UNIVERSIDADE
BEIRA INTERIOR

# Invariants and Code Contracts in an Online Classroom Environment

**Rui Pedro de Almeida Barata**

Dissertação para obtenção do Grau de Mestre em

**Engenharia Informática**

(2º ciclo de estudos)

Orientador: Prof. Doutor Simão Melo de Sousa

**outubro de 2023**

# Declaração de Integridade

Eu, Rui Pedro de Almeida Barata, que abaixo assino, estudante com o número de inscrição M11467 do Mestrado em Engenharia Informática da Faculdade de Engenharia, declaro ter desenvolvido o presente trabalho e elaborado o presente texto em total consonância com o Código de Integridades da Universidade da Beira Interior.

Mais concretamente afirmo não ter incorrido em qualquer das variedades de Fraude Académica, e que aqui declaro conhecer, que em particular atendi à exigida referenciação de frases, extratos, imagens e outras formas de trabalho intelectual, e assumindo assim na íntegra as responsabilidades da autoria.

Universidade da Beira Interior, Covilhã 06/10/2023

Rui Pedro de Almeida Barata

*”Rejoice with your family in*
*the beautiful land of life.”*
- Albert Einstein

Too my family, that is everything to me.

# Acknowledgments

Gostaria de expressar os meus sinceros agradecimentos a todas as pessoas e instituições que desempenharam um papel fundamental na realização deste trabalho de dissertação, que representa um marco importante na minha jornada académica.

Em primeiro lugar, ao meu orientador, Simão Melo de Sousa, a sua orientação experiente, a sua dedicação incansável e a sua orientação crítica foram inestimáveis ao longo de todo o processo de investigação. As suas sugestões e conselhos ajudaram a moldar este trabalho e a aprimorar a sua qualidade. A sua paciência em responder às minhas dúvidas e a sua capacidade de incentivar o pensamento crítico foram cruciais para o meu desenvolvimento académico. Agradeço também à unidade de Investigação RELEASE pela disponibilidade e recursos fornecidos que tornaram este projeto possível.

Quero estender a minha gratidão aos meus colegas e professores da Universidade, cujo conhecimento partilhado e colaboração enriqueceram o meu percurso académico e contribuíram para o desenvolvimento profissional e pessoal.

Quero também dedicar uma parte especial dos meus agradecimentos à minha família. Aos meus pais e irmã, o meu amor e gratidão são eternos. O vosso apoio inabalável, desde o início da minha jornada académica até este momento, foi um motor constante de motivação. Sem a vossa confiança em mim e o vosso encorajamento, este trabalho não teria sido possível.

Aos meus avós, tios e primas, agradeço pela vossa constante presença nas minhas conquistas. As vossas palavras de incentivo e o vosso carinho sempre foram um conforto nas horas mais desafiadoras.

Por último, mas não menos importante, quero expressar um agradecimento profundo e caloroso à Barbara. Ao longo deste percurso, enfrentei desafios e momentos de dúvida, mas tu estiveste sempre ao meu lado. A tua compreensão, paciência e apoio incondicional foram fundamentais para o meu sucesso. Obrigado por seres a minha fonte de inspiração e força.

A todos, o meu profundo agradecimento por terem feito parte desta jornada académica e por terem contribuído para o sucesso deste trabalho. Sem a vossa ajuda e apoio, este projeto não teria alcançado os resultados que aqui apresento.

# Resumo

A educação em programação evoluiu significativamente com o surgimento de ambientes de sala de aula online, oferecendo vantagens e desafios. Esta dissertação explora a integração de contratos de código e invariantes na educação em programação, com foco na linguagem OCaml. O principal objetivo é desenvolver uma ferramenta que traduza os contratos de código e invariantes escritos pelos alunos em *assertions* executáveis, melhorando a confiabilidade e correção do código.
Para atingir esse objetivo, são apresentadas várias contribuições-chave. Primeiramente, uma análise aprofundada dos ambientes de sala de aula online, destacando a importância de abordar a confiabilidade do software nesse contexto. A plataforma Learn-OCaml é introduzida como um recurso educacional valioso, oferecendo uma combinação única entre plataformas de concursos e Massive Open Online Courses (MOOCs).
A dissertação explora os contratos de código e invariantes, elucidando a sua importância para garantir que o código se comporte conforme o previsto. Examina as ferramentas de contrato de código existentes e a relevância dos ambientes de sala de aula online. É ainda discutido como a linguagem de especificação *Gospel* modificada pode ser usada para integrar contratos de código em programas OCaml.
A fase de implementação do projeto é detalhada, delineando o processo de tradução de componentes como précondições, póscondições, invariantes e variantes a partir de especificações do *Gospel* em código OCaml executável. São ainda reconhecidas as limitações da ferramenta, nomeadamente na manipulação de quantificadores.
A dissertação conclui com um resumo das conquistas e contribuições, abordando o objetivo da investigação de melhorar a confiabilidade e correção do software na educação em programação online. Trabalhos futuros são propostos, incluindo o suporte a quantificadores e a incorporação da ferramenta desenvolvida na plataforma Learn-OCaml, promovendo a avaliação automatizada e a comparação de *assertions*.
Este trabalho representa um passo significativo em direção à melhoria da qualidade da educação em programação em salas de aula online, capacitando os alunos a escreverem código mais confiável e correto, ao mesmo tempo oferecendo aos educadores ferramentas melhoradas para avaliação e *feedback*.

# Palavras-chave

Educação de Programação; Contratos de Código; Especificação Comportamental; Plataformas de Ensino Online; Programação Funcional; OCaml; Correção de Software

# Resumo alargado

O ensino de programação passou por uma transformação substancial, impulsionada pela proliferação de plataformas de sala de aula online. Esta dissertação é uma exploração abrangente da integração de contratos de código e invariantes na educação em programação, com foco específico na linguagem de programação OCaml. O principal objetivo é desenvolver uma ferramenta que traduza contratos de código e invariantes escritos pelos estudantes em *assertions* executáveis, melhorando assim a confiabilidade e correção do software produzido pelos estudantes.

O capítulo *Introduction* delineia os objetivos da investigação e destaca as contribuições feitas no campo da educação em programação. O objetivo principal é criar uma ferramenta que possa automatizar a verificação do código escrito pelos estudantes por meio de contratos de código e invariantes. O capítulo estabelece a importância deste trabalho no contexto dos ambientes de sala de aula online.

No capítulo *Related Work*, é apresentada uma visão abrangente do trabalho relacionado na área. Começa por discutir a evolução dos ambientes de sala de aula online e os desafios e vantagens específicos que estes apresentam para a educação em programação. O capítulo explora várias plataformas de ensino online, categorizando-as em plataformas viradas para concursos e MOOCs. É introduzida a plataforma Learn-OCaml, que combina elementos de ambos os tipos. O capítulo também fornece informações sobre contratos de código e invariantes, destacando o seu papel na garantia da correção do software.

O capítulo *Starting Point* apresenta a instância personalizada do Learn-OCaml, uma plataforma online popular para educação em OCaml. O capítulo fornece uma visão geral das características do Learn-OCaml, incluindo fichas de trabalho, uma ferramenta de atribuição de exercícios e a integração de perguntas de múltipla escolha. É introduzida a linguagem de especificação *Gospel*, que é fundamental para a tradução de contratos de código e invariantes. O capítulo conclui apresentando uma ferramenta chamada *Cameleer*, parte essencial do projeto que estende o Gospel para facilitar a implementação de contratos de código em ficheiros com código fonte OCaml.

O capítulo *Implementation* detalha a implementação da tradução de contratos de código e invariantes a partir das especificações do *Gospel* em afirmações executáveis em código OCaml. Divide o processo em três componentes: précondições, póscondições e invariantes/-variantes. Para cada componente, o capítulo explica como a especificação do *Gospel* é analisada e transformada em código OCaml. As limitações da ferramenta também são discutidas, nomeadamente o desafio de lidar com quantificadores.

O capítulo *Conclusion* serve como um resumo de toda a dissertação. Resume os objetivos principais, conquistas e contribuições feitas no campo da educação em programação. É proposto trabalhos futuros, que inclui a adição de suporte para quantificadores e a integração da ferramenta na plataforma Learn-OCaml, possibilitando a avaliação automatizada e a comparação de *assertions*.

Em resumo, esta dissertação representa um avanço significativo na melhoria da qualidade do ensino de programação em salas de aula online. Ao automatizar a verificação do código

escrito pelos estudantes por meio de contratos de código e invariantes, os estudantes podem produzir código mais confiável e correto, enquanto os educadores obtêm ferramentas aprimoradas para avaliação e *feedback*.

# Abstract

Programming education has evolved significantly with the advent of online classroom environments, offering both advantages and challenges. This dissertation explores the integration of code contracts and invariants into programming education, with a focus on the OCaml language. The primary objective is to develop a tool that translates student-written code contracts and invariants into executable assertions, enhancing software reliability and correctness.

In the pursuit of this objective, several key contributions are presented. First, an in-depth analysis of online classroom environments, highlighting the importance of addressing software reliability in this context. The Learn-OCaml platform is introduced as a valuable educational resource, offering a unique blend of contest-like challenges and comprehensive lessons.

The dissertation delves into code contracts and invariants, elucidating their significance in ensuring code behaves as intended. It surveys existing code contract tools and the relevance of online classroom environments. The research demonstrates how a modified Gospel specification language can be harnessed to integrate code contracts into OCaml programs.

The implementation phase of the project is detailed, outlining the process of translating components like preconditions, postconditions, invariants, and variants from Gospel specifications into actionable OCaml code. Limitations of the tool, especially in handling quantifiers, are acknowledged.

The dissertation concludes with a summary of achievements and contributions, addressing the research problem of enhancing software reliability and correctness in online programming education. Future work is proposed, including the addition of quantifier support and the incorporation of the developed tool into the Learn-OCaml platform, fostering automated grading and assertion comparison.

This work represents a significant step toward improving the quality of programming education in online classrooms, empowering students to write more reliable and correct code while offering educators enhanced tools for assessment and feedback.

# Keywords

Programming Education; Code Contracts; Behavioral Specification; Online Teaching Platforms; Functional Programming; OCaml; Software Correctness

# Contents


**1 Introduction** **1**

1.1 Objectives . . . 1
1.2 Contributions . . . 2
1.3 Document Organization . . . 3

**2 Related Work** **5**

2.1 Online Classroom Environments . . . 5
2.1.1 Challenges and Advantages of Online Education . . . 6
2.1.2 Online Teaching Platforms . . . 7
2.1.3 The Best of Both Worlds: Learn-OCaml . . . 9
2.2 Code Contracts and Invariants . . . 11
2.2.1 Components . . . 11
2.2.2 Code Contract Tools . . . 12
2.3 Conclusion . . . 13

**3 From Learn-OCaml to Code Contracts Verification** **15**

3.1 Our Instance of Learn-OCaml . . . 15
3.1.1 Worksheets . . . 15
3.1.2 Exercise Assignment Tool . . . 16
3.1.3 Multiple Choice Questions . . . 17
3.2 The OCaml Specification Language: GOSPEL . . . 20
3.2.1 Cameleer . . . 22
3.3 Conclusion . . . 23

**4 Proposed Solution** **25**

4.1 Translating Components . . . 25
4.1.1 Precondition . . . 25
4.1.2 Postcondition . . . 27
4.1.3 Invariants and Variants . . . 29
4.2 Assessments . . . 31
4.3 Conclusion . . . 31

**5 Conclusion** **33**

5.1 Future Work . . . 33

**Bibliografia** **35**

# List of Figures

2.1 Learn-OCaml exercise view. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.2 Learn-OCaml exercise feedback view. . . . . . . . . . . . . . . . . . . . . . . . . . 10

3.1 Flowchart of how the assignment of exercises works.[1] . . . . . . . . . . . . . . . 17
3.2 Flowchart of how the translation of the difficulty assignment to exercise assignment works. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.3 Teacher control center. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.4 Interface of multiple choice questions. . . . . . . . . . . . . . . . . . . . . . . . . 19

# Acronyms

**AI** Artificial Intelligence

**AST** Abstract Syntax Tree

**IDE** Integrated Development Environment

**JML** Java Modeling Language

**MCQ** Multiple Choice Question

**MOOC** Massive Open Online Course

**QOL** quality of life

**REPL** read-eval-print loop

**TFPIE** Trends in Functional Programming in Education

# Chapter 1

# Introduction

Online learning platforms have become increasingly important in recent years as more and more people, both academic and industry focused, turn to the internet for education and professional development [1]. One of the main advantages of online learning is convenience: students can access course materials and complete assignments from anywhere with an internet connection, at a time that is convenient for them. This can be especially useful for people who have busy schedules or who live in areas where access to traditional education is limited.

The Learn-OCaml project offers a unique platform that provides students with hands-on programming experience, enabling them to grasp the intricacies of functional programming paradigms effectively [2]. Its benefits extend beyond mere code execution; it fosters an interactive learning environment, allowing students to experiment, test, and refine their code within the confines of well-defined assignments and exercises.

In the context of online education, the reliability and correctness of software are of paramount importance. Educational software must not only facilitate learning but also ensure that students can trust the outcomes of their programming efforts. However, the advent of tools like ChatGPT and GitHub Copilot [3, 4], while aiding in code generation, poses an increasing danger. Students might inadvertently rely on these tools without fully comprehending the code they produce, potentially hindering their learning experience and jeopardizing software quality.

The central objective of this dissertation is to investigate and address the challenges posed by the evolving landscape of online education, focusing on the Learn-OCaml project. Specifically, I3 aim to explore how the integration of invariants and code contracts can enhance the reliability, correctness, and overall learning experience in online OCaml classrooms. Invariants and code contracts serve as powerful tools to ensure that the code produced and executed by the students aligns with the their logic and specifications.

## 1.1 Objectives

The primary objective of this dissertation is to design and create a tool that can seamlessly integrate into the Learn-OCaml platform. This tool will play a pivotal role in enhancing the learning experience of OCaml programming students. Specifically, it will serve as a bridge between the abstract concept of code contracts and invariants, and their practical application in code development.

To achieve this, the tool will translate code contracts and invariants authored by students into executable assertions. These assertions will function as vigilant sentinels, diligently scrutinizing the code produced by students to ensure that it adheres to the specified contracts.

By automating this validation process, the tool aims to provide real-time feedback, allowing students to identify and rectify deviations from their intended logic promptly. This not only reinforces the importance of understanding code but also promotes a culture of code correctness and reliability.
In addition to the development of the code contracts and invariants tool, this dissertation will detail the contributions made to the Learn-OCaml platform itself. These contributions have enabled a deeper understanding of the platform's architecture and functionality, ultimately enhancing its usability and effectiveness for online OCaml education.
Furthermore, this research will encompass a thorough exploration of state-of-the-art research in the realm of invariants and code contracts. These findings will serve as the foundation for incorporating these concepts into the source code of OCaml programs.

## 1.2 Contributions

This dissertation encompasses a series of notable contributions aimed at enhancing the Learn-OCaml platform, advancing the field of OCaml programming education, and addressing the critical issues surrounding code correctness and understanding:

**Enhancements to Learn-OCaml** To gain a comprehensive understanding of the Learn-OCaml platform, we embarked on the development of two significant projects within the platform:

- **Automatic Exercise Assignment Tool** The first project involved the creation of an innovative tool capable of automatically assigning a unique set of exercises to each student. This assignment is based on individual proficiency levels and the difficulty of exercises. By tailoring exercises to match a student's current skill level, this tool contributes to a personalized and effective learning experience within the platform.
- **Integration of Multiple Choice Questions** In the second project, we extended the Learn-OCaml platform by incorporating multiple choice questions. These questions provide an additional interactive dimension to the learning process, enabling students to test their knowledge and enhance their problem-solving abilities.

**Research Publications** This dissertation is complemented by the publication of two research contributions:

- **Trends in Functional Programming in Education (TFPIE) 2023** A paper titled "Mastering Functional Programming, Algorithms, and Data Structures in OCaml" is among the contributions. This paper explores pedagogical strategies and insights gained through the application of Learn-OCaml in functional programming education.
- **INFORUM 2023** A communication presentation titled "Learn-OCaml Estendido com Contratos" highlights the idea of extending Learn-OCaml with code contracts. This communication discusses the integration of contracts and their impact on the platform, fostering code reliability and student understanding.

**Code Contracts Translation** The cornerstone of this dissertation's contributions is the development of a robust tool capable of translating code contracts, into executable assertions. This tool will play a pivotal role in ensuring that the code written by students aligns with their specified contracts. By automating the validation process, it promotes code correctness and reliability within the Learn-OCaml environment.
These contributions collectively represent a commitment to the enhancement of online OCaml education and the advancement of code correctness through the application code contracts.

## 1.3 Document Organization

This dissertation is organized into distinct chapters, each contributing to a comprehensive understanding of the research problem and its solution. The document's structure is as follows:

- **Introduction** - This chapter introduces the background, research problem, objectives, and contributions of the dissertation. It provides a roadmap for the entire document.
- **Related Work** - In this chapter, we explore the existing literature and related work in the fields of online classroom environments, programming education, code contracts, and invariants.
- **From Learn-OCaml to Code Contract Verification** - This chapter delves into the initial setup, presenting our customized instance of Learn-OCaml, worksheets, exercise assignment tools, multiple-choice questions, and the specification language, GOSPEL, which forms the foundation for our work.
- **Proposed Solution** - Here, we detail the process of translating code contracts and invariants from GOSPEL specifications into OCaml code.
- **Conclusion** - In this final chapter, we summarize the key findings, achievements, and contributions of the dissertation. We also outline potential avenues for future work.

# Chapter 2

# Related Work

In this chapter, we will review related literature and analyze existing tools in the field of online classroom environments, OCaml programming education and code contracts and invariants. The aim of this chapter is to provide a comprehensive overview of existing solutions and their strengths and weaknesses, and to lay the foundation for the next steps of this project.

**Online Classroom Environments** have evolved rapidly in response to the growing demand for flexible and accessible education [5]. In the digital era, students and educators find themselves immersed in virtual learning spaces where collaboration and knowledge sharing transcend geographical boundaries. Of particular interest is the integration of OCaml into online education, and the associated advantages and challenges that this presents. Yet, these platforms also introduce a set of challenges, such as maintaining the integrity of educational content and ensuring students genuinely understand the code they produce, particularly when automated tools like ChatGPT [3] and GitHub Copilot [4] are involved.

**Code Contracts and Invariants** are essential components in guaranteeing the correctness and reliability of software. Grounded in the principles of formal verification, these concepts serve a vital function in building a robust foundation of confidence within software code. Our examination encompasses prior research efforts that have employed code contracts and invariants in diverse contexts, illuminating their capacity to enhance code comprehension and uphold software integrity.

## 2.1 Online Classroom Environments

The evolution of education in the digital era has witnessed a transformative shift towards online classroom environments, redefining the way knowledge is disseminated and acquired [5]. Online education platforms have become instrumental in providing accessible, flexible, and technology-driven learning experiences, transcending geographical boundaries and time constraints.
The significance of online education is underscored by its ability to adapt to diverse learning styles and accommodate learners from various backgrounds. These platforms offer a plethora of resources, interactive tools, and collaborative opportunities, fostering engagement and enhancing the learning process [6]. However, this digital transformation is not without its complexities, particularly when it comes to teaching a language as distinctive as OCaml [7].
OCaml, renowned for its strong typing, functional programming paradigm, and elegance, presents both advantages and challenges in the context of online education. While it em-

powers students with a deeper understanding of programming principles, it also demands a comprehensive grasp of its unique features. Thus, the integration of OCaml into online learning environments necessitates careful consideration of pedagogical strategies and technological solutions that facilitate effective teaching and learning.

### 2.1.1 Challenges and Advantages of Online Education

The integration of online education platforms into the modern educational landscape has ushered in numerous benefits, but it also comes with its own set of challenges. Understanding and addressing these challenges is essential to harness the full potential of online learning environments.

Online education confronts several challenges:

- **Lack of Interaction.** One of the most conspicuous challenges is the absence of interaction. In online settings, students and educators are often geographically dispersed, making it challenging to develop personal connections, facilitate real-time collaboration, and promptly address queries [8].

  To alleviate this issue, online platforms incorporate a range of interactive features such as discussion forums, video conferencing, and virtual office hours. These tools aim to bridge geographical distances and foster engagement among participants.

- **Self-Discipline and Time Management.** Online learning frequently demands a high degree of self-discipline and time management. Students must adapt to self-paced schedules and the absence of fixed classroom hours [9].

  Effective time management strategies and study plans, along with the provision of clear schedules and deadlines by educators, can assist students in developing essential self-discipline skills.

On the flip side, online education offers distinct advantages:

- **Accessibility and Flexibility.** Online education dismantles geographical barriers, making education accessible to a global audience. Learners gain the flexibility to study at their own pace and convenience.

- **Multimodal Learning Resources.** Online platforms offer an extensive array of multimedia resources, including videos, interactive simulations, and adaptive quizzes. This rich diversity of resources enriches the learning experience and caters to diverse learning styles.

- **Data-Driven Personalization.** The digital nature of online education allows for data collection and analysis. This, in turn, enables personalized learning experiences that are tailored to individual student needs, fostering better engagement and comprehension.

When we delve into the realm of OCaml in education, we encounter a unique combination of advantages and instructional challenges. OCaml, renowned for its expressive power and pedagogical value, introduces distinctive benefits to programming education
OCaml's functional programming paradigm encourages students to think differently about problem-solving and program design, fostering a deep understanding of fundamental programming principles. Its strong typing and type inference system provide immediate feedback to students, reducing common programming errors and promoting code reliability.
However, teaching OCaml comes with its set of challenges. OCaml's unique syntax and functional programming concepts may pose an initial challenge to students accustomed to imperative languages. Moreover, there's limited availability of resources that facilitate interactive learning in OCaml.

### 2.1.2 Online Teaching Platforms

Online teaching platforms have revolutionized the way education is delivered and accessed in the digital age. These platforms have become instrumental in providing flexible, interactive, and accessible learning experiences to students across the globe [10]. In the realm of online education, two distinct categories of platforms have emerged, each with its unique approach and characteristics.
The first category includes contest-like platforms, where learners engage in coding challenges and usually competitions, often fostering a spirit of friendly rivalry and problem-solving. The second category encompasses MOOCs, which offer a wide array of courses on diverse subjects, making education available to a broad audience.

#### 2.1.2.1 Contest-like Platforms

**Codeforces** is a renowned competitive programming platform known for its rigorous coding contests and challenges. It has gained popularity for its strong community of competitive programmers and a diverse range of algorithmic problems. Codeforces regularly hosts timed contests that attract participants from around the world. These contests are designed to test participants' coding skills, speed, and problem-solving abilities, making it an excellent platform for those looking to excel in competitive coding [11].

**DOMjudge** is a specialized online judging system used in programming competitions and contests [12]. It provides a structured environment for organizing and conducting coding competitions, both locally and remotely. DOMjudge supports a wide range of programming languages and offers features for managing problem sets, judging submissions, and scoring contests. It is often utilized by educational institutions and programming communities to host coding events.

**Mooshak** is another online judging system designed for running programming contests and competitions [13]. It offers a user-friendly interface for participants to submit their code solutions and receive real-time feedback. Mooshak is commonly used in educational settings,

particularly in universities, to organize coding contests and coursework assessments. It provides instructors with the tools needed to create, manage, and evaluate coding assignments and competitions effectively [14].

It's important to note that contest-like platforms typically focus on assessing problem-solving abilities and code efficiency rather than providing extensive educational materials. As a result, students using these platforms may find themselves lacking in-depth feedback on their coding practices or a detailed means of communication with instructors or peers. While the competitive nature of these platforms can be motivating, it may not always align with the requirements of a structured educational environment. While some efforts have been made to adapt such platforms for use within an online classroom environment [15, 16, 17, 18], educators often seek supplementary tools or platforms to complement the learning experience, especially when aiming to deliver comprehensive programming courses.

#### 2.1.2.2 MOOCs

**Coursera** stands as a prominent figure in the world of MOOCs, offering an extensive selection of courses across various disciplines [19]. Partnering with universities and institutions worldwide, Coursera provides learners with access to high-quality content [20]. Students can choose from a diverse array of courses, spanning computer science, business, arts, sciences, and more. Coursera accommodates both free and paid courses, with the opportunity to earn certificates and even full degrees through selected programs.

**edX** was founded by Harvard University and MIT, it is an open-source MOOC platform that collaborates with universities and institutions globally [21]. It hosts a wide spectrum of courses, covering subjects ranging from computer science to humanities. edX offers flexible learning pathways [22], including verified certificates and MicroMasters programs that allow students to earn academic credit towards advanced degrees.

**Fun-Mooc** , short for France *Université Numérique* - Massive Open Online Courses, is a notable MOOC platform with a focus on courses offered by French institutions [23]. It serves as a hub for a wide range of courses. Fun-Mooc's offerings encompass diverse academic disciplines, making it an accessible resource for learners seeking knowledge and skills in various fields. While they once featured an OCaml course [24], it's important to note that course availability on MOOC platforms can change over time, and some courses may no longer be offered.

While MOOCs undoubtedly provide a wealth of knowledge and resources to a broad audience, they often prioritize the collective learning experience rather than tailored support for individual students. The scale at which MOOCs operate can make it challenging to provide personalized assistance to every learner [25]. As such, educators and institutions often complement MOOCs with additional resources and strategies to ensure that students receive the personalized attention they may need to succeed.

### 2.1.3 The Best of Both Worlds: Learn-OCaml

Learn-OCaml is a sophisticated and purpose-built online learning platform meticulously designed to cater specifically to the needs of OCaml programming education. This unique platform serves as a bridge between the world of contest-like programming environments and MOOCs, offering a comprehensive and interactive educational experience tailored to OCaml. Learn-OCaml distinguishes itself through several key advantages:

- **A web-based approach.** The platform is completely web based, the students do not need to install anything on their own machine and the client-server architecture in which the server has minimal commitment compared to the client has a notable impact on the platform's stability and scalability, but also on it's use.
- **Focus on student's quality of life (QOL).** Moreover it provides each student with an Integrated Development Environment (IDE), the front-end of a compiler for syntax and typing verification and a read-eval-print loop (REPL) so students can experiment with their solutions before requesting the server to evaluate and store the solution (figure 2.1).
- **Comprehensive Lessons.** Beyond the conventional scope of contest-like platforms, Learn-OCaml goes the extra mile by providing meticulously structured lessons. These lessons can cover a wide spectrum of OCaml topics, facilitating a deep and holistic understanding of the language. Unlike typical coding challenges, Learn-OCaml equips students with the foundational knowledge and skills required for mastery.
- **Powerful Feedback System.** Learn-OCaml boasts a robust feedback system that transcends simple correctness checks (figure 2.2). While contest platforms often focus on the outcome, Learn-OCaml places equal emphasis on the coding process. It provides detailed feedback not only on the accuracy of the code but also on coding style, and efficiency. This depth of feedback fosters a rich learning experience and encourages students to refine their coding skills.
- **Interactive Teacher-Student Interaction.** A hallmark feature of Learn-OCaml is its capacity to facilitate direct and interactive communication between teachers and students. Instructors have the unique ability to access and review the code submitted by each student for every exercise. This level of engagement allows for personalized guidance, immediate clarification of doubts, and constructive feedback. Such direct teacher-student interaction is typically absent in conventional MOOCs and sets Learn-OCaml apart as an exceptional educational platform.

Learn-OCaml's dedication to delivering comprehensive lessons, offering in-depth code analysis, and enabling seamless teacher-student interaction defines it as a dynamic and effective platform for OCaml programming education. This fusion of features embodies the "best of both worlds", affording students the challenge and engagement of contest-like platforms while providing the instructional depth and feedback-rich experience often found in MOOCs

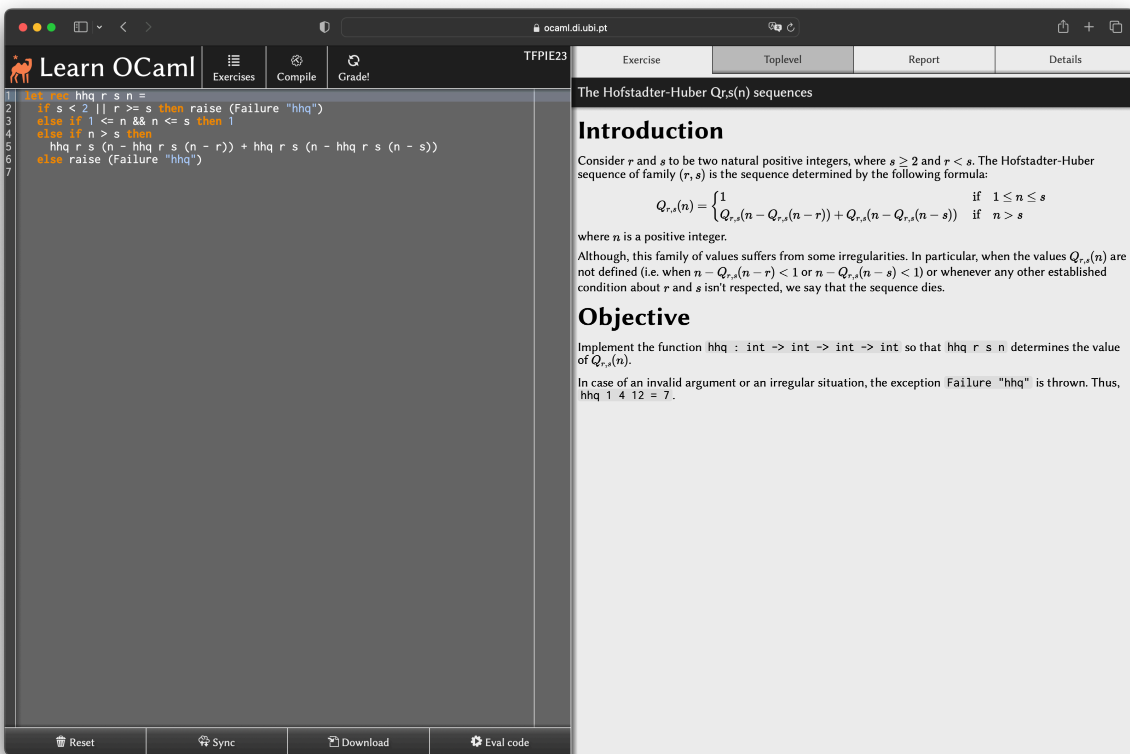


Figure 2.1: Learn-OCaml exercise view.

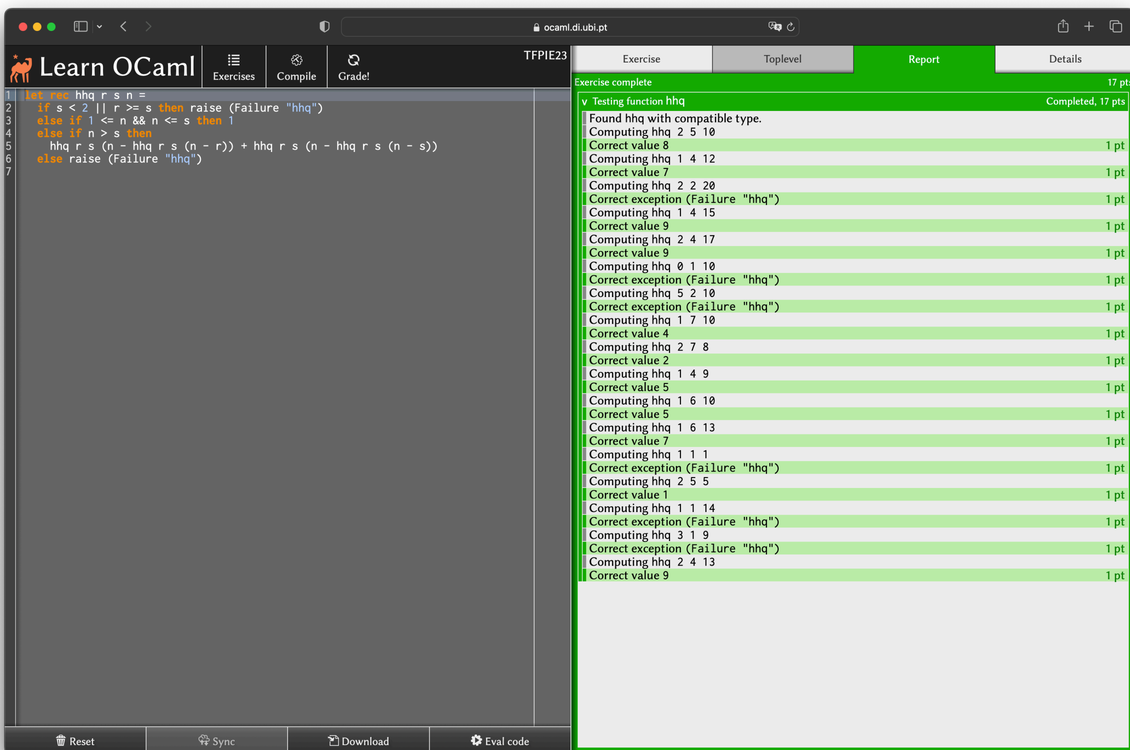


Figure 2.2: Learn-OCaml exercise feedback view.

[26]. It stands as a testament to the commitment to excellence in OCaml education. Therefore, we have chosen Learn-OCaml as the foundational platform for this dissertation, recognizing its capacity to provide a robust and interactive learning environment.

## 2.2 Code Contracts and Invariants

In the ever-evolving landscape of programming and software development, the pursuit of code correctness and reliability remains a fundamental objective. Achieving this goal necessitates not only the mastery of programming languages and algorithms but also a keen focus on the soundness of the code itself. This is where the concepts of code contracts and invariants come into play.
Code Contracts and Invariants encompass a set of conditions and expectations that code is required to meet throughout its execution. These contracts, typically consisting of preconditions, postconditions, invariants, and variants, serve as the foundation for ensuring that software behaves as intended.

### 2.2.1 Components

**Preconditions** are conditions that must be satisfied before a function or method is executed [27]. They define the valid input state for a particular function, ensuring that the function can safely perform its intended operations. Preconditions act as a contract between the caller and the function, specifying the expectations for the input.
In a function that calculates the square root of a number, a precondition might specify that the input number must be non-negative. Therefore, the function should not be called with a negative value.

**Postconditions** are conditions that describe the expected state or behavior of a function after it has executed [27]. They define what the function guarantees to accomplish or the state it ensures to leave behind. Postconditions serve as a contract between the function and its caller, specifying what the caller can rely on after the function call.
In a sorting function, a postcondition might state that the output array is sorted in ascending order. This ensures that the caller can depend on the sorted result.

**Invariants** are conditions that hold true at the beginning and end of each iteration of a loop [27]. They help ensure that the loop's logic is correct and that it makes progress towards its termination. Loop invariants act as a contract within the loop, guiding its execution.
In a loop that calculates the factorial of a number, a loop invariant could state that the product of the calculated factorials so far is equal to the factorial of the processed elements. This ensures the correctness of the factorial calculation.

**Variants** are conditions that measure or bound the progress of a loop or recursive function, ensuring their termination [28]. These essential components guarantee that a spe-

cific metric decreases with each iteration, preventing the occurrence of infinite loops or unbounded recursive calls.
In a loop that calculates the factorial of a number, a loop variant could be the decreasing value of the number being processed. With each iteration, the number is decremented by 1, ensuring that the loop progresses towards its termination. This variant guarantees that the loop doesn't run indefinitely, leading to the successful calculation of the factorial.
Consider a recursive function for calculating the nth Fibonacci number. A variant in this context could be the decreasing value of $n$. In each recursive call, $n$ is reduced by 1, ensuring that the function approaches the base case where $n$ equals 0 or 1. This variant guarantees the termination of the recursive calls, allowing the function to calculate the Fibonacci number efficiently.

Code contracts enhance the comprehensibility of a system's intended behavior and instill confidence when making modifications to the code. This confidence stems from the knowledge that any alterations conflicting with the established contracts will be promptly detected and preempted. In essence, code contracts serve as a safeguard, ensuring that the code adheres to its specified requirements and maintains its integrity, even in the face of modifications or updates.

### 2.2.2 Code Contract Tools

Contract-based programming tools provide a means of specifying these assumptions and guarantees in the form of contracts. In this section, we will explore some examples of contract-based programming tools and discuss their usage for various purposes such as runtime checks, proofs, and documentation generation. By understanding these tools, we can gain insight into how contract-based programming can be used to improve the quality of software and make software development more efficient and effective.

**Microsoft Code Contracts** are a set of tools and libraries that provide a way to express and check design-by-contract assumptions in .NET code [29].
These contracts can then be checked at runtime, either by throwing an exception when a contract is violated or by generating warnings for contracts that may be violated.
Microsoft Code Contracts also provide a set of static analysis tools that can generate proof of correctness for the contracts in your code [30]. These tools use formal verification techniques to analyze the contracts and generate a proof of their correctness.
Furthermore, the contracts serve as a valuable source for generating documentation that enhances the comprehensibility and maintainability of the code, especially for individuals who may not have prior knowledge of the code base [31]. This documentation can play a crucial role in facilitating collaboration and communication among all stakeholders involved in the project.

**Java Modeling Language (JML)** is a specification language for Java programs that aims to provide a high-level, formal specification of the behavior of Java modules (classes,

methods, and data) [32, 33, 34]. JML provides a syntax for expressing assertions about the behavior of Java code and is used to write formal specifications that can be used to reason about and validate the behavior of Java programs. The specifications written in JML can be automatically checked using automated tools like assertion checking compiler **jmlc** [35] which converts JML annotations into runtime assertions, a documentation generator **jml-doc** [36] which produces Javadoc documentation augmented with extra information from JML annotations, and a unit test generator **jmlunit** [37] which generates JUnit test code from JML annotations.

**Eiffel** is an object-oriented programming language that was first developed by Bertrand Meyer in 1986 [38]. It is designed to be both simple and efficient, with a focus on strong typing and software reliability. The language is known for its design by contract methodology, where classes define a contract specifying the required inputs and outputs for methods, and the conditions that must be met for the methods to execute successfully [39]. This contract-based approach enables automatic checking of program correctness, making it easier to identify and prevent errors before they become issues in production.
The language is known for its readability and conciseness. Eiffel has been used in a variety of industries, including finance, health care, and telecommunications.

**Why3** is a software verification platform that supports code contract-based programming [40]. Code contracts, such as pre- and postconditions, are specified in a program and can be used to verify properties about the program's behavior. Why3 provides a language for writing specifications for these contracts, which can then be used to generate proof obligations that can be automatically verified. By combining the expressiveness of code contracts with the power of automatic verification, Why3 enables developers to write more reliable and trustworthy software.

In this section was discussed the use of code contracts in software development and the various tools that employ contract-based programming. We explored Microsoft Code Contracts, Java Modeling Language, Eiffel Language, and the Why3 tool, and discussed how they can be used to produce documentation, run-time checks, and proofs of correctness. Code contracts have become an important tool in software development and have helped to improve the quality and reliability of software systems. The use of code contracts can help to make the code easier to understand and maintain, and can be especially useful for stakeholders who may not be familiar with the code base.

## 2.3 Conclusion

In the pursuit of enhancing online programming education and incorporating code contracts and invariants, this chapter has undertaken a comprehensive exploration of related work. Through an extensive review of existing literature and tools, we have uncovered valuable insights and innovations that provide essential context for our research.

Online classroom environments have evolved significantly, catering to diverse learning needs and preferences. We have observed the emergence of contest-like platforms and MOOCs, each offering unique advantages and challenges in programming education. While contest-like platforms focus on evaluating code, MOOCs provide structured courses on a large scale. This dichotomy has led to the development of platforms like Learn-OCaml, combining the best of both worlds, specifically tailored to OCaml education.
In examining code contracts and invariants, we find that these essential software engineering concepts play a pivotal role in ensuring code correctness, reliability, and maintainability. While educators and developers have recognized their significance, challenges persist in conveying these concepts effectively to students.
By building upon these foundations and integrating code contracts and invariants within the Learn-OCaml platform, we aim to contribute to the advancement of online OCaml programming education.

# Chapter 3

# From Learn-OCaml to Code Contracts Verification

In this chapter, we will delve deeper into our starting point, examining the contributions that we have made thus far to the Learn-OCaml Project and the basis of the tool we're going to develop. Our goal is to provide a comprehensive overview of the work that has been done to date, highlighting both the strengths and limitations of our approach.

## 3.1 Our Instance of Learn-OCaml

Our customized instance of Learn-OCaml serves as a tailored environment designed to enhance the OCaml programming education experience. These features encompass the integration of versatile tools, including exercise assignment and Multiple Choice Questions (MCQs) utilities, alongside interactive worksheets. Together, they create a dynamic and immersive OCaml programming education ecosystem that empowers learners with the tools they need to succeed and thrive.

### 3.1.1 Worksheets

My first encounter with Learn-OCaml was a task assigned to me by my advisor. At the time, my advisor had created five worksheets as part of their educational materials for the subject "Functional Programming". However, they wanted to make these worksheets more accessible and interactive for students, so they tasked me with transforming them into Learn-OCaml worksheets.
Developing an exercise in Learn-OCaml involves creating a set of seven files, each serving a specific purpose.

**descr.html/md** which contains the description of the exercise in HTML or markdown which is then presented in HTML. This file provides the user with information about what they are expected to do in the exercise.

**meta.json** contains metadata about the exercise including the title, type, difficulty, identifier, requirements and author/s of the exercise.

**prelude.ml** which contains the definitions that are known to the user. In the web-application, the prelude is displayed after the exercise description. This file is loaded into the environment before any other .ml files from the exercise directory.

**prepare.ml** which contains definitions that are unknown to the user. This file can be used to redefine functions or execute code. For example, it can be used to redefine some standard library functions, or define variables that students will need. **prepare.ml** is loaded directly after the **prelude.ml** file but before the students code in the environment.

**template.ml** which contains a template of code for the exercise. This file is loaded into the editor by default if the user does not have a saved code for the current exercise.

**solution.ml** which contains the teacher's solution for the exercise. This file can be used to test the student's code by comparing its results with teacher's code.

**test.ml** which contains the test program used to check and grade the user's code. This file has access to many functions that allow for the introspection of code.

The flexibility of the exercise file structure in Learn-OCaml allows for creative and interesting ways of grading student solutions. The graders (the OCaml functions that correct student's solutions) can be tailored to meet specific learning objectives and to evaluate a student's solution in a manner that aligns with the desired outcomes. The ability to control the correction of the solution and its form, style (e.g recursive, iterative), and even elegance, is made possible by the these graders. With this level of customization, Learn-OCaml can provide not only a means of checking the correctness of a student's solution, but also a way to evaluate their understanding of the material and the thought processes they used in arriving at their solution. This provides a comprehensive assessment of the student's progress, allowing for a more personalized learning experience.

### 3.1.2 Exercise Assignment Tool

Designing a tool capable of attributing exercises based on students' knowledge was our next task in the Learn-OCaml project. This tool is able to analyze the students' level and assign them exercises from each worksheet that are appropriate for their current level of knowledge, allowing for a more personalized and efficient learning experience.
This tool operates by first reading the level assigned to each student by the teacher, which is based on the student's current year in university and the teacher's selected exercises. The exercises are grouped into three categories: basic, intermediate, and advanced. Based on each student's level, the tool then assigns a certain number of exercises from each difficulty. This configuration can be chosen by the teacher, for example:

- Level 1 students - are required to complete 3 basic level exercises and 2 intermediate level exercises.
- Level 2 students - must complete either 1 basic level exercise, 3 intermediate level exercises, and 1 advanced level exercise or 2 basic level exercises, 2 intermediate level exercises, and 1 advanced level exercise.

- Level 3 students - must complete either 2 intermediate level exercises and 3 advanced level exercises or 3 intermediate level exercises and 2 advanced level exercises.

If there are less selected exercises than the number of exercises required in the selected configuration, the tool assigns all of them. If there are extra exercises, the tool selects the required number of exercises to meet the selected configuration, as demonstrated in figures 3.1 and 3.2.

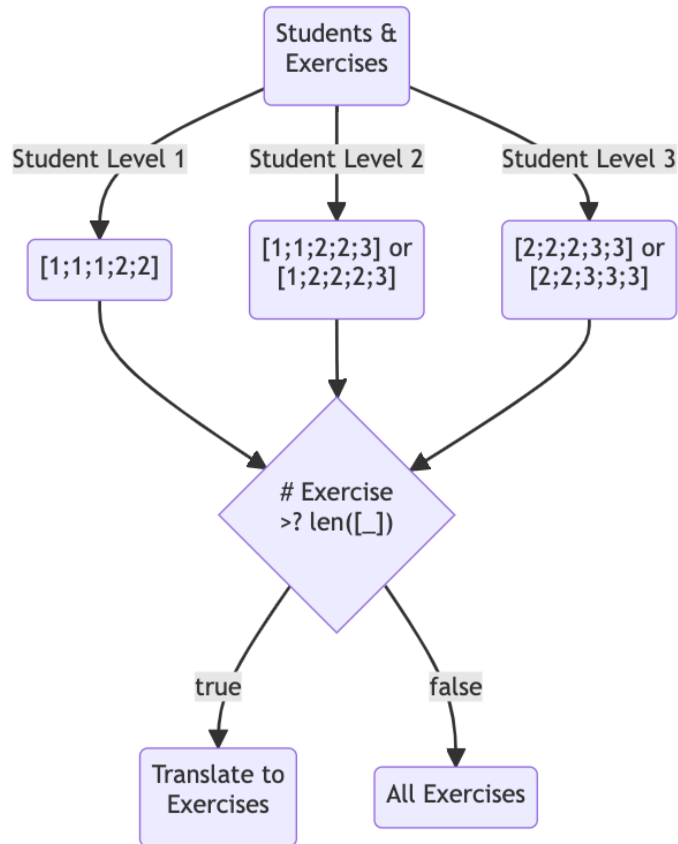


Figure 3.1: Flowchart of how the assignment of exercises works.[1]

This tool has greatly improved the efficiency of managing large classrooms. By determining the number of exercises each student must complete, the tool is able to streamline the process of assigning exercises and keep track of students' progress. Teachers are able to easily assign entire worksheets to their students based on their level of knowledge and monitor their progress throughout the year, making the management of large classrooms much easier (figure 3.3).

### 3.1.3 Multiple Choice Questions

In our next task for Learn-OCaml, we focused on updating the platform to support MCQs. This involved a number of updates to both the frontend and backend components to ensure that the platform could effectively manage and display MCQs for students to answer.

[1]1 -> Basic exercise
2 -> Intermediate exercise
3 -> Advanced exercise

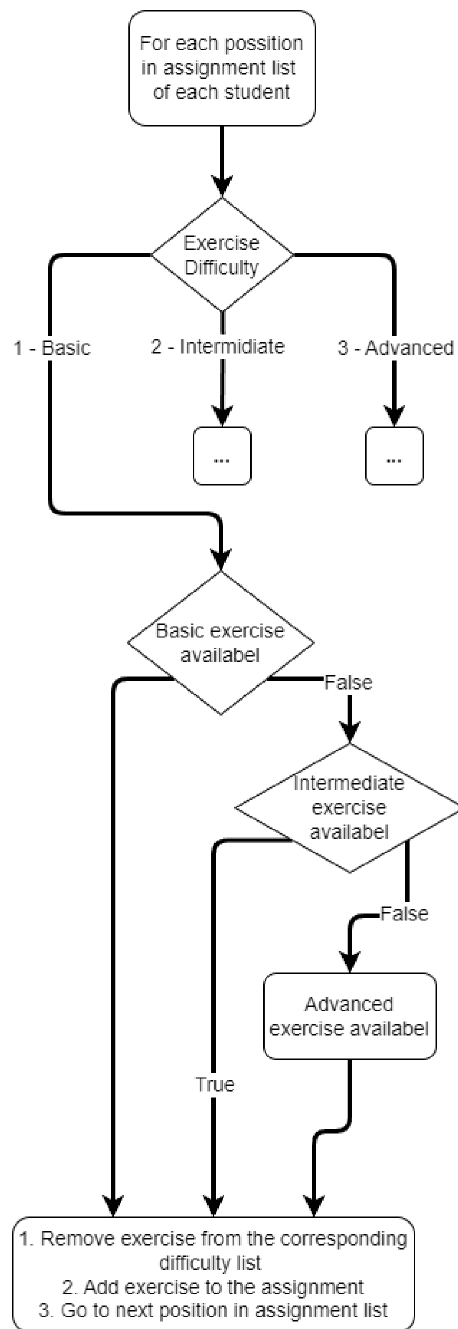


Figure 3.2: Flowchart of how the translation of the difficulty assignment to exercise assignment works.

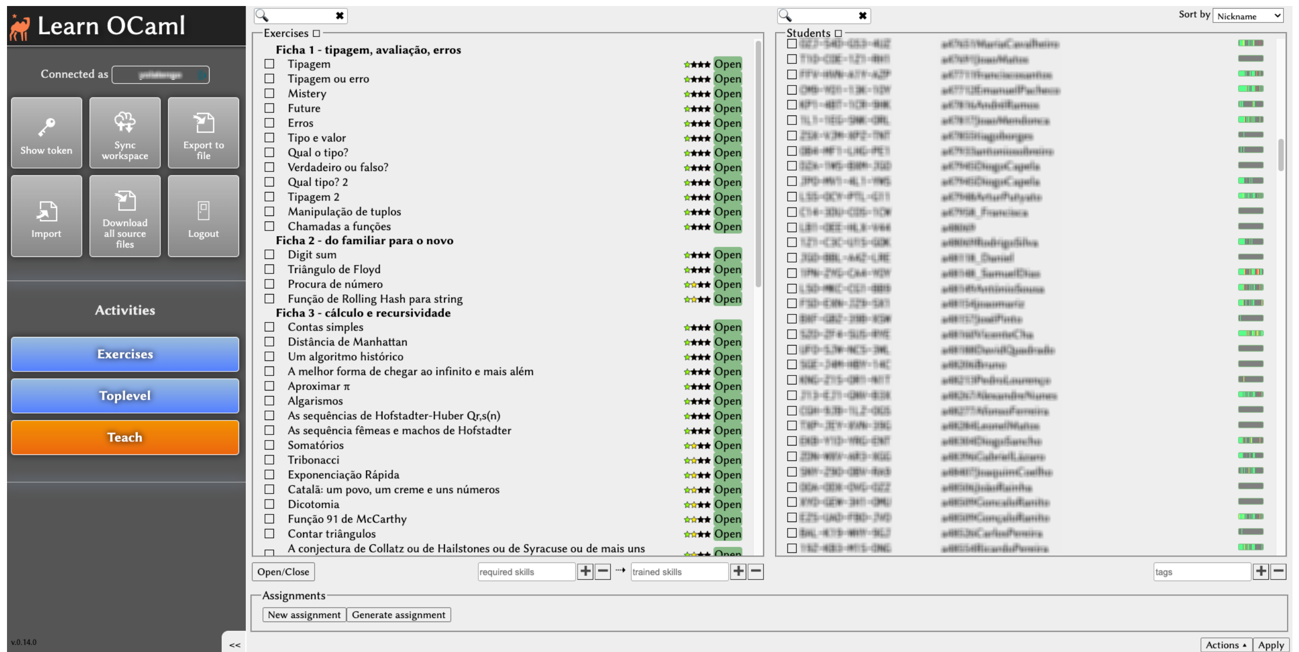


Figure 3.3: Teacher control center.

One of the key challenges in this task was ensuring that the interface was user-friendly and intuitive for students. This task entailed careful consideration in the design and arrangement of the multiple choice questions to ensure optimal user experience. The final result of the updated interface can be seen in 3.4.

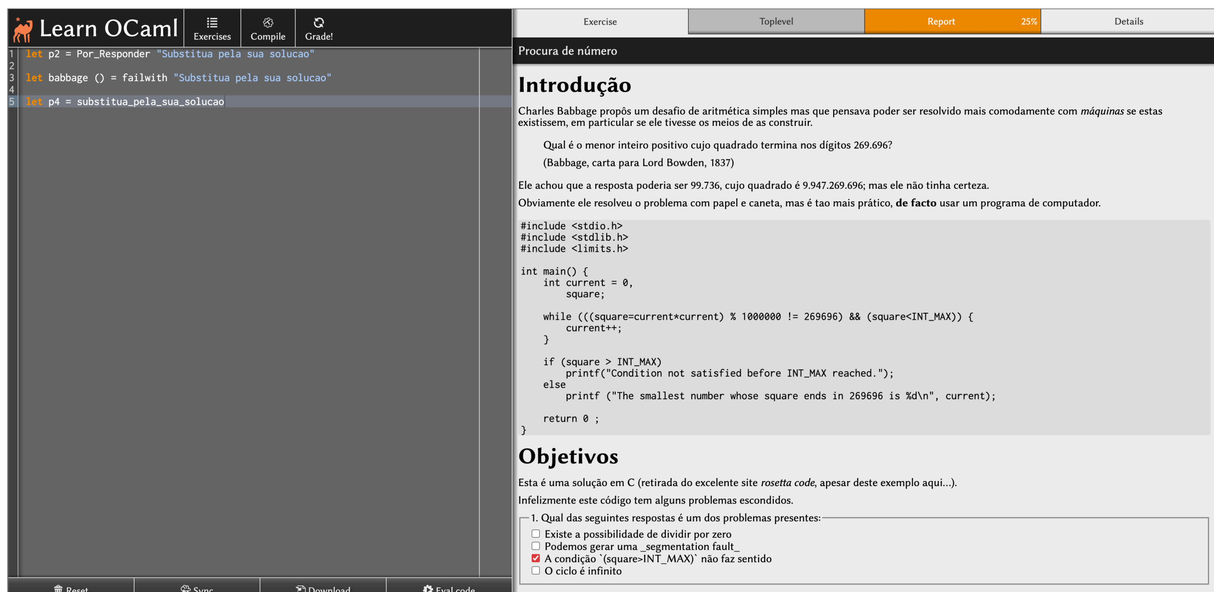


Figure 3.4: Interface of multiple choice questions.

The successful integration of MCQs into the Learn-OCaml platform represents a significant advancement in our OCaml programming education efforts. These MCQs offer a vital dimension to the learning experience, roviding students with a structured and interactive approach to test their knowledge and understanding.
Our customized instance of Learn-OCaml represents a pivotal step forward in the realm of

OCaml programming education. By enhancing the platform with specialized tools and interactive worksheets, we've created an enriched learning environment tailored to the unique needs of OCaml students.

## 3.2 The OCaml Specification Language: GOSPEL

In programming and software engineering, precision and clarity are paramount when expressing the intended behavior of code. To address this, a robust specification language becomes essential. Such a language provides a structured and formal means of articulating the expectations and constraints of a program, ensuring that code aligns with its intended functionality.

For our mission in enhancing OCaml programming education, we require a specification language that not only offers expressiveness but is also tailored to the intricacies of OCaml. Enter Gospel, a specification language designed explicitly for OCaml interfaces. Gospel significantly simplifies the process of defining code contracts and behavioral properties, making it particularly suited for OCaml's functional programming paradigm.

Unlike other specification languages, Gospel streamlines the specification process by employing Separation Logic as its foundation. Separation Logic provides a straightforward semantics for Gospel's constructs, enhancing the verification process and code correctness. It addresses the need for describing side effects, including the scope and nature of mutable state manipulations [41].

Gospel, which stands for "Generic OCaml SPEcification Language", offers a concise and accessible for students to write code contracts.

Gospel specifications are succinctly embedded within specially designated comments, initiated with the '@' character. These comments can be affixed to type declarations or value declarations, offering a means to specify the desired behavior of the associated signature items.

To illustrate how Gospel facilitates precise specification, let's consider the straightforward example used in Gospel's own website [42]. Imagine we're tasked with defining a generic interface for polymorphic, limited capacity containers in OCaml. This interface would encompass operations such as creating a container with a specified capacity, checking if it's empty, clearing its contents, adding elements while ensuring a capacity limit, and verifying the presence of an element within the container in listing 1.

Gospel allows us to precisely specify the contract of these functions using special comments marked with the @ character. For instance, consider the $create$ function. We specify that it returns a container ($t$) and lay down specific conditions in listing 2.

This specification provides critical information: it states that the provided capacity must be positive, the returned container's capacity must match the provided value, and the container should be initially empty. With such precise specifications, Gospel empowers developers to communicate and enforce code contracts, ensuring that code behaves exactly as intended.

Moreover, Gospel has laid the foundation for the development of a suite of powerful tools designed to enhance the OCaml programming experience. These tools harness Gospel's formal

**Listing 1** Module that specifies the polymorphic container.

```
type 'a t
(** The type for containers. *)

exception Full

val create: int -> 'a t
(** [create capacity] is an empty container whose maximum capacity
    is [capacity]. *)

val is_empty: 'a t -> bool
(** [is_empty t] is [true] iff [t] contains no elements. *)

val clear: 'a t -> unit
(** [clear t] removes all values in [t]. *)

val add: 'a t -> 'a -> unit
(** [add t x] adds [x] to the container [t], or raises [Full] if
    [t] has reached its maximum capacity. *)

val mem: 'a t -> 'a -> bool
(** [mem t x] is [true] iff [t] contains [x]. *)
```

**Listing 2** Specification of create function.

```
val create: int -> 'a t
(*@ t = create c
    requires c > 0
    ensures t.capacity = c
    ensures t.contents = Set.empty *)
```

semantics and its capability to specify OCaml interfaces.

**Cameleer**, extends Gospel to implementation files. This tool facilitates semi-automated deductive verification of OCaml programs, ensuring that the specified code contracts and invariants are upheld during program execution [43].

**Ortac**, focuses on runtime assertion checking. Built on Gospel's specifications, Ortac generates verifying code for test suites and program monitors, providing real-time feedback on code correctness and helping to identify and rectify any deviations from the specified behavior [44].

**Why3gospel** is a Why3 plugin allows developers to verify that a program's proof aligns with the Gospel specifications before extracting it to OCaml. By seamlessly integrating Gospel with Why3, it streamlines the process of ensuring that code contracts and invariants are faithfully adhered to throughout the development lifecycle [45].

### 3.2.1 Cameleer

Of this tools, Cameleer stands as a transformative tool bridging the gap between Gospel specifications and OCaml implementation files, with an emphasis on facilitating seamless integration.

Gospel, offers a robust platform for specifying code contracts and type invariants. Cameleer extends this functionality by enabling these Gospel specifications to directly influence OCaml implementation files, adding loop invariants, variants and many other interesting details. It streamlines this process, making it accessible and practical for OCaml programmers.

The core mechanism of Cameleer involves taking Gospel-annotated OCaml programs and translating them into an equivalent representation in WhyML, the programming language of the Why3 framework [40].

To further illustrate how Cameleer leverages Gospel specifications within OCaml implementation files, let's delve into a practical example directly from Cameleer's page [46]. Consider the following OCaml code snippet, which defines a function for calculating Fibonacci numbers in listing 3.

**Listing 3** Specifing Fibbonaci algorithm using Cameleer.

```
let fibonacci n =
  let y = ref 0 in
  let x = ref 1 in
  for i = 0 to n - 1 do
    (*@ invariant !y = fib i && !x = fib (i+1) *)
    let aux = !y in
    y := !x;
    x := !x + aux
  done;
  !y
(*@ r = fibonacci n
      requires n >= 0
      ensures  r = fib n *)
```

These comments serve as formal specifications for the function, articulating the preconditions, postconditions, and loop invariants. Cameleer focus on extracting and utilizing these Gospel specifications, translating them into WhyML specifications that can be further employed in automated deductive verification processes. This capability empowers OCaml programmers to seamlessly integrate formal specifications into their codebases, ensuring precise behavior and contributing to enhanced code correctness and reliability.

## 3.3 Conclusion

In the course of this chapter, we've explored how Gospel, in tandem with Cameleer, plays a pivotal role in the seamless integration of formal specifications into OCaml code. This powerful synergy facilitates the direct translation of Gospel specifications into OCaml implementation files, ensuring that the code aligns precisely with its intended behavior, as articulated in the specifications.

Building upon this integration, we have harnessed the modified Gospel from Cameleer to develop a unique tool. This tool is designed to take Gospel specifications, which were seamlessly incorporated into OCaml code, and translate them into assertions. It will not only streamlines the specification process but also ensures that code behaves in accordance with the formalized expectations, fostering code quality and software reliability in the context of OCaml programming.

# Chapter 4

# Proposed Solution

In this chapter, we will delve into the core process of translating Gospel specifications into essential components of code contracts and invariants. These components include preconditions, postconditions, invariants, and variants. Each plays a crucial role in ensuring code correctness and reliability by defining specific conditions and expectations that the code must meet.

## 4.1 Translating Components

In this section, we delve into the practical process of translating Gospel specifications into code contracts and invariants. Each of these components—preconditions, postconditions, invariants, and variants—plays a fundamental role in ensuring the correctness and reliability of OCaml code. We will outline the specific steps involved in translating these components, highlighting their significance in maintaining software integrity
The translation of components from Gospel specifications to OCaml code is a systematic process facilitated by a modified version of Gospel [47]. Common to each of these components is the approach we employ to update untyped Abstract Syntax Tree (AST) nodes marked as functions within the code.
We initiate the translation process by iterating through the $s_structure$, which represents a list of $s_structure_items$ in the AST. For each $s_structure_item$, we examine the $sstr_desc$, if this descriptor denotes a $Str_value$ it marks the beginning of a function in the AST. Within this descriptor, we validate the presence of a Gospel specification within the $s_value_bindinglist$. If a valid specification is found, we proceed to address it; otherwise, we move on to the next item in $s_structure$ (Listing 4).
To transform these specifications into executable code, we delve into the translation of data structures, particularly from $term$ to $s_expression$. This translation bridges the gap between the formal specifications expressed in Gospel and their integration into the OCaml codebase, thereby ensuring that each component is effectively enforced for code correctness and reliability (Listing 5.

### 4.1.1 Precondition

Preconditions, within the context of code contracts and invariants, serve as vital conditions that must be satisfied before a function or method is executed. They are instrumental in ensuring that these functions and methods operate under specific, well-defined circumstances. The significance of preconditions lies in their role as safeguards that prevent unintended or erroneous execution, contributing to the overall reliability and robustness of code.

**Listing 4** Data structures involved with the process of finding specification.

```
(...)
s_structure = s_structure_item list

and s_structure_item = {
  sstr_desc : s_structure_item_desc;
  sstr_loc : Location.t;
}

and s_structure_item_desc =
  (...)
  | Str_value of rec_flag * s_value_binding list
  (...)

and s_value_binding = {
  spvb_pat : Parsetree.pattern;
  spvb_expr : s_expression;
  spvb_attributes : attributes;
  spvb_vspec : val_spec option;
  spvb_loc : Location.t;
}

type val_spec = {
  sp_header : spec_header option;
  sp_pre : term list;
  (...)
  sp_post : term list;
  (...)
}
```

**Listing 5** Data structures involved with the process of translating specification.

```
(...)
type term = { term_desc : term_desc; term_loc : Location.t }

and term_desc =
  | Ttrue
  | Tfalse
  | Tconst of constant
  (...)

and s_expression = {
  spexp_desc : s_expression_desc;
  spexp_loc : Location.t;
  spexp_loc_stack : Location.t list;
  spexp_attributes : attributes; (* ... [@id1] [@id2] *)
}

and s_expression_desc =
  | Sexp_ident of Longident.t loc
  | Sexp_constant of constant
  (...)
```

Preconditions act as gatekeepers, verifying that the input conditions are met and that the function or method can proceed safely. By enforcing these conditions, preconditions aid in the early detection of potential issues and errors, reducing the likelihood of unexpected behavior or runtime errors within the codebase.
The process begins by identifying the presence of preconditions within Gospel specifications. Specifically, we focus on the *val_spec* section, where function specifications are defined. Within this section, we examine the *sp_pre*, which contains a list of preconditions associated with a particular function.
Once identified, these preconditions are systematically translated from their Gospel representation in *term* into the *s_expression* format. This translation is crucial to bridge the gap between formal specifications and actionable OCaml code. The translated preconditions are then seamlessly integrated at the beginning of the respective function or method within the OCaml code.
Taking the listing 6 as an example the resulting code is in listing 7

### 4.1.2 Postcondition

Postconditions play a pivotal role in the realm of software development, providing a means to validate and assert the expected outcomes of functions and methods. These conditions define the properties, constraints, or expectations that must hold true after the execution of a specific code segment. While they are integral to ensuring the correctness and reliability of software, their importance extends beyond mere verification.
In essence, postconditions serve as a form of documentation that clarifies the expected behavior and outcomes of functions, facilitating not only verification but also comprehension

**Listing 6** Code before the precondition is translated.

```
let f n0 =
  let n = ref n0 in
  let r = ref 1 in
  while (!n <> 0) do
  (*@
    invariant n0 >= !n
    invariant !n >= 0
    invariant fact n0 = !r * fact !n
    variant !n
   *)
    r := !r * !n;
    n := !n - 1
  done;
  !r
(*@
    r = f n0
      requires n0 >= 1
      ensures result = fact n0
*)
```

**Listing 7** Code after the precondition is translated.

```
let f n0 =
  assert (n0 >= 1);
  let n = ref n0 in
  let r = ref 1 in
  while (!n <> 0) do
  (*@
    invariant n0 >= !n
    invariant !n >= 0
    invariant fact n0 = !r * fact !n
    variant !n
   *)
    r := !r * !n;
    n := !n - 1
  done;
  !r
(*@
    r = f n0
      ensures result = fact n0
*)
```

and maintenance of code. By providing a clear and unambiguous description of what a function accomplishes, postconditions enable developers to write code with greater confidence and foster collaboration by making the code's behavior transparent to others.
Similar to the translation of preconditions, the process begins by identifying the presence of post-conditions for each function. This time, our focus is on the $sp_post$ and $sp_header$ sections, which contain a lists of postconditions, and the header for the specification associated with a particular function.
Once postconditions are identified, a new function, we call as a "wrapper function", is created. This function bears the same name as the main function but appends a $_post$ suffix. For example, if the main function is named $f$, the corresponding wrapper function would be $f_post$.
The body of the wrapper function is constructed using information from the Gospel specification's $sp_header$. Within the wrapper function, the main function (e.g., $f$) is invoked, and its result is captured and assigned to a variable based on $sp_header$. This captures the actual result produced by the main function's execution. The heart of the postcondition validation lies in the assertions incorporated after the call to the main function, within the wrapper function.
Taking the listing 6 as an example the resulting code is in listing 8.

**Listing 8** Code after the postcondition is translated.

```
let f n0 =
  (...)

let f_post n0 =
  let result = f n0 in
  assert (result = fact n0)
```

### 4.1.3 Invariants and Variants

Invariants are conditions or properties that remain unchanged throughout the execution of a program. They provide assurances that specific properties or relationships between program components hold true during the program's execution. Invariants act as essential guards against unintended changes to program state and are instrumental in preserving program correctness.
Variants, on the other hand, are conditions that measure or bound the progress of loops or recursive functions. They play a pivotal role in guaranteeing the termination of program elements. Variants ensure that a certain metric decreases with each iteration, ultimately leading to the termination of loops or recursive processes.
To translate invariants and variants we initiate the process by iterating through $spvb_expr$ of $s_value_binding$ within the Gospel specification. This iteration allows us to identify the presence of loops. Within the identified loops, we specifically check for the existence of a $loop_spec$ in the Gospel specification. The $loop_spec$ denotes the presence of loop-related specifications (Listing 9).

**Listing 9** Relevant data structures for loop specification.

```
type loop_spec = { loop_invariant : term list; loop_variant : term list }
(...)

and s_expression_desc =
  (...)
  | Sexp_while of s_expression * s_expression * loop_spec option
  | Sexp_for of
      Parsetree.pattern
      * s_expression
      * s_expression
      * direction_flag
      * s_expression
      * loop_spec option
```

For each identified invariant within a loop, we proceed to translate it into assertions. These assertions encapsulate the invariant conditions, ensuring that the specified properties or relationships hold true at the beginning and end of the loop.

For variants, we take a different approach. Instead of translating directly into assertions, we create separate variables for each identified variant condition. These variables are initialized at the beginning of the loop. At the end of each iteration, the value of each variable is compared with its value at the beginning of the loop. It is essential that these comparisons ensure the decrease of the variant's value, confirming progress toward loop termination. In addition, there's one more assertion to check if the variants are greater than zero.

Taking the listing 6 as an example the resulting code is in listing 10

**Listing 10** Code after the invariants and variants are translated.

```
let f n0 =
  (...)
  while (!n <> 0) do
    assert (n0 >= !n);
    assert (!n >= 0);
    assert (fact n0 = !r * fact !n);
    assert (!n > 0);

    let variant_1 = !n in

    r := !r * !n;
    n := !n - 1;

    assert (n0 >= !n);
    assert (!n >= 0);
    assert (fact n0 = !r * fact !n);
    assert (variant_1 > !n)

  done;
  (...)
```

## 4.2 Assessments

The developed tool has proven effective in translating Gospel specifications into assertions within a subset of OCaml programs, we developed a worksheet with 6 exercises (including the example used above) that the specifications can be translated and tested in using this tool.

One point that we aim to improve is the handling of quantifiers within specifications. Quantifiers, such as $universal(\forall)$ and $existential(\exists)$ quantifiers, play a crucial role in specifying complex logical conditions and mathematical expressions. As of the current state of development, the tool does not provide an efficient solution for incorporating quantifiers into the translated code (more details in the 5.1 section).

## 4.3 Conclusion

In this chapter, we have delved into the practical implementation of a tool designed to translate Gospel specifications into actionable assertions within OCaml code. We started by providing an overview of the translation process, emphasizing the critical role this tool plays in enhancing the reliability and correctness of OCaml programs.

Throughout the chapter, we explored the translation of various components, including preconditions, postconditions, invariants, and variants, all of which are essential for specifying the expected behavior and correctness properties of OCaml functions and methods.

We also acknowledged the tool's current limitations, particularly in the handling of quantifiers within specifications. This limitation serves as a stepping stone for future enhancements and developments, as we strive to expand the tool's capabilities and accommodate more intricate logical conditions.

As we move forward, the tool's integration into the Learn-OCaml platform holds promise for enriching the educational experience of OCaml programming students. By providing automated verification and feedback mechanisms, this tool empowers learners to write more reliable and correct code while enhancing their understanding of Gospel and specification-driven development.

# Chapter 5

# Conclusion

In this dissertation, we explored the integration of code contracts and invariants into the OCaml programming education environment. We began by delving into the landscape of online classroom environments, specifically focusing on the Learn-OCaml project. Our primary objective was to enhance the educational experience in OCaml programming by introducing tools and methodologies that promote software reliability and correctness.

Throughout our research, we emphasized the vital importance of software reliability and correctness in educational software. The proliferation of tools like generative Artificial Intelligence (AI), such as GitHub Copilot and ChatGPT, has raised concerns about students not fully comprehending the code they write. This issue motivated our exploration of code contracts and invariants as a means to address this challenge. We aimed to empower students to write more reliable code while gaining a deeper understanding of program behavior.

We developed a tool that seamlessly integrate code contracts and invariants into OCaml programs, allowing students to express their expectations about program behavior explicitly. These tools facilitate automated checking of code against these specifications. Additionally, we expanded the platform's capabilities by introducing features like automatic exercise assignment based on student proficiency levels and the incorporation of multiple-choice questions to enhance the learning experience.

Gospel provided a structured framework for describing the expected behavior of OCaml functions and methods, enabling us to bridge the gap between code and specifications. We leveraged a modified version of Gospel to translate code contracts and invariants into actionable elements within OCaml code.

As we conclude this dissertation, we have not only achieved our primary objectives but have also paved the way for future advancements in OCaml programming education. Our work demonstrates the potential of integrating code contracts and invariants into the learning process, offering a more comprehensive and interactive educational experience. We look forward to further developments in this field and the broader impact of our contributions on the world of OCaml programming.

## 5.1 Future Work

While this dissertation has achieved significant milestones in the integration of code contracts and invariants into the OCaml programming, there remain several avenues for future work and enhancements. Here, we outline some promising directions for future research and development.

**Add Support for Quantifiers** One of the key challenges we faced during this research was the efficient handling of quantifiers within code contracts. Incorporating support for quantifiers, such as $universal()$ and $existential()$ quantifiers, would be a valuable addition to our toolset. This would enable more expressive and comprehensive specifications, allowing students to reason about program behavior at a higher level.

**Incorporate the Tool into Learn-OCaml** While we have developed tools for code contract translation and specification integration, the ultimate goal is to seamlessly incorporate these tools into the Learn-OCaml platform. This would require developing a mechanism for grading student code based on the presence and correctness of assertions. This endeavor would enhance the platform's capabilities, making it a more powerful and interactive tool for teaching OCaml programming.

**Real-world Validation** While our research has introduced innovative tools and methodologies for integrating code contracts and invariants into programming education, it's important to acknowledge that we have yet to conduct real-world validation in a classroom environment. At the time this dissertation was developed, the tools were still in the prototype stage and not fully ready for deployment in an active classroom setting. Therefore, we recognize the need for thorough validation and assessment in a real educational context.

In conclusion, this dissertation lays the foundation for a promising approach to programming education by integrating code contracts and invariants. While our research has made significant strides in this direction, there is ample room for further exploration and refinement. The future work outlined here represents an exciting roadmap for enhancing the educational experience and promoting software reliability in programming education.

# Bibliography


[1] A. D. Dumford and A. L. Miller, "Online learning in higher education: exploring advantages and disadvantages for engagement," *Journal of Computing in Higher Education*, vol. 30, no. 3, pp. 452–465, Dec. 2018. [Online]. Available: http://link.springer.com/10.1007/s12528-018-9179-z 1

[2] B. Canou, G. Henry, Ç. Bozman, and F. Le Fessant, "Learn OCaml, An Online Learning Center for OCaml," in *OCaml Users and Developers Workshop 2016*, Nara, Japan, Sep. 2016. [Online]. Available: https://inria.hal.science/hal-01352015 1

[3] OpenAI, "Chatgpt." [Online]. Available: https://openai.com/chatgpt 1, 5

[4] GitHub, "Github copilot · your ai pair programmer." [Online]. Available: https://github.com/features/copilot 1, 5

[5] European Parliament. Directorate General for Parliamentary Research Services., *Rethinking education in the digital age.* LU: Publications Office, 2020. [Online]. Available: https://data.europa.eu/doi/10.2861/84330 5

[6] S. Dash, S. Samadder, A. Srivastava, R. Meena, and P. Ranjan, "Review of Online Teaching Platforms in the Current Period of COVID-19 Pandemic," *Indian Journal of Surgery*, vol. 84, no. S1, pp. 12–17, Apr. 2022. [Online]. Available: https://link.springer.com/10.1007/s12262-021-02962-4 5

[7] M. C. Lewis, D. Blank, K. Bruce, and P.-M. Osera, "Uncommon Teaching Languages," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. Memphis Tennessee USA: ACM, Feb. 2016, pp. 492–493. [Online]. Available: https://dl.acm.org/doi/10.1145/2839509.2844666 5

[8] J. Gillett-Swan, "The Challenges of Online Learning: Supporting and Engaging the Isolated Learner," *Journal of Learning Design*, vol. 10, no. 1, p. 20, Jan. 2017. [Online]. Available: https://www.jld.edu.au/article/view/293 6

[9] A. H. Huang, "Challenges and Opportunities of Online Education," *Journal of Educational Technology Systems*, vol. 25, no. 3, pp. 229–247, Mar. 1997. [Online]. Available: http://journals.sagepub.com/doi/10.2190/DE8W-DA78-FH16-5K89 6

[10] D. Benta, G. Bologa, S. Dzitac, and I. Dzitac, "University Level Learning and Teaching via E-Learning Platforms," *Procedia Computer Science*, vol. 55, pp. 1366–1373, 2015. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1877050915015987 7

[11] M. Mirzayanov. [Online]. Available: https://codeforces.com/ 7

[12] "Domjudge." [Online]. Available: https://www.domjudge.org/ 7

[13] J. C. Paiva, "Mooshak," Oct 2016. [Online]. Available: https://mooshak2.dcc.fc.up.pt/ 7

[14] J. P. Leal and F. Silva, "Mooshak: a Web-based multi-site programming contest system," *Software: Practice and Experience*, vol. 33, no. 6, pp. 567–581, May 2003. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/spe.522 8

[15] M. T. Pham and T. B. Nguyen, "The DOMJudge Based Online Judge System with Plagiarism Detection," in *2019 IEEE-RIVF International Conference on Computing and Communication Technologies (RIVF)*. Danang, Vietnam: IEEE, Mar. 2019, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/8713763/ 8

[16] T. Meggendorfer, "Domtutor." [Online]. Available: https://tobias.meggendorfer.de/domtutor.html 8

[17] M. Mirzayanov, O. Pavlova, P. Mavrin, R. Melnikov, A. Plotnikov, V. Parfenov, and A. Stankevich, "Codeforces as an Educational Platform for Learning Programming in Digitalization," *OLYMPIADS IN INFORMATICS*, pp. 133–142, Dec. 2020. [Online]. Available: https://ioinformatics.org/journal/v14_2020_133_142.pdf 8

[18] J. P. Leal and F. Silva, "Using mooshak as a competitive learning tool," in *The 2008 Competitive Learning Symposium*, 2008. 8

[19] "Coursera." [Online]. Available: https://www.coursera.org/ 8

[20] C. Severance, "Teaching the World: Daphne Koller and Coursera," *Computer*, vol. 45, no. 8, pp. 8–9, Aug. 2012. [Online]. Available: http://ieeexplore.ieee.org/document/6272250/ 8

[21] "edx." [Online]. Available: https://www.edx.org/ 8

[22] C. P. Rosé, O. Ferschke, G. Tomar, D. Yang, I. Howley, V. Aleven, G. Siemens, M. Crosslin, D. Gasevic, and R. Baker, "Challenges and opportunities of dual-layer MOOCs: Reflections from an edX deployment study," in *Proceedings of the 11th international conference on computer supported collaborative learning (CSCL 2015)*, vol. 2. International Society of the Learning Sciences, 2015, pp. 848–851. 8

[23] "Fun mooc." [Online]. Available: https://www.fun-mooc.fr/en/ 8

[24] R. Di Cosmo, Y. Regis-Gianas, and R. Treinen, "Introduction to functional programming in OCaml." [Online]. Available: https://www.fun-mooc.fr/en/courses/introduction-functional-programming-ocaml/ 8

[25] S.-W. Kim, "MOOCs in Higher Education," in *Virtual Learning*, D. Cvetkovic, Ed. InTech, Dec. 2016. [Online]. Available: http://www.intechopen.com/books/virtual-learning/moocs-in-higher-education 8

[26] A. Hameer and B. Pientka, "Learn-OCaml Extensions and Repository for Article Teaching the Art of Functional Programming using Automated Grading (Experience

Report),” McGill University, Tech. Rep., Jul. 2019, type: dataset. [Online]. Available: https://dl.acm.org/do/10.1145/3342529/abs/ 11

[27] C. Marchéé, “Basics of deductive program verification,” https://www.lri.fr/~marche/MPRI-2-36-1/2019-2020/lecture1-raw.pdf, 2019, accessed: 2023–09-26. 11

[28] C. Marché, “More advanced topics in program verification,” https://www.lri.fr/~marche/MPRI-2-36-1/2019-2020/lecture2-raw.pdf, 2017, accessed: 2023–09-26. 11

[29] M. Fahndrich, M. Barnett, and F. Logozzo, “Embedded Contract Languages,” in *ACM SAC - OOPS*. Association for Computing Machinery, Inc., Mar. 2010, edition: ACM SAC - OOPS. [Online]. Available: https://www.microsoft.com/en-us/research/publication/embedded-contract-languages/ 12

[30] M. Fähndrich, “Static verification for code contracts,” in *Static Analysis: 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings 17*. Springer, 2010, pp. 2–5. 12

[31] Jun. 2017, publication Title: Microsoft Research. [Online]. Available: https://www.microsoft.com/en-us/research/project/code-contracts/ 12

[32] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and others, “JML reference manual,” 2008. 13

[33] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. Rustan, M. Leino, and E. Poll, “An overview of JML tools and applications1 1www.jmlspecs.org,” *Electronic Notes in Theoretical Computer Science*, vol. 80, pp. 75–91, Aug. 2003. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1571066104808107 13

[34] G. T. Leavens and Y. Cheon, “Design by Contract with JML,” 2006. 13

[35] Y. Cheon, *A runtime assertion checker for the Java Modeling Language*. Iowa State University, 2003. 13

[36] L. Friendly, “The design of distributed hyperlinked programming documentation,” in *Hypermedia Design: Proceedings of the International Workshop on Hypermedia Design (IWHD’95), Montpellier, France, 1–2 June 1995*. Springer, 1996, pp. 151–173. 13

[37] Y. Cheon and G. T. Leavens, “A Simple and Practical Approach to Unit Testing: The JML and JUnit Way,” in *ECOOP 2002 — Object-Oriented Programming*, ser. Lecture Notes in Computer Science, B. Magnusson, Ed. Berlin, Heidelberg: Springer, 2002, pp. 231–255. 13

[38] B. Meyer, “Eiffel: A language and environment for software engineering,” *Journal of Systems and Software*, vol. 8, no. 3, pp. 199–246, Jun. 1988. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0164121288900222 13

[39] R. B. Findler and M. Felleisen, "Contracts for higher-order functions," in *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*. Pittsburgh PA USA: ACM, Sep. 2002, pp. 48–59. [Online]. Available: https://dl.acm.org/doi/10.1145/581478.581484 13

[40] J.-C. Filliâtre and A. Paskevich, "Why3 — Where Programs Meet Provers," in *Programming Languages and Systems*, ser. Lecture Notes in Computer Science, M. Felleisen and P. Gardner, Eds. Berlin, Heidelberg: Springer, 2013, pp. 125–128. 13, 22

[41] A. Charguéraud, J.-C. Filliâtre, C. Lourenço, and M. Pereira, "GOSPEL—providing OCaml with a formal specification language," in *Formal Methods–The Next 30 Years: Third World Congress, FM 2019, Porto, Portugal, October 7–11, 2019, Proceedings 3*. Springer, 2019, pp. 484–501. 20

[42] "Your First Specification | Gospel." [Online]. Available: https://ocaml-gospel.github.io/gospel/getting-started/first-spec 20

[43] M. Pereira and A. Ravara, "Cameleer: A Deductive Verification Tool for OCaml," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, pp. 677–689. 22

[44] J.-C. Filliâtre and C. Pascutto, "Ortac: Runtime Assertion Checking for OCaml (Tool Paper)," in *Runtime Verification*, ser. Lecture Notes in Computer Science, L. Feng and D. Fisman, Eds. Cham: Springer International Publishing, 2021, pp. 244–253. 22

[45] "Why3Gospel," Sep. 2023, original-date: 2020-11-20T15:51:49Z. [Online]. Available: https://github.com/ocaml-gospel/why3gospel 22

[46] "Cameleer," Sep. 2023, original-date: 2020-05-31T23:49:09Z. [Online]. Available: https://github.com/ocaml-gospel/cameleer 22

[47] "ocaml-gospel/gospel at implementations_gospel." [Online]. Available: https://github.com/ocaml-gospel/gospel/tree/implementations_gospel 25