Shaping a Modern Programming Paradigms Course for Advanced University Students

Leonardo Montecchi[1][0000−0002−7603−9695]

Department of Computer Science
Norwegian University of Science and Technology (NTNU)
Sem Sælands vei 7–9, 7034 Trondheim, Norway
leonardo.montecchi@ntnu.no

Abstract. Programming is one of the core disciplines in Computer Science (CS) and Computer Engineering (CE) courses, and it is increasingly permeating the curricula of other study programs. After a classical introduction to programming, and a course on object-oriented programming, some students will attend an advanced course on *programming languages*, where features of different programming paradigms are discussed. Due to the emphasis on theory and semantics aspects, such courses often employ old or experimental languages that have little practical application, resulting in low engagement of students. Unfortunately, most of the research work is focused on introductory programming, which has a more established syllabus and larger possibilities for interventions. In this paper, we analyze the current status of the *programming languages* course at our university, and we investigate possibilities for renewing the syllabus with modern languages and tools. We first review the current topics addressed by the course, and then we discuss possible content changes, with the aim to shape a more engaging course. The paper ends with a plan for implementing and evaluating the new version of the course starting from the next academic year.

Keywords: Programming languages · programming paradigms · course planning · advanced students.

1 Introduction

Programming is one of the core disciplines in Computer Science (CS) and Computer Engineering (CE) courses, and it is increasingly permeating the curricula of other study programs. After undergoing a classical introduction to programming, and a course on object-oriented programming, some students will attend an advanced course on *programming languages*, where features of different programming paradigms are discussed, with an emphasis on the semantics of execution.

Knowledge of different styles of programming, such as functional programming or different concurrency models, is considered a core skill by the ACM Computer Science Curricula 2023, meaning that such topics fall in the category

of "topics that every Computer Science graduate must know" [5], at least from an overview level. However, these topics are not covered in a standardized way, and not all the CS/CE programs actually offer such a course.

Due to the emphasis on theory aspects, those advanced programming lanugages courses often employ old or experimental languages that have little practical application, resulting in low engagement of students. Unfortunately, while many opportunities of improvement of teaching practices exist for such courses, most of the research work is focused on introductory programming, which has a more established syllabus and larger possibilities for interventions.

In this paper we analyze the current status of the *programming languages* course at our university, and we investigate possibilities for renewing the syllabus with modern languages and tools. We first review the current topics addressed by the course, and then we discuss possible content changes with the aim to shape a more engaging course. The work in this paper is driven by the overarching reasearch question *"What is an engaging course plan for a programming languages course for advanced university students?"*. Following a design thinking [2] approach, we have combined a divergent step, in which candidate topics and languages to be included in the course have been identified, and a convergent step, in which we have defined a concrete plan for a revised version of the course. The paper ends with a roadmap for implementing and evaluating the new version of the course starting from the next academic year.

The rest of the paper is organized as follows. Section 2 introduces the necessary background, including a brief description of the TDT4165 course and the challenges with its current structure. Section 3 discusses the methodology we adopted to shape a new version of the course, including a discussion of currently covered topics and of candidate programming languages for the revised version. Section 4 introduces and discusses the details of the new course plan, while Section 5 provides an overview of the roadmap towards its implementation and evaluation. Related work are discussed in Section 6. Finally, conclusions are drawn in Section 7.

2 Background

The TDT4165 "Programming Languages" course at NTNU [9] is a third-year course offered to different programs, having typically around 120 enrolled students. The course introduces students to programming language paradigms, semantics and computational models. The current version of the course is based on the Oz [7] programming language and its Mozart environment. Oz is a language originally developed for teaching and research, which is associated to the well-known "CTMCP" book that covers language theory [12].

One of the challenges we are facing is that the Oz/Mozart environment is becoming outdated, because it is not being maintained since several years, and the course suffers from the heavy dependence on that environment. The version on which the book is based, Mozart 1.4, has not been updated since 2013. The development of a new Mozart2 version started in 2012, but is currently still

"alpha quality", as mentioned in the official GitHub repository[1], and the latest update was published in 2018. Being an alpha version, some important parts that were included in the original Mozart are not yet implemented in Mozart2 (e.g., the debugger), leaving students with a frustrating and incomplete experience. While Oz/Mozart has been a fundamental step in the teaching of programming languages, its current limitations are evident. Oz/Mozart authors themselves discuss the history of the language, and its current limitations in [13].

The main challenge in moving to another language(s) is to find a solution that i) has a good practical support for programming, ii) covers the content of the course, and iii) has a good theory support.

3 Methodology

3.1 Overview

The approach adopted in this paper is inspired to the *design thinking* process, which has been described in different ways [2]. The work in this paper mainly corresponds to the "Inspiration" and "Ideation" phases in [1]; a discussion on the "Implementation" phase is reported in Section 5.

One of the characteristics of design thinking is the cyclic alternation between two ways of working towards the solution: *divergent thinking* (analyze), where the objective is to gather as many insights as possible; and *convergent thinking* (synthesize), where the objective is to synthesize the obtained information and to produce consolidated coherent solutions.

Our investigation was based on the following requirements:

- R1: The new course plan should cover at least the same topics that are covered today.
- R2: The new course plan should be based on programming languages with development tools (e.g., compilers, editors) that are multi-platform and actively maintained.
- R3: The new course plan should expose students to multiple languages having different features.

Divergent Thinking. For our divergent thinking activity we gathered information from multiple sources. In particular, we first reviewed related work from the literature (discussed in Section 6 later). We then defined a detailed conceptual map of the topics currently covered by the course (Section 3.2), and finally we selected a list of candidate programming languages to be considered for the revised version of the course (Section 3.3).

Convergent Thinking. As part of our convergent thinking activity, we mapped the candidate programming languages identified in the previous step, and we created small proof-of-concept examples that could serve as assignments in the course. Mostly, we converted existing exercises that are based on Oz to the new candidate languages, and we investigated their feasibility.

[1] `https://github.com/mozart/mozart2`

3.2 Current Curriculum

The current curriculum covered by the course is briefly discussed in the following.

1. *Introduction to Language Theory.* This module of the course provides an introduction to language theory, and the main steps that involve the definition and usage of a programming language. This includes a brief overview of grammars and the parsing process. In particular, the difference between lexical analysis and parsing is discussed.
2. *Declarative Programming.* This module introduces students to declarative and functional programming, through a restricted version of Oz called the Declarative Sequential Kernel Language (DSKL). Formal semantics for this part of the language is given through abstract machine semantics.
3. *Higher-Order Programming.* This module goes deeper into functional programming, in particular discussing higher order programming, function values (closures), and common abstractions that are used in functional programming, such as list folding.
4. *Memory, Scope, Exceptions.* This module collects some topics related to memory management. This includes some details on scoping of variables, bindings, as well as a formal semantics of exceptions. Among the other things, it discusses the difference between recursion and iteration, and tail recursion optimization.
5. *Declarative Data Structures.* This module discusses the definition of data structures in the declarative functional programming model. The module discusses how common data structures such as queues and stacks can be represented in effective ways, without resorting to explicit state.
6. *Threads.* This module introduces concurrency and gives a formal semantics of threads in Oz, still using abstract machine semantics. This module is however oriented to the dataflow concurrency model only, which is typical of Oz.
7. *Lazy Evaluation.* This module introduces the concept of lazy evaluation, that is, the possibility of delaying computation of values. This concept is useful for processing (potentially) infinite data structures in parallel, referred to as streams.
8. *Logic Programming.* The last module of the course introduces logic programming, using simple examples in Prolog. This module was originally in Oz as the rest of the course, but it was adapted to Prolog given the lack of support for logic programming in the current version of Oz/Mozart.

3.3 Candidate Programming Languages

In this section we discuss the programming languages that have been considered for the course, and the reasoning behind their inclusion or exclusion. The discussion is organized in three category, as follows.

Popular, but unsuitable languages. The first category includes languages that are well-known and widely used by both students and professionals, such as `Java`, `Javascript`, `Python`, and `C/C++`. While those languages would have the advantage that students do not need to learn new tools, they are very broad multi-paradigm languages, meaning that students would have *too much* flexibility, and it would be difficult for them to stick with the paradigms introduced in the couse, and for us teachers to verify if they are doing so. In particular, the course focuses in large part on functional programming and *immutability* of variables, which is not easily enforced in those languages.

Popular, potentially suitable languages. A few programming languages have gained popularity recently, and students frequently express their wish to learn them. It is the case for example of `Rust`, `Elixir`, and `Scala`. All of them support immutable variables in some form, and they are quite oriented to functional programming, which makes them potentially suitable for the course.

`Rust`[2] is a multi-paradigm language targeted to embedded systems and systems programming. It is designed to compete with `C/C++`, providing a similar flexibility on memory access and low-level programming, but with an increased type safety. Variables in `Rust` are immutable by default, which aligns with the content of the course. While the use of `Rust` for a similar course is discussed in [3], one of the drawbacks is its broadness in terms of functionality, which would make it difficult to define a subset for the initial part of the course, and to make sure that students will adhere to that. In particular, some specific features related to memory management, make the use of `Rust` for the entire course particularly challenging.

`Elixir`[3] is a functional, concurrent programming language that focuses on metaprogramming and flexibility. It builds on top of `Erlang`, and runs on the same virtual machine (BEAM). `Elixir` also uses immutable variables and has a convenient syntax for *records*, simpler than the original one used in `Erlang`. While being more oriented to functional programming and to the topics of the course, `Elixir` has also a wide range of features that would need to be restricted for a proper adoption in the course.

`Scala`[4] is a strong statically typed language originated from `Java` and fully compatible with it; it runs in fact on the Java Virtual Machine (JVM). Many of design decisions are intended to address criticism of `Java`, and it has a strong emphasis on combining object-oriented programming and functional programming. `Scala` is actually already used for one of the assignments in the current version of the course, although the focus of the assignments is on threads and concurrency. One of the peculiarities of `Scala` is that it supports both mutable ("`var`") and immutable ("`val`") variables. While this could be interesting for discussing how the two features can be combined, using it is not ideal for the initial

[2] `https://www.rust-lang.org/`
[3] `https://www.elixir-lang.org/`
[4] `https://www.scala-lang.org/`

part of the course, where students should be constrained to using the functional paradigm.

Less popular, but suitable languages. A language that is suitable for the course should enforce the functional programming paradigm and be relatively small, so that a core subset can be identified for formal reasoning about its semantics.

One option is to use some "dialect" of `Lisp`, which is often used in advanced courses on programming languages. Languages from the same family as `Lisp` include for example `Scheme`[5] and `Racket`[6]. While these languages are suitable to cover the content of the course, we identified two main limitations: i) their syntax is quite unique and very unusual with respect to languages used in the industry; and ii) teaching material (e.g., books) based on them is usually of a more theoretical nature than the current version of the course. We recall that the course is offered to different study programs, including some that are outside the computer engineering or computer science area.

Few other languages exist that: i) have been used in the industry, ii) enforce the functional programming style, and iii) are simple enough to be adapted to the course. `Erlang`[7] has a strong industrial background, being originally developed as proprietary software within the Ericsson company, and later released as open-source software in 1998. Despite being more than 25 years old, it is still actively maintained, the latest release being from July 2024. It is a functional high-level programming language, strongly oriented to concurrency. The `Erlang` runtime is freely available for the most common operating systems, and specialized plugins are available for modern editors such as IntelliJ and VSCode. Furthermore, it has an extensive documentation and a relatively active community. Balancing all these reasons, we considered `Erlang` to be the most suitable one among the candidate languages to be used as basis for a revised version of the course.

4 Revised Course Plan

In our *convergent thinking* step we synthesized the information we had gathered, and we planned a revised version of the course, with the aim to satisfy the requirements described in Section 3.1.

We decided to base a large part of the revised version of the course on the `Erlang` language, for its simplicity and availability of tools. Besides identifying modern languages to cover the course topics, this process also led to the revision of the course plan as a whole, which is discussed in the following. Topics marked by a star (*) are either completely new or heavily modified.

Introduction to Language Theory. The theory content of this module is kept essentially the same: students are introduced to grammars and to the main

[5] `https://www.scheme.org/`
[6] `https://www.racket-lang.org/`
[7] `https://www.erlang.org`

steps that are required to process and execute a language. Details of parsing algorithms are not introduced there, but are instead typically addressed in a Compiler Construction course later in the study program.

Currently, students are exposed to examples (in Oz) about lexical analysis and parsing. Such examples show how these two steps can be performed for a simple language, such as the one used as input in a simple calculator program. One of the problems with this module is that students struggle with understanding the practical aspects of grammars.

In the revised version of the course, students will be exposed to the `leex`[8] and `yecc`[9] tools, both of which are `Erlang`-based. These two tools are, respectively, analogous to the well-known "lex" and "yacc" tools for the C language: `leex` takes as input a regular expressions and generates a lexer; `yecc` takes as input a grammar in BNF format and generates the code to parse that grammar; both tools generate `Erlang` code. This module will introduce students to language theory and, at the same time, have them familiarize with the `Erlang` environment.

Declarative Programming. This module will also be based on `Erlang`, and it is actually one of the main reasons of selecting this language as the basis for the new version of the course. `Erlang` is a quite strict language, which is appropriate for this module, where students still need to adapt to the new paradigm of functional programming. This means, for example, using recursion instead of imperative loops for iterative tasks.

Furthermore, we believe that, at least informally, most of the Oz DSKL can be mapped to a subset of `Erlang` (a kind of "Erlang Kernel Language"). This would allow the course to keep the theoretical content on DSKL semantics, while providing examples in a more realistic language to students. In fact, all the statements in the DSKL have a direct correspondence in `Erlang`, with a similar semantics. The only (notable) exception is the *freeze* semantics of dataflow variables, which is instead typical of Oz and does not exist in `Erlang`. In this new version of the course, dataflow variables are addressed later in the newly introduced *Concurrency Models* module.

Discussing a semantics that is similar, but not exactly the same, to the Oz DSKL, also leaves room for discussions and exercises on how to formally define a semantics for a language different than Oz.

Higher-Order Programming. At this point students should start being familiar with both `Erlang` and functional programming. In this module, they will be introduced to higher-order programming, with examples and teaching material also based on `Erlang`. Among the other things, `Erlang` has a good support for the specification of types in functions[10], as well as automated inference of

[8] `https://www.erlang.org/doc/apps/parsetools/leex.html`
[9] `https://www.erlang.org/doc/apps/parsetools/yecc.html`
[10] `https://www.erlang.org/doc/system/typespec.html`

function parameters types. While types are not particularly addressed by this course, providing a better overview on types and dicussing types specifications for functions, enables a better understanding of higher-level programming.

This module is also well-suited to start comparing different languages, so students will be asked in exercises to identify the same concepts in languages they already know. Most of the programming languages that are popular today, such as `Java` or `Python`, are in fact multi-paradigm, so students should be able to identify functional programming concepts in most of the languages they know.

Functional Programming Patterns*. This module is considered to be a new module. While part of the content is extracted from the previous version of the "Higher-Order Programming" module, we decided to create a separate module with a more coherent focus. While the previous module focuses on the concept of higher-order programming and closures, in this module we focus on patterns for functional programming. In a certain sense, this module could be seen as the equivalent of a module on design patterns for a course on object-oriented programming.

Most of this module can be delivered using `Erlang`. However, we plan to discuss some specific topics using `Haskell`, which is often considered the reference for reasoning on advanced functional programming. Topics like *currying* or concepts from category theory [6] are better discussed using `Haskell`. Also, this would be the occasion for preparing students to use `Haskell` in subsequent modules of the course such as the one on "Lazy Evaluation".

Memory, Scope, Exceptions. This module houses a heterogeneous set of concepts that are not easily fitting other modules. While we did not modify the theoretical content of this module, we believe that it can be exploited for a more in-depth comparison of different programming languages.

In this case, popular multi-paradigm programming languages are well-suited for this task, as they exhibit considerable differences in terms of memory management, scoping of variables, and error-handling behavior. Exercises and examples should provide students with a broad understanding on how such concepts are realized in different languages, such as `Java`, `Python`, and `C`. Furthermore, it is worth discussing memory management in `Rust`, which has some unique features such as its "ownership" and "borrowing" concepts, which are deeply related to memory management and scoping.

Declarative Data Structures. The content of this module is substantially unaltered from the previous version of the course. Its objective is to discuss how advanced data structures can be defined in a declarative (i.e., immutable) way. It basically discusses two of the four ways to package data abstractions [10], namely immutable ADT (Abstract Data Types) and immutable objects. This module can be delivered in any languages that strictly follows the functional paradigm, such as `Erlang`.

Explicit State*. This module was added in the new version of the course, and it focuses on discussing explicit state and its effects. The theoretical part of this module is available in the pensum book [12], so no major alterations are needed to the recommended reading material. Before taking this course, students have already taken courses where they have programmed with explicit state, such as in object-oriented programming. Therefore, no specific exercises are planned for this modules. However, students should be encouraged to identify explicit state constructs in multi-paradigm languages such as `Python`.

Concurrency Models*. This new module stems from the observation that students know very little about concurrency when they take this course. In the previous version of the course, students were only introduced to dataflow concurrency, which is typical of Oz and few other languages. Since one of the learning outcomes of the course is to give students "the ability to understand and compare existing and future languages.", a broader view of concurrency models should be provided. This module addresses three common models of concurrency: *shared-state*, *message-passing*, and *dataflow*. All the the three of them are covered by the book that is already adopted for the course [12].

Shared-State Concurrency is the concurrency model used in imperative and object-oriented programming languages, such as `Java`. In case students have some experience with concurrency, they probably experienced this flavor of concurrency. A full coverage of concurrency problems and solutions is out of the scope of this course, however, a brief introduction to this model and its dangers will be provided. For shared-state concurrency, examples will be given in a traditional language such as `Java`.

Message-Passing Concurrency is a programming style in which a program consists of independent entities that interact by sending each other messages asynchronously, i.e., without waiting for a reply [12]. Support for message-passing concurrency is one of the distinguishing features of `Erlang`. Therefore, at this point of the course introducing this concurrency model should follow quite naturally from students' experience with the language in the rest of the course.

Dataflow Concurrency is a concurrency model that is distinctive of Oz and its "dataflow variables". While this model is used in some domain-specific contexts such as distributed data processing and workflows, none of the most common general purpose programming languages support it directly. One of the few ways for students to experience dataflow concurrency other than Oz is to use the `GPars` library[11], which is available for both `Groovy` and `Java`.

Another possibility for this module is to run it using `Scala`, which has libraries to support all the three concurrency models: shared-state, using basic thread con-

[11] `https://www.gpars.org/`

structs; message-passing using the `Actors` library[12]; and dataflow using `GPars`, since `Scala` is fully compatible with `Java` libraries.

Lazy Evaluation. Lazy evaluation is an important module of this course, introducing students to computation using streams. Besides the theoretical importance, streams are used in practice in several practical applications, from data processing to embedded systems.

This module will be run by comparing two languages that have been already introduced earlier in the course: `Haskell`, which implements *lazy* evaluation by default, and `Erlang`, which instead uses *eager* evaluation (i.e., the opposite, where values are always computed immediately). Besides discussing the difference between these two features of a language, it will also be shown how to emulate lazy evaluation in a language that does not support it.

Logic Programming. Logic programming is currently introduced using `Prolog`, which is of course the reference language for this programming paradigm. While using `Prolog` works well for introducing the paradigm, we find that it does not provide students with a real feeling of the applicability of logic programming as part of larger applications. In the revised version of the course, we will complement a theoretical introduction to logic programming with exercises using practical libraries in `Python`, such as `pytholog`[13] or `swipy`[14].

Table 1. Summary of the modules of the revised version of the course, and programming languages that are used in each of them.

	Module	*Language*	*Notes*
1	Introduction to Language Theory	`Erlang`	Using `leex` and `yecc` for grammars and overview of parsing.
2	Declarative Programming	`Erlang`	—
3	Higher-Order Programming	`Erlang`	Types specifications (`-spec`) help reasoning about functions.
4	Functional Programming Patterns	`Erlang`	Specific advanced topics using `Haskell`.
5	Memory, Scope, Exceptions	—	Compare popular multi-paradigm languages. Further, `Rust`'s *ownership* and *borrowing* concepts provide an advanced view on scoping and memory management.
6	Declarative Data Structures	`Erlang`	—
7	Explicit State	—	Reason on popular multi-paradigm languages such as `Python`.
8	Concurrency Models	`Java`, `Erlang`, `GPars`	`Scala` (with `GPars`) can also be used as a single-language alternative.
9	Lazy Evaluation	`Haskell`	Also discuss emulation of lazy evaluation in eager languages such as `Erlang`.
10	Logic Programming	`Python`	Introduce the theory in `Prolog`. Use `Python` libraries for exercises.

[12] `https://docs.scala-lang.org/overviews/core/actors.html`

[13] `https://github.com/MNoorFawi/pytholog`

[14] `https://github.com/SWI-Prolog/packages-swipy`

The revised version of the course has 10 modules (Table 1), which fit well a typical semester consisting of 14 weeks of lectures. The course combines having a reference language for most parts of the course, so that students can feel comfortable with something known, with occasionally introducing other languages to enable a broader view of common features of programming languages.

5 Implementation and Evaluation

The course plan introduced in Section 4 will be gradually implemented from the next instance of the course, which will be held in Autumn 2025. To manage the risk in completely changing the course curriculum, the new course plan will be implemented gradually in three phases:

Phase 1. The teaching material for the theoretical part is keep in the current version. Exercises in the new languages are introduced for specific parts of the course. Modules 8, 9, and 10 are the potential candidates for this first phase, since their modification does not affect other modules.

Phase 2. The teaching material is modified so that the examples and the practical activities of the course are mostly based on the new core language. This involves modifying modules 1, 2, 3, and 4.

Phase 3. The rest of the course is adapted to the new version. The remaining modules are adapted to the new content (5 and 6), and the material for module 7 is finalized. The course is now completely adapted to the new version.

At each phase, the feedback from students will be collected through questionnaires and discussions with the reference group, to evaluate the impact of the introduced changes, and adopt corrective actions if needed. Besides evaluating the satisfaction of students, the evaluation aims at assessing potential risks in the modification of the course, and in particular: i) the risk of introducing multiple programming languages in the same course; and ii) the risk of detaching the theoretical framework of the adopted book [12] from the practical examples and exercises.

6 Related Work

Most of the literature on teaching programming languages focuses on introductory courses, which are typically given in the first or second year of bachelor programs (e.g., "Introduction to Programming" and "Object-Oriented Programming"). Those courses are typically larger (in terms of students), their syllabuses are more standardized, and they are often run by a large team of teachers and teaching assistants. All these characteristics make them more convenient for running interventions, collecting data, and analyzing the results. Conversely, the TDT4165 course is a specialized course, such that many computer science programs in other universities do not even offer it.

Some of the most relevant works on the topic of this project are from the authors of CTMCP and Oz/Mozart themselves [11,10], and they are from the same period as the CTMCP book [12].

The work in [4] provides an overview of common problems and open challenges in the teaching of programming languages, from a Computing Education Research (CEdR) perspective. Among the other things, the authors confirm that *"An enormous amount of attention in computing education has focused on traditional CS1 courses."*[15] and that *"Despite some noticeable treatment in the literature, notional machines do not feature prominently in curricula or texts for computing courses."*. With "notional machines", the authors refer to something analogous to the "abstract machine semantics" introduced in the CTMCP book [12], and used in the course to define the semantics of languages.

There exist a few other works in the literature that discuss topics related to the objectives of our work. The authors of [14] discuss the choice of the programming language to be used in a programming course, going though the languages that are most appropriate for each of the different paradigms. The focus of that work is however on choosing the *first* language that students will learn, i.e., the one to be used in an *introductory* course. Also, the work is from 15 years ago, and thus it cannot reflect the current panorama of programming languages. The authors of [8] discuss the design and evaluation of a course similar to TDT4165 using the C# language[16]. They mention that time limitations prevented them to use multiple programming languages, but unfortunately they do not provide further details on the challenges they encountered.

Interestingly, the most relevant source on this topic is gray literature in the form of a blog post [3]. The author mentions that *"Unlike in some fields of computer science, in programming languages there isn't widespread agreement on the undergraduate curriculum"*. The author then reports his experience in renewing a similar course, and discusses the choices they made. The course ended up combining Haskell[17] and Rust[18], coupled with some theory material that was created ad-hoc for the course. It should be noted, however, that the author [3] used a different theory foundation than the one currently used in the TDT4165 course. Nevertheless, even if the work in [3] is gray literature, it should be considered a respectable source, because it appeared in the "PL Perspective" blog. PL Perspective is the blog of ACM SIGPLAN, the Special Interest Group on Programming Languages of the Association for Computing Machinery (ACM), one of the main computer science associations worldwide, organizing leading research conferences in programming languages and software engineering.

[15] CS1 and CS2 designate the first two courses in the introductory sequence of a computer science major.
[16] C# Documentation, `https://learn.microsoft.com/en-us/dotnet/csharp/`
[17] Haskell Language, `https://www.haskell.org/`
[18] Rust Programming Language, `https://www.rust-lang.org/`

7 Conclusion

In this paper we have discussed the planning of a new version of the course TDT4165 "Programming Languages" at NTNU. We first discussed the challenges we are facing with the current version of the course, and then we described the methodology we used to shape a new, modern, version of the course. We first analyzed the topics currently covered by the course, and then investigated possible programming languages that could be used to cover and emphasize the topics discussed in the different modules.

The investigation resulted in a new version of the course with a total of 10 modules, some of which have been revised in the content or introduced as new. The new version of the course combines a base language that is used for the core topics of the course, as well as specifically selected languages to cover advanced topics. We believe this new structure enhances the learning objective of being able "to understand and compare existing and future languages". We proposed a realistic roadmap and discussed some insights for the implementation and the evaluation of the new course plan. As future work, we plan to perform a thorough evaluation of the new course, and to create teaching material to support the proposed vision of a modern programming language course.

In the long term, there will need to be a reflection on the role of AI (Artificial Intelligence) on programming. While such reflection goes beyond the scope of a single programming course, the popularity of AI is leading to new ways of defining programs, which will need to be acknowledged and discussed by a course that aims to provide students with the skills to "understand and compare existing and future languages".

Acknowledgments. This work has been partially supported by the SFU-2016/10002 Excited Centre of Excellent IT Education and the Norwegian Directorate for Higher Education and Skills. The author would like to thank Marie Hayashi Strand and Christoffer Stensrud for their initial work on investigating candidate programming languages.

References

1. Brown, T.: Design thinking. Harward Business Review (2008)
2. Grönman, S., Lindfors, E.: The process models of design thinking: A literature review and consideration from the perspective of craft, design and technology education. Techne serien - Forskning i slöjdpedagogik och slöjdvetenskap **28**(2), 110–118 (April 2021), `https://journals.oslomet.no/index.php/techneA/article/view/4352`
3. Hsu, J.: Re-imagining the "programming paradigms" course. SIGPLAN PL Perspectives (January 28th 2021), available on line at: `https://blog.sigplan.org/2021/01/28/re-imagining-the-programming-paradigms-course/` (last accessed October 25, 2024)
4. Krishnamurthi, S., Fisler, K.: Programming Paradigms and Beyond. In: Fincher, S.A., Robins, A.V. (eds.) The Cambridge Handbook of Computing Education Research. Cambridge University Press (2019)

5. Kumar, A.N., et al.: Computer Science Curricula 2023. Association for Computing Machinery, New York, NY, USA (2024)
6. Milewski, B.: Category Theory for Programmers. Blurb, Incorporated (2019), also available online: https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/ (last accessed October 25, 2024)
7. Mozart Consortium: The Mozart Programming System. http://mozart2.org/ (2013)
8. Ortin, F., Redondo, J.M., Quiroga, J.: Design and evaluation of an alternative programming paradigms course. Telematics and Informatics **34**(6), 813–823 (2017), sI: IT Education & Training
9. TDT4165: Programming Languages. NTNU, https://www.ntnu.edu/studies/courses/TDT4165/2024 (Autumn 2024), last accessed October 25, 2024.
10. Van Roy, P.: Programming paradigms for dummies: what every programmer should know. In: New computational paradigms for computer music, pp. 9–47. Editions Delatour France (2009)
11. Van Roy, P., Haridi, S.: Teaching programming broadly and deeply: The kernel language approach. In: Cassel, L., Reis, R.A. (eds.) IFIP TC3 / WG3.2 Conference on Informatics Curricula, Teaching Methods and Best Practice (ICTEM 2002) July 10–12, 2002, Florianópolis, SC, Brazil, pp. 53–62. Springer US (2003)
12. Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press (2004)
13. Van Roy, P., Haridi, S., Schulte, C., Smolka, G.: A history of the oz multiparadigm language. Proceedings of the ACM on Programming Languages **4**(HOPL), 1–56 (Jun 2020). https://doi.org/10.1145/3386333
14. Vujošević-Janičić, M., Tošić, D.: The role of programming paradigms in the first programming courses. Teaching of Mathematics **XI**(2), 63–83 (2008)