problem for object types, but allows assigning the null value to variables of pointer types (ISO/IEC 14882:2020). Therefore, pointer types in C++ suffer from the same problem of `null` dereference. In fact, pointer types in C++ are inherently unsafe from the point of view of the type system (Pierce, 2002), and `null` dereference is only one of the related problems.

For primitive types, Java and C# have object counterparts (e.g., `Integer` in Java (Gosling et al., 2023) and `int?` in C# (ECMA-334) for `int`). These object types can be used as optional containers of primitive type values.

The fact that absence of a value is only temporary is detected, for example, for `final` variables in Java. A variable declared as `final` must be definitely assigned with a value at declaration (for local and static (class) variables), or no later than in a constructor (for non-static (instance) variables) (Gosling et al., 2023). However, it only refers to the `final` variables. If a variable can change its value, definite assignment is not required by the compiler.

Dynamically typed programming languages (e.g., JavaScript (ECMA-262), Python (Pyt, 2023), and Perl (Per, 2023)) also allow assigning an absent value (`null`, `undefined`, etc.) to any variable. Therefore, any variable in these languages is optional. However, it does not cause additional problems because variables in dynamically typed languages can store values of any type at all, and avoiding incorrect usage of the value is the programmer's responsibility.

Languages with strong functional influence support the optional monad (e.g., `Option` in Scala (É, 2023), optionals in Swift (App, 2023) and nullable types in Kotlin (Jet, 2023)). This type is type-safe in the sense that the value of an object of this type cannot be used without explicit checking that the object does contain an actual value.

However, for historical reasons, all object types in Scala can also contain the null value treated as in Java (Odersky et al., 2023). Thus, different optional types co-exist in Scala. One of them is compatible with Java and has its drawbacks, whereas the second one is not compatible with both Java and the first one. In Swift (App, 2023) and Kotlin (Jet, 2023), the optional type is constructed more consistently: only variables explicitly declared as having an optional type can contain the null value. Assigning the null value to other objects is prohibited and checked at the compilation time.

SQL treatment of the optional types should also be noted. In SQL, a column can contain the null value (unless it is prohibited by the `NOT NULL` constraint) too (ISO/IEC 9075-2:2023), but its semantics is peculiar. Whereas most programming languages treat `null` as absence of a value (or, technically speaking, a special value meaning inappropriateness of other values), SQL interprets `NULL` as an unknown value among the values of the same type (ISO/IEC 9075-2:2023). In particular, it means that the expression `NULL = NULL` is not true, while in most other programming languages it is. Therefore, although the null values in SQL and other languages implement the same pattern, the difference in their semantics creates problems in object-relational mapping.

Related Patterns

The Null Object pattern, described by Kevlin Henney, suggests using a real object of the necessary class to mark an absent value (Henney, 2002). This object should implement all methods of the class, but the methods should do nothing and return suitable default values (Henney, 2002). Applying the Null Object pattern does not require any language support, but puts the responsibility for handling the special case on the Null Object itself. The client that receives this object may even be unaware that it is working with the Null Object (unless it checks it explicitly). As a result, it is difficult to guarantee that the default no-op action and the default values are indeed suitable. Suitability rather depends on the client logic, which is not known to the Null Object. On the contrary, the Optional Value pattern is based on the client's awareness about absence of the value and (in its type-safe form) on the enforcement of the condition check by the programming language.

The optional type in its type-safe form is a special case of a monad. It was one of the first described monads under the name of the Maybe monad (Wadler, 1992a). The optional type can also be treated as an iterable collection that cannot contain more than one element.

Formalisation

Since the *Optional Value* pattern is a special case of the Variant pattern (Section 3.6), the former can be formalised similarly to the latter, substituting two underlying types:

an arbitrary type T and $Unit$ (empty type).

$$
\begin{aligned}
Optional = \forall T.\{\exists S, \{\forall R.\{ \\
&OptCreateValue : T \to S, \\
&OptCreateEmpty : Unit \to S, \\
&OptApply : S \to (T \to R) \to (Unit \to R) \to R, \\
&OptMutate : Ref\ S \to \\
&\to (AssignableOnce\ T \to R) \to (Unit \to R) \to R\}\}\}
\end{aligned}
$$

Instead of $AssignableOnce\ Unit$, this definiton uses simply $Unit$ because assigning to the Unit type does not make sense.

3.8. Type Intersection

Type intersection naturally corresponds to multiple inheritance in object-oriented languages since a subclass of several classes is also a subtype of all of them. However, multiple inheritance from classes is rarely supported in object-oriented languages. Multiple inheritance from interfaces is typically supported and provides the opportunity to create type intersection. Since type intersection is a natural consequence of multiple inheritance, there is no need to describe a separate pattern.

4. ELEMENTS OF COMPUTATION

The present section describes basic computational patterns that provide the foundation for describing program behaviour. In particular, the Traversable Once and the Assignable Once patterns establish links between program behaviour and data composition (Batdalov and Nikiforova, 2018). Since the assignment patterns, even though they represent basic behaviour as well, are numerous and complicated, their discussion is postponed until Section 5.

4.1. Generalised Functions

A function is the primary unit of behaviour in most programming languages. However, the features of functions may differ between languages. The present section aims to find the general case of what behaves like a function in various languages.

Context

A computation consisting of logically separate pieces with clearly defined inputs and outputs (as virtually any computation does).

Problem

To reason about programs, one needs to split the whole computation into separate pieces. The relationships between these pieces involved in the reasoning may be pretty complex, and the pieces should be suitable for them.

Forces

Reusing computation A piece of computation may be used in multiple other pieces of code.

Logically separate computation Considering a piece of computation to be a logically separate operation can be helpful in reasoning about a program even if this operation is never reused (Martin, 2008).

Meta-computation A computation may be defined in terms of other computations. For example, a mapping function can take an arbitrary transformation (not known beforehand) and apply it to every element in a sequence.

Inputs and outputs Reasoning about programs is more straightforward when pieces of computation are treated as transformations of inputs to outputs.

Type safety The type system of a language should be able to check if a value passed as an input or an output is of the type expected by the receiver either at the compile time (for statically typed languages) or at the run time (for both statically and dynamically typed languages). The same force applies to meta-computations, meaning that computation pieces should also have types.

Computation state A computation may need to keep a state between calls. For example, a computation that generates a numerical sequence may store the current position in the sequence.

Side effects The state used by a piece of computation may be external to the computation itself and shared with other pieces. In this case, changes in the state may affect other pieces of computation (side effects).

Solution

The primary use cases are covered by conventional functions, which exist in virtually any general-purpose programming language. Conventional functions satisfy the forces *Reusing computation*, *Logically separate computation*, *Inputs and outputs*, and *Type safety*. If functions are first-class objects (which is not the case in any language), they also satisfy the *Meta-computation* force (higher-order functions in type theory (Pierce, 2002)). In non-purely functional languages, conventional functions usually satisfy the force *Side effects*. However, they do not have an isolated state: any external state they use is shared with other calls of the same function (and usually the outside world).

Some languages support anonymous functions (also known as lambda-expressions). They are useful in the context of the *Meta-computation* force as it is often impractical to give a separate name to each function to be passed as an argument, especially because these functions are often short and simple. Instead, a lambda-expression may be defined in place (e.g., `values.map(n => n * n)` to square each element of a sequence). Since lambda-expressions exist within another scope (e.g., inside a function) that may have a limited lifetime, the *Side effects* force requires careful implementation to avoid accessing a state that does not exist anymore. In practice, it often involves creating a closure (an implicitly created functor, see below) that holds the necessary state (Meyers, 2015). However, lambda-expressions themselves yet do not support the *Computation state* force in full because they still do not have an isolated state.

A solution that satisfies all mentioned forces is a functor. Functors are objects which can hold some data and also have a special method for calling them as if they were functions (e.g., `operator()` in C++ (ISO/IEC 14882:2020), `apply` in Scala (Odersky et al., 2023), etc.). When a functor is used as a function, this method is called instead. Thus, from a user's point of view, functors behave like functions but can also hold some state. The language-level support for functors require existence of a designated method and substitution of calls to this method instead of calls to the object itself.

Functors can also be defined automatically by a compiler. In particular:

- A closure is a functor that holds values or references of the variables that a function or a lambda expression uses from the enclosing scope. The function inside this functor references the closure's data members instead of the real variables. However, an inaccurate implementation of such a closure may lead to memory leaks (Glasser, 2013).

- A function that returns a sequence of values one by one (e.g., a generator in Python (Pyt, 2023) or an iterator in C# (ECMA-334)) should hold the state of iteration between calls. The state is held in an automatically generated functor.

Thus, a few kinds of objects may serve as functions:

- conventional functions,
- user-defined functors,
- autogenerated functors for closures and generators.

Obviously, in the course of a program's evolution, different kinds of these generalised functions may replace each other. From a user's point of view, these forms are interchangeable at the call site and differ only in implementation. Therefore, the design principle 'Program to an interface, not an implementation' (Gamma et al., 1995) requires them to have a common interface. This common interface is the functional type discussed in the next paragraph.

In type theory, the functional type is defined by the types of the arguments and the return value of a function (Pierce, 2002). In terms of the current pattern, the forces *Type safety* and *Meta-computation* require the presence of the functional type. It is important that, in systems with subtyping, the types at the declaration site and at the usage site may differ. If a higher-order function expects a functional argument of type `A => B` (i.e., a function that takes a value of type `A` and returns a value of type `B`), then any function of type `A' => B'`, where `A'` is any *super*type of `A` and `B'` is any *sub*type of `B`, will be suitable instead. The reason is that, when the higher-order function calls its argument, it will always pass a value of type `A`, which is suitable if the argument function is ready to accept any value of the supertype `A'`. The argument function will always return a value of type `B'`, which is suitable when the higher-order function expects a value of its supertype `B`. In terms of subtyping, the functional type is contravariant with respect to its parameters and covariant with respect to its return value (Pierce, 2002). Support for functors at the language level requires that the manually and automatically generated functors also belong the the same functional type as conventional functions.

Consequences

Benefits

Complex relationships Higher-order functions and holding state between calls allow defining complex relationships between pieces of computation.

Isolated state The state held in a functor is shared between calls to the same functor but isolated from other functors, even of the same class.

Easy evolution If a program is written in terms of the functional type, it is easy to switch between different implementations of this type (e.g., replace a function with a functor when a state gets required).

Liabilities

Difficult implementation The most general case is problematic in implementation and reasoning both on the programming and compiler development sides. In most cases, it is just redundant.

Complicated hierarchy It is difficult to abstract from a particular implementation (e.g., a conventional function) and write a program in terms of the functional type from the beginning.

Existing Languages

Using functions as values, lambda-expressions and closures are relatively recent features in mainstream object-oriented language but supported by now in virtually all of them. However, the general case described in this section is not always supported.

C++ allows passing functions as values using function pointers, which have a corresponding type (e.g., `int (*)(int, int)`) (ISO/IEC 14882:2020). C++ also allows creating an arbitrary functor by defining the function call operator `operator()` in a class (ISO/IEC 14882:2020). A functor may be defined automatically for a lambda expression. Neither functors nor pointers to them belong to the function pointer type; it includes only conventional functions that cannot hold any state. Since C++ 11, the standard library contains the `std::function` type, which covers both function pointers and functors (ISO/IEC 14882:2011). Having different types is beneficial since it allows differentiating between stateless functions and arbitrary callable objects. However, this means that a functor cannot be passed when a function is expected, especially in the old code, where there was no opporunity to use `std::function`. Furthermore, the function pointer type is invariant with respect to its parameters and return type, i.e., a function with another signature cannot be passed even if it is totally safe. The `std:function` type provides variance according to type theory rules. Thus, recent versions of C++ support the general pattern but have historical artifacts that are not fully compatible with it.

Java uses functional interfaces (interfaces with only one method declared) for implementation of the functional type (Gosling et al., 2023). Therefore, Java does not have a ‘default’ method that is called when an object is applied as a function. As a result, switching from a function to a functor requires changes at all call sites. According to the general subtyping rules in Java, functional interfaces are covariant with respect to their return types but invariant with respect to their parameter types (because method overriding with extension of parameter types is not supported (Gosling et al., 2023)).

C# supports the functional type, which includes functions and lambda expressions and has type theoretical variance rules (ECMA-334). However, C# does not allow to define arbitrary functors.

The languages designed for full support of the functional style, such as Scala (Odersky et al., 2023), Swift (App, 2023) and Kotlin (Jet, 2023) support the described pattern in full.

Related Patterns

Erich Gamma et al. described the Command pattern, which generalises conventional functions. This generalisation has the following features: it is a first-class object, uses inheritance, may store internal state, and supports undo, redo and logging (Gamma et al., 1995). Frank Buschmann et al. described the Command Processor pattern that may implement parts of this functionality (Buschmann et al., 2013). The current pattern may be treated as an adaptation of the Command pattern minus what is included in the Command Processor pattern for the problems of the present work. The part included in the Command Processor pattern is discussed in Section 4.2.

The Traversable Once pattern (see Section 4.4) includes functions that return multiple values piecewise. Such functions may be implemented using the current pattern as a piecewise return requires holding a state between calls.

Higher-order functions may be used to implement inversion of control (Fowler, 2005).

Formalisation

A generalised function is a universal type that behaves like a normal function.

$$\begin{aligned} \mathit{GenFunction} = \forall T_1.\forall T_2.\{\exists F, \{ & \\ & \mathit{FuncApply} : F \to T_1 \to T_2\}\} \end{aligned}$$

T_1 is the type of a formal parameter and T_2 is the type of a result. F is a concrete type that implements a general function (e.g., a normal function or a functor). An object of type F may be applied to a value of type T_1 to get a value of type T_2.

As usually in type theory, multi-parameter functions are not considered because they are easily reducible to single parameter functions.

4.2. Executors

During program execution, one may need to execute different parts of code in different environments. An executor is an abstraction of an environment that can run code.

Context

Executing different pieces of computation in a heterogeneous environment (e.g., on different hardware, in different threads, or with different values of global variables).

Problem

Computation needs to manage in which environments to execute various pieces of the computation.

Forces

Multiple systems Different pieces of computation may be executed on different machines.

Multiple execution units Different pieces of computation may be executed on different execution units (e.g., threads or thread pools) on the same machine.

Different global objects Different pieces of computation may need access to different global objects. For example, pieces of code executed on different machines have different clocks. However, it is also possible in the local case: for example, testing code that reaches out to an external system may require creating a fake companion.

Restricted computation Computation executable in a different environment may be restricted. For example, it would be unsafe if a remote system accepted arbitrary code for execution.

Managing environment Code that manages where to execute other pieces of code is itself executed in some environment.

Solution

Introduce the abstraction of an executor that can perform computation. It can represent either a remote system (the *Multiple systems* force) or an execution unit on the same system (the *Multiple execution units* force).

In order to address the *Different global objects* force, associate an execution context with an executor. An execution context should contain references to all available global objects (singletons), but different execution contexts may have different global objects with the same name.

In order to address the *Restricted computation* force, define an interface that an executor is ready to provide.

The *Managing environment* force means that executing code on another executor should be explicit. By default, a called function is executed on the same executor, but a caller may override it.

Consequences

Benefits

Unified interface Inter-environment calls are performed in a unified manner.

Replacing singletons Singletons can be replaced easily by creating a new execution context, which may allow avoiding tight coupling. Tight coupling to singletons is a common design problem, hard to eliminate (Sommerlad, 2007).

Interaction styles Since an executor may represent an external system, the ability to execute code on it provides support for various interaction styles (see Section 2.11).

Liabilities

Counter-intuitive interface The relationship between the current environment, the executing environment, and the executable code may be counter-intuitive because the more habitual case of same-environment calls has only two participants.

Singleton implementation problems If execution contexts are responsible for managing singletons, they inherit all their implementation difficulties, such as safe destruction (Alexandrescu, 2001).

Existing Languages

The notion of an executor is usually implicit at the language level but frequently used in libraries. For example, Java standard library contains the Executor and ExecutorService interfaces, implemented by various thread pools (Ora, 2023).

Remote execution is usually implemented as Remote Procedure Calls (RPC), such as CORBA (OMG, 2021), gRPC (gRP, 2021), WCF (Mic, 2021), or SOAP (W3C, 2007). Inter-process communication on the same machine, as in D-Bus (Pennington et al., 2023), is architecturally similar. An executor in such systems (remote or a concurrent process) is often represented by an object implementing some interface (cf. the *Restricted computation* force) or participates in the message exchange. The interface is usually defined using an interface definition language.

The executors mentioned above naturally have different global objects, such as system clocks. At the same time, explicit replacing of global objects is common in testing environments. For example, they often allow managing time in fake clocks to test code behaviour in time without actually waiting.

Related Patterns

The code executed by an executor implements the Generalised Functions pattern (see Section 4.1).

Various patterns based on systems' interaction, such as Façade, Proxy (Gamma et al., 1995), Blackboard, Broker, Forwarder-Receiver, Master-Slave, Pipes and Filters, Proxy, or Publisher-Subscriber (Buschmann et al., 2013) may be implemented using executors.

The global objects held in an execution context are essentially singletons (Gamma et al., 1995). An execution context restricts the scope of such singletons so that different singletons with the same name can co-exist, but only one is accessible from a particular point in a program.

Formalisation

The general executor can be formalised in the following way:

$$
\begin{aligned}
\mathit{Executor} = \forall T_1.\forall T_2.\{\exists E, \{ \\
&\mathit{ExecExecute} : E \to (\mathit{GenFunction}\ T_1\ T_2) \to T_1 \to T_2, \\
&\mathit{ExecContext} : E \to (\mathit{Map}\ \mathit{Id}\ \mathit{Top})\}\}
\end{aligned}
$$

T_1 and T_2 (the types of an argument and a result) define the interface that the executor supports. They can be Top (an arbitrary value) if the executor can execute any code. An executor can also support multiple interfaces (be able to call functions of different types), but it is omitted for simplicity.

$\mathit{ExecExecute}$ takes an executor, a generalised function, and an argument and executes the generalised function on an executor, producing the return value.

$\mathit{ExecContext}$ retrieves an execution context, which is interpreted as a map from identifiers to arbitrary values.

4.3. Data Access Delegation

Many well-known design patterns are based on delegation. Behaviour delegation is usually trivial (it is just defining a function that calls another function) and does not require a separate discussion. However, the present section describes a specific case of delegation: delegation of data access.

Context

A computation that involves multiple participants (logical parts with relatively independent behaviour).

Problem

Computing some data should look at the usage site like direct data access.

Forces

Computable properties Some properties of objects may logically look like data attributes but not be stored in the object itself.

Changing program logic In the course of program evolution, data that used to be stored may become computable.

Hiding implementation details The place where data are stored (e.g., in the object directly, its base class, or inside its field) is an implementation detail and may be changed without changing the interface.

Toilsome definitions Manual definition of multiple computable properties may be toilsome when they are computed similarly (e.g., all are stored inside the same compound field).

Side effects A computation may have side effects which are not expected from regular data access.

Solution

Interpret data properties of an object as a logical relationship without assuming that these data are held in the object. Interpret any data property access (both for reading and writing if allowed) as a computation, being it a trivial retrieving a field from a compound data structure or a complex computation. Consider the actual storage and computation an implementation detail that may change. This solution addresses the *Computable properties*, *Changing program logic*, and *Hiding implementation details* forces.

It is optional to define both a read accessor (getter) and a write accessor (setter). Read-only properties have only a getter, and write-only properties (rare but possible) have only a setter.

The *Toilsome definitions* force can be addressed by the automatic creation of delegations according to certain rules (e.g., automatically created a getter and a setter for a field declared as public or redirect all requests to missing fields to the same object).

Inheritance can be considered a specific application of this solution. Inheritance can be reduced to composition (an object contains an object of its base class) and delegation (all data requests and calls to methods of the base class are delegated to the base object). It is not how inheritance is usually implemented, but the logical relationship may be described this way. In this sense, the preference of composition to inheritance mentioned by Erich Gamma et al. (Gamma et al., 1995) can be explained by better control over delegation. The automatic delegation of everything to the base object is unpredictable and can cause unexpected effects.

The *Side effects* force requires a convention (enforced or not) that the function to which data access is delegated does not have side effects. There may also be other conventions corresponding to what is typically expected from data access, e.g., guaranteed time complexity at most linear with the size of the data.

Consequences

Benefits

Easy transition It is easy to change how data are stored or computed without changes on the usage site.

Naturally looking data access Addressing data properties looks like taking data attributes independently of how they are implemented.

Unexpected delegation The automatic creation of delegations may make the behaviour of an object unpredictable as it is difficult to expect what is delegated and what is not. Furthermore, delegation to a member with the same name may also accidentally cause semantic mismatches if the true meaning of this member is different.

Unexpected behaviour The conventions related to the absence of side effects and time guarantees are difficult to enforce. However, if they are not followed, the data access may have unexpected behaviour.

Existing Languages

Computable properties are supported, for example, in C# (ECMA-334), JavaScript (ECMA-262), TypeScript (Mic, 2023), Scala (Odersky et al., 2023), and Python (Pyt, 2023). They allow writing getters and setters to work with arbitrary data (not necessarily provide access to a data member with the same name). The properties defined with getters and setters on the usage site behave like actual data attributes.

The interpretation of inheritance as composition plus delegation applies to virtually any object-oriented programming language (even if it is not used explicitly).

Scala documents the convention about the absence of side effects. In Scala, a method without parameters may be declared either with empty parentheses or without them (`foo` vs `foo()`). These declarations are functionally equivalent, but the declaration without parentheses is interpreted as a data accessor and a promise not to have side effects (Spiewak and Copeland, 2023).

Related Patterns

Many patterns described by Erich Gamma et al. (Gamma et al., 1995) and other researchers are based on behaviour delegation. The current pattern describes data access delegation, which complements behaviour delegation.

If a computable property can be changed, delegation effectively provides an assignable slot (see Section 4.5).

Formalisation

If type T has a computable property $x : U$, the getter and setter may be defined as:

$$x : T \to U$$

$$xAssignable : Ref\ T \to AssignableOnce\ U$$

The compiler may then substitute access to this field (like $u = t.x$ or $t.x = u$) with calls to these functions ($u = t.x()$ and $t.xAssignable().Assign(u)$ correspondingly).

As discussed in Section 3.1, generalising the mutation operation requires specifying the most specific type as the argument and the most abstract type as the return value. This approach should work with any T. For specific types in T's place, it is possible to define narrower-typed overrides (e.g., accepting *Sink T* instead) and thus provide more flexibility to their users.

For types encapsulating a compound data structure (e.g., record- or map-based classes), getters and setters (if they exist) may have a trivial implementation. Reading is just accessing a field, and writing is a partial assignment (see Section 5.3). In other cases, the implementation may be more complex and type-specific.

4.4. Traversable Once

The Traversable Once pattern is a generalisation of the Iterator pattern, described by Erich Gamma et al. (Gamma et al., 1995). Unlike the Iterator pattern, it does not assume that the values to iterate over are held in an aggregate (collection). Instead, the values can be retrieved or computed as they are requested. In this case, the opportunity to traverse the same set of values the second time is not guaranteed. That is the reason to call the pattern Traversable *Once*. However, multiple traversals are not prohibited by the pattern, they are only not guaranteed. If the underlying structure allows multiple traversals, they can be performed by getting a number of Traversable Once's, each of which is traversed once only.

Context

Software logic involves processing of a series of values of the same type.

Problem

Repeat the same operation or transformation on a series of values, independently of where these values come from.

Forces

Independence of the source Processing of a series of values is logically independent of the origin of these values. They can be stored in computer memory as a kind of collection, received from an external source or simply calculated on the fly (e.g., members of a numerical sequence).

Big amount of data The amount of data coming from an external source or calculated on the fly may be big or even infinite. This makes impractical or impossible to store all values in memory, which requires to process them one by one.

Values transformation Transformation (mapping) of a sequence of values may give another sequence or sequences of values. In this case, it is often impractical to compute mapping before the resulting sequence is actually used (eager vs lazy evaluation).

Complicated value retrieving Retrieving a value from a sequence may involve intensive computations or interaction with peripheral devices or other computers. In this case, it may be undesirable to block the current thread until the next value is received.

Basic operation Traversal of a sequence of values is perceived as a basic operation. Therefore, implementing this operation with a long and complex sequence of actions is undesirable.

Solution

Create an object that can return the next value in a sequence or report that the sequence is exhausted.

The participants of the pattern are:

- Traversable Once, which provides a common interface for retrieving a sequence.
- Generator, which retrieves a sequence of values from its source (a collection, a peripheral device, computation, etc.).
- Mapper (optional), which transforms a sequence of values into another sequence.
- Provider, which creates the Generator and/or Mapper.
- Client, which consumes the values provided by the Traversable Once.

The class diagram of the solution is presented in Figure 4.1. Generator and Mapper implement the common interface TraversableOnce so that instances of those classes can be used interchangeably. A mapper references one or more subsequent Traversable Once's, which produce values to be transformed. TraversableOnce, Generator and Mapper can be generic types, whose parameter is the type of traversable values. Methods of the TraversableOnce interface and its subtypes can return their results either synchronously or asynchronously. The name `getValues` is given to the method returning the Traversable Once as an example. Actually, any method of the Provider can return a Traversable Once.

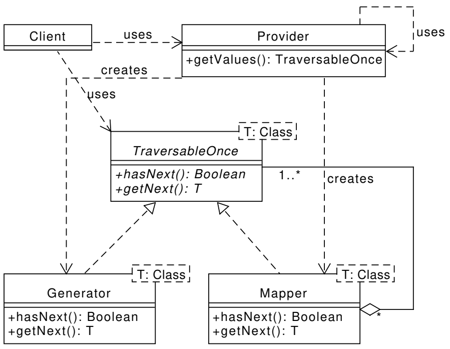
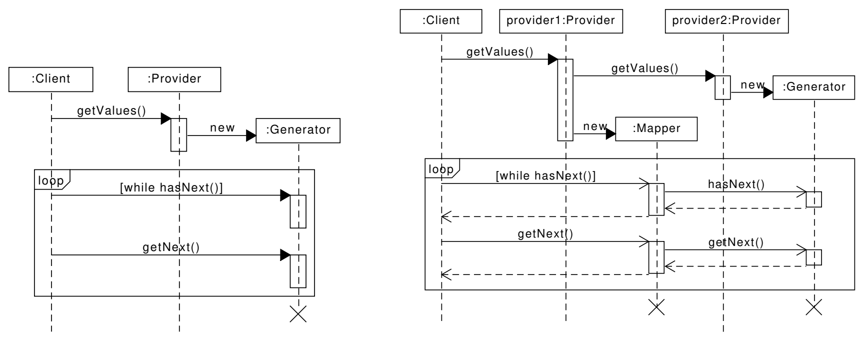


Figure 4.1. Class diagram of the Traversable Once pattern

Two example sequence diagrams are shown in Figure 4.2. A client requests a Traversable Once from a provider, the latter creates one and passes it to the client. After that, interaction between the client and the Traversable Once is independent of the provider. Since the pattern only guarantees one traversal, the mappers and generators cease to exist after traversal has finished.



(a) Synchronous traversal

(b) Asynchronous traversal with an intermediate mapper

Figure 4.2. Sequence diagrams of the Traversable Once pattern

This solution is functionally sufficient for implementation of different kinds of sequence traversal. However, it does not address the problem of excessively complex implementation of a basic operation (force *Basic operation*). To solve the problem, a programming language can hide the pattern complexity behind special syntactical constructs. Examples of these constructs are the `for` loop, generator functions and functional transformations. With these constructs, the participants of the pattern are not created and used explicitly by the programmer, but implicitly by the compiler.

Consequences

Benefits

Sequential processing The pattern facilitates processing a sequence of values one by one. The values do not have to be stored in memory all at once, but can be received from external source or calculated as necessary.

Functional transformation Using mappers facilitates lazy evaluation of the results of functional transformations of value sequences.

Hiding complexity Language-level support of the pattern allows hiding its complexity behind natural and easily understandable constructs.

Liabilities

Counter-intuitive lazy evaluation Lazy evaluation may lead to counter-intuitive behaviour when side effects are involved (Pierce, 2002). For example, in Scala, such behaviour caused switching almost all transformations to eager evaluation instead of previously used lazy one (Odersky and Spoon, 2023).

Sequential processing only A restriction of the described pattern is that it describes strictly sequential processing of values only. In the asynchronous case, parallel processing is important too.

Existing Languages

Synchronous Examples

Iterable collections are widely used in programming languages and libraries: arrays, lists, sets, etc. Abstractions of iterable collections, such as the `Iterable` interface in Java (Gosling et al., 2023), capture their common ability to provide the items that they contain to the client one by one. Collections usually hold the items for a relatively long time and support multiple traversals. In terms of the described pattern, iterable collections are Providers that generate Traversable Once's on request.

Generators functions or simply generators (such as in Python (Pyt, 2023), C# (ECMA-334), JavaScript (ECMA-262), etc.) form another approach to iteration. A generator function conceptually produces a sequence of values, but returns one value on each call instead of all values at once. Thus, iteration over the sequence of values can be performed by a series of subsequent calls. Unlike collections, generator functions do not require the produced values to be stored in memory in advance. Instead, they can retrieve them as needed or calculate on the fly. In terms of the described pattern, generator functions are `getValues` methods (maybe, called differently) of some providers.

Scala generalises both approaches in the `TraversableOnce` type (after which the name of the pattern is chosen). `TraversableOnce` is a supertype of `Traversable`s (abstraction of collections) and `Iterator`s (objects that can be used to implement generator-like behaviour) (Odersky and Spoon, 2023). This abstraction captures the idea of iteration without any guarantees on the object state after the iteration has finished. Since the state after iteration is unknown, the opportunity to iterate again is not guaranteed (hence the name).

Asynchronous Examples

The examples given above may have asynchronous counterparts. For example, in Python and JavaScript, generators may be both synchronous and asynchronous (ECMA-262; Pyt, 2023). Asynchronous generators return values asynchronously and thus can be used for iteration over values that are not immediately available.

Scala's `TraversableOnce` does not have an asynchronous counterpart in the standard library. An analogue that can be used asynchronously is `Iteratee` (Kiselyov, 2012). `Iteratee`s are purely functional and do not implement the described pattern (since it involves changing state). However, they can be used for the same purposes.

Another remarkable use is the `Observable` type in the ReactiveX library (ReactiveX). An `Observable` generates a sequence of values, which is provided to a client asynchronously (ReactiveX). `Observable`s cannot be used in the `for` loop, but allow consuming the values in the reactive style (e.g., subscribing to an `Observable`).

Related Patterns

Gang-of-Four's Iterator pattern covers the synchronous case of iterable collections. It decouples the collection interface (the Aggregate) from its traversal logic (the Iterator), but still assumes that Iterators are generated by Aggregates (Gamma et al., 1995). The Traversable Once pattern describes a slightly more general case, where existence of a collection is not necessary.

The Traversable Once pattern is a form of inversion of control, described by Martin Fowler (Fowler, 2005), since the control flow is partially controlled by the object that provides a sequence of values.

The Pipes and Filters architectural pattern, described by Frank Buschmann et al., provides a structure for systems that process a stream of data (Buschmann et al., 2013). Data are processed by filters and transferred between filters by pipes (Buschmann et al., 2013). Since the Pipes and Filters pattern assumes incremental processing of data (Buschmann et al., 2013), pipes and filters may be implemented using the Traversable Once pattern.

The Traversable Once pattern is a special case of monads, which are used in functional programming to mimic 'impure' features such as state, exceptions, and continuations (Wadler, 1992a,b).

Formalisation

The Traversable Once pattern can be formalised in the following way:

$$\begin{aligned} TraversableOnce = \forall T.\{\exists R, \{ & \\ & TraversableCreate : T \to R, \\ & TraversableAdvance : OneTimeRef\ R \\ & \ \to Optional\ Pair\ T\ R\}\} \end{aligned}$$

T is a type parameter, the type of produced values. R is a concrete type that implements a Traversable Once.

$TraversableCreate$ creates an object from a single value. Subsequent traversing this object should return the same value and stop. $TraversableAdvance$ performs actual traversing: it returns a pair of a value and a new Traversable Once or signals the end of traversal by returning an empty value. $OneTimeRef$ is a reference that can be used only once, like an rvalue reference in C++ (ISO/IEC 14882:2020). It communicates the fact the the advance operation is destructible: only the output R object can be used for the subsequent traversing, but not the original one.

As an alternative, Traversable Once may be defined as mutable in place (cf. having an implicit output parameter in Section 2.7):

$$\begin{aligned} MutTraversableOnce = \forall T.\{\exists R, \{ & \\ & MutTraversableCreate : T \to R, \\ & MutTraversableAdvance : Ref\ R \to Optional\ T\}\} \end{aligned}$$

It is equivalent to the first definition if the R return value is always stored at the location of the original object.

Thus defined type allows implementing the monadic functions (see Section 1.3):

1. *map*: given a transformation $T \to U$ and a $TraversableOnce$, construct another $TraversableOnce$ that holds the original once inside. $TraversableAdvance$ of the new class should call $TraversableAdvance$ of the underlying object and apply the given transformation before returning the value.

2. *unit*: apply $TraversableCreate$.

3. *join*: given a $TraversableOnce$ (the outer one) that produces $TraversableOnce$s (the inner ones), construct an object that holds the outer $TraversableOnce$ and the last produced inner one (or an empty $TraversableOnce$ at the beginning). $TraversableAdvance$ of the new class should call $TraversableAdvance$ of the inner $TraversableOnce$ and return the produced value or, if the inner $TraversableOnce$ is empty, request a new one from the outer $TraversableOnce$. If the outer one is exhausted, traversing stops.

It is easy to check that none of these functions requires iterating twice over the same $TraversableOnce$ and that the functions defined in this way obey the monadic laws. Therefore, $TraversableOnce$s are indeed monads.

4.5. Assignable Once

The Assignable Once pattern is used to request a value by giving the value producer a slot, where the value may be stored. Since the slot may not be available after being used, the pattern is called Assignable *Once*. As with the Traversable Once pattern, 'Once' does not prohibit multiple assignments, but rather emphasises that they are not guaranteed. When the application logic allows multiple assignments, Assignable Once's may be requested multiple times.

Context

A value calculated in the course of program execution is to be stored for future use.

Problem

Finding the target location for an assignment operation and computation of the value to be assigned may be apart in code and time, making direct assignment difficult.

Forces

Separation in code Producing a value and consuming it may be apart in code.

Separation in time A request for a value and actual producing it also may be apart in time, i.e., a value does not have to be provided immediately.

Limited time Despite the previous force, the time span between the request and producing a value is not infinite. The value consumer cannot guarantee that it will wait for the value endlessly (unless it is explicitly required by the program logic).

Object initialisation A particular case when the time frame for assigning a value is restricted is object or variable initialisation. An initial value is to be assigned before initialisation completes so that the variable is certainly initialised. In some cases, program logic also requires that the value is not changed after initialisation (constants, final class members, all values in purely functional languages, etc.).

Solution

Create an assignable slot by means of encapsulating it in an object, whose interface allows storing a value once. The slot is created by the value consumer. After the slot has been used, the consumer is notified that maintaining the slot in the writable state is not needed anymore. If a single assignment restriction is not needed in a particular situation, the value consumer can create multiple Assignable Once's.

The participants of the pattern are:

- Producer, which provides a value at a moment defined by its own logic.
- Consumer, which creates an assignable slot.
- Slot, which allows storing a value once.

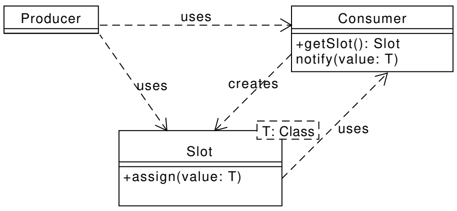


Figure 4.3. Class diagram of the Assignable Once pattern

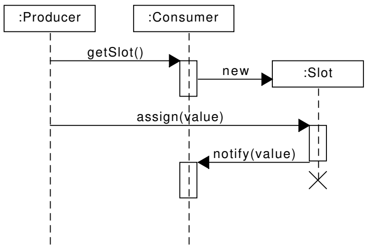


Figure 4.4. Sequence diagram of the Assignable Once pattern

The class diagram of the solution is shown in Figure 4.3. Figure 4.4 gives a sample sequence diagram. A producer retrieves a slot from a consumer and assigns a value at a later time. It should be noted that the slot does not have a method to retrieve the assigned value. Instead, the slot provides the value to the consumer in the way defined by the latter (in the given example, using the `notify` callback, but other solutions are valid too) and ceases to exist.

Consequences

Benefits

Decoupling from implementation The pattern forms a general abstraction of an assignable slot decoupled from the nature of the particular slot (a variable, a constant, a future, etc.). In the general case, this abstraction is transferrable between parts of code and in time.

Leak avoidance The restriction to a single assignment protects against leaking the assignable slot out of the scope of controllable usage. However, it does not prohibit multiple assignments at all since multiple slots can be generated when the program logic requires them.

Liabilities

Potential unsafety Even with the single assignment restriction, a distance or a time between creation and usage of a slot requires careful work to avoid unsafe operations.

Existing Languages

Variable is a basic concept in many programming languages. In terms of the described pattern, variables are simple consumers that can generate multiple assignable slots. Constants or final variables can provide only one assignable slot, which should be used during variable or class initialisation. However, constants and variables do not have a notification mechanism except for certain ordering guarantees (such as the *happens-before* relationship in Java (Gosling et al., 2023)).

Pointer and reference types (as in C++ (ISO/IEC 14882:2020)) are other non-object-oriented forms of assignable slots. Essentially, they are pointers to memory chunks for storing values. Pointers are transferrable and long-lived, so they allow dealing with forces Separation in code and Separation in time. However, the time frame of pointer usage is not controlled, the producer may attempt to assign a value even after the memory is deallocated. Therefore, pointers cannot help with forces Limited time and Object initialisation as the value consumer is responsible for maintaining validity of the reference during an infinite time span. Pointers do not support notification about value assignment either. Moreover, due to pointer unsafety, programming languages typically restrict their usage (e.g., in C#, dealing with pointers is available in the unsafe mode only (ECMA-334), and in Java, it is hidden in the undocumented package `sun.misc`). However, the pointers do exist in these languages, but are operated implicitly by the compiler and the run-time environment.

In Scala, the `Promise` data type can be interpreted as an Assignable Once for the asynchronous case. Whereas the `Future` data type contains a value that may eventually be available, the `Promise` data type provides a slot to assign such a value (Haller et al., 2023).

`Promise`s in JavaScript implement the same pattern. Resolving promises can be interpreted as assigning to an assignable slot, and this event gets known to the promise creator if it has attached a callback using method `then()` (ECMA-262).

`Subject` in ReactiveX is an example of an object that encapsulates the assignment operation, but allows multiple assignment (ReactiveX). In terms of the described pattern, it is a consumer that generates and provides multiple assignable slots.

Related Patterns

The Assignable Once pattern can be used to implement a specific aspect of inversion of control, providing a slot to store a value instead of retrieving a value and storing it on the caller side.

The Observer pattern, described by Erich Gamma et al., provides an interface for notification of observers about changes in the state of an observed object (Gamma et al., 1995). Its purposes are similar to the ones of the Assignable Once pattern, but it describes rather a long-time relationships between a publisher and multiple subscribers. The Assignable Once pattern, on the contrary, describes a single assignment.

Formalisation

The only requirement to the Assignable Once pattern is the opportunity to assign a value once. It can be formalised in the following way:

$$\begin{aligned} AssignableOnce = \forall T.\{\exists A, \{ \\ & Assign : OneTimeRef\ A \rightarrow T \rightarrow Unit\}\} \end{aligned}$$

T is a type parameter, the type of assignable values. A is a concrete type that implements an Assignable Once.

The only defined operation is *Assign*, which accepts a single-use reference (like an rvalue reference in C++ (ISO/IEC 14882:2020)) and a value and performs the assignment. The operation is destructible: the assignable slot cannot be used afterwards (but can be generated again if the underlying type supports it). The function does not return a value as its goal is the side effect.

This formalisation does not contain any operations of creating an assignable slot because there is no generic way of doing that. Creating assignable slots depends on the nature of the slot and is delegated to the specific subtypes. Examples of this are considered in the Section 5.

5. ASSIGNMENT

The present section describes various patterns for assignment applied in programming languages (Batdalov and Nikiforova, 2021).

Programs access values stored in memory by names, and giving a name to a value, i.e., assignment is one of the most fundamental operations in programming languages. In this work, the assignment is understood in a broad sense; it refers to any operation that changes the association between names and values. Such operations include:

- initialising a variable or a constant,
- reassigning the value of a variable, and
- passing objects between contexts, i.e., passing an argument to a function or returning a function's return value.

These operations change the association between names and values in a program, but this association can be changed in different ways. For example, a function argument can be passed either by value (creating a copy of the passed value) or by reference (sharing the same value between the caller and the callee). The present section aims to identify different types of assignment used in programming languages and describe them as patterns.

Programming languages usually provide similar opportunities for different assignment operations. For example, if a programming language allows passing arguments either by value or by reference, the same two options for assigning a variable will exist. Therefore, the identified patterns cover all mentioned assignment operations, collectively mentioned as 'assignment'. The distinction between the patterns is focused on their goal and the opportunities they provide to the programmer.

5.1. Value Assignment

Value assignment is the most straightforward kind of assignment when the source value is directly stored in the target. Its description is almost trivial in the simplest case, but even the base case is useful for discussing complications and trade-offs.

Context

Multi-step computations, which appear in virtually any program.

Problem

At a particular execution stage, processing logic depends on values already calculated, but maybe, at another time or in another code part. Recalculating these values every time when they are needed is inefficient.

Forces

Computational dependencies The result of a computation step can be an input for other computation steps.

Multiple dependencies Different computation steps executed at different times may depend on the same input.

Intermediate results semantics Intermediate results may have their own semantics and meaning for a programmer.

Linked data structures Data may be stored in a linked data structure, i.e., occupy several non-contiguous chunks of memory.

Expensive copying Copying a compound type value may be expensive in terms of CPU and memory resources.

Non-copyable data Some data, such as particular resource handles, should not be copied to avoid adverse effects of duplication (such as concurrent access to a resource that does not support it or use after release).

Multi-threaded environment Computations may be performed in a multi-threaded environment, allowing access to the same data from concurrent execution flows.

Solution

The prototypical solution, which satisfies forces *Computational dependencies* and *Multiple dependencies* , is to copy the source data and store the copy in the target, as shown in Figure 5.1.

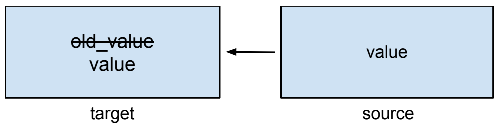


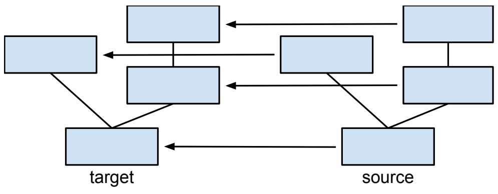
Figure 5.1. Value assignment for simple types

It should be noted that the source of an assignment does not have to be a named variable. It can be the result of a complicated expression instead. In this case, value assignment associates the result with a name, which may clarify its meaning (force *Intermediate results semantics*). Considering code quality, giving names to intermediate values make sense even if a value is used only once (Martin, 2008).

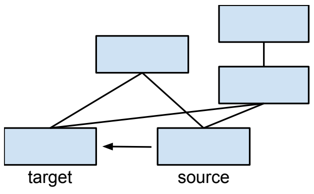
Moreover, value assignment of an unnamed intermediate result often allows avoiding creating a copy, which may be expensive (force *Expensive copying*). The compiler may optimise the code and construct the result of an expression directly at the target. However, such optimisation is not always possible, and in some cases, copying is unavoidable.

If the data are stored in a linked data structure (force *Linked data structures*), only the structure's base (e.g., the root node of a tree) can be copied directly to the target.

Other parts (e.g., non-root nodes) should be duplicated in new chunks of memory and correctly linked (deep copying). Deep copying is challenging to implement in the general case (Bishop, 2008). Many languages provide built-in support only for shallow copying: the base is copied to the target but continues referencing the old nodes. The distinction between deep and shallow copying is illustrated in Figure 5.2. Shallow copying is cheaper and easier to implement, but the coexistence of the source and the target means that changes in one copy affect the other one. Essentially, shallow copying is partly the value assignment and partly the referential assignment (see Section 5.2).



(a) Deep copy



(b) Shallow copy

Figure 5.2. Deep vs shallow copying

The drawbacks of shallow copying and expensiveness of copying in general (force *Expensive copying*) led to an alternative solution, the move assignment. If the source is not used after the assignment, we can save the source data to the target and ignore what remains at the source. In this case, shallow copying is safe. Using shallow copying for the move assignment is shown in Figure 5.3. This solution can often give a significant performance gain. The move assignment may also allow applying the value assignment to non-copyable data (force *Non-copyable data*).

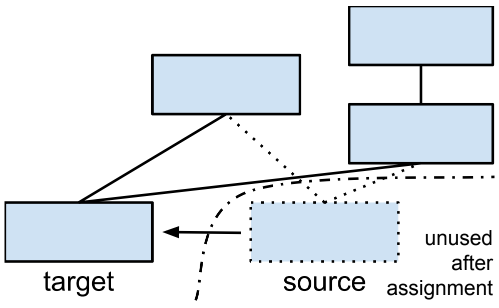


Figure 5.3. Move assignment example

Consequences

Benefits

Value reusing A value created at one location can be used at another one.

Absence of shared state Since the target copy is independent on the source, each copy can be modified independently without the risk of unexpected behaviour. Examples of such behaviour include:

- race conditions in a multithreaded environment (but they are still possible if the same object is used in different threads),
- accidental modification of an argument passed to a function, and
- use of a memory chunk after freeing it.

Move assignment optimisation If the conditions for the move assignment application hold, this technique may save a significant amount of resources.

Liabilities

Spending resources for copying The value assignment is often more expensive than the referential assignment. Even the move assignment may be expensive if the base is big.

Implementation difficulty Creating a universal implementation of deep copying for the general case with arbitrarily complex data structures is challenging. In practice, this means that creating a deep copy method for a complex type is the type creator's responsibility. Programming languages and standard libraries may provide such facilities for well-known types but not for the general case.

Limitedness of the move assignment The move assignment may require a separate implementation. It may not be more efficient than the copy assignment. The conditions when it is possible may not be obvious (Meyers, 2015).

Usage difficulty Variability of the value assignment and complicated conditions of applicability of different options put a heavy burden on programmers. It is not always obvious which option will be applied by the compiler, how to affect its choice, and how to balance efficiency and safety.

Existing Languages

Assignment to a variable of a primitive type, such as an integer number, is the value assignment in most languages. However, such languages as Java (Gosling et al., 2023), JavaScript (ECMA-262), C# (ECMA-334), and others treat all non-primitive types as reference types, and assignment to such variables is referential assignment (see Section 5.2). In such languages, creating and referencing a copy of an object can simulate the value assignment. Copying usually should be implemented by the programmer.

C++ supports the value assignment even for complex types, provides default implementations of copying, and allows overriding or deleting the default implementation for a specific type (ISO/IEC 14882:2020). C++ standard allows compilers to eliminate copying return values of functions and construct the return values directly at the targets when possible (return value optimisation) (ISO/IEC 14882:2020).

C++ 11 differentiates between copy and move assignment and chooses between them automatically according to a complex set of rules (ISO/IEC 14882:2011). These rules not straightforward and the compiler's choice may often be non-obvious (Meyers, 2015), but in the end, a programmer has a high degree of control over the kind of assignment.

The same approaches work for passing objects between contexts. Passing a value of a primitive type to a function creates a copy that can be modified independently in most languages. C++ differentiates between passing by value and by reference and, for by-value arguments, applies the same techniques as during the value assignment.

The explicit support for the move assignment in C++ is quite a rare example, but many languages apply this approach implicitly. If the result of a complex computation is passed as a function parameter or returned from a function, the temporary variable that stores the result is no longer needed. As a result, the compiler may move from this variable instead of copying its value. Essentially, the explicit marking a value as movable in C++ (`std::move`) is just forced treating it as a temporary value that the program can destroy after usage (Meyers, 2015).

Related Patterns

There is often a trade-off between the value assignment and the referential assignment (see Section 5.2). The forces affecting these patterns are mostly the same, and the balance of benefits and liabilities dictates the choice.

The value assignment target is not necessarily a variable. The Assignable Once pattern can generate an assignable slot on the fly and pass it to the place where it will be assigned (Batdalov and Nikiforova, 2018). The Partial Assignment pattern for mutable types (see Section 5.3) is an example of such a situation.

Formalisation

The present work treats constant and variable names as type-theoretical references (`Ref`, `Source`, or `Sink`, depending on the allowed operations; see Section 1.3). They should be distinguished from Java referential types, which actually represent the next level of indirection discussed in Section 5.2. In type theory, references formalise the ability to mutate the program state (Pierce, 2002). However, in everyday programming, this basic referencing and dereferencing is usually implicit, and programmers are used to treat symbols as values or assignable slots depending on the context. In this sense, a value assignment $a = b$ when both symbols are assigned type T means that symbol a should have type $Sink\ T$, and symbol b should have type $Source\ T$.

In order to formalise the value assignment, the *Sink* type should be able to generate assignable slots:

$$SinkAssignable : Sink\ T \rightarrow AssignableOnce\ T$$

With this function available, the mentioned assignment consists of the following:

1. Generating an assignable slot from a.
2. Using the assignable slot (see Section 4.5) to assign a value from b.

The whole operation can be written as Assign(SinkAssignable(a), Get(b)).

5.2. Referential Assignment

Referential assignment means that the same memory location can be reached through multiple symbols (names) in a program. It is often more efficient than the value assignment and has additional use cases, but also has specific drawbacks related to shared state. The referential assignment can also be called aliasing in the sense that one symbol is an alias of another. However, the referential assignment has a broader sense because the source does not have to be a specific aliased object; it can also be a temporary object that holds the result of an expression.

Context

Multi-step computations, usually involving complex types.

Problem

Processing logic depends on previously calculated values, but copying data is too expensive or unsafe for applying the Value assignment pattern.

Forces

Computational dependencies The result of a computation step can be an input for other computation steps.

Multiple dependencies Different computation steps executed at different times may depend on the same input.

Expensive copying Copying a compound type value may be expensive in terms of CPU and memory resources.

Non-copyable data Some data, such as particular resource handles, should not be copied to avoid adverse effects of duplication (such as concurrent access to a resource that does not support it or use after release).

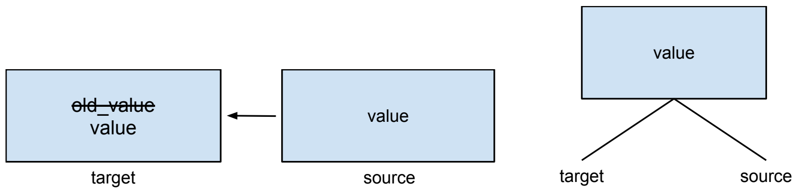
Multi-threaded environment Computations may be performed in a multi-threaded environment, allowing access to the same data from concurrent execution flows.

Immutable data types A data type may be immutable, i.e., prohibit modifications after creating a value.

Lifetime considerations The source and the target may have a different lifetime.

Solution

Create a data type that holds a reference to the actual data (e.g., a memory address, a handle, or another indirection mechanism). Implement the referential assignment by copying the source reference to the target reference. Both references may continue existing



(a) Value assignment (b) Referential assignment

Figure 5.4. Value assignment vs referential assignment

after the assignment and will refer to the same memory location. The difference between the value assignment and the referential assignment is illustrated in Figure 5.4.

The referential assignment avoids the costs of copying a significant amount of data (force *Expensive copying*). However, it comes at the cost of state sharing and the related adverse effects. These costs can be reduced by limiting what a reference can do. For example, a `const` reference and a pointer to `const` in C++ allow reading a value but do not allow modifying it (ISO/IEC 14882:2020).

The lifetime difference of the source and the target (force *Lifetime considerations*) may lead to dangling references when the referential assignment is applied. A solution that addresses this problem may include a memory management system that ensures against dangling references. The most popular system of this kind is garbage collection. Other options include smart pointers (tracking the ownership of a location, i.e., which part(s) of code is/are responsible for its release when unused) or the absence of such a system and putting the responsibility on the programmer's shoulders (as with raw pointers and references in C++).

Consequences

Benefits

Value reusing A value created at one location can be used at another one.

Distant modifications A value created at one location can be modified at another one. Output parameters and passing around assignable slots may be implemented in this way.

Resource saving For compound data types, copying a reference is typically much cheaper than copying the whole object.

Shared access to non-copyable data If there is a reason to prohibit copying a resource handle (or other similar data), a single non-copyable object may serialise and handle all requests to the resource. Then having multiple references to this object allows accessing the resource from different parts of code despite the inability to copy.

Efficiency for immutable types With immutable types, the cost of state sharing is negligible because the shared state cannot be modified. If we have guarantees that a type is immutable, we can replace all value assignment operations with the referential assignment without risk. The only thing to assess is whether the referential assignment is cheaper indeed. Unfortunately, non-purely functional programming languages rarely provide guarantees of immutability. Therefore, tracking possible adverse effects is the programmer's responsibility.

Liabilities

Adverse effects of shared state Distant modifications of a shared state may lead to such effects as race conditions in a multithreaded environment, accidental modification of an argument passed to a function, and use of a memory chunk after freeing it.

Memory management Without a sophisticated memory management system, such as garbage collection or smart pointers, the referential assignment may lead to memory leaks or dangling references.

Existing Languages

Such languages as Java (Gosling et al., 2023), JavaScript (ECMA-262), C# (ECMA-334) and others treat all non-primitive types as reference types. Assignment to variables of such types and passing them between functions is the referential assignment because different names may refer to the same object in memory. The same logic applies to passing objects between functions.

C++ treats the assignment operation as the value assignment (see Section 5.1) but has two derived types that one can use for the referential assignment: references and pointers (plus smart pointers in the standard library) (ISO/IEC 14882:2020). The initialisation of a reference (references cannot be reassigned) or a value assignment to a pointer is referential assignment of the referenced or pointed to object. Similarly, functions' arguments and return values can be passed by reference or as pointers. Thus, referential assignment is implemented through the value assignment. However, treating references as values and existence of different types of values in C++ may be confusing when it comes to which rules apply to a specific variable (e.g., a reference to an rvalue may itself be an lvalue) (Meyers, 2015).

Related Patterns

There is often a trade-off between the value assignment (see Section 5.1) and the referential assignment. The forces affecting these patterns are mostly the same, and the balance of benefits and liabilities dictates the choice.

The shallow copying, described in Section 5.1, combines the value assignment and the reference assignment.

The Assignable Once pattern (Batdalov and Nikiforova, 2018) can be implemented using the referential assignment. A reference passed to another context makes up an assignable slot that can be assigned later.

Formalisation

The referential assignment assumes that symbols as references are part of the program state and can refer to different addresses at different times. Then, the referential assignment is just the value assignment of references, and a separate formalisation is not required. It precisely reflects how referential assignment is typically *implemented* in programming languages. The described pattern provides a different *view* of this mechanism, but the underlying implementation and formalisation do not change.

5.3. Partial Assignment

Partial assignment arises when a part of a compound data structure should be changed.

Context

Modification of compound objects, such as arrays, records, and maps.

Problem

Processing logic defines a new value as a transformation of a previously calculated one, affecting only a part of the data structure. Copying the unaffected parts may be expensive and unnecessary.

Forces

Local transformation The transformation (or its single step) is restricted to a part of the data structure (e.g., one element of an array) and does not affect the other parts.

Dropping the old value The old value (before the transformation) may be unneeded afterwards.

Expensive copying Copying a compound type value may be expensive in terms of CPU and memory resources.

Immutable data types A data type may be immutable, i.e., prohibit modifications after creating a value.

Solution

For mutable types, locate the affected part of a data structure (e.g., an array element or a record field), create it if it does not exist. Force *Local transformation* should allow that. Treat the affected part as the target and apply either the value assignment (see Section 5.1) or the referential assignment (see Section 5.2). Figure 5.5 illustrates the partial assignment for arrays. Locating the affected part can be expressed directly in the syntax (e.g., `a[i] = b`, i.e., the i^{th} element is treated as an independent target in the assignment operation) or not (e.g., in Java, `map.put(key, value)`, i.e., it is an operation on the map as a whole and not on the map entry). The former case is further referred to as the assignable slot syntax.

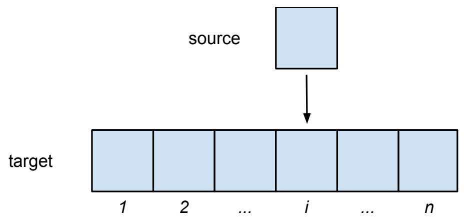


Figure 5.5. Partial assignment example

With mutable types, the partial assignment is possible only if the force *Dropping the old value* applies. Otherwise (i.e., if both the old and the new value are required), one cannot avoid copying the whole data structure.

The solution above does not apply to immutable types (force *Immutable data types*) because the state of an immutable object cannot change. An alternative solution is a copy-change operation: build a data structure that shares the unaffected part with the source and contains new values or references in the affected part. Figure 5.6 shows an example of changing the first element of a linked list.

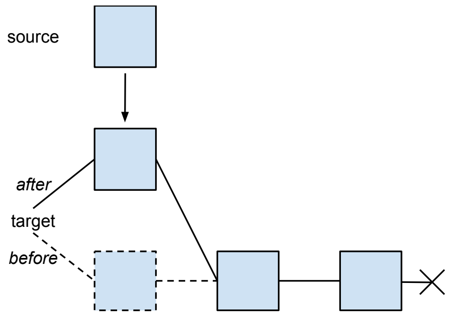


Figure 5.6. Copy-change for immutable types

Depending on the kind of modification, the copy-change operation may require extensive copying. For example, if one modified the last element of a linked list, it would require copying almost the whole list. On the other hand, immutable types do not require the force *Dropping the old value*: the copy-change operation allows keeping and using both the old and the new value.

Consequences

Benefits

Partial value reusing Non-changing parts of a compound data type are reused instead of building the whole structure from scratch.

Assignable slot syntax The `a[i] = b` syntax (when used) clearly shows the locality of a change.

Careful choice of algorithms Algorithms on mutable and (especially) immutable data types should be chosen with considering the conditions of the partial assignment applicability to avoid excessive copying.

Limitedness of the assignable slot syntax Assignable slots are usually simple pointers to the locations where a value can be stored, but not all partial assignment kinds allow defining such pointers. In particular, such pointers do not exist in the case of immutable types.

Ambiguity of the assignable slot syntax Since the partial assignment may involve creating the affected part (e.g., inserting a new key in a map), the exact meaning of the assignable slot syntax may not be obvious. For example, in C++, the statement `value = map[key]` may modify the map, and the statement `map[key] = func()` may leave the map modified even if the function call throws an exception.

Existing Languages

The partial assignment is typical for container and container-like objects that maintain some form of key-value association. Examples are arrays, lists, records (classes, structs), and maps in C++ (ISO/IEC 14882:2020), Java (Ora, 2023), JavaScript (ECMA-262), Scala (Odersky and Spoon, 2023), and other languages. The key-value (or index-value) association provides the necessary locality to apply the partial assignment. Thus, these types support assigning a value to a particular element or entry. It can be noted that sets, another popular type of containers, do not support the partial assignment for an obvious reason: even though we have locality, there is nothing to assign.

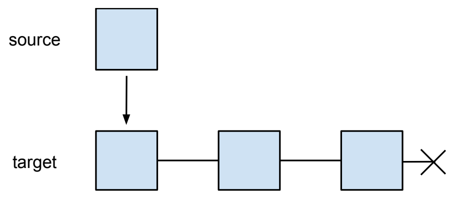
With immutable types, balancing the desire to do partial modifications with avoiding excessive copying may be complicated. For example, Scala standard library contains the Vector class: a sophisticated immutable counterpart of an array, which supports access by an arbitrary index (Odersky and Spoon, 2023). Internally, it is implemented as a tree with the branching factor 32 to reduce copying during a copy-change operation (Odersky and Spoon, 2023).

Related Patterns

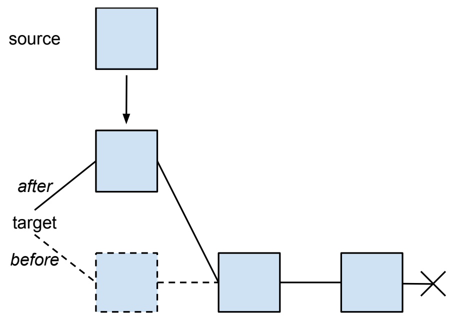
Implementation of the partial assignment includes applying the value assignment (see Section 5.1) or the referential assignment (see Section 5.2). When the affected part of the data structure is found, the remaining task is the same as for these two patterns.

On the other hand, one can treat the partial assignment as either the value assignment or the referential assignment to the whole data structure. When a value is assigned to an element of a mutable object, it is logically equivalent to calculating a new value of the whole object and storing it in the same memory location. A copy-change operation on an immutable object means calculating a new value, storing it at a different memory location but referencing it in the same variable.

Scala collections library creators describe this relationship as (partial) interchangeability of a mutable `val` (an unchangeable reference to an object of a mutable type) and an immutable `var` (a changeable reference to a value of an immutable type). Figure 5.7 illustrates this equivalence. However, the equivalence is not complete: some partial assignment operations on mutable Scala collections do not have direct equivalents and require more complicated syntax for immutable types (Batdalov and Ņikiforova, 2017).



(a) Mutable `val`: `target` always references the same object, but the state of the object changes



(b) Immutable `var`: the state of an object cannot change, but `target` may reference another object instead

Figure 5.7. Interchangeability of a mutable `val` and an immutable `var`.

The Iterator pattern (Gamma et al., 1995) can be used to locate the affected part (one or more elements) of the data structure.

The assignable slot used in the partial assignment may be implemented using the Assignable Once pattern (see Section 4.5).

Formalisation

The partial assignment is reducible to the ability of compound types to generate assignable slots. Subsections of Section 3 contain examples of such operations:

$$SeqIterateAssignable : Ref(Seq\ T) \to TraversableOnce\ AssignableOnce\ T$$

$$MapIterateAssignable : Ref(Map\ K\ V) \to$$

$$\to TraversableOnce\ (Pair\ K\ (AssignableOnce\ V))$$

$$MMapIterateAssignable : Ref(MultiMap\ K\ V) \to$$

$$\to TraversableOnce(Pair\ K\ (TraversableOnce\ (AssignableOnce\ V)))$$

$$LVarMutate : Ref(LabelledVariant\ T_1 \dots T_n) \to (AssignableOnce\ T_1 \to R) \to$$

$$\dots \to (AssignableOnce\ T_n \to R) \to R$$

$$OptMutate : Ref(Option\ T) \to (AssignableOnce\ T \to R) \to (Unit \to R) \to R$$

There are other operations of this kind. For example, most sequences support index-based operations. As discussed in Section 3.1, the generic definition of a sequence excludes them, but specific subtypes may support such operations. For such sequences (e.g., arrays), generating an assignable slot by index is possible. Other types may provide other ways of generating an assignable slot.

If an assignable slot referring to a part of a compound type is available, the partial assignment is performed by calling the *Assign* function described in Section 4.5.

5.4. Destructuring

Destructuring allows decomposing a compound data structure and assigning its elements to multiple targets in one statement. It is syntactic sugar and does not add new functionality. However, it may simplify code, especially when the source data structure comes from a call to another function.

Context

Multi-step computations, whose intermediate results are of compound data types.

Problem

Processing logic calculates a compound value but later uses its parts independently from each other. Assigning the value to a temporary variable and then assigning its parts to the variables used on subsequent steps is wordy and may involve unnecessary copies or moves.

Forces

Value or referential assignment semantics The semantics of destructuring may be of either the value assignment (when we are interested in the data held in the compound structure) or the referential assignment (when we are interested in their location).

Ignored values The following data processing may need only a part of the data held in the compound structure.

Arbitrary data structures The problem statement is general enough to be applied to arbitrary data structures, including user-defined types.

Solution

Allow the target of other assignment operators discussed in this paper to consist of multiple symbols. To make it obvious which symbol receives which part of the data, the target symbols should be organised in the same shape as the source data structure (e.g., `(a, b) = somePair`). In addition, a special symbol (that cannot serve as an actual identifier) or absence of a symbol can mark ignoring a part of the data (force *Ignored values*).

An optional extension that addresses force *Arbitrary data structures* is the opportunity to define the rules of destructuring when a new type is defined. If this extension is not supported, destructuring may be applied only to standard types.

Consequences

Benefits

Shorthand assignment to multiple targets Data may be extracted from a compound data structure in a single statement (instead of multiple statements that retrieve the required fields one by one).

Avoiding unnecessary copies With a single operator, the compiler is free to apply optimisation to avoid unnecessary copying or moving (e.g., the return value optimisation, discussed in Section 5.1). With an intermediate variable, it may be more difficult.

User-defined types If the optional extension is supported, destructuring applies to user-defined types as well.

Liabilities

Less readable code The destructuring syntax may look unusual or complicated because the result of a single expression does not frequently consist of multiple independent parts.

Implementation difficulty Destructuring is orthogonal to other choices of the type of assignment and thus multiplies the number of options that a compiler should support.

Separate support for each type Destructuring rules depend on the semantics and internal structure of a data structure and hardly can be defined in a generic way. Thus, they should be defined for each type separately. Many types logically should not allow destructuring at all.

Existing Languages

The destructuring assignment for standard data types (including sometimes standard generics, such as iterables) is supported, among others, in C++ (ISO/IEC 14882:2020), JavaScript (ECMA-262), TypeScript (Mic, 2023), and Python (Pyt, 2023).

Destructuring of arbitrary data types is supported, for example, in Scala and Kotlin:

- Kotlin supports the `component1()`, `component2()`, etc. operators, which are called for each target symbol in a row (Jet, 2023). This approach allows defining a way to destructure a user-defined type, but a given type should have a fixed number of components.
- Scala has a developed mechanism of pattern matching, which includes destructuring (if a value matches a pattern, then parts of the value may be extracted and assigned to the symbols used in the pattern). Any type with method `unapply` or `unapplySeq` (or both) can be used in pattern matching (Odersky et al., 2023). The `unapplySeq` method allows destructuring a type with a non-fixed number of components (e.g., an array).

Related Patterns

The assignment to individual variables during destructuring can be either the value assignment (see Section 5.1) or the referential assignment (see Section 5.2).

Formalisation

The present work uses Scala's formalisation of destructuring assignment, where methods that create an object from individual values (in Scala, `apply`) have counterparts doing the reverse operation (in Scala, `unapply` and `unapplySeq`). With such counterparts, it is possible to decompose an object into individual values and assign them to multiple assignable slots. The syntax of a language must also provide a way to write this operation down, but the syntactical issues are outside the scope of the present work.

The composing operations discussed in the present work are:

- $SeqCreate$ (Section 3.1),
- $SetCreate$ (Section 3.2),
- $MSetCreate$ (Section 3.3),
- $MapCreate$ (Section 3.4),
- $MMapCreate$ (Section 3.5),
- $LVarCreate_x$ (Section 3.6),
- $OptCreateValue$ (Section 3.7),
- $TraversableCreate$ (Section 4.4), and
- $MutTraversableCreate$ (Section 4.4).

Their decomposing counterparts are not included in the descriptions above to avoid cluttering the presentation. However, types that support the destructuring assignment should have them. The same is valid for composing and decomposing operations defined in specific subtypes.

5.5. Unboxing

Unboxing allows representing a series of calls returning boxed values (for example, calls to the API, each of which returns `Promise<T>`) as a simple sequence of operations. Similarly to destructuring, the unboxing assignment is only syntactic sugar but makes a program syntax shorter and more concise.

Context

Multi-step computations whose intermediate results are enclosed (boxed) in an object of another type. The prototypical example of such a type is a future value type (`Future` or `Promise`). However, the same pattern also applies to other types containing values (even simple containers, though applying the pattern to containers may be misleading).

Problem

Processing logic produces boxed values, for example, a future value (a value that may be available later). However, further processing does not depend on the boxing and depends only on the value itself. The current unavailability of the value only means that the computation should continue later when the value is available. In other words, the processing logic depends only on the enclosed values, and the boxing affects only the execution flow. Managing the boxing and the value in the same code creates much boilerplate code and distracts the attention from the central processing logic.

Forces

Single-value processing logic Unboxing is applicable when the processing logic depends only on the boxed values. If several values are boxed, each of them must be processed independently from others.

Unavailability of the value In the general case, the boxing types may not allow retrieving a value directly (the accessors either do not exist or return a value only under certain conditions, such as completing the operation that would return a future value). Instead, they accept a function that should be applied to the value when (or if) it is available. This function contains the subsequent steps of computation.

Long series of calls A function may contain multiple calls using data returned by the previous ones. Chaining such calls leads to difficult to read statements.

Collapsible types The types typically used with unboxing semantically allow flattening. For example, from the practical point of view, `Promise<Promise<Promise<T>>>` is not distinguishable from `Promise<T>` because we are interested in the final value. However, for some other types, it is not the case. For example, `List<List<T>>` is semantically different from `List<T>`.

Solution

Introduce an operator that looks like assigning a boxed value to a variable but, in fact, modifies the execution flow. All subsequent steps of computation are converted to a separate callback and passed to the boxing object. Thus, we have a naturally looking code that expresses the processing logic, but the actual execution runs according to the force *Unavailability of the value.*

Force *Collapsible types* requires an additional operation after the code block has been executed: flattening the result. Explanation of this requires understanding the types involved in unboxing. If the boxed type is `Promise<T>` (a future value of type `T`), then the unboxed variable will have type `T`. The rest of the code block takes the variable of type `T` and produces a value of another type `U`. Then the whole function will return a value of type `Promise<U>`. The same is true for other boxing types (not only for `Promise`s).

A series of multiple calls each of which returns a future value technically returns a repeatedly boxed type, such as `Promise<Promise<T>>`. However, this type does not make much practical sense and is immediately flattened to `Promise<T>`.

If non-collapsible types, such as `List`, are involved, we can theoretically imagine a situation when a list of lists is not flattened to a flat list. However, this option is not supported by the known uses.

Consequences

Benefits

Logical sequence The unboxing assignment represents the logical sequence of operations how the programmer imagines it.

Long series of calls simplification If long chains of calls are involved (force *Long series of calls*), unboxing dramatically simplifies the code and reduces nesting.

Liabilities

Difficulty to reason about the code The divergence between the written code and the final execution flow may hide what is happening during the code execution and hinder reasoning about the code.

Flattening vs non-flattening If a non-collapsible type is involved, it may not be obvious whether the result is flattened.

Existing Languages

The `await` operator in C# (ECMA-334), JavaScript (ECMA-262), TypeScript (Mic, 2023), and Python (Pyt, 2023) performs unboxing for the future value type (`Task`, `Promise`, or `Future`). These languages also allow using `await` with custom types (by defining specific methods that the compiler internally calls), but `await`'s semantics is still tightly coupled with future values.

Scala applies a more general case of unboxing with for-comprehensions that can be used with arbitrary types that have methods `map`, `flatMap` and `filter` (for example, all standard collections) (Odersky et al., 2023).

In the choice provided by force *Collapsible types*, all known uses prefer to flatten the result as much as possible. For example, a series of three sequential `await`ed calls in TypeScript will return `Promise<T>` (where `T` is the type of the actual value) and not `Promise<Promise<Promise<T>>>`. Similarly, Scala applies the `map` method only at the last translation step but `flatMap` at each preceding one.

Related Patterns

The assignment assumed in unboxing may be of the kinds described above (value assignment, referential assignment, partial assignment, or destructuring). Applying the unboxing pattern means that the value will become an argument to a function instead of the source of an assignment. However, as discussed in the introduction, the same patterns apply to passing values between contexts, so they are applicable in this case too.

Formalisation

The unboxing assignment is based on the transformation of a statement of the kind $x : T =_{unbox} y[: BoxT]; z$ to $y.flatMap/map(x : T => z)$. This (or equivalent) transformation is performed by the compiler. z can access x both before and after the transformation. The choice between $flatMap$ and map in all known uses is subject to the following rule: the last unboxing in a code block uses map, and all previous ones use $flatMap$. If the type of y does not support $flatMap$ or map when necessary, a compilation error occurs.

6. VALIDATION

The present work illustrates the capabilities provided by the described type system using the example of the author's implementation of an emulator of MIX, a mythical computer invented by Donald Knuth for his "The Art of Computer Programming" book series (Knuth, 1997). Despite being a 'toy' project, it shows potential improvements even in such an expressive language as Scala (Batdalov and Ņikiforova, 2017). Since higher expressiveness is associated with easier implementation of a solution, the expected practical consequence of the proposed improvements is reducing the implementation complexity. This section discusses the features of the developed type system in the context of how they could simplify the mentioned implementation.

Since a goal of the present work is to support code evolving (and not only simplify the code written once), the discussion of potential improvements is divided into three parts: static code simplification, how this system could have evolved from a simpler prototype, and how it can evolve in the future.

The discussion in the present section remains, to a certain degree, speculative. There are two main reasons for that: the present work describes a type system but not an actual programming language, and the considered code changes are different from the actual history of the system. However, despite that, this discussion demonstrates the potential aid that the described type system could bring.

6.1. Project Description

Donald Knuth invented the MIX imaginary computer for his book series "The Art of Computer Programming" (Knuth, 1997). MIX never existed as a physical machine but combines features of typical computers of the 1960s. "The Art of Computer Programming" contains a well-defined MIX hardware description, a system of commands, and an assembly language (MIXAL). Donald Knuth wrote most programs and answers to exercises in his masterpiece in MIXAL. Future editions of "The Art of Computer Programming" will replace MIX with a newer MMIX computer (Knuth, 2005). However, this work is still in progress, and MIX preserves its value for readers despite its outdated architecture.

A crucial feature of MIX is its incomplete determinism. According to Knuth's description, MIX can work as a binary or a decimal computer (or, more precisely, the number of possible values of a MIX byte can be arbitrary between 64 and 100), and a correct program should not depend on the byte size (Knuth, 1997). Similarly, the input/output operations are asynchronous, and programs should work correctly independently on the input/output speed (e.g., reading from the memory area where an input operation is storing data without checking that the operation has finished leads to undefined behaviour). This indeterminism creates challenges in programming in the low-level assembly language.

The author developed a web-based emulator of MIX (Batdalov and Ņikiforova, 2017). One of its primary design goals was handling the mentioned indeterminism. Unlike other emulators of MIX known to the author, which are binary only, the author's one allows executing programs in the binary or the decimal mode and verifies the correctness of input/output synchronisation. Another supported feature is the opportunity to record every state during program execution and switch between them forward and backward. These features provide program correctness verification tools, differentiating the developed emulator from others.

As the primary implementation language, the author chose Scala. According to the author's Master's thesis research (Batdalov, 2017), Scala is one of the most expressive modern languages, which makes it a natural choice for studying the boundaries of its expressiveness. However, the author later rewrote the front-end part in TypeScript due to difficulties in interaction between Scala-transpiled code and JavaScript front-end libraries. The source code of the author's implementation is available at `https://github.com/linnando/MIXEmulator`. A working copy of the emulator is available at `https://www.mix-emulator.org`.

Emulator Design Overview

The emulator consists of the web-based front-end, written in TypeScript, and the back-end, written in Scala and transpiled into JavaScript using Scala.js. Other front-ends (e.g., a JVM-based one) may be added as future extensions, thanks to the cross-compilation facilities of Scala. The back-end part, in turn, consists of the virtual machine and the assembler. The assembler uses the virtual machine because "The Art of Computer Programming" defines the translation of MIXAL programs in terms of the MIX computation model (e.g., an assembler should evaluate a MIXAL arithmetic expression to the execution result of a corresponding MIX subprogram) (Knuth, 1997). Figure 6.1 shows these main building blocks.

The front-end and the assembler implementations are straightforward and not discussed in detail. However, the virtual machine implementation according to the described requirements is somewhat tricky. The design applied to achieve these goals is described further.

The first complication is byte size variability. The MIX computation model operates several data types: byte, index (the contents of an index register), word (the contents of the accumulator or the extension register or of a memory cell), and double word (the contents of the accumulator and the extension register taken together) (Knuth, 1997). The relationship between the types is stable, e.g., an index consists of a sign and two bytes, and a word consists of a sign and five bytes (MIX does not use the two's complement signed integer representation, which is standard for modern computers). However, the byte size is variable: a byte may hold 64 to 100 values. Changes in the byte size entail corresponding changes in the other type sizes.

Figure 6.2 shows the classes representing these primitive data types in the author's emulator. Each type has an abstract class and two implementations for the two supported

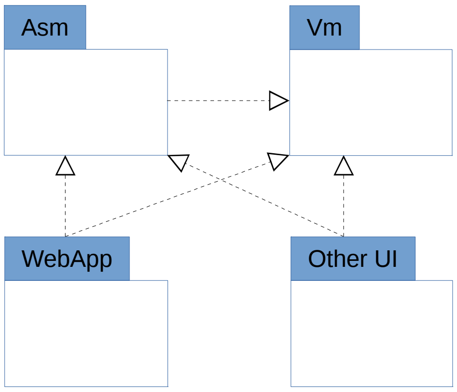


Figure 6.1. Emulator building blocks.

emulator modes: binary (a byte may hold 64 values) and decimal (100 values). Having multiple implementations of an abstract concept is typical in object-oriented software.

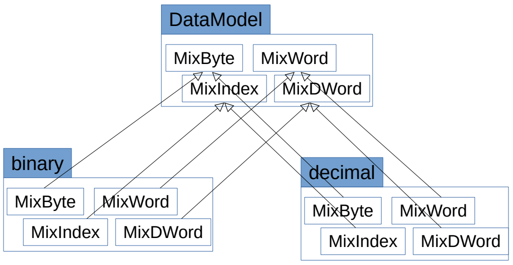


Figure 6.2. Mode-dependent data types in MIX emulator.

However, unlike in usual object-oriented class hierarchies, the mentioned implementations are not interchangeable. It is impossible to use the binary implementation of one type with the decimal implementation of another. For example, a binary memory cell cannot store a decimal word because the latter may contain a value that is too big. It means abstract types cannot have an interface defined in terms of abstract types. It would allow intermixing implementations for different modes, which would be wrong.

Instead, the author's implementation uses Scala's family polymorphism, a mechanism for implementing families of interdependent data types (Odersky et al., 2006). It ensures that types are used only with members of their family. Its implementation involves the following components:

1. An abstract class (the abstract model) should contain declarations of the abstract data types, forming the abstraction of the type family.

2. The same abstract class should contain type variables, which represent not yet defined concrete data types. The types represented by the type variables inherit from the abstract data types.

3. Declarations of the abstract data type interfaces (method parameters and return values) may include the declared type variables.

4. A singleton object (the concrete model, one per family) should contain concrete data types implementing the previously defined abstract data types. Each type variable gets an implementation there.

5. The operations on the concrete types should agree with the previously declared abstract interfaces, substituting the corresponding concrete types for the type variables.

For example, the abstract model defines the `AbstractMixWord` data type and the `I` type variable (as well as others). `I` inherits from `AbstractMixIndex`. Extracting the address part from a MIX word is defined as a method of `AbstractMixWord`: `def getAddress: I`. It means that an implementation of `AbstractMixWord` should define the `getAddress` operation, which will return the implementation of `AbstractMixIndex` defined in the same concrete model. The binary implementation returns the binary `MixIndex`, and the decimal implementation returns the decimal `MixIndex`. Thus, classes interact with classes of the same family, and the emulator can easily switch between the binary and the decimal families depending on the current mode.

Another design complication is related to input/output synchronisation. A MIX program can initiate an input/output operation, but the execution continues without waiting for the operation to finish (Knuth, 1997). Eventually, an input operation stores data from the device in memory, and an output operation does the opposite, but the moment when it will happen is unknown. Before an input/output operation is complete, using the same memory (e.g., reading from a memory area where a concurrent input operation is storing data) may lead to undefined behaviour (Knuth, 1997). A correct program should consider this factor and synchronise the main execution flow with the input/output operations.

However, synchronisation correctness is challenging to verify in an emulator, particularly during step-by-step execution. Due to the high performance of modern computers, any input/output operation will likely have finished before the program proceeds to the next step. Therefore, synchronisation errors may not manifest themselves during debugging.

Therefore, the developed emulator aims to verify that a program guarantees that the execution result does not depend on the speed of input/output operations. The verification logic applies memory locks inspired by SQL data locks (ISO/IEC 9075-2:2023). When a program initiates an input operation, which reads data from a device and stores them in memory, the emulator prohibits any attempt to write to the same memory area or read from it before the operation has been completed (an exclusive lock). Similarly, when a program starts writing memory data to a peripheral device (an output operation), the emulator prohibits writing to the same memory area. However, reading from the same memory area is allowed (a shared lock). Fulfilment of the conditions imposed by the locks guarantees that the program gives the same result independently of the input/output operations speed.

The emulator applies corresponding exclusive or shared locks when an input/output operation begins and stores them as part of the memory state. If a program tries to perform a prohibited action before the operation is guaranteed to finish, the emulator raises an exception. After the program has checked the operation completion, the emulator updates the memory or device contents and releases the lock.

Detection of the moment when the emulator should release a lock is a separate problem. Unlike multi-threading environments in modern programming languages, MIX does not have a special synchronisation command. The author's emulator recognises the 'busy wait' synchronisation pattern: a conditional jump to the current address when the addressed device is busy (in MIXAL notation, `JBUS *(DEVICE_NUMBER)`). It means the program stays in the loop until all input/output operations on this device have finished. The emulator treats this command as a signal to release all locks related to the corresponding device.

The final design feature worth emphasising is tracking virtual machine states (e.g., memory and register contents) during the program execution. The emulator allows the user to go back during the execution and inspect previous machine states (the feature inspired by Online Python Tutor (Guo, 2013)). It requires recording the machine state after each instruction and the ability to switch between them back and forth.

The author's emulator uses immutable types to reduce memory requirements for storing all encountered states. Immutable data structures typically use shared memory storage, in which two compound objects may share the memory that keeps the same data. Figure 2.1 schematically shows a simplified example of this concept.

Being a functional language, Scala provides rich support for immutable data structures. Scala's standard library contains a variety of immutable data structures for different purposes (Odersky and Spoon, 2023). The language supports various operations that facilitate using immutable data structures, for example, the automatically defined method copy on case classes and expansion of the shortcut assignment operators (e.g., `+=`, `-=`) so that they work equivalently for both mutable and immutable data structures (Odersky et al., 2023). These operations provide a simple and concise syntax similar to one for mutable data structures. However, instead of mutating a data structure, they create another one with the mutated data. It is precisely what the emulator needs because MIX processor instructions mutate the machine state, but storing all states requires emulating the mutation with a sequence of immutable states. These features make Scala's immutable types extremely useful in implementing the emulator.

The requirements defined for the emulator posed a few challenges in the emulator design and implementation. Scala's advanced features proved to be very helpful in solving these challenges. However, the implementation experience also discovered a few potential improvements that could make this task easier. The subsequent sections discuss how the types described in the present work could serve this task.

6.2. Code Simplification

The structural types described in Section 3 have feature parity for mutable and immutable implementations. Section 3 describes only the most generic compound types, but it is assumed that the definition of more specific types should follow the same approach. Among other things, the feature parity means that both mutable and immutable types support the partial assignment (assigning to a part of a compound object; see Section 5.3).

Partial assignment to a mutable compound type is virtually universally supported. It provides a convenient way to change only a part of a compound object (e.g., `values[index] = new_value` or `obj.field = new_value`) and leave the remaining part intact. This operation reflects an essential feature of a mutable compound object: the partial assignment is reducible to the assignment to a component. A component (e.g., an element or a field) of a mutable compound object is a mutable object itself, so we can take the address of the corresponding piece of memory and use it as the assignment destination. Sometimes, this approach leads to unexpected side effects (e.g., in C++, `map[key] = get_value()` will create an entry in the map even if `get_value()` raises an exception and the assignment does not happen). However, despite these edge cases, partial assignment to a mutable compound type is simple and usually works out of the box.

Immutable compound types are different. Even though the partial update operation is natural for them, too, it is not as simple as updating a component. Instead, partial assignment to an immutable compound type involves rebuilding the whole data structure, as shown in Figure 6.3. We cannot simply update a node in the tree (in this example, assigning 9 instead of 1); we need to build a new tree with the updated node and store a reference to the new tree.

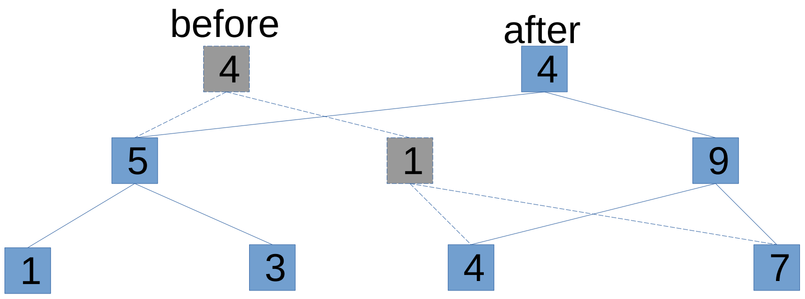


Figure 6.3. Updating an immutable tree.

Thus, changing the state of the MIX Emulator requires complicated statements like

```
copy(forwardReferences = forwardReferences.updated(symbol,
  forwardReferences(symbol) :+ counter))
```

If immutable data structures supported partial assignment directly, as described in the present work, the state could be updated more easily. The present work proposes partial assignment to immutable types using assignable slots (see Section 4.5). For a mutable compound data type, the assignable slot is usually a direct pointer to the corresponding element of the compound object. For an immutable type, an assignable slot has a more complicated behaviour: assigning a value to this slot creates a new instance of the compound type and stores a reference to the new instance in the same variable. Figure 6.4 shows the sequence diagram of this process.

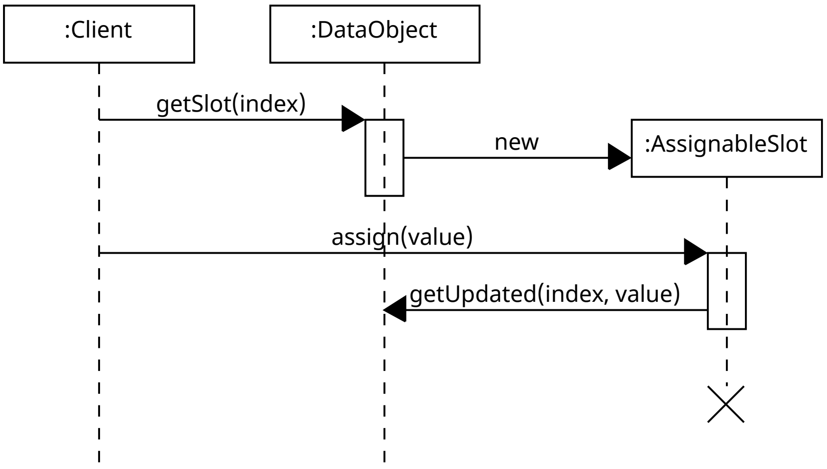


Figure 6.4. Sequence diagram of a partial assignment to an immutable type.

Then the code updating the state of the emulator could look as simple as in the mutable case:

```
forwardReferences(symbol) :+= counter
```

6.3. Evolving from a Simpler Prototype

This section discusses simplifications that could have appeared if the emulator had initially had a simplified design and had evolved to the current state later. This mental experiment is not pure because, in reality, the application did not pass through these stages. However, this way of development is typical for various applications, so reasoning about such changes is possible and sound.

As described in Section 6.1, most emulators of MIX work only as binary machines, but the author's emulator supports binary and decimal modes. Let us consider how the software would have evolved if the emulator had initially been binary only and then was extended to support the decimal mode.

As discussed in Section 6.1, the emulator's modes are supported by having independent implementations of a few basic types:

- `MixByte` (one byte),

- `MixIndex` (the contents of an index register, two bytes and a sign),
- `MixWord` (the contents of the accumulator register, the extension register, or a memory cell, five bytes and a sign),
- `MixDWord` (the contents of the accumulator and the extension registers taken together, ten bytes and a sign),
- `RegisterState` (the state of all virtual machine registers),
- `MemoryState` (the state of the virtual machine memory), and
- `VirtualMachineBuilder` (the class that allows creating a virtual machine in a series of independent operations).

Each of these classes has a binary and a decimal version as shown in Figure 6.2. The emulator chooses the binary or decimal version depending on the current mode.

If the only difference between these versions were in their implementation, introducing a new mode would be easy: it would be sufficient to treat the existing (binary) classes as implementations, declare abstract interfaces that those implementations implement, and define new concrete classes for the decimal mode. Many programmers have developed a habit of doing the first two steps from the beginning, even when only one implementation exists. Introducing new independent implementations is familiar and does not cause problems.

However, the situation in this project is more complicated because the interfaces of these classes are different, too. For example, getting a memory cell value from the binary `MemoryState` will always return a binary `MixWord`, and the function storing a value in an index register accepts only a binary `MixIndex`. Covariant return types can address the former issue: the getter function could be declared as `def get(address: Short): MixWord` in the abstract class and as `def get(address: Short): binary.MixWord` in the concrete binary class. However, this does not work with the latter issue because covariant parameter types are unsafe (see Section 1.2). Therefore, a setter function declared as `def updatedI(index: Int, value: MixIndex): RegisterState` in the abstract class would have to accept an arbitrary implementation of `MixIndex`. However, passing a decimal `MixIndex` to its binary implementation could fail as the decimal index may contain a value too large for the binary register. So, the traditional interface-implementation relationship would require checking the passed value type at the run time, which is error-prone and inefficient. As a result, the usual way of defining one interface and multiple implementations for different modes is not sufficient for this problem.

The current implementation addresses this problem by using family polymorphism in Scala (Odersky et al., 2006). The `ProcessingModel` class encapsulates all mode-dependent classes and allows using them as type variables in arbitrary declarations as described in Section 6.1. There are different implementations of `ProcessingModel` for the binary and the decimal modes. This mechanism provides compile-time safety and run-time efficiency. However, it has a significant drawback. The whole family of inter-related classes (e.g., the binary implementation of `ProcessingModel`) must be contained in one singleton object. For a non-trivial logic, it creates enormous objects, violating the single responsibility principle (Batdalov and Ņikiforova, 2017).

The interpretation of inheritance in terms of existential types, discussed in Section 2.7, provides an alternative approach to this problem. This approach allowed subtyping method parameters in the case of depth subtyping, provided that the relationship between parameter and return types guarantees type safety. The same can be done for family polymorphism. The only difference is that all types in the family should be existential types (not only the class encompassing a function). Then, the type system can check the usage correctness statically.

Here is how the rules given in Section 2.7 could apply to the function storing a value in an index register:

- In the current implementation, this function is a member of the `AbstractRegisterState` class, and its signature is `def updatedI(index: Int, value: I): RS`. It means that the function accepts the register index and a value of the index type and returns the updated state of registers (`I` and `RS` are the type variables for implementations of `MixIndex` and `RegisterState` correspondingly). The distinction between `RegisterState` and `AbstractRegisterState` is required to fulfil Scala's typing rules; in the proposed system, one class should be sufficient for both purposes.
- Since the return type of `updatedI` is the class where this function is defined (the register state), the return type is an existential type. It means that the return type is unknown until the concrete type of the second parameter is specified.
- The binary implementation of `RegisterState` should define the function that accepts a binary index value and returns a binary `RegisterState`. The decimal implementation should define the function that accepts a decimal index value and returns a decimal `RegisterState`.
- At the call site, the return type is inferred based on the second argument type. An attempt to call the function with a decimal index value for a binary register state (or vice versa) will cause a compilation error.

This way, the interface and multiple implementations can be defined in related but separate classes, making adding a new implementation as convenient as in the typical object-oriented case.

In more complicated cases, programming such families would be challenging because type safety guarantees require defining types carefully. Careless declarations could easily lead to extending return types (i.e., if the type system cannot prove that a method returns a binary word, its return type will be inferred as an abstract `MixWord`, making such a method virtually unusable). However, with proper typing, this approach can simplify family polymorphic code.

Another potential simplification in the described scenario (adding the decimal mode to the initially binary-only emulator) is the treatment of singletons and executors. The current implementation passes the processing model (binary or decimal) as a parameter to functions that use it. In a binary-only emulator, basic objects of the processing model would probably have been globally accessible as singletons unless the developer had anticipated the emergence of an alternative processing model. Moving from such a design to one with multiple processing models would require significant refactoring in different parts of the code.

As an alternative, Section 4.2 describes executors, which, among other things, hold singletons. Singletons are reachable as global objects, but the program can replace them during execution. Then, the code that chooses the emulator mode could set the processing model in the executor, and the rest of the code could access it as before.

6.4. Potential Future Development

This section describes how features proposed in the present work could facilitate future development of the author's emulator of MIX. As in the previous section, this discussion is, to a certain degree, abstract because the emulator does not support the described improvements yet. However, the described type system could make their implementation easier.

A possible improvement of the emulator is related to its interpretation of input/output operations determinacy. As described in Section 6.1, the emulator verifies the correctness of input/output synchronisation to ensure that the program executes correctly independently on the input/output operations speed. The emulator locks a memory chunk used for an input/output operation and raises an exception if the program executes an instruction with an undefined outcome before the input/output operation is guaranteed to be complete. Figure 6.5 schematically shows the execution flow when an input/output operation is involved.

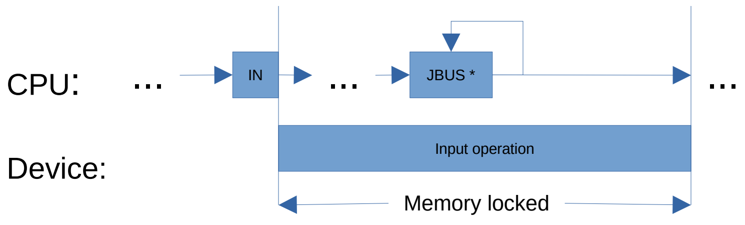


Figure 6.5. MIX Emulator input/output execution flow.

However, a drawback of this approach is that it requires absolute determinacy (except for the numerical input/output speed). The author's emulator recognises only the busy-wait synchronisation pattern (staying at the same address while a device is busy). MIX has two input/output-related conditional instructions: `JRED` (jump if a device is ready) and `JBUS` (jump if a device is busy) (Knuth, 1997). However, in the author's emulator, the former raises an exception always, and the latter in all cases except if the jump target coincides with the current address. The reason is that `JBUS *(DEVICE_NUMBER)` (stay at the same address if the device is busy) is the only conditional operation after which the emulator state is deterministic. Even though other uses of `JRED` and `JBUS` are possible, the emulator cannot determine the exact virtual machine state after their execution.

As a result, only a program that does the same independently of the input/output speed can pass the synchronisation checks. For example, the emulator will reject a program that periodically types `WAITING...` on the terminal until an input/output operation finishes. Since it is not known how many times this line should be written, the behaviour of this program is non-deterministic. Such a strong condition is too restrictive and invalidates many reasonable programs.

The emulator could be more permissive if it could track state boundaries instead of the exact state. For example, a particular memory chunk may contain either the values it had before an input operation or the values read from the device (and the emulator does not know for sure which of the two values is there before input/output synchronisation). As long as the difference in possible states stays reasonable, the execution continues (what exactly is reasonable is a matter of choice). Thus, the emulator could execute some non-fully deterministic programs but still identify erroneous behaviour.

The same idea of state boundaries could potentially be helpful with the indeterminacy of the byte size. Instead of executing a program in the binary and decimal modes separately, the emulator can track the calculation results and know that some bytes hold deterministic values and values of other bytes are unknown. It is how 'The Art of Computer Programming' presents the program execution (Knuth, 1997). This way, one program run is sufficient to find possible errors and provides better coverage than just two modes (binary and decimal).

However, tracking state boundaries is complicated and unreasonable in most cases because most programs are deterministic. The emulator could track the exact state as long as possible and switch to the state boundaries mode when the behaviour is not deterministic anymore (and then back when the state becomes deterministic again). The logic of tracking the exact state and tracking state boundaries would be significantly different, so it makes sense to implement it in different classes. It is an example of the State pattern (Gamma et al., 1995). Section 2.9 describes chameleon objects, which would be helpful in this case. The deterministic and non-deterministic implementations of the virtual machine would be different classes that can convert to each other when needed. From the client's point of view, there would be only one abstract virtual machine, though its behaviour is actually different at different stages.

CONCLUSION

The present work is devoted to establishing a link between design patterns and expressive opportunities of programming languages. As known from the literature and experience, patterns-based design solutions are often associated with higher complexity, even though their original goal was the opposite: handle complexity. The complexity of a solution may be inherent to a problem, but the complexity of patterns-based solutions is often higher than necessary. The proposed hypothesis is that the excessive complexity is partially caused by the insufficient expressiveness of programming languages, which struggle to convey mental constructs represented by patterns. In order to address this problem, the present work proposes a set of generalisations describing general cases of various programming languages' constructs and formalises them using type-theoretical formalisms.

The tasks defined for the present work are fully fulfilled:

1. Analysis of programming languages' evolution trends and previously described difficulties in design patterns' implementation demonstrated problematic points even in modern programming languages.

2. Based on this analysis, the objectives and requirements for the desired type system are formulated.

3. The patterns of data composition and basic computation primitives, which provide generalisations of commonly used programming languages' constructs, are described.

4. The described patterns are formalised using type-theoretical apparatus.

5. An emulator of the MIX computer is implemented in Scala to be used as an example practical project.

6. It is demonstrated how the proposed types could be beneficial in implementing the example project and its evolution.

The main result of the present work is the developed system of types representing basic structural and computational patterns. The patterns describe general cases of typical constructs in programming languages (though actual languages may currently support a pattern in whole or only its special cases) and thus provide flexibility in modelling programmers' thoughts. Possession of such primitives in real programming languages could facilitate the implementation of design patterns and, more importantly, further development of an existing system.

Additional results of the work are:

1. The pattern-form descriptions of the described constructs explain how and why these constructs are used.

2. The list of known uses compares and contrasts implementations of the described patterns in various languages. It facilitates finding equivalent or similar constructs in different languages and understanding their limitations.

3. The methodology of describing language or library primitives as patterns to cover general cases and then formalising them as types can also be used for other constructs not covered by the present work.

4. The developed emulator of MIX can be used in learning. It has certain benefits compared with other emulators, such as support for the decimal mode (in addition to the binary one) and verification of input/output synchronisation correctness.

The following conclusions can be made based on the conducted study:

1. Programming languages' constructs often represent only special cases of general patterns they implement. However, more expressive languages can support these patterns in their complete form.

2. More general cases of language constructs can be identified by describing them as patterns.

3. The described patterns are suitable for type-theoretical formalisation, which potentially allows them to serve as language constructs.

4. Even most expressive programming languages, such as Scala, have room for improvement concerning the described patterns.

The work can be continued in the following directions:

1. The same methodology can be applied to other constructs. For example, the present work does not cover such essential aspects of a programming language as object lifecycle and memory management.

2. The general patterns described in the present work can be implemented in new and existing programming languages.

Bibliography

Martin Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer New York, 2012. ISBN 978-1-4419-8598-9.

Moez A. AbdelGawad. A comparison of NOOP to structural domain-theoretic models of object-oriented programming. Preprint available at https://arxiv.org/abs/1603.08648, 2017.

Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Angel Shlomo. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press USA, 1977. ISBN 978-0-19-501919-3.

Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. C++ in-depth series. Addison-Wesley, 2001. ISBN 978-0-201-70431-0.

Apache Hadoop 3.3.6. Apache Software Foundation, 2023a.

Apache Spark 3.5.0. Apache Software Foundation, 2023b.

The Swift Programming Language (5.9). Apple, Inc., 2023. URL `https://docs.swift.org/swift-book/documentation/the-swift-programming-language/`.

Pavol Bača and Valentino Vranić. Replacing object-oriented design patterns with intrinsic aspect-oriented design patterns. In *Proceedings of the 2nd Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC)*, pages 19–26. IEEE, 2011. doi: 10.1109/ecbs-eerc.2011.13.

Ruslan Batdalov. Inheritance and class structure. In Pavel P. Oleynik, editor, *Proceedings of the First International Scientific-Practical Conference Object Systems — 2010*, pages 92–95, 2010. URL `https://cyberleninka.ru/article/n/inheritance-and-class-structure/pdf`.

Ruslan Batdalov. Is there a need for a programming language adapted for implementation of design patterns? In *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP '16)*, pages 34:1–34:3. Association for Computing Machinery, 2016. ISBN 978-1-4503-4074-8. doi: 10.1145/3011784.3011822.

Ruslan Batdalov. Comparative analysis of object-oriented programming languages in the context of language expressiveness. Master's thesis, Riga Technical University, 2017.

Ruslan Batdalov and Oksana Nikiforova. Towards easier implementation of design patterns. In *Proceedings of the Eleventh International Conference on Software Engineering Advances (ICSEA 2016)*, pages 123–128. IARIA, 2016.

Ruslan Batdalov and Oksana Ņikiforova. Implementation of a MIX emulator: A case study of the Scala programming language facilities. *Applied Computer Systems*, 22(1): 47–53, 2017. doi: 10.1515/acss-2017-0017.

Ruslan Batdalov and Oksana Nikiforova. Three patterns of data type composition in programming languages. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs (EuroPLoP '18)*, pages 32:1–32:8. Association for Computing Machinery, 2018. ISBN 978-1-4503-6387-7. doi: 10.1145/3282308.3282341.

Ruslan Batdalov and Oksana Nikiforova. Elementary structural data composition patterns. In *Proceedings of the 24th European Conference on Pattern Languages of Programs (EuroPLoP '19)*, pages 26:1–26:13. Association for Computing Machinery, 2019. ISBN 978-1-4503-6206-1. doi: 10.1145/3361149.3361175.

Ruslan Batdalov and Oksana Nikiforova. Patterns for assignment and passing objects between contexts in programming languages. In *Proceedings of the 26th European Conference on Pattern Languages of Programs (EuroPLoP '21)*, pages 4:1–4:9. Association for Computing Machinery, 2021. ISBN 978-1-4503-8997-6. doi: 10.1145/3489449.3489975.

Ruslan Batdalov, Oksana Nikiforova, and Adrian Giurca. Extensible model for comparison of expressiveness of object-oriented programming languages. *Applied Computer Systems*, 20(1):27–35, 2016. doi: 10.1515/acss-2016-0012.

Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1–2):70–118, October 2005. ISSN 0004-3702. doi: 10.1016/j.artint.2005.05.003.

Judith Bishop. *C# 3.0 Design Patterns*. O'Reilly Media, Sebastopol, CA, USA, 2008. ISBN 978-0-596-52773-0.

Grady Booch. The well-tempered architecture. *IEEE Software*, 24(4):24–25, 2007. ISSN 0740-7459. doi: 10.1109/ms.2007.122.

Jan Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998. ISSN 0896-8438.

Thouraya Bouabana-Tebibel, Stuart H. Rubin, and Miloud Bennama. Formal modeling with SysML. In C. Zhang, J. Joshi, E. Bertino, and B. Thuraisingham, editors, *Proceedings of the 13th IEEE International Conference on Information Reuse and Integration (IEEE IRI)*, pages 340–347. IEEE Syst Man & Cybernet Soc (IEEE SMC); Soc Informat Reuse & Integrat (SIRI); IEEE, 2012. ISBN 978-1-4673-2284-3.

Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing*, volume 4 of *Pattern-Oriented Software Architecture*. Wiley, 2007a. ISBN 978-0-470-06530-3.

Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture, On Patterns and Pattern Languages*, volume 5 of *Pattern-Oriented Software Architecture*. Wiley, 2007b. ISBN 978-0-470-51257-9.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, A System of Patterns*, volume 1 of *Pattern-Oriented Software Architecture*. Wiley, 2013. ISBN 978-1-118-72526-9.

Travis Carlson and Eric Van Wyk. Type qualifiers as composable language extensions for code analysis and generation. *Journal of Computer Languages*, 50:49–69, 2019. ISSN 2590-1184. doi: 10.1016/j.jvlc.2018.10.008.

Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135, 1989.

James O. Coplien. *Software Patterns*. SIGS management briefings. SIGS, 1996. ISBN 978-1-884842-50-4.

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction To Algorithms*. MIT Press, 2001. ISBN 978-0-262-03293-3.

Ivica Crnković, Severine Séntilles, Aneta Vulgarakis, and Michel R. V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615, 2011. ISSN 0098-5589. doi: 10.1109/tse.2010.83.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991. ISSN 0164-0925. doi: 10.1145/115372.115320.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

ECMA-335. *Common Language Infrastructure (CLI)*. Ecma International, sixth edition, 2012. Standard.

ECMA-404. *The JSON Data Interchange Syntax*. Ecma International, second edition, 2017. Standard.

ECMA-334. *C# Language Specification*. Ecma International, sixth edition, 2022. Standard.

ECMA-262. *ECMAScript® 2023 Language Specification*. Ecma International, 14th edition, 2023. Standard.

Scala API Docs. École Polytechnique Fédérale de Lausanne, 2023. URL `https://www.scala-lang.org/api/3.3.1/`.

Francesca Arcelli Fontana, Stefano Maggioni, and Claudia Raibulet. Understanding the relevance of micro-structures for design patterns detection. *Journal of Systems and Software*, 84(12):2334–2347, 2011. ISSN 0164-1212. doi: 10.1016/j.jss.2011.07.006.

Francesca Arcelli Fontana, Stefano Maggioni, and Claudia Raibulet. Design patterns: a survey on their micro-structures. *Journal of Software-Evolution and Process*, 25(1): 27–52, 2013. ISSN 2047-7481. doi: 10.1002/smr.547.

Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. *SIGPLAN Not.*, 34(5):192–203, May 1999. ISSN 0362-1340. doi: 10.1145/301631.301665.

Martin Fowler. Inversion of control. Available at https://www.martinfowler.com/bliki/InversionOfControl.html, June 2005.

Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2012. ISBN 978-0-13-306521-3.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995. ISBN 978-0-201-63361-0.

Joseph Gil and David H. Lorenz. Design patterns and language design. *Computer*, 31(3): 118–120, 1998. ISSN 0018-9162. doi: 10.1109/2.660196.

David Glasser. An interesting kind of JavaScript memory leak. Available at https://blog.meteor.com/an-interesting-kind-of-javascript-memory-leak-8b47d2e7f156, August 2013.

The Go Programming Language Specification. Google, 2023.

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification. Java SE 8 Edition*, 2015.

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. *The Java Language Specification. Java SE 21 Edition*, 2023.

gRPC documentation. gRPC Authors, 2021. URL `https://grpc.io/docs/`.

Philip J. Guo. Online Python Tutor: Embeddable web-based program visualization for CS education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1868-6. doi: 10.1145/2445196.2445368.

Christian Haack, Erik Poll, Jan Schäfer, and Aleksy Schubert. Immutable objects for a Java-like language. In Rocco De Nicola, editor, *Programming Languages and Systems: Proceedings of the 16th European Symposium on Programming (ESOP'2007)*, volume 4421 of *Lecture Notes in Computer Science*, pages 347–362, Berlin, Heidelberg, July 2007. Springer. ISBN 978-3-540-71316-6. doi: 10.1007/978-3-540-71316-6_24.

Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. *Scala Futures and Promises*, 2023. URL `http://docs.scala-lang.org/overviews/core/futures.html`.

Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. *ACM Sigplan Notices*, 37(11):161–173, 2002. ISSN 0362-1340. doi: 10.1145/583854.582436.

Huahai He and Ambuj K. Singh. Query language and access methods for graph databases. In Charu C. Aggarwal and Haixun Wang, editors, *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*, pages 125–160. Springer US, Boston, Massachusetts, USA, 2010. ISBN 978-1-4419-6045-0. doi: 10.1007/978-1-4419-6045-0_4.

Kevlin Henney. Null object. In *Proceedings of the Seventh European Conference on Pattern Languages of Programming, EuroPLoP*, 2002.

Tony Hoare. Null references: The billion dollar mistake. Available at https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare, August 2009.

Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

ISO/IEC 7185:1990. *Information technology — Programming languages — Pascal*. ISO/IEC, 1990. Standard.

ISO/IEC 14882:2011. *Information technology — Programming languages — C++*. ISO/IEC, 2011. Standard.

ISO/IEC 9899:2018. *Information technology — Programming languages — C*. ISO/IEC, 2018. Standard.

ISO/IEC 14882:2020. *Information technology — Programming languages — C++*. ISO/IEC, 2020. Standard.

ISO/IEC 9075-2:2023. *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*. ISO/IEC, 2023. Standard.

Kotlin Language Documentation 1.9.0. JetBrains, 2023. URL `https://kotlinlang.org/docs/kotlin-reference.pdf`.

Rod Johnson, Juergen Hoeller, Keith Donald, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Alef Arendsen, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, Mark Fisher, Sam Brannen, Ramnivas Laddad, Arjen Poutsma, Chris Beams, Tareq Abedrabbo, Andy Clement, Dave Syer, Oliver Gierke, Rossen Stoyanchev, Phillip Webb, Rob Winch, Brian Clozel, Stephane Nicoll, Sebastien Deleuze, Jay Bryant, and Mark Paluch. *Spring Framework Documentation 6.0.12*. VMWare, Inc., 2023.

Peter Kelly. *Applying Functional Programming Theory to the Design of Workflow Engines*. PhD thesis, University of Adelaide, January 2011.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the European conference on object-oriented languages (ECOOP '97)*, pages 220–242, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69127-3.

Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture, Patterns for Resource Management*, volume 3 of *Pattern-Oriented Software Architecture*. Wiley, 2013. ISBN 978-1-118-72523-8.

Oleg Kiselyov. Iteratees. *Functional and Logic Programming*, pages 166–181, 2012. doi: 10.1007/978-3-642-29822-6_15.

Donald E. Knuth. *The Art of Computer Programming: Fundamental algorithms*. Addison-Wesley series in computer science and information processing. Addison-Wesley, 1997. ISBN 978-0-201-89683-1.

Donald E. Knuth. *The Art of Computer Programming, Volume 1, Fascicle 1: MMIX — A RISC Computer for the New Millennium*. Addison-Wesley, 2005. ISBN 978-0-201-85392-6.

Scala 3 Reference. LAMP/EPFL, 2023. URL `https://docs.scala-lang.org/scala3/reference/index.html`.

Craig Larman. *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development*. Pearson, 2016. ISBN 978-93-325-5394-1.

Chris Lattner and Vikram S. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO 2004)*, pages 75–86, New York, NY, USA, 2004. IEEE. doi: 10.1109/CGO.2004.1281665.

António Leitão and Sara Proença. On the expressive power of programming languages for generative design: the case of higher-order functions. In *Proceedings of the 32nd International Conference on Education and Research in Computer Aided Architectural Design in Europe (eCAADe)*, pages 257–266, 2014.

George Leontiev, Eugene Burmako, Jason Zaugg, Adriaan Moors, Paul Phillips, Oron Port, and Miles Sabin. SIP-23 - literal-based singletone types. Scala Improvement Proposal, 2019.

Hui Lin and Bo Huang. SQL/SDA: a query language for supporting spatial data analysis and its web-based implementation. *IEEE Transactions on Knowledge and Data Engineering*, 13(4):671–682, 2001. ISSN 1041-4347. doi: 10.1109/69.940739.

Fredrik Skeel Løkke. Scala & design patterns. Master's thesis, University of Aarhus, March 2009.

Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin Series. Pearson Education, 2008. ISBN 978-0-13-608325-2.

Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C#*. Robert C. Martin Series. Pearson Education, 2006. ISBN 978-0-13-279714-6.

Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, Sebastopol, CA, USA, 2015. ISBN 978-1-491-90399-5.

Greg Michaelson. *An introduction to functional programming through lambda calculus*. Dover Publications, Inc., 2011. ISBN 978-0-486-47883-8.

Develop Service-Oriented Applications with WCF. Microsoft, 2021. URL `https://docs.microsoft.com/en-us/dotnet/framework/wcf/`.

TypeScript Documentation. Microsoft Corporation, 2023. URL `https://www.typescriptlang.org/docs/`.

John C. Mitchell. Types, theory of. In Anthony Ralston, Edwin D. Reilly, and David Hemmendinger, editors, *Encyclopedia of Computer Science*, pages 1806–1810. John Wiley and Sons Ltd., Chichester, UK, 4th edition, 2003. ISBN 978-0-470-86412-8.

Riley Moher, Michael Gruninger, and Scott Sanner. What's in a (data) type? Meaningful type safety for data science. In Renata Guizzardi, Jolita Ralyté, and Xavier Franch, editors, *Research Challenges in Information Science*, pages 20–38, Cham, 2022. Springer International Publishing. ISBN 978-3-031-05760-1. doi: 10.1007/978-3-031-05760-1_2.

Johan Montagnat, Tristan Glatard, and Diane Lingrand. Data composition patterns in service-based workflows. In *Workshop on Workflows in Support of Large-Scale Science, 2006 (WORKS'06)*, pages 1–10, July 2006. doi: 10.1109/WORKS.2006.5282350.

Miguel P. Monteiro and João Gomes. Implementing design patterns in Object Teams. *Software: Practice and Experience*, 43(12):1519–1551, 2013. ISSN 1097-024X. doi: 10.1002/spe.2154.

Peter D. Mosses. Formal semantics of programming languages: An overview. *Electronic Notes in Theoretical Computer Science*, 148(1):41–73, 2006. ISSN 1571-0661. doi: 10.1016/j.entcs.2005.12.012.

Martin Odersky. *Scala Language Specification: Version 2.6*, 2007.

Martin Odersky. Functional programming principles in Scala. Massive open online course, 2016. URL `https://www.coursera.org/learn/progfun1`.

Martin Odersky and Lex Spoon. *Scala Collections*, 2023. URL `http://docs.scala-lang.org/overviews/collections/introduction.html`.

Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. An overview of the Scala programming language. Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

Martin Odersky, Philippe Altherr, Vincent Cremet, Sébastien Doeraene, Gilles Dubochet, Burak Emir, Philipp Haller, Stéphane Micheloud, Nikolay Mihaylov, Adriaan Moors, Lukas Rytz, Michel Schinz, Erik Stenman, and Matthias Zenger. *Scala Language Specification: Version 3.4*, 2023. URL `https://scala-lang.org/files/archive/spec/3.4/`.

Unified Modeling Language Version 2.5.1. OMG, 2017. URL `https://www.omg.org/spec/UML/2.5.1`. Standard.

Common Object Request Broker Architecture Version 3.4. OMG, 2021. URL `https://www.omg.org/spec/CORBA/3.4/`. Standard.

Java® Platform, Standard Edition Core Libraries Release 21. Oracle, 2023. URL `https://docs.oracle.com/en/java/javase/21/core/java-core-libraries1.html`.

Jorge Luis Ortega-Arjona. *Patterns for Parallel Software Design.* Wiley Software Patterns Series. Wiley, 2010. ISBN 978-0-470-970874.

Peter Padawitz. Swinging UML - how to make class diagrams and state machines amenable to constraint solving and proving. In A. Evans, S. Kent, and B. Selic, editors, *Proceedings of the 3rd International Conference on the Unified Modeling Language (UML' 2000)*, volume 1939 of *Lecture Notes in Computer Science*, pages 162–177, Heidelberger platz 3, D-14197 Berlin, Germany, 2000. IEEE Comp Soc; IEEE Tech Council Complex Comp; ACM SIGSOFT; Kennedy Carter; Project Technol Inc; Retional Software Inc, Springer-Verlag Berlin. ISBN 3-540-41133-X.

David Lorge Parnas, John E. Shore, and David Weiss. Abstract types defined as classes of variables. In *Proceedings of the 1976 Conference on Data : Abstraction, Definition and Structure*, pages 149–154. Association for Computing Machinery, 1976. doi: 10.1145/800237.807133.

Havoc Pennington, Anders Carlsson, Alexander Larsson, Sven Herzberg, Simon McVittie, and David Zeuthen. *D-Bus Specification Version 0.42*, 2023. URL `https://dbus.freedesktop.org/doc/dbus-specification.html`.

Language Reference: Perl 5 version 38.0 documentation. Perl 5 Porters, 2023. URL `http://perldoc.perl.org/5.26.1/index.html`.

Michael L. Perry. *The Art of Immutable Architecture: Theory and Practice of Data Management in Distributed Systems.* Apress, Berkeley, California, USA, 2020. ISBN 978-1-4842-5955-9. doi: 10.1007/978-1-4842-5955-9.

PHP 8.2 Language Reference. PHP Group, 2023.

Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002. ISBN 978-0-262-16209-8.

Alex Potanin, Johan Östlund, Yoav Zibin, and Michael Ernst. Immutability. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 233–269. Springer, Berlin, Heidelberg, January 2013. ISBN 978-3-642-36946-9. doi: 10.1007/978-3-642-36946-9_9.

Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages: Design And Implementation.* Prentice Hall PTR, fourth edition, 2001. ISBN 978-0-13-027678-0.

The Python Language Reference 3.12.0. Python Software Foundation, 2023.

Dimitri Racordon and Didier Buchs. Implementing a language with explicit assignment semantics. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL 2019)*, page 12–21, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-6987-9. doi: 10.1145/3358504.3361227.

ReactiveX. *ReactiveX*, 2023. URL `http://reactivex.io/intro.html`.

The Rust Reference. Rust Team, 2023. URL `https://doc.rust-lang.org/reference/`.

Nathanael Schärli, Andrew P. Black, and Stéphane Ducasse. Object-oriented encapsulation for dynamically typed languages. *ACM SIGPLAN Notices*, 39(10):130–149, October 2004. ISSN 0362-1340. doi: 10.1145/1035292.1028988.

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Wuyts Roel. Composable encapsulation policies. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'2004)*, pages 26–50, June 2004. ISBN 978-3-540-22159-3. doi: 10.1007/978-3-540-24851-4_2.

Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, volume 2 of *Pattern-Oriented Software Architecture.* Wiley, 2013. ISBN 978-1-118-72517-7.

Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45, 1986.

Peter Sommerlad. Design patterns are bad for software design. *IEEE Software*, 24(4): 68–71, 2007. ISSN 0740-7459. doi: 10.1109/ms.2007.116.

Daniel Spiewak and David Copeland. *Scala Style Guide*, 2023. URL `https://docs.scala-lang.org/style/index.html`.

Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, sep 1996. ISSN 0360-0300. doi: 10.1145/243439.243441.

Martin Thomson and Cory Benfield. HTTP/2. RFC 9113, June 2022. URL `https://www.rfc-editor.org/info/rfc9113`.

Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems (GRADES '16)*, New York, New York, USA, 2016. Association for Computing Machinery. ISBN 978-1-4503-4780-8. doi: 10.1145/2960414.2960421.

Janina Voigt, Warwick Irwin, and Neville Churcher. Class encapsulation and object encapsulation: An empirical study. In P. Loucopoulos and L. Maciaszek, editors, *Proceedings of the 5th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'2010)*, pages 171–178, 2010. ISBN 978-989-8425-21-8.

Extensible Markup Language (XML) 1.1. W3C, second edition, 2006. URL `https://www.w3.org/TR/xml11/`.

SOAP Version 1.2. W3C, second edition, 2007. URL `https://www.w3.org/TR/soap12/`.

XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. W3C, 2012. URL `https://www.w3.org/TR/xmlschema11-2/`.

Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992a.

Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. Association for Computing Machinery, 1992b.

Uwe Zdun and Paris Avgeriou. A catalog of architectural primitives for modeling architectural patterns. *Information and Software Technology*, 50(9-10):1003–1034, 2008. ISSN 0950-5849. doi: 10.1016/j.infsof.2007.09.003.

Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. Interface representation patterns: Crafting and consuming message-based remote APIs. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs (EuroPLoP '17)*, pages 27:1–27:36. Association for Computing Machinery, 2017. ISBN 978-1-4503-4848-5. doi: 10.1145/3147704.3147734.

APPENDICES

Appendix 1

Definitions and Abbreviations

Base type — a type treated as a set of elementary values, which are not decomposed into simpler types (see Section 1.2). Examples: Booleans, numbers, characters.

Closure — a functor that holds values or references of the variables that a function or a lambda expression uses from the enclosing scope (see Section 4.1).

Compile-time type — the type bound to an identifier at the compile time (see Section 1.2).

Compound type — a type that is a composition of other types (see Section 1.2). Examples: arrays, sets, maps.

Data transfer representation — a representation that is agnostic to the programming languages and paradigms used to interpret the data (Zimmermann et al., 2017). Examples: REST, JSON, XML.

Depth subclassing — subclassing a member in a subclass (Pierce, 2002).

Design pattern — a description of a particular recurring design problem and a well-proven generic scheme for its solution (Buschmann et al., 2007b).

Dynamic (dynamically typed) language — a language that does not support compile-time types for identifiers (see Section 1.2).

Existential type — an abstract type whose definition declares properties of the type but does not specify the type representation and only assumes the latter exists (Pierce, 2002). Examples: sequence, set, map (abstract classes allowing multiple implementations).

Expressiveness of a programming language — the measure of the breadth of ideas that can be expressed using the language (Leitão and Proença, 2014).

Family polymorphism — Scala-specific feature for modelling families of types which vary together covariantly (Odersky et al., 2006).

Functor — an object that holds data but can be called as if it were a function (see Section 4.1).

Generic — see *Universal type.*

Immutable object — an object whose state cannot be changed after creation (Potanin et al., 2013).

Liskov substitution principle — the requirement that any program using class `A` continues to work correctly if any `A`'s subclass `B` is substituted for `A` (Martin and Martin, 2006).

Marshalling — conversion of a data object to its data transfer representation (Zimmermann et al., 2017).

MIX — an imaginary computer, invented by Donald E. Knuth for his book series 'The Art of Computer Programming' (Knuth, 1997).

Monad — an abstraction of computation, representing impure features in a purely functional environment (Wadler, 1992b).

Nominal type system — a type system in which structurally equivalent types with different names are different (Pierce, 2002).

Polymorphism — the opportunity, provided by a type system, to use a single piece of code with multiple types (Pierce, 2002).

Recursive type — a type that participates in its own definition (Pierce, 2002).

Run-time type — a type bound to an object at the run time (see Section 1.2).

Singleton — a class that has only one instance (Gamma et al., 1995).

Static (statically typed) language — a language in which identifiers have compile-time types (see Section 1.2).

Structural type system — a type system in which a type is determined by its structure, and types with the same structure but different names are equivalent (Pierce, 2002).

Subtype relationship — a relationship between types that means that any value of a subtype is also a value of its supertype (Pierce, 2002).

Type — a collection of computable values that have certain properties in common (Mitchell, 2003).

Type erasure — removal of the compile-time type information from the compiled code Pierce (2002).

Type system — a tractable syntactic method for proving that a program is free from certain types of errors by classifying language phrases according to the kinds of values that they compute (Pierce, 2002).

Type theory — a field of computer science that studies type systems existing in programming languages (Pierce, 2002).

UML — Unified Modeling Language, a language providing system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes (OMG, 2017).

Unmarshalling — conversion of a data transfer representation to a data object (Zimmermann et al., 2017).

Universal type — a generic (parameter-polymorphic) type whose definition uses variables in place of actual underlying types and can be instantiated with particular types as needed (Pierce, 2002). Examples: arrays, sets (as long as types of keys and values are unknown).

Variance — the subtyping relationship between different parameterisations of the same universal type (Pierce, 2002).

Width subclassing — a type of inheritance, in which a subclass has all members of its superclass, as well as new members (Pierce, 2002).

Appendix 2

List of Described Patterns

Name	Section	Examples
Assignable once	4.5	Variable, pointer, promise
Data access delegation	4.3	Computable property
Destructuring	5.4	Assignment of parts of a single compound value to multiple variables Scala: pattern matching
Executors	4.2	Thread, thread pool, remote proxy
Generalised functions	4.1	Function pointer, functor, closure Functional languages: general functional type
Map	3.4	Record, hash map, tree map
Multimap	3.5	Hash multimap, tree multimap
Multiset	3.3	Hash multiset, tree multiset
Optional value	3.7	Nullable type, optional type
Partial assignment	5.3	Assignment to an array or map element
Referential assignment	5.2	C++: assignment to a pointer or reference Java-like languages: reference type assignment
Sequence	3.1	Array, linked list, tuple
Set	3.2	Hash set, tree set
Traversable once	4.4	Iterator, generator, stream ReactiveX library: observable
Unboxing	5.5	`await` Scala: for-comprehension
Value assignment	5.1	C++: value assignment Java-like languages: primitive type assignment
Variant type	3.6	C++: `std::variant` Scala: `Either` TypeScript: union type Enumeration

Ruslan Batdalov was born in 1983 in Izhevsk, Russia. He obtained a mathematician's and system programmer's qualification in Applied Mathematics and Informatics from Kazan State University (2003, with distinction) and a Master's degree in Computer Systems from Riga Technical University (2017, with distinction). He has worked at "Smart Home" Ltd., CBOSS, Moscow City telephone network, Riga Technical University, and Rietumu Banka. Currently, he is a software engineer at Google. Since 2010, he has been a member of ACM. His research interests are related to design patterns and the expressiveness of programming languages.