NOVA
NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

DIOGO RAFAEL REBOCHO SILVÉRIO
Bachelor Degree in Science and Computer Engineering

EFFICIENT DECLARATIVE PROGRAMMING IN OCAML

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2021

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

EFFICIENT DECLARATIVE PROGRAMMING IN OCAML

DIOGO RAFAEL REBOCHO SILVÉRIO

Bachelor Degree in Science and Computer Engineering

Adviser: Dr. Artur Miguel de Andrade Vieira Dias
Assistant Professor, NOVA University Lisbon

Co-adviser: Dr. Mário José Parreira Pereira
Postdoctoral Researcher, NOVA University Lisbon

Examination Committee

Chair: Doutor Carlos Augusto Isaac Piló Viegas Damásio
Professor Associado, Universidade NOVA de Lisboa

Rapporteur: Doutor Simão Patrício Melo de Sousa
Professor Associado com Agregação, Faculdade de Engenharia da Universidade da Beira Interior

Advisers: Doutor Artur Miguel de Andrade Vieira Dias
Professor Auxiliar, Universidade NOVA de Lisboa

Doutor Mário José Parreira Pereira
Investigador Pós-doutorado, Universidade NOVA de Lisboa

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2021

Efficient Declarative Programming in OCaml

Copyright © Diogo Rafael Rebocho Silvério, NOVA School of Science and Technology, NOVA University Lisbon.
The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.9.6) [20].

This dissertation is dedicated to my parents.
They have done everything possible and impossible to make
sure I have had the best life one could ever ask for.
There is not a moment in my life when they were not there to
support me, and I fear the day I must say otherwise.
Their love is the kind that makes one wish for eternity.

Acknowledgements

I would like to thank the OCaml software foundation for funding the LEAFS-OCaml project. I would also like to express my gratitude to both my advisors for believing in me and helping me whenever I needed. Finally, I would like to thank all the teachers that have taught me throughout my life. I wouldn't be the person I am today without them, and I certainly wouldn't have been capable of writing this dissertation.
Teachers are the pillars of society.

“We can only see a short distance ahead,
but we can see plenty there that needs to be done.”
(Alan Turing)

Abstract

Software gets more complex each day, specially with the growing popularity of IoT, Parallel Computing and interactive web applications. The opportunity for error grows hand in hand with the ever growing complexity of these systems. Declarative programming allows developers to focus on solving a problem and avoid the complexity of dealing with programming low-level elements, such as memory management.

Unfortunately, some pure declarative programs tend to perform poorly from an executional point of view compared with their equivalent imperative counterparts. These are more machine-friendly which turns the use of declarative programming less appealing to the general audience.

The goal of this dissertation is to create several OCaml PPX (PreProcessor eXtension) rewriters that transform purely declarative OCaml programs into equivalent ones with better executional performance. These PPX rewriters allow the user to automatically improve the performance of their code. Our biggest aim is to promote the use of the declarative paradigm. The motivation is two-fold: one, developers can enjoy natural, intuitive, and elegant declarative solutions to their problems; second, increase code correctness, decrease code size and ease code verification.

Keywords: Efficient Declarative Programming, OCaml PreProcessor eXtensions, Code Optimization, Program Transformations

Resumo

O software tem-se tornado cada vez mais complexo, especialmente com o aumento de popularidade do IoT, da Computação Paralela e de aplicações interactivas da Web. A oportunidade de erro cresce de mãos dadas com a complexidade crescente destes sistemas. A programação declarativa permite aos programadores concentrarem-se na resolução de um problema e evitar a complexidade de ter de lidar com a programação de elementos de baixo nível, como a gestão de memória.

Infelizmente, alguns programas declarativos puros tendem a ter um mau desempenho do ponto de vista da execução em comparação com os seus equivalentes imperativos. Estes são menos custosos de executar para uma máquina, o que torna o uso da programação declarativa menos atraente para o público geral.

O objectivo desta dissertação é criar vários reescritores OCaml PPX (PreProcessor eXtension) que transformam programas OCaml puramente declarativos em programas equivalentes com melhor desempenho de execução. Estes rescritores deixam o utilizador melhorar o desempenho do seu código automaticamente. O nosso maior objectivo é promover o uso do paradigma declarativo. A motivação é dupla: primeiro, os programadores podem desfrutar de soluções declarativas naturais, intuitivas e elegantes para os seus problemas; segundo, melhorar a correção do código, diminuir o seu tamanho e facilitar a sua verificação.

Palavras-chave: Programação Declarativa Eficiente, OCaml PreProcessor eXtensions, Optimização de Código, Transformação de Programas

Contents

List of Figures **x**

List of Tables **xi**

List of Listings **xii**

1 Introduction **1**

1.1 Context 1
1.2 Motivation 2
1.3 Problem Statement 3
1.4 Contributions 5
1.5 Document Structure 5

2 Background **7**

2.1 Program Transformations 7
2.2 Programming Paradigms 7
2.3 Imperative and Declarative Programming 9
2.4 OCaml 10
2.5 OCaml's PreProcessor eXtensions 11

3 Related Work **14**

3.1 The current state of the PPX world 14
3.1.1 `ppx_tools` 15
3.1.2 `ocaml-migrate-parsetree` 15
3.1.3 `ppxlib` 16
3.2 Continuation Passing Style and Defunctionalization 16
3.3 Automatic Memoization using Higher Order Functions 17
3.4 Hash-Consing 19

4 Memoization **22**

4.1 Memoization 23

4.2 Memoization with PPX 24
4.3 Memoization PPX Performance Evaluation 26
4.3.1 Number of Computations 26
4.3.2 Execution Time 26
4.3.3 Memory Consumption 27
4.3.4 Results for large inputs 28

5 Hash-Consing 30
5.1 Hash-Consing 30
5.2 Hash-Consing with PPX 32
5.2.1 Hash-Cons PPX Customization 34
5.3 Hash-Consing PPX Performance Evaluation 36
5.3.1 Memory Consumption 39
5.3.2 Execution Time 40

6 Defunctionalization 41
6.1 Defunctionalization 42
6.2 Defunctionalization with PPX 43
6.3 Defunctionalization PPX Performance Evaluation 45
6.3.1 Execution Time 45
6.3.2 Memory Consumption 46

7 Conclusions 47
7.1 Contributions 47
7.2 Future Work 48
7.3 Final Remarks 48

Bibliography 49

List of Figures

2.1 Relation between programming concepts, paradigms and languages. 8
2.2 OCaml's compilation pipeline, adapted from [24]. 12

4.1 Fibonacci function calls tree. Dashed nodes represent repeated computations. 24

List of Tables

4.1 Number of computations of both versions of the Fibonacci function. 26
4.2 Execution time of both versions of the Fibonacci function. 27
4.3 Memory consumption of both versions of the Fibonacci function. 27
4.4 Test results summary of the memoized Fibonacci function. 28

5.1 Memory consumption of creating a normal tree and a hash-consed tree. . . 39
5.2 Execution time of checking if two normal and hash-consed trees are equal. 40

6.1 Execution time of both versions of the tree height function. 45
6.2 Memory consumption of both versions of the tree height function. 46

List of Listings

2.1 List length. 9
2.2 Power set. 10
2.3 Add One PPX. 12
3.1 Memoize Function for non recursive functions. 17
3.2 Factorial function. 18
3.3 Factorial Factory function [35] and new Factorial function[15]. 18
3.4 Y combinator implementation in OCaml[35]. 19
3.5 Memoize Function for recursive and non recursive functions. 19
3.6 Structurally equal and physically different variables. 20
4.1 Fibonacci function in OCaml. 23
4.2 Fibonacci function with memoization in OCaml. 24
4.3 Fibonacci function with extension node. 24
4.4 Memoized Fibonacci function. 25
5.1 OCaml tree type. 31
5.2 Structurally equal and physically different trees. 31
5.3 Hash-cons in OCaml. 31
5.4 Equal function for trees and hash-consed trees in OCaml. 32
5.5 Term type with the hash-cons extension node. 32
5.6 Hash-consed term type. 33
5.7 Term type with the hash-cons extension node and the node to customize the name of variables. 34
5.8 Hash-consed term type with customized variable names. 34
5.9 Term type with the hash-cons extension node, the node to customize the name of variables and the node to customize the hash function. 35
5.10 Hash-consed term type with customized variable names and customized hash function. 35
5.11 Code to create a tree. 37
5.12 Code to create a tree node and an hash-consed tree node. 38
6.1 Factorial function in OCaml. 42

6.2 Defunctionalized factorial function in OCaml. 42
6.3 Tree height function with extension node. 43
6.4 Defunctionalized tree height function. 44

1

Introduction

This first chapter contains an introduction to the work done in this dissertation. It provides some context to the thesis, states its motivation and describes the problem at hand. Then it expresses the expected goals and finally announces the remaining document structure.

1.1 Context

Declarative programming is a programming paradigm that mainly shifts the user's focus from the control flow of the program to the logic of the program. The program can be seen as a theory and its computation as a deduction from said theory[19]. Programming languages that fall under this paradigm have simple syntax and semantics which translates into several advantages such as increased expressiveness, reduced code size[12], ease of code analysis, readability and verification. They also do not have a notion of state and minimize or completely eliminate the occurrence of side effects. This eliminates a major source of errors and improves code correctness according to its specification[14].

This programming paradigm has always had a significant role in both computer science and software engineering. Moreover, in the past recent years it has been rising in popularity because of its benefits in areas such as parallel and concurrent computing[13] and also front-end web development, particularly in React, a declarative JavaScript library for building user interfaces, which is used in Facebook, the most popular social media platform in the world, with over 2 billion users[40]. The paradigm's characteristics are valuable in the areas mentioned because they allow for programmers to abstract cumbersome implementation details that are not part of the actual program logic. The qualities mentioned allow for natural, intuitive and elegant declarative solutions to solve some programming problems. These would otherwise be verbose, harder to understand and more prone to error if written in a language that falls under another programming paradigm, *e.g.*, imperative programming.

Whilst declarative programming has many good qualities, it is not perfect and has its negative aspects that hinder its usage. Unfortunately, sometimes declarative solutions

end up performing poorly from an executional point of view in comparison with more machine-friendly ones. This is because they require more resources, like time and memory, to achieve the same results. The computation may result in inadmissible computation time or in a crash due to stack overflow. A possible solution to overcome the performance issues that may arise with the use of declarative programming is to manually rewrite the declarative code into an imperative one in order to regain the control flow. This allows the user to improve the executional performance along with removing the possible occurrence of stack overflow crashes. However, this approach is not trivial as it requires a complete change of the original problem and thought process resulting in a semantic gap between the solution specification and the resulting code. This alteration ends up lowering the quality of the code because it drastically lessens readability, simplicity and verification. Furthermore, finding these performance issues and making the required alterations to overcome them is left at the responsibility of the programmer even when originally there was a perfectly correct solution.

1.2 Motivation

As previously mentioned, one of the advantages of declarative programming is the abstraction of the control flow of a program. This is becoming more relevant with the rise of software development areas such as concurrent and parallel programming, the Internet of Things and web development. This is because of the ever increasing complexity of code in these areas. More devices and people connected to the internet and multi-core processors with a bigger amount of cores inevitably demand more robust and complex code.

In parallel programming several computations are performed simultaneously in multiple threads. To take advantage of this feature, developers must divide a program into smaller parts for each thread. Doing this adds complexity to the code development and specially to debugging, as code is no longer sequential and therefore it is much harder to find and correct errors. In this study[33], the authors asked students to develop parallel programs and then categorized the errors committed in order to study if there was a need for better debugging in parallel programming. The results obtained indicate that 49.68% of errors were all related to the decomposition of the problem and that students spent on average 52 minutes for each of the 155 bugs reported. As the authors put it: "This is also a good indication that it really is difficult to locate errors and correct them when dealing with a number of processes executing concurrently and communicating asynchronously[...]". By using declarative programming, students would only have to worry about correctly coding the necessary functions to solve the problems without having to think about parallelism management. The abstract solution will run in a single thread, and is also suitable for simple automatic parallelization.

In the area of web development, declarative programming can also help simplify the development of applications. A few years ago, websites were mostly static and mainly

focused on displaying data. However, much has changed recently and most websites are now dynamic single-page applications (SPA) with interactive user interfaces (UI). These applications interact with the web browser in order to dynamically rewrite parts of the website instead of loading entire new web pages. To do this, developers must manually change parts of the document object model (DOM), an abstraction model that represents the user interface elements as a tree structure, so that the UI matches the state of the application. Considering that HTML is very verbose and has semantically similar constructs, making these alterations increases the chance of making mistakes by developers[31]. The process of updating and rendering these interactive UIs can be automated by declarative programming and in fact this is what React[10] does. The library has a declarative API that allows the developer to tell React what state the UI should be in and it automatically changes the DOM to match this state. This takes the responsibility of manipulating attributes, handling events and manually updating the DOM away from developers and consequently eliminates the errors that stem from this responsibility. For this reason, React's declarative API makes designing and debugging UIs simpler. Web programming also relies on a variety of technologies like JavaScript, HTML, CSS and many others. The developers must be proficient in all of them and must maintain all the software artifacts created with each one synchronized. Furthermore, the client and server are usually developed separately. All these properties create several software components prone to error. In order to mitigate these errors, the Ocsigen framework[1], more specifically, the tierless language Eliom[34] was created in order to unify the development of the client and server by taking advantage of the expressiveness offered by the functional language.

Declarative programming is also an important tool in learning programming and problem solving. Its expressiveness and high level semantics allow students to better express themselves and consequently better formally describe problem solutions. Students can then solve more interesting and challenging problems, seeing that their abstraction skills improve, and they learn to use them as a design tool. This also means that students are more capable of succeeding in future programming courses since success is more based on the ability to use abstraction as a tool than the ability to code[16].

Leaving only the logical component to the developer and shifting the responsibility of the program's control flow from the programmer to the compiler and language implementation decreases the complexity of the code. Meaning less error opportunities and more robust code. Also, the use of declarative programming helps developers learn how to use abstraction as a design tool[16] which aids problem solving in any programming paradigm. This is why declarative programming is now more important than ever and should not be overlooked.

1.3 Problem Statement

As already mentioned declarative programming has numerous advantages, namely its expressive power combined with reduced code size and safety. These qualities have

become more relevant than ever as nowadays software systems are present everywhere as the world is ever more dependent on them and software systems ever more dependent on each other. Today, most applications are connected to the internet which means users and other programs expect these to be available all the time and any error can be very costly and have a cascade effect. In addition, software runs on more powerful central processors with multiple cores that should be leveraged with parallel computing. This means applications must be more robust and safe but still be fast. To achieve these standards, software systems and applications have become significantly more complex. The added complexity makes code harder for developers to understand, safety harder to achieve and consequently software systems become harder to develop and maintain.

Declarative programming substantially alleviates these problems but the poor performance attached to some of the code written in this paradigm turns it into a non-viable solution. In general, declarative programs work without problems. There are many people programming with OCaml and other declarative languages successfully. The execution problems arise only for large datasets or when the solution involves a recursive algorithm. Objectively, the great difference is that many imperative programs run with constant spatial complexity at the stack level while declarative programs run with linear complexity because of recursion. This is the origin of stack overflow errors for very large datasets.

A representative example would be to calculate the length of a list. The imperative solution does not spend stack memory while the recursive solution spends stack memory proportional to the length of the list. From the user's point of view, both solutions work well and are fast. However, with the recursive solution, the user observes a stack overflow if they try to calculate the length of a list with millions of elements. The Fibonacci function is a specific example of recursion that easily encounters performance issues. This is because the Fibonacci function has exponential computational complexity so when executed even with a relatively small input like 40 it performs approximately 2^{40} computations that require so much time it becomes infeasible to use. Thankfully there are techniques that can be applied to reduce the computational time complexity of this and other functions from exponential time $O(2^n)$ to even linear time $O(n)$.

The goal of this dissertation is to apply these techniques automatically making use of the OCaml PPX technology. This way users must not worry about taking into account underlying computational details that are irrelevant to the problem at hand. We also want to promote the use of declarative programming by making it as efficient as other programming paradigms. We want to be able to use this programming paradigm and its features to our advantage without having to deal with performance issues and crash errors. There are techniques that mitigate these problems but these must be applied by the developer, they are usually not easy to understand, and they require time allocated in mundane issues that could be spent in the actual program's logic.

There are tools like compilers that do perform some optimizations which alleviate some of these problems. For example, the *ocamlopt* compiler with *flambda* or *gcc* with its

optimization options *o2* and *o3*. But not all of them are activated by default and are not always easy to use or supported in all systems.

Some of these are also external tools that require the installation and management done by the developer. So we want to be able to easily write purely declarative OCaml programs without any execution problems including performance ones without the need for external tools.

1.4 Contributions

The major contributions of this dissertation are identifying the weak points of declarative programming, analyse how to best improve its pragmatic characteristics and apply techniques that will improve pure declarative definitions efficiency and promote its usage by eliminating this obstacle. To do this we developed three PreProcessor eXtensions rewriters. These are program transformation tools for OCaml that will automatically improve the efficiency of purely declarative programs by applying code optimization techniques such as memoization, hash-consing or transformation into first-order definitions. With our approach the developer must only write a small annotation wherever they desire in the code and PPX will parse the code's AST and produce a new improved one with the efficiency enhancements during the compilation process. This dissertation is also part of the LEAFS project. It is meant to promote the declarative paradigm among university students as well as teach them about program transformations and the power of optimization techniques.

1.5 Document Structure

Given the initial introduction to the dissertation we now present the document's overall structure.

- Chapter 2 - Background: in this chapter, key concepts and information about technology related with the thesis is briefly presented and discussed. The concepts presented include program transformations, programming paradigms, the imperative and declarative programming, the OCaml language and PPX.
- Chapter 3 - Related Work: in this chapter we present and briefly discuss some work being done to improve the efficiency of declarative programming and the PPX development process. We present crucial tools to work with PPX. Finally, we present common program transformation techniques such as continuation passing style, defunctionalization, memoization, hash-consing as well as other optimization transformations performed by compilers.
- Chapter 4 - Memoization: in this chapter we present the Memoization PPX. We introduce the problem. Discuss why memoization is a solution and how it solves

the problem. Present the PPX and discuss why it is useful to the problem and the benefits of using it. Then we present some benchmarks and discuss the performance of the new code transformed by the PPX.

- Chapter 5 - Hash-Consing: in this chapter we present the Hash-Consing PPX. We introduce the problem. Discuss why hash-consing is a solution and how it solves the problem. Present the PPX and discuss why it is useful to the problem and the benefits of using it. Then we present some benchmarks and discuss the performance of the new code transformed by the PPX.

- Chapter 6 - Defunctionalization: in this chapter we present the Defunctionalization PPX. We introduce the problem. Discuss why defunctionalization is a solution and how it solves the problem. Present the PPX and discuss why it is useful to the problem and the benefits of using it. Then we present some benchmarks and discuss the performance of the new code transformed by the PPX.

- Chapter 7 - Conclusions: in this chapter we reflect about the dissertation as a whole. We discuss the contributions made with it and how future work may develop.

2 Background

In this chapter, key concepts and information about technology related with the thesis is briefly presented and discussed. The concepts presented include program transformations, programming paradigms, the imperative and declarative programming, the OCaml language and the PPX.

2.1 Program Transformations

A program transformation is the process of turning one program into another. The transformation technique takes the original program as input and returns a modified output program. Usually, the transformed program is required to be semantically equivalent to the original. Many program transformations are performed by means of annotations[6] in the code which later get expanded by a compiler or by an external tool. They can also be done through a translator[21] which parses a program and then builds a new one based on a modified grammar. Another option, is through processing and templating engines that take programs as input, compare them to a template code and make the necessary transformations to make sure the input program conforms with the template[32]. The annotations process consists in automatically altering only parts of the code specified by the user, usually to enhance pragmatic characteristics of the code. While the other mentioned processes may modify the entire program. This process is done by modifying the AST or other data structures created by the compiler in an intermediate step of the compilation pipeline. An example is shown in figure 2.2. In this dissertation we will be using PPX as the tool to transform programs which is based on annotations as it is the official way to extend OCaml code.

2.2 Programming Paradigms

Programming paradigms are a product of programming language categorization and a way of gathering different programming concepts and principles together in different sets. Each paradigm consists in a set of concepts and each programming language implements

one or more of these paradigms. A visual representation of this philosophy is shown in figure 2.1. On the left, the small circles represent different programming concepts and the set of irregular shapes that contain these circles represent programming paradigms. On the right, the different sets of irregular shapes represent different programming languages that implement a set of programming paradigms. Notice that some concepts are outside all programming paradigms and some are inside more than one. This means that some concepts are not implemented by any programming paradigm and some are implemented by several paradigms, respectively. Also, many programming languages are specialized and only support one paradigm while others combine several paradigms. These are called multi-paradigm languages.

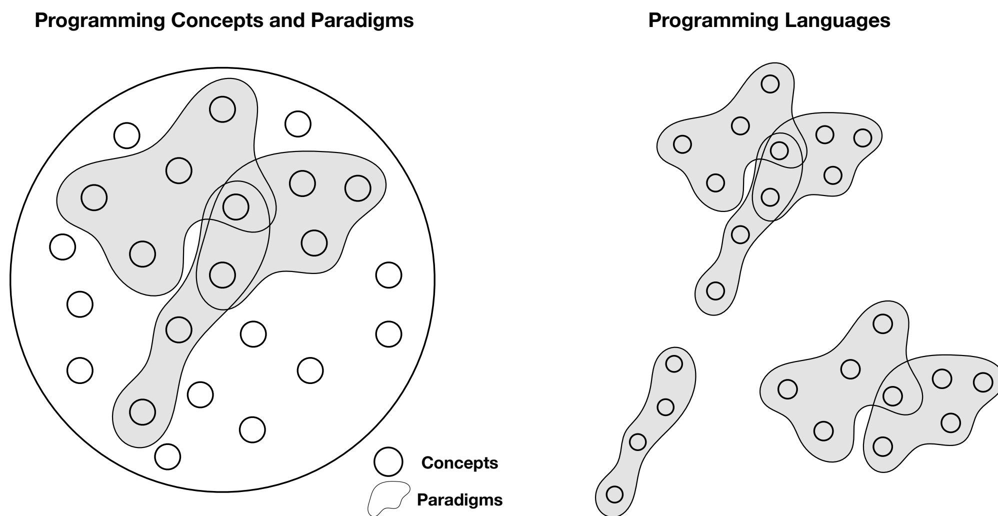

Figure 2.1: Relation between programming concepts, paradigms and languages.

In [37], the author presents a taxonomy of programming paradigms with 27 distinct paradigms organized in a graph that shows how they are related based on concepts. Theoretically there are more paradigms than these but many such as the empty paradigm or paradigms with only 1 concept are not very interesting. All the ones mentioned have good implementations and practical applications[37]. When a programming language implements a paradigm this means that the language was intentionally designed to support the set of concepts. Some other concepts implemented through external libraries or frameworks are not considered.

From all these paradigms we are interested in imperative and declarative programming, particularly in their differences, the advantages and disadvantages of each one.

2.3 Imperative and Declarative Programming

Imperative and declarative programming are often considered opposites of each other on the programming paradigm spectrum. Even though both paradigms have significant differences, this idea that both are complete opposites is not correct since they are actually very close to each other in the terms of the set of concepts that comprise them[37].

Imperative programming is characterized by explicitly telling a computer "how" to accomplish a goal. A program written using this paradigm is responsible for its own control flow and the logic is based on the use of mutable variables. The program evolves by going from one state to another state based on the statements in the code. The direct access to the program's control flow and mutable variables make it possible for imperative programs to be extremely efficient in both memory and time consumption. But in order to achieve such high quality programs, the programmer must be very knowledgeable about computer science, the language implementation and the compiler that is used. This also leaves more room for error and more often than not these features translate into memory leaks, undesired side effects and race conditions in parallel programming. Some of the most commonly known languages that implement the imperative paradigm are Java, C, C++, C# and Python.

Declarative programming is a programming paradigm characterized by telling a computer "what" to do to accomplish a goal. Unlike imperative programming the actual process of how to achieve such goal is not described in the program and is instead left to the language and compiler implementations to deal with. The notion of state and direct control flow does not exist in declarative programming. By removing direct access to control flow and mutable variables, declarative programming reduces development time and increases program safety and readability since developers no longer have to worry about cumbersome aspects outside the scope of the logical thought process. Naturally declarative programming also has its flaws, sometimes suffering from highly inefficient code in terms of time complexity and making it more complex to express input/output operations that are inherently not functional and representing state when needed.

Let us demonstrate 2 examples of declarative programs, one trivial and another one a little more sophisticated. The first one is the length of a list and is shown in listing 2.1.

Listing 2.1: List length.

```
1 let rec length l =
2   match l with
3     | [] -> 0
4     | x::xs -> 1 + len xs
```

The length of a list can be easily defined as zero when the list is empty or 1 plus the length of its tail when it is not. Notice that we do not specify anything about the control flow of the program. We also do not use state to keep track of the list's length during its computation. We are iterating all elements and counting them, but we do not specify how we are doing it. With imperative programming we would have to specify the iteration

process. We would also have to create and manage a mutable variable to keep track of the number of elements of the list.

The second example is the power set of a set. The algorithm to calculate the power set is complex. It iterates through every element and constructs sets along the way while avoiding repetitions. By using declarative programming we avoid having to think about how to implement this algorithm. We must only implement a mathematical definition. Given that S is a set and x is an element of this set, the power set is the union of all the subsets of S that contain x and all those that do not. The correspondent OCaml code is shown in 2.2

Listing 2.2: Power set.

```
let rec power l =
  match l with
    | [] -> [[]]
    | x::xs -> power xs @ List.map (fun l -> x::l) (power xs)
```

Notice that just like in the previous example, we do not specify any aspect of the programs' control flow. We simply wrote the corresponding code of the mathematical definition above. Both functions solve a problem solemnly using the problem's properties. As stated the code examples are simple and elegant solutions but suffer from performance issues. When computing these functions with large data both end up crashing due to a stack overflow error.

Taking advantage of declarative programming expressiveness and safety while complementing these features with imperative programming's time and space efficiency is the main goal of this dissertation. To achieve this, it is ideal to work with a multi-paradigm programming language that implements both paradigms such as OCaml.

2.4 OCaml

OCaml which stands for Objective Caml is a general purpose, multi-paradigm programming language with an emphasis on expressiveness and safety. Its high expressiveness makes OCaml programs short and easy to read as it is closer to human language and high level semantics. Its safety is guaranteed by compilers that verify programs' type soundness before they can be executed. The most prominent features of the language are: a powerful type system equipped with parametric polymorphism and type inference; user-definable algebraic data types which can be defined as combinations of records and sums, and pattern-matching to operate over such data structures; automatic memory management through garbage collection; it has portable bytecode compilers that allow creating stand-alone applications, a foreign function interface that allows OCaml code to interoperate with C and interactive use of OCaml via a "read-evaluate-print" loop; a sophisticated module system; an object-oriented layer featuring multiple inheritance, parametric and virtual classes; and efficient native code compilers[27].

Being a multi-paradigm programming language, OCaml supports all three imperative, object-oriented and functional paradigms. This allows for one to quickly change between the three and take advantage of the benefits of each when it is most convenient. OCaml also has its own native way of extending the language via extension points called PreProcessor eXtensions or simply PPX which we further explain in section 2.5.

The characteristics presented, especially the fact that OCaml supports imperative and declarative programming, make the language ideal to build a program transformation tool to improve the execution of declarative programs.

2.5 OCaml's PreProcessor eXtensions

The PPX OCaml feature supports two types of extension points. Extension attributes are meant to provide supplementary information about a node in the AST while extension nodes are meant to generate code. The compiler has some builtin attributes such as the *@@@warning* attribute meant to toggle an OCaml compiler warning and the *@@deprecated* attribute meant to indicate that a module should no longer be used. Extension nodes are the vehicle to generate new code, so we're going to focus on them and on the tools used to expand these nodes.

PreProcessor eXtensions or simply PPX rewriters are OCaml programs that take another program as argument and expand its extension nodes transforming the input program into a new program. These extension points represent generic placeholders in the abstract syntax tree of the program. They are rejected by the OCaml's type checker as they are expected to be expanded by tools, *e.g.*, PPX rewriters, that interpret the extension nodes and transform the input syntax tree into a new syntax tree. The new syntax tree no longer has extension nodes. This means the type checker verifies if the PPX rewriter fails to find and interpret the extension nodes.

Extension nodes have an identifier and a payload. Their general syntax in OCaml is `[%id payload]` but also have an infix form when the payload is of the same kind, *e.g.*, `let%foo x = 2 in x + 1` is equivalent to `[%foo let x = 2 in x + 1]`[18]. The identifier is the name that the PPX rewriter will look for when parsing the AST. The payload is a fragment of OCaml code. When the PPX finds the identifier it uses the payload as input and applies the transformation.

OCaml's compiler pipeline as shown in figure 2.2, generates intermediate files throughout the compilation process and branches in the end producing different results. The bytecode can be run by a portable interpreter and the native code compiler generates specialized executable binaries suitable for high-performance applications. As mentioned PPX rewriters do not actually transform OCaml programs directly. They do it by transforming the untyped abstract syntax tree generated by the language's compiler. This happens during the preprocessing stage resulting in a modified untyped AST. This is highlighted in figure 2.2.

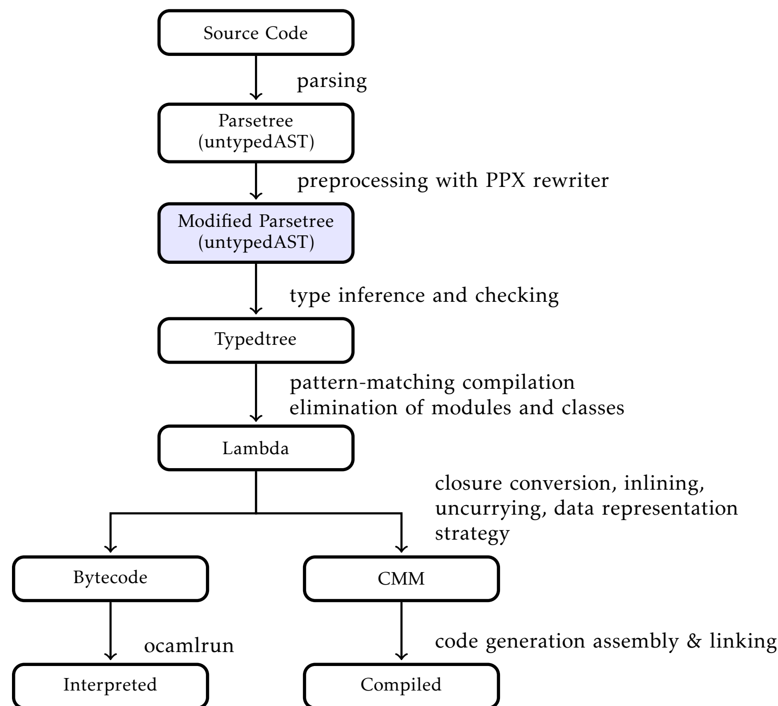

Figure 2.2: OCaml's compilation pipeline, adapted from [24].

Let us look at an example of a PPX. The Add One PPX is a simple PPX rewriter that looks for the extension node *addone* and adds 1 to an expression. It is shown in listing 2.3.

Listing 2.3: Add One PPX.

```
open Ast_mapper
open Ast_helper
open Asttypes
open Parsetree
open Longident

let rec expr_mapper mapper expr =
  begin match expr with
    | { pexp_desc = Pexp_extension ({ txt = "addone"; loc }, pstr); _} ->
      begin match pstr with
        | PStr [{ pstr_desc = Pstr_eval (expression, _); _}] ->
          Exp.apply (Exp.ident {txt = Lident "+"; loc=(!default_loc)})
            [(Nolabel, (expr_mapper mapper expression));
             (Nolabel, Exp.constant (Pconst_integer ("1", None)))]
        | _ -> raise (Location.Error (Location.error ~loc "Syntax_error_in_expression_mapper"))
      end
    | x -> default_mapper.expr mapper x;
  end

let addone_mapper argv =
{
```

```
22    default_mapper with
23    expr = expr_mapper;
24  }
25
26  let () = register "addone" addone_mapper
```

To program a PPX we use OCaml's auxiliary compiler libraries indicated at the beginning of listing 2.3. The libraries allow the identification and manipulation of the OCaml's AST. The function in line 7 is an expression mapper. The mapper extends the default OCaml's mapper and looks for extensions nodes in the AST. This one, in particular, looks for extension nodes with the id `addone`. If such nodes aren't found then the default mapper is called to perform an identity transformation. This does not alter anything in the AST. When an *addone* node is found then the mapper checks if its payload is an expression. If it is, then it proceeds to construct the new nodes in the AST that represent the add one arithmetic operation. The PPX rewriter is very verbose but the user doesn't need to see or know this code. They must only know the purpose of the PPX. Using this PPX with the expression `[%addone 1 + 2]` would result in the new expression `(1 + 2) + 1`.

3

Related Work

In this chapter we present and briefly discuss some work being done to improve the efficiency of declarative programming and the PPX development process. We present crucial tools to work with PPX. Finally, we present common program transformation techniques such as continuation passing style, defunctionalization, memoization, hash-consing as well as other optimization transformations performed by compilers.

3.1 The current state of the PPX world

Camlp4 was a software system for writing extensible parsers for programming languages and was the tool of choice to extend the OCaml language and perform program transformations[4]. It was deprecated in August 2019 when it officially stopped supporting OCaml releases after release 4.08[5]. Now the tool of choice is the PPX rewriter. As mentioned before, PPX rewriters are OCaml programs that take another program as input, expand its extension nodes performing the adequate changes to the program's AST and return a modified program as output. Both OCaml compilers *ocamlc* and *ocamlopt* support PPX with the *-ppx* command line option. This executes the PPX given as argument to the compiler with the respective input program. The PPX transforms the program's AST during the preprocessing stage of the compilation pipeline as shown in figure 2.2. The compilers then continue the compilation process with the new modified AST. Several PPX rewriters can be passed to the compiler which are applied sequentially and the output of one will be the input of the next.

Despite being OCaml programs, PPX rewriters are more cumbersome to write than the usual ones. This is because coding a PPX is usually done using OCaml's compiler libraries that provide no stability guarantee and require extensive knowledge of the OCaml AST. With every OCaml release there is a chance the AST and structure may change. This leads to broken PPX rewriters and consequently requires developers to relearn the language's AST constructs. This happens if the developer is directly using OCaml's compiler libraries but there are other ways to develop PPX that have emerged and mitigate or eliminate these problems. We shall begin by exploring the `ppx_tools` which is the starting base of

PPX development.

3.1.1 ppx_tools

The ppx_tools is a toolbox made for developing PPX rewriters. It is a findlib package installed and executed through the ocamlfind driver[28]. The ppx_tools offers several utilities that facilitate the debugging stage of the PPX development process:

- ppx_metaquot - is a ppx filter that helps writing other PPX by allowing the developer to use a concrete syntax of expressions to create or destroy AST fragments.
- rewriter - is an important utility that applies the PPX to a program chosen by the user and returns the modified equivalent program. This is a crucial utility that lets the user know if the PPX is behaving as expected.
- Ast_mapper_class - is a module that implements an API similar to Ast_mapper from OCaml's compiler libraries. The difference is that it implements open recursion using a class.
- dumpast - is another crucial utility that parses OCaml and returns the corresponding AST representation. This tool is useful to learn the OCaml AST representation and helps creating PPX code.
- genlifter - is a tool that generates a virtual lifter class for one or more OCaml type constructors.

From all these utilities, rewriter and dumpast are the most important ones as they are essential to debugging PPX. Both *ocamlc* and *ocamlopt* compilers have the *-dparsetree* flag which is similar to dumpast as they all extract the AST of a program. Unfortunately, the syntax is harder to understand, it is more verbose and the resulting AST cannot be written directly to a file. Besides that, dumpast has the option to construct AST fragments directly in the command line through the *-e*, *-p* and *-t* flags that dump the AST of an expression, pattern or type expression, respectively.

3.1.2 ocaml-migrate-parsetree

As mentioned before the AST syntax and structure may change between OCaml versions which breaks PPX rewriters. The ocaml-migrate-parsetree is a library that solves this problem. It has a version of each AST for each major OCaml version as well as conversion functions that allow switching between versions. This library also has a driver functionally that allows it to link several PPX at once into a single executable in order to perform the minimum number of AST conversions and speed up the process. In addition, there is a library called ppx_tools_versioned that extends ocaml-migrate-parsetree to ppx_tools. This allows for PPX that were written using the aid of ppx_tools to be ported

to ocaml-migrate-parsetree[29]. This tool is essential in PPX maintenance, as it keeps it from breaking with new language versions.

3.1.3 ppxlib

The ppxlib is a library that exposes a higher level of abstraction for writing PPX[30]. With this library, PPX rewriters are no longer perceived as transformation programs applied at compile time. Instead, the extension points expanded by PPX are now seen as compile time functions that are evaluated in a top-down way. This has the advantage of turning the rewriting process into a single pass process instead of it being a one pass per PPX. This provides a better mental model for developers and users of PPX rewriters and makes it easier to reason about how a group of PPX can be composed together. This way the user no longer needs to understand low-level details about the PPX rewriters in order to understand how to use them simultaneously. The ppxlib also provides safety guarantees by checking that all attributes are interpreted and any typing mistake is caught instead of being ignored by the compiler during the preprocessing phase of the compilation pipeline. The library is also based on a specific version of the AST exposed by ocaml-migrate-parsetree library which means that every time OCaml is updated, ppxlib and the PPX rewriters built with it do not break. The PPX rewriters dependent on ppxlib only break when the version of the AST used with ppxlib is updated.

Unfortunately, the documentation for ppxlib is very scarce and thus insufficient to understand how to use the library to develop PPX rewriters. For this reason, and taking into account that ocaml-migrate-parsetree offers the possibility to port PPX to different OCaml versions, ppxlib will not be used in this dissertation.

3.2 Continuation Passing Style and Defunctionalization

Now that we have discussed the work being done to help the development of PPX rewriters and how it affects the work being done in this dissertation, we move onto two notable program transformation techniques. Continuation Passing Style and Defunctionalization are two examples of important program transformation techniques. These techniques more commonly used at compile time can improve code declarative efficiency, code robustness and also code correctness proofs.

Continuation Passing Style is a programming style where a function always returns its result by passing it onto another function[39]. It transforms non-tail recursive function calls into tail recursive calls, turns intermediate computations[17] and makes the order of the argument evaluation explicit[8, 17].

Non-tail recursive functions are a form of inefficient declarative programming because the recursive call is the last thing to be executed. In the meantime, the function keeps using the call stack space to store intermediate data like arguments and return addresses. This means the stack space required to execute a recursive function grows as the length

of the recursion grows. When the recursion is long enough, the call stack pointer exceeds the stack limit and causes a stack overflow error which generally results in a program crash. With continuation passing style, functions no longer return values but instead accept another argument, a continuation. This is a single argument function that by definition invokes the continuation tail recursively, giving it as argument the result of the first function[17]. Compilers perform an optimization on these functions turning them into loops which use constant stack space[26] eliminating stack overflow errors. This is the case for the OCaml compiler but not all compilers do the same. The Java compiler does not optimize tail recursive calls and the C compiler only performs these optimizations when using the O2 level of optimization.

Defunctionalization is a program transformation technique that transforms higher order programs into first order programs[36]. In higher order programs, functions may be anonymous, passed as arguments, and returned as results. In first order programs, all functions are named and each call refers to the callee by its name[9]. Functions are more commonly represented at runtime as closures, but they can alternatively be represented by tree like data structures when a higher order program has been defunctionalized into a first order program. These data structures are constructors containing the values of free variables of the function abstraction. Defunctionalization can ease the automatic correctness proof of programs[38] since in first order defunctionalized programs all continuations can be inspected no matter where they are defined. Function values can also be inspected while in higher order programs that is not possible[9].

Both CPS and Defunctionalization are good examples of program transformations. CPS is a useful technique to boost declarative programs efficiency and make them more robust since it eliminates stack overflow errors. Defunctionalization, on the other hand, may not provide any efficiency improvements directly but it is still an important technique that deserves recognition as it does improve the automatic correctness proof of programs[2].

3.3 Automatic Memoization using Higher Order Functions

We can easily write a naive implementation of a higher order function that memoizes other functions. We just need the higher order function to perform memoization on the function passed as argument. The code that automatically memoizes a non recursive function is very simple and is presented in listing 3.1.

Listing 3.1: Memoize Function for non recursive functions.

```
1 let memo f =
2   let cache = Hashtbl.create 15 in
3     fun x ->
4       try Hashtbl.find cache x
5       with | Not_found ->
6         let y = f x in
```

```
7        Hashtbl.add cache x y; y
```

The higher order function memo receives a function as argument and returns a new memoized version of it. This new function constructs an auxiliary hash table that stores previous computations. Then it declares an anonymous function that searches previously computed results in the table. If they are found then they are returned immediately but if they are not then they are computed, stored in the hash table and finally returned. As we can see, applying memoization to non recursive functions is easy and can be done in an intuitive and naive way. But we want a way to memoize all functions, including recursive ones.

The reason why the code in listing 3.1 is not advantageous for recursive functions is because it can only memoize the final result of the function. All recursive computations made during the execution are not stored in the hash table. This means that the complexity of recursive functions does not change and it will take just as much time to compute the final result as their non memoized counterpart. Fortunately, it is possible to memoize a recursive function including its recursive computations by combining the code in listing 3.1 with the Y combinator. A combinator is a function which takes one or more pure functions as inputs, returns a new function as output and does not use any functions that are not provided as input. Making a small alteration in the function's definition can greatly reduce computational complexity, even from exponential to linear.

Consider the recursive definition of the factorial function presented in listing 3.2.

Listing 3.2: Factorial function.

```
1 let rec fact n = if n = 0 then 1 else n * fact (n-1)
```

The reason why the memo function in listing 3.1 does not memoize the recursive computations made by the function fact is because fact is already defined by calling on its non memoized version. The same happens for any recursive function defined by calling upon itself. So let us rewrite fact in a way that allows us to define it without recursively calling upon itself.

Listing 3.3: Factorial Factory function [35] and new Factorial function[15].

```
1 let factFactory f n = if n = 0 then 1 else n * f (n-1)
2 let new_fact = factFactory fact
```

The factFactory, in listing 3.3, is an auxiliary function very similar to the fact function in listing 3.2. When applied with fact as argument it is a new equivalent definition for the factorial function. By doing that, we obtain the function new_fact. Notice that new_fact is almost the same function as fact defined in listing 3.2. It just has the base case written twice. We only changed the name to new_fact to ease interpretation but both fact in listing 3.2 and new_fact in listing 3.3 compute the same result. Now, the new recursive definition of the factorial function does not call upon itself. The recursion exclusively appears in the definition in line 2 of listing 3.3. This peculiarity means that fact is a fixed point of factFactory[15].

A fixed point is an element of the function's domain that is mapped to itself by the function. This means that $x = f(x) = f^n(x)$, where n is the number of times f is applied to x. In this case, the function `fact` is a fixed point of `factFactory` since $\text{fact} = \text{factFactory}(\text{fact}) = \text{factFactory}^n(\text{fact})$. Now that we have a new definition of the factorial function that does not call upon itself we only need to eliminate the recursion in line 2. To do to this we can use the fixed point Y combinator[15].

A fixed point combinator is a higher order function that given a function f returns a fixed point of f. This means that $\text{fix } f = f(\text{fix } f) = f^n(\text{fix } f)$, where n is the number of times f is applied to fix f. The Y combinator is an implementation of a fixed point combinator represented in lambda calculus by $Y = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$[15]. Lambda calculus cannot express recursion directly because all functions are anonymous. The Y combinator is especially important because it allows for recursion to be expressed. This means that we are able to express recursion in OCaml without having to have functions call upon themselves.

The Y combinator can be expressed in OCaml by the following expression.

Listing 3.4: Y combinator implementation in OCaml[35].

```
1 let y f = let rec g x = f g x in g
```

Now we have a way to express recursive functions without calling upon themselves. We can combine the y combinator code in listing 3.4 and the auto memoize code in listing 3.1 and finally create a function that automatically memoizes any function, including recursive ones[22]. The corresponding OCaml code is presented in listing 3.5.

Listing 3.5: Memoize Function for recursive and non recursive functions.

```
1  let memo f =
2    let cache = Hashtbl.create 15 in
3    let rec g x =
4      try Hashtbl.find cache x
5      with | Not_found ->
6        let y = f g x in
7        Hashtbl.add cache x y; y
8    in g
9
10 let memo_fact = memo factFactory
```

This technique will be used to create a memoization PPX. This PPX will automatically memoize functions appropriately tagged. It is presented in chapter 4. In the next section we discuss Hash-Consing, a technique used to share data that is structurally equal.

3.4 Hash-Consing

Hash-consing is a technique to share data that is structurally equal. Structurally equal data is data that has the same value or has the same structure, in the case of data structures. It differs from physically equal data which is data that is only equal if it is stored

in the same physical memory location. This technique allows minimizing memory allocation, a rather computational expensive action, and it can also speed up fundamental operations and data structures. This saves time and space consumed during runtime[11]. For example, both variables in listing 3.6 are structurally equal because they hold the same value 1 but when tested for physical equality the result is *false* because both values are stored in different memory locations.

Listing 3.6: Structurally equal and physically different variables.

```
let a = ref 1
let b = ref 1
a = b (* true *)
a == b (* false *)
```

This happens because every time a reference is created a new memory block is allocated. If there are other allocated references with the same value which is the case with *a* and *b* in listing 3.6, then memory is being wasted since the same values are being stored in different locations. Hash-consing solves this problem by sharing the same memory block for variables that share the same value. Besides the advantage of saving memory space, hash-consing can also speed up fundamental operations and data structures by several orders of magnitude when sharing is maximal, that is two values are shared when they are structurally equal[11].

This technique is used in [11], where the authors present a type-safe modular hash-consing OCaml library that solves problems that arise with an ad hoc implementation. As stated in [11], performing hash-consing with a good hash function already presents considerable performance enhancements but still has four drawbacks. First, there is no way to distinguish between values that are hash-consed and those that are not; Second, one would like to improve data structures containing hash-consed terms, such as sets or dictionaries implemented using hash tables or balanced trees by using pointers to get $O(1)$ hashing or total ordering functions over hash-consed terms. Unfortunately, this is not possible because the OCaml compiler does not give access to the pointer value and because the OCaml garbage collector is likely to move allocated blocks underneath, thus changing the values of pointers; Third, there is no way for the garbage collector to reclaim terms that are not alive anymore since they are referenced in the hash-consing table; Fourth, the hash-consing function computes the hash value associated to its argument twice when it is not in the hash table and also wastes space in the buckets of the hash table by storing the pointer twice.

The OCaml library presented performs the hash-consing technique but does not suffer from any of these issues. This is achieved by making a few modifications to the way the library is designed and implemented. Without those drawbacks the authors show that using their hash-consing library greatly reduces the amount of memory used but that it comes with the cost of reducing the speed performance. However, this trade-off can be overcome by combining hash-consing with the use of memoization. The version of the

code with hash-consing and memoization performs better regarding both time and space. It is more than 10 times faster and consumes 100 times less memory than the version with neither.

Normally improving a program's performance has the trade-off of increasing memory consumption but this library overcomes this cost. Given that the goal of this dissertation is to achieve efficient declarative programming, these promising results show that the use of this library should be considered when improving OCaml code efficiency. In particular, when using data structures like sets and dictionaries or when defining recursive functions. This work is also very important for the dissertation because it is used as a guide for the implementation of a hash-consing PPX presented in chapter 5.

4 Memoization

In this chapter we discuss memoization, a technique used to speed up program executions by storing repeated computations. We show an example where the problem arises and the technique can be applied. We describe it and show how it solves the problem. Then we present the Memoization PPX and how memoization is incorporated into OCaml code. At last, we present the results of applying the Memoization PPX and the respective benchmarks.

As stated in chapter 2, declarative programming has several advantages. Simple syntax, increased expressiveness, reduced code size, ease of code analysis, readability and verification. There is also no notion of state and no occurrence of side effects. But certain problems may arise when it comes to efficiency. One of those problems is repeated computations. Note that this is different from repeated code. Repeated code is when the exact same code is written more than once. It can happen by accident or it might be intended to increase readability and comprehension. Regardless, usually it doesn't have a meaningful effect on code efficiency. Repeated computations, however, happen when during execution time the process repeats the exact same computation more than once. It may have a harsh impact on efficiency as it means the code is taking longer than needed to conclude. A perfect example that presents such a problem are recursive functions. Recursive functions can easily be implemented in declarative programming. They can almost be copied from their mathematical definition without making alterations. But because of recursion, during each iteration these functions must repeat every computation done in previous iterations. This means the function has a high time complexity. For example, the Fibonacci function with argument 1, makes a single computation; with argument 10, it already makes 177 computations; and with argument 20, it makes 21891 computations. The number of computations grows much faster than the size of the argument. Memoization can solve this problem by stopping repeated computations from happening. Instead of repeating computations, we can store previous them and access them later when necessary. On the following section, we shall describe and explain more in depth how this technique works.

4.1 Memoization

Memoization is an optimization technique that can speed up computer programs by avoiding repeated computations. Previous computations are stored in memory so that they may be consulted later when necessary, removing the need to repeat them. Every computation gets tabulated in a data structure. Usually, a hash table to make use of the instant access speed ($O(1)$ time complexity). Every time the function needs an old computation, it first checks if it has already been computed before. If that is the case, the computation value is returned, if not, then the computation is performed and its value is stored in the data structure for future consultation.

Donald Michie[23], divides memoized functions into 2 distinct parts: a "rule part" when computational procedures happen, and a "rote part" where a look-up table is used. To perform memoization, a "rote part" is added to the function. The final result of a function evaluation is unaltered but the computation process is eased. This technique is quite powerful, being able to reduce a function's time complexity from $O(2^n)$ to one of $O(n)$. To illustrate this technique, let us look at the Fibonacci function.

The Fibonacci function, also known as Fibonacci number or Fibonacci sequence, is defined as the result of the sum of the previous two results. Its recursive definition is as follows: $F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}, \quad \text{with n} > 1, \text{n} \in \mathbb{N}$. When implementing it in OCaml we use a slightly modified version of the function to increase readability. This change has no impact on the computational complexity of the function which is the topic at hand. The Fibonacci function definition in OCaml is shown in listing 4.1. This is the definition we will use throughout the rest of the chapter.

Listing 4.1: Fibonacci function in OCaml.

```
1 let rec fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

If we look at the function we can see that it is recursive and that each function call makes 2 subsequent recursive calls. This indicates that the function has an exponential time complexity, $O(2^n)$. The function also runs into a problem we've mentioned before. It repeats computations. To better visualize it, let us take a look at the function call tree shown in 4.1.

As we can see, calling the Fibonacci function with an argument of $n = 0$, $n = 1$ or $n = 2$ yields no repeated computations. When the function is called with an argument of $n = 3$, we can see that there is one repeated computation. The $fib(1)$ is calculated twice. When the Fibonacci function is called with an argument of $n = 4$ the function suddenly performs 4 unnecessary computations. The number of redundant computations grows with the size of the argument and so does the execution time. We will discuss this in depth, in section 4.3. Thus, the Fibonacci function is an example of a function that greatly benefits from memoization. We want to eliminate repeated computations highlighted in figure 4.1. To apply this technique to the function we must add a table to save all the computations.

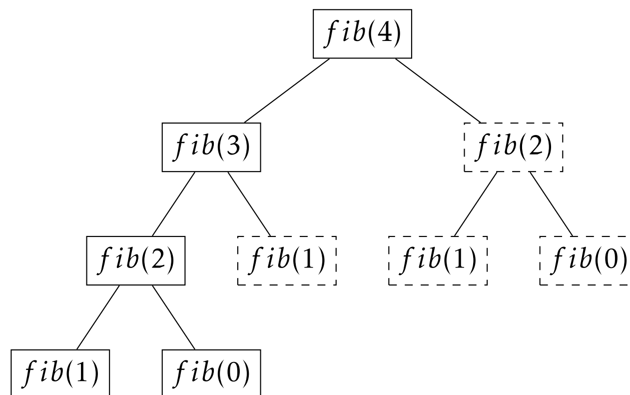

Figure 4.1: Fibonacci function calls tree. Dashed nodes represent repeated computations.

The table must speed up the computation time but it mustn't alter the function results. The Fibonacci function with memoization applied to it is shown in listing 4.2.

Listing 4.2: Fibonacci function with memoization in OCaml.

```
let fib =
  let cache = Hashtbl.create 15 in
  let rec fib n =
      try Hashtbl.find cache n
      with | Not_found ->
        let y = if n < 2 then 1 else fib(n-1) + fib(n-2) in
        Hashtbl.add cache n y; y
  in fib
```

The new Fibonacci function is very similar to the previous version. It performs the same calculation and the results are unaltered. But now the function has an auxiliary data structure that saves all computations. Also, before calculating anything, the function first checks if the value has been stored in the data structure in order to avoid repeated computations. The transformation from the Fibonacci function in listing 4.1 to the memoized version in listing 4.2 is what we want to automate with our Memoization PPX. In the following section, we discuss how the Memoization technique can be automatically applied to functions using PPX.

4.2 Memoization with PPX

The automatic memoize PPX rewriter is a program transformation that automatically memoizes a function in OCaml. To use the PPX, the user must first write an extension node in their OCaml code. The extension node `%memo` must be written at the beginning of a let-definition right after the *let*, as shown in the example in listing 4.3.

Listing 4.3: Fibonacci function with extension node.

```
let %memo rec fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

Then the user can use the *rewriter* utility of `ppx_tools` tool mentioned in chapter 3, with the PPX and target program as argument to transform the program. This lets the user check the changes that were made to the code. The user can also use the PPX during the compilation process with the *-ppx* flag. The resulting code from applying the memoize PPX to the code in listing 4.3 is shown in listing 4.4.

Listing 4.4: Memoized Fibonacci function.

```
1  let fib =
2    let cache = Hashtbl.create 17 in
3    let rec fib n =
4      try Hashtbl.find cache n
5      with
6      | Not_found ->
7          let res = if n < 2 then 1 else (fib (n - 1)) + (fib (n - 2)) in
8          (Hashtbl.add cache n res; res)
9    in
10   fun x -> fib x
```

The new Fibonacci function defined in listing 4.4 has a cache that will save all the intermediate computations of the function in order to avoid repeating them. This auxiliary structure is implemented using a hash table from the OCaml standard library. The modified function first looks at its cache to check if a specific computation has been made before. If it has, then all it must do is retrieve the result from the cache. If it has not been computed yet, then it is computed and the result is stored in the cache for future computations. This optimization technique will result in performance improvements in the code's execution time because it greatly reduces the number of computations made when the function is called. All the redundant `fib` calls highlighted in figure 4.1 no longer happen with the new code. In fact, the function's time complexity is reduced from exponential $O(2^n)$ to linear $O(n)$, where n is the input.

As it can be seen in listing 4.4, the resulting memoized function becomes much more verbose and harder to understand because of the extra optimization code. Thanks to PPX the user does not need to understand the extra added code or the technique used since they do not code it themselves. In fact, the user does not even have to see the resulting code in listing 4.4. We only show it to demonstrate the result of the PPX application and to show how verbose the code becomes. All the user has to do is write the extension node `%memo` as shown in listing 4.3 and then compile the code with the PPX attached. The improved code is then ready to be executed.

As we have said, the new code has noticeable performance improvements in its executional time, reducing the function's time complexity from exponential $O(2^n)$ to linear $O(n)$. To show these results we have executed both of the codes in listings 4.3 and 4.4 and compared the results. These are shown in the following section where we evaluate the new code's performance.

4.3 Memoization PPX Performance Evaluation

To show that the automatic memoization PPX does improve the code's performance we executed the codes in listings 4.3 and 4.4 and measured the number of computations made, the execution time and, the total memory consumption of each one. Both codes were executed in the exact same experimental environment. They were executed in a computer with a 1,6 GHz Dual-Core Intel Core i5 processor and a 8 GB 2133 MHz LPDDR3 memory. There were no other programs being executed other than background operating system processes. Each test was performed five times and the median of the results was selected to avoid skewed results.

4.3.1 Number of Computations

The following table shows the number of computations made by each definition of the Fibonacci function.

Table 4.1: Number of computations of both versions of the Fibonacci function.

Argument	Fibonacci	Memoized Fibonacci
1	1	1
10	177	11
20	21891	21
30	2692537	31
40	331160281	41
50	40730022147	51

As it can be seen in table 4.1, the number of computations made by each function execution for the same input is tremendously different. When the argument is 1 both functions only make 1 computation since both immediately hit the base case of the recursive definition and therefore make no recursive calls. But when the input increases past the base case, the number of computations made by each one starts to diverge. The number of computations made by the Fibonacci function without any modifications increases exponentially while the memoized function has a linear growth in relation to the growth of the argument size. The difference in the number of computations performed translates into a difference in execution time which will be analysed in the next section. It will help understand the advantage of using memoization.

4.3.2 Execution Time

The following table shows the time it took for each definition of the Fibonacci function to terminate its execution.

As it can be seen in table 4.2, the difference in execution time is very small when the argument ranges from 1 to 20 and can be considered insignificant. When the argument increases to 30, we can the see first major difference in times. The normal definition of the Fibonacci function took about 200 times more time to execute than its memoized

Table 4.2: Execution time of both versions of the Fibonacci function.

Argument	Fibonacci (s)	Memoized Fibonacci (s)
1	0,000001	0,000005
10	0,000003	0,000017
20	0,000062	0,000030
30	0,007300	0,000033
40	0,604344	0,000030
50	88,97804	0,000042

version and about 117 times more time to execute than itself with an argument difference of only 10. When the argument is 40, the normal definition of the Fibonacci function already takes a little over half a second to compute while its memoized version's execution time barely increases with the argument growth. Half a second may not seem like a lot of time for humans but for a computer in 2022 to take this much time to compute a single function value is unacceptable. When the argument increases to 50, the execution time of the normal Fibonacci becomes inadmissible as it takes about 1 minute and a half to terminate its execution. This presents an astonishing contrast with the memoized version's execution time which is about 2 million times faster.

Overall, the execution time of the normal definition of the Fibonacci function increases exponentially while its memoized version has a barely noticeable growth. The number of computations and execution time of both functions show that memoization is an optimization technique capable of outstanding results when it comes to improving the execution time performance of code. Given the fact that memoization is performed at the expense of a cache, we must also analyse the amount of memory it consumes. It is important to understand how much extra memory is allocated and if justifies the time benefits that we have just witnessed.

4.3.3 Memory Consumption

The following table shows the total amount of memory that is allocated during the execution of each definition of the Fibonacci function.

Table 4.3: Memory consumption of both versions of the Fibonacci function.

Argument	Fibonacci (MB)	Memoized Fibonacci (MB)
1	0,001053	0,002274
10	0,001053	0,004074
20	0,001053	0,005981
30	0,001053	0,007889
40	0,001053	0,010361
50	0,001053	0,012268

As it can be seen in table 4.3, the memory allocated in the execution of the normal Fibonacci function stays the same regardless of the size of the argument. This is expected

as the definition of the function does not allocate any memory thus an increase in recursive calls does not alter this number. On the other hand, the memoized definition of the Fibonacci function allocates more memory when the argument grows. This happens because the memoized version is storing the result of every recursive call in the cache which means that when the argument increases, the number of recursive calls, entries in the cache, and consequently the amount of memory consumed all increase too.

The memory consumed in each execution was very small but it does increase with the argument which should be taken into consideration when using memoization. Memoization is an optimization technique with outstanding results in execution time performance but does come with this cost that cannot be overlooked. It is a trade-off between time and space efficiency.

4.3.4 Results for large inputs

So far we have only seen the results of the execution of the two function definitions for an argument of up to 50. With an argument of this size the execution of the non memoized Fibonacci already hits inadmissible times so it is pointless to continue to test the function for larger inputs. The performed tests were enough to compare both definitions and conclude that the memoized definition of the Fibonacci function is more time efficient than the non memoized one at the cost of space efficiency. But we have only tested the function for relatively small arguments. The new definition should keep its efficiency even for larger inputs so that it can be considered a viable optimization. In order to check if the optimization results hold for larger inputs, we have executed the memoized function with much larger inputs and measured the results as before. These tests were made in the exact same experimental environment as the previous ones.

The following table shows the number of computations made, the execution time and the total memory consumption during the execution of the memoized definition of the Fibonacci function.

Table 4.4: Test results summary of the memoized Fibonacci function.

Argument	Computations	Execution time (s)	Memory consumption (MB)
1	1	0.000005	0.002274
10	11	0.000003	0.004074
100	101	0.000094	0.022858
1000	1001	0.000969	0.208427
10000	10001	0.008982	2.159721
100000	100001	0.114179	21.076309
1000000	1000001	1.854013	206.740295
10000000	Stack Overflow Error		

As it can be seen in table 4.4, the number of computations follows the same growing behaviour as before and the function computes extremely fast up until the input hits one million. At this point the execution time becomes slower but that is expected for such a

large input. The memory consumption also grows with the size of the input and achieves considerably large numbers when the input is a hundred thousand or larger. The cost benefit analysis between the execution time and memory consumption ultimately must be done by the user of the PPX. But, considering that most computers nowadays have at least 4 GB of RAM we consider that these are highly satisfactory results.

When the input reaches 10 million the table no longer has data about the function's performance and instead presents the "Stack Overflow Error". The function fails to compute successfully with such a large input and terminates with an error. This happens because the recursion becomes excessively deep and the space occupied by the function in the call stack exceeds its limit. The error comes from the recursive nature of the function and can be solved by turning it into a loop definition and making the stack explicit. As we can see the PPX did not solve all efficiency problems of declarative programming but it did make a remarkable contribution. The automatic memoization PPX was successful in making declarative code more efficient in time.

5

Hash-Consing

In this chapter we discuss hash-consing, a technique used to reduce the memory consumed by program executions and also speed them up by sharing data that is structurally equal. We show an example of where the problem arises and the technique can be applied. We describe it and show how it solves the problem. Then we present the Hash-Consing PPX and how hash-consing is incorporated into OCaml code. At last, we present the results of applying the Hash-Consing PPX and the respective benchmarks.

As stated in chapter 2, declarative programming has several advantages. Simple syntax, increased expressiveness, reduced code size, ease of code analysis, readability and verification. There is also no notion of state and no occurrence of side effects. But certain problems may arise when it comes to efficiency. One of those problems is memory consumption. Memory consumption itself isn't a problem. It is perfectly normal and is a crucial part of computation. However, some computations may result in excessive memory usage.

5.1 Hash-Consing

Hash-consing is an optimization technique used to decrease memory consumption of computer programs. It can also be used to speed up expensive operations and consequently the program execution[7]. This technique involves sharing identical immutable values in memory so that only a single copy of semantically equivalent objects is kept[3]. Homologous to memoization, before a value is created, a lookup is performed on a data structure to check if there's already an equal value saved. If that is the case, then such value is returned, if not, then the value is created and is saved in the data structure for future consultation. A data structure saves all values created and never repeats a value that is already saved, instead it reuses them. Thus saving memory during the program execution. This technique can also improve time performance. When comparing values there is a need to traverse two entire data structures in order to conclude they're equal. This is has $O(n)$ time complexity given that every element in the structure must be verified. But the complexity can quickly grow into exponential if there are nested

data structures. Hash-consing can be used to solve this problem because we can achieve maximal sharing. This means every structurally equal value is shared. Thus checking if two values are equal becomes an operation with $O(1)$ time complexity since structural equality can be replaced with physical equality. To illustrate this technique, let us look at an example of its implementation in OCaml.

We will use the tree data type as an example to illustrate the problem. The OCaml code that represents a char tree type is shown in listing 5.1.

Listing 5.1: OCaml tree type.

```
type tree =
    | Leaf
    | Node of tree * char * tree
```

A tree can either be a Leaf or a Node. A Node has a left branch of type tree, a character and a right branch of type tree. This is a typical tree type definition in OCaml. Now let us create two structurally equal trees and perform both structural and physical equality. The respective code is shown in listing 5.2.

Listing 5.2: Structurally equal and physically different trees.

```
let t1 = Node (Leaf, 'e', Leaf)
let t2 = Node (Leaf, 'e', Leaf)

t1 = t2 (* true *)
t1 == t2 (* false *)
```

As expected the trees are structurally equal and physically different. This happens because the content of the trees is equal but they were defined as different values. Now let us implement hash-cons in OCaml so these trees can share their content and thus become physically equal. The Hash-cons implementation in OCaml is shown in listing 5.3.

Listing 5.3: Hash-cons in OCaml.

```
let table = Hashtbl.create 251
let hashcons x =
    try Hashtbl.find table x
    with | Not_found -> Hashtbl.add table x x; x

let t1 = hashcons (Node (Leaf, 'e', Leaf))
let t2 = hashcons (Node (Leaf, 'e', Leaf))

t1 = t2 (* true *)
t1 == t2 (* true *)
```

As we've mentioned before the process is homologous to memoization. Before creating a value we first check if it's already been created and stored in the hash table. If yes, then we reuse it. If not, then we create it and store it. Now when we create both trees, they no longer have the same value repeated, they are actually sharing it. And we can see

that they are structurally and physically equal. If we create a smart constructor that performs hash-consing, we ensure that all values of type tree are hash-consed and we achieve maximal sharing[11]. With maximal sharing achieved physical and structural equality become equivalent. We can use this to improve the tree equality function. In order to compare two equal trees that are not hash-consed we must compare every node of each tree. The function is shown in listing 5.4.

Listing 5.4: Equal function for trees and hash-consed trees in OCaml.

```
let rec equal t1 t2 =
  match t1, t2 with
    | E, E -> true
    | Node (l1, c1, r1), Node (l2, c2, r2) -> equal l1 l2 && c1 = c2 && equal r1 r2
    | _ -> false

let hcEqual t1 t2 =
  match t1, t2 with
    | E, E -> true
    | Node (l1, c1, r1), Node (l2, c2, r2) -> l1 == l2 && c1 = c2 && r1 == r2
    | _ -> false
```

Given that we have achieved maximal sharing the equal function can be replaced with the simpler more efficient version hcEqual. This is possible because of the assumption that sub-trees are hash-consed. We've eliminated the recursion in the equal function and thus the time complexity of the equal function is reduced from $O(n)$ to $O(1)$.

We want to automate this technique with our Hash-Consing PPX. In the following section, we discuss how the Hash-Cons technique can be automatically applied to OCaml code using PPX.

5.2 Hash-Consing with PPX

The automatic hash-consing PPX rewriter is a program transformation that automatically performs hash-cons in OCaml. To use the PPX, the user must first write an extension node in their OCaml code. The extension node %hashcons must be written at the beginning of a type-definition right after the *type*, as shown in the example in listing 5.5.

Listing 5.5: Term type with the hash-cons extension node.

```
type%hashcons term =
  | Var of int
  | Lam of term
  | App of term * term
```

Then the user can use the *rewriter* utility of ppx_tools tool mentioned in chapter 3, with the PPX and target program as argument, to transform the program. This lets the user check the changes that were made to the code. The user can also use the PPX

during the compilation process with the *-ppx* flag. The resulting code from applying the hash-cons PPX to the code in listing 5.5 is shown in listing 5.6.

Listing 5.6: Hash-consed term type.

```
type term =
    | Var of int * int
    | Lam of int * term
    | App of int * term * term

let unique = function
    | Var (u,_) -> u
    | Lam (u,_) -> u
    | App (u,_,_) -> u

module X = struct
    type t = term

    let hash = function
        | Var (_,u_0) -> 19 * (Hashtbl.hash u_0)
        | Lam (_,u_0) -> 19 * (unique u_0)
        | App (_,u_0,u_1) ->
            ((19 * (Hashtbl.hash u_0)) + (19 * (unique u_1))) land max_int

    let equal t1 t2 =
        match (t1, t2) with
            | (Var (_,l_0),Var (_,r_0)) -> l_0 == r_0
            | (Lam (_,l_0),Lam (_,r_0)) -> l_0 == r_0
            | (App (_,l_0,l_1),App (_,r_0,r_1)) -> (l_0 == r_0) && (l_1 == r_1)
            | _ -> false
end

module W = (Weak.Make)(X)
let nodes = W.create 5003
```

The application of the Hash-cons PPX modified the original type `term` and produced some more code. Let us analyse all the alterations. First, the type term was modified and now each constructor has an additional argument. This argument is an unique identifier used to distinguish all terms. The `unique` function is used to extract the unique identifier from a value of type term. This function is auxiliary to the `hash` function. Then we create a module. This module is created to build an implementation of the OCaml standard library's weak hash set structure[25]. We use a weak hash set because the OCaml's garbage collector cannot retrieve values from an hash table that are not being used anymore. This is because these values are always referenced in it[11]. This weak hash set data structure uses OCaml's weak pointers so that when a value stops being useful, it can be collected by the garbage collector[7]. This way we avoid having a memory leak created by values being indefinitely stored in an hash table even after having no references to them except the one from the table itself. Next we define the type of the elements that will be stored in the

table. Then we define the hash function. It performs a standard hash code for each type of term. The Lam and App terms use the unique function to get their identifier while the Var term uses the hash function from the OCaml standard library. But the hash function may be customized by the user. We will present this option in subsection 5.2.1. Next we have the equal function. Thanks to the assumption that all sub-terms are hash-consed and thus maximal sharing is achieved, the function simply performs physical equality of the sub-terms. With the module concluded, we then build the structure and create a new empty weak hash set with initial size 5003.

As it can be seen in listing 5.6 the code becomes more extensive and the optimization portion has no semantic contribution to what is being implemented by the user. Thanks to PPX the user doesn't need to implement this extra code. The user must only write the extension node %hashcons and then they may take advantage of the optimization benefits.

As we have said, the hash-cons PPX has customization options. In the following subsection we will show what these encompass and how the user can be use them.

5.2.1 Hash-Cons PPX Customization

Users may customize the hash-cons code in two ways. They may customize the name of the variables used and the function that creates an hash code inside the hash function. In order to do the former, users must attach another extension node inside the type being hash-consed. The node is named %hcons and it should contain the custom name inside it. It must be attached as a component of a product type before the component to be customized. In the listing 5.7 we can see an example of how to use it.

Listing 5.7: Term type with the hash-cons extension node and the node to customize the name of variables.

```
1 type%hashcons term =
2     | Var of int
3     | Lam of [%hcons id_lam] * term
4     | App of [%hcons id_app] * term * term
```

This piece of code is quite similar to the one in listing 5.6 except in this one we have extra extension nodes. There are two extra %hcons extension nodes. The first %hcons node has id_lam inside as payload. This indicates that inside the hash-cons code the default variable name for the term of Lam will be replaced with the name chosen. In this case, it will be id_lam. The second %hcons behaves the exact same way but the name will be id_app. The resulting code is presented in listing 5.8

Listing 5.8: Hash-consed term type with customized variable names.

```
1 type term =
2     | Var of int * int
3     | Lam of int * term
4     | App of int * term * term
5
```

```
6  let unique = function
7      | Var (u,_) -> u
8      | Lam (id_lam,_) -> id_lam
9      | App (id_app,_,_) -> id_app
10
11 module X = struct
12     type t = term
13     let hash = function
14         | Var (_,u_0) -> 19 * (Hashtbl.hash u_0)
15         | Lam (_,id_lam) -> 19 * (unique id_lam)
16         | App (_,id_app,u_1) ->
17             ((19 * (unique id_app)) + (19 * (unique u_1))) land max_int
18
19     let equal t1 t2 =
20         match (t1, t2) with
21             | (Var (_,l_0),Var (_,r_0)) -> l_0 == r_0
22             | (Lam (_,l_id_lam0),Lam (_,r_id_lam0)) -> l_id_lam0 == r_id_lam0
23             | (App (_,l_id_app0,l_1),App (_,r_id_app0,r_1)) ->
24                 (l_id_app0 == r_id_app0) && (l_1 == r_1)
25             | _ -> false
26 end
27
28 module W = (Weak.Make)(X)
29 let nodes = W.create 5003
```

It is shown that the names chosen by the user are now being used in the equal and hash functions. As mentioned before, the user may also customize the function that produces an hash code by choosing another one. In order to do this the user must attach another extension node inside the type being hash-consed. The node is named %hash and it must contain the custom function name inside it. It must be attached as an attribute[18] next to the component of the product type which the user wants the function to be applied to. In listing 5.9 we can see an example of how to use it.

Listing 5.9: Term type with the hash-cons extension node, the node to customize the name of variables and the node to customize the hash function.

```
1 type%hashcons term =
2     | Var of int
3     | Lam of [%hcons id_lam] * term
4     | App of [%hcons id_app] * (term[@hash Hashtbl.hash]) * term
```

This piece of code is similar to the one in listing 5.7 but with an added attribute node. Attributes share the same notion of identifier and payload as extension nodes[18]. The hash node has Hashtbl.hash inside as payload. This indicates that inside the hash-cons code the default function that produces an hash code will be replaced with the function chosen by the user. In this case, it will be the hash function from the OCaml module Hashtbl. The resulting code is presented in listing 5.10

Listing 5.10: Hash-consed term type with customized variable names and customized hash function.

```
type term =
    | Var of int * int
    | Lam of int * term
    | App of int * term * term

let unique = function
    | Var (u,_) -> u
    | Lam (id_lam,_) -> id_lam
    | App (id_app,_,_) -> id_app

module X = struct
    type t = term
    let hash = function
        | Var (_,u_0) -> 19 * (Hashtbl.hash u_0)
        | Lam (_,id_lam) -> 19 * (unique id_lam)
        | App (_,id_app,u_1) ->
            ((Hashtbl.hash id_app) + (19 * (unique u_1))) land max_int

    let equal t1 t2 =
        match (t1, t2) with
            | (Var (_,l_0),Var (_,r_0)) -> l_0 == r_0
            | (Lam (_,l_id_lam0),Lam (_,r_id_lam0)) -> l_id_lam0 == r_id_lam0
            | (App (_,l_id_app0,l_1),App (_,r_id_app0,r_1)) ->
                (l_id_app0 == r_id_app0) && (l_1 == r_1)
            | _ -> false
end

module W = (Weak.Make)(X)
let nodes = W.create 5003
```

It is shown that the custom function chosen by the user is being used inside the hash function. As we can see in line 17, the default function `unique` has been replaced with `Hashtbl.hash` chosen by the user.

As we have said, the new code has noticeable performance improvements in its executional memory consumption. It also has executional time improvements provided by an equal function with $O(1)$ complexity thanks to maximal sharing. To show these results we will execute code using the hash-cons PPX and without. Then we will compare and analyze the results. These are shown in the following section where we evaluate the new code's performance.

5.3 Hash-Consing PPX Performance Evaluation

To show that the automatic hash-cons PPX does improve the performance of the code, we executed two programs that consists on creating two trees and performing structural

and physical equality between the both of them. One of them has the normal version of the code and the other has an hash-consed version of the it. We then measured the number of words allocated in the major heap[24] since the program was started. We also measure the execution time of the structural and physical equality operations. The code was executed in a computer with a 1,6 GHz Dual-Core Intel Core i5 processor and a 8 GB 2133 MHz LPDDR3 memory. There were no other programs being executed other than background operating system processes. Each test was performed five times and the median of the results was selected to avoid skewed results.

Both versions of the code share some code but they also differ as one of them uses the hash-cons code created by the PPX. The normal version of the code is shown in listing 5.11.

Listing 5.11: Code to create a tree.

```
type tree = E | N of tree * char * tree

let rec equal t1 t2 =
   match t1, t2 with
      | E, E -> true
      | N (l1, c1, r1), N (l2, c2, r2) ->
         equal l1 l2 && c1 = c2 && equal r1 r2
      | _ -> false

let node l c r = N (l, c, r)

let leaf_x = node E 'x' E

let rec create_tree n =
   if n = 0 then leaf_x, leaf_x
   else
      let n' = Random.int n in
      let l1, l2 = create_tree n' in
      let r1, r2 = create_tree (n - n' - 1) in
      node l1 'm' r1, node l2 'm' r2
```

First, there's a definition for the tree type. A tree is either empty or a node with a left and right branch of type tree as well. Between them, it can hold a value of type char. Second, there's a function that checks if two trees are equal. If both are empty, then they are equal. If very node on both trees has the same value and the trees have the exact same shape, then they are also true. If neither of these conditions is met, then they are different. Third, there's a function to create a tree node. Fourth, there's a function to create a leaf node. Finally, there's a function that creates two identical trees. It receives the desired tree height as argument. If the argument is zero, then the tree is just a leaf. If it is bigger than that, the function creates a random integer using OCaml's Random module. This integer is a number between zero and the value of n. Then the function uses this number to create both branches of the tree. Every node has the same char value and both subtrees

are equal. This is to ensure we can properly evaluate the structural and physical functions performance since we want the equal function to perform the entire recursion.

The hash-consed version of the code has some differences to make use of the technique. The differences in this version of the code are shown in listing 5.12. The omitted code is the same as in listing 5.11.

Listing 5.12: Code to create a tree node and an hash-consed tree node.

```
type tree = E | N of int * tree * char * tree

let node =
  let cpt = ref 1 in
  fun l c r ->
    let n0 = N (!cpt, l, c, r) in
    let n = W.merge nodes n0 in
    if n == n0 then incr cpt;
    n

let equal t1 t2 =
  match t1, t2 with
    | E, E -> true
    | N (_, l1, c1, r1), N (_, l2, c2, r2) ->
      l1 == l2 && c1 == c2 && r1 == r2
    | _ -> false
```

The tree type is modified and the equal function is provided by the PPX to accommodate the hash-cons technique. This change has already been discussed in section 5.2. The new node function is a smart constructor[11] created in order to also take advantage of hash-consing. It creates a reference value that will be the unique identifier for each hash-consed value. The anonymous function creates a hash-consed tree node. Then it stores the node inside the table. If the node already exists on the table, then the table returns an instance of said node. If it doesn't, then the table stores the node and returns it[25]. Then the functions checks if the node was already stored in the table. If it was then it increments the unique identifier. Finally it returns the created node.

The two different codes that were compared were just shown in listing 5.11 and 5.12. The test to compare their performance, comprises of simply calling the create_tree with the same argument for both versions of the code. Then both structural and physical equality are performed. The structural equality is done by the equal function and the physical equality is performed by the == function built into OCaml. We measured the time taken to execute both programs and the amount of major words[24] allocated. In the following subsection, we present the results for the spacial performance of the Hash-Cons PPX.

5.3.1 Memory Consumption

The following table 5.1 shows the number of words allocated in the major heap and the corresponding memory consumed by them during the execution of the program. According to OCaml, to get the number of bytes that a word occupies one must multiply it by 8 on 64-bit machines. This is how we calculate the megabytes consumed during the program execution presented in table 5.1.

Table 5.1: Memory consumption of creating a normal tree and a hash-consed tree.

Tree Height	**Normal Create Tree**		**Hash-consed Create Tree**	
	Words	**Memory (MB)**	**Words**	**Memory (MB)**
1.000	0	0	11.853	0,09
10.000	0	0	38.303	0,29
100.000	762.059	5,81	224.050	1,71
1.000.000	7.912.232	60,37	3.704.620	28,26
10.000.000	79.985.603	610,24	41.679.617	317,99
100.000.000	799.958.379	6.103,20	339.641.822	2.591,26

As it can be seen in the table 5.1, the number of words allocated in the major heap increases with the height of the tree. This is normal and expected because as the tree increases in height, the memory consumption also increases as more nodes are stored in memory. The normal create tree doesn't allocate words in the major when the argument is 1.000 and 10.000. It is worth noting tho that minor words are still allocated. We are just not interested in measuring those. For the same arguments, the hash-consed version allocates 11.853 and 38.303, respectively. This happens because the hash-consed version uses a table to save the tree and the table itself also occupies memory. When the height of the tree is 100.000, the normal create tree function starts allocating major words. It allocates 762.059 which translates into $5,81$ megabytes. The hash-consed version allocates 224.050 which translates into $1,71$ megabytes. With a tree of this height, it's not a very noticeable difference yet. Both versions don't consume a lot of memory. When the height of the tree is 1.000.000, the normal version consumes $60,37$ megabytes and the hash-consed version consumes $28,26$ megabytes. At this point, the difference in memory consumption starts to become notable. For this specific program it might look small but if we had more trees and we run a more complex program this difference will easily scale to bigger values. When the height is 10.000.000 the normal version consumes $610,24$ megabytes and the hash-consed version $317,99$ megabytes. The difference is quite big and we can see that hash-consing the tree reduces memory consumption by half. When the height is 100.000.000 the normal version consumes $6.103,20$ megabytes and the hash-consed version $2.591,26$ megabytes. The difference in memory consumption with a tree of this height is immense. At this point, both versions consume tremendous amounts of memory but this highlights the benefit of using hash-cons. Even if $2.591,26$ megabytes is a lot of memory consumed, it is still much smaller than the $6.103,20$ megabytes consumed by the normal version. It is less than half.

Overall, the memory consumption of the normal version increases linearly in relation to the growth of the tree. But the hash-consed version is able to consistently reduce the memory consumed by the tree in half when compared to its counter-part. This contrast shows that hash-cons is a technique capable of substantial results when it comes to improving memory consumption of code. As we have said, hash-cons is also capable of reducing execution time. This is because when maximal sharing is achieved, structural equality can be replaced with physical equality. In the following subsection, we present the results for the time performance of the Hash-Cons PPX.

5.3.2 Execution Time

The following table 5.2 shows the time it took for each equality to terminate its execution for the normal version of the tree and its hash-consed version.

Table 5.2: Execution time of checking if two normal and hash-consed trees are equal.

Height	Equality of Normal Trees		Equality of Hash-consed Trees
	Structural (s)	Physical (s)	Physical (s)
1.000	0,000021	0,000001	0,000001
10.000	0,000325	0,000001	0,000001
100.000	0,002772	0,000001	0,000001
1.000.000	0,024883	0,000001	0,000001
10.000.000	0,290202	0,000001	0,000001
100.000.000	2,9951	0,000001	0,000001

The first thing we can see in table 5.2 is that there is no version of the structural equality for the hash-consed tree. This is because we achieved maximal sharing and both equalities became equivalent. Therefore we only used the physical equality as it is faster. If we look at the physical equality column under the normal tree we can also see that it has constant execution time. It is neither influenced by the height of the tree nor the hash-cons technique. This is expected as it is essentially just checking the address of both trees in memory. Only the structural equality is influenced by the height of the tree. The execution time becomes longer when the height of the tree increases. The difference in execution time may appear small from a human perspective but for a computer it is significant. The time it takes to complete this function with a tree of height 1.000 is already 21 times faster than the physical equality. This difference just keeps getting wider with growth of the tree. When the tree has a height of 100.000.000 the hash-consed version performs 3 million times faster than the normal version.

The hash-cons version of the code that creates a tree performed better than it's counter-part in terms of memory consumption. It consumed about half of the memory that the normal one consumed. The technique was also able to fasten the equality between trees. The hash-consed version performed between 21 to 3 million times faster than its counter-part. Overall, the automatic hash-cons PPX was successful in making declarative code more efficient in both space and time.

6 Defunctionalization

In this chapter we discuss defunctionalization, a technique that transforms higher order functions into first order functions facilitating the automatic correctness proof of programs. We show an example where it can be applied in order to facilitate the correctness proof. We describe the technique and discuss how it helps with deductive program verification. Then we present the Defunctionalization PPX and how defunctionalization is incorporated into OCaml code. At last, we present the results of applying the Defunctionalization PPX and the respective benchmarks.

As stated in chapter 2, declarative programming has several advantages. Some of those are improved code robustness, ease of code analysis and verification. But the paradigm doesn't always make it easy for code to easily be verified or automate proofs that assert code correctness. As stated in chapter 1, errors happen frequently when developing software. Declarative programming helps mitigate some of those by removing the control flow and the notion of state of the program from the user and by stopping the occurrence of side effects. Users may also use specialized debug tools but neither of these is able to actually prove that the program is correct. That is, the program behaves exactly how it was specified by the programmer. In order to make sure that the program works as intended, programmers can perform formal verifications. These are techniques that prove that a program satisfies certain properties specified by the programmer. They prove the program is correct. Programs may be manually proved correct but the process may be also automated. Defunctionalization is a program transformation that translates higher-order functions into first-order ones. This is an interesting transformation for automated formal program verification because one can employ off-the-shelf automated program verifiers to prove the correctness of the generated first-order program[38]. The program transformation can help programmers automate the process of proving their code correct. It may seem like this topic falls outside of the scope of this dissertation given that it's an efficacy problem and not an efficiency one. But we believe that it is still is an important program transformation because it facilitates the application of automatic program verifiers. Consequently, it improves the efficiency of of software development as programmers spend less time verifying the correctness of their code. In the following

section, we discuss what defunctionalization encompasses.

6.1 Defunctionalization

Defunctionalization is a technique used to transform higher-order programs into first-order programs. In higher-order programs, functions can be anonymous, occur as arguments to other functions, as the result of functions and can be values assigned to variables[36]. In first-order programs, functions are always named and are invoked by their name[9]. Our defunctionalization method follows the one proposed by Jonh C. Reynolds[36]. In the resulting first-order program, first-order functions have an auxiliary constructor data type containing the values of free variables and function abstractions become case expressions. The technique can be used to automate the correctness proof of the program. Most parts of defunctionalized programs can be transformed into logical statements that correlate the input and output. These can then be reasoned about and proven correct. The defunctionalized program can also be verified through automatic verification using a first-order verifier[38].

To illustrate the technique, let us look at an example of a function being defunctionalized. We will use the factorial function as an example. An implementation in OCaml is shown in listing 6.1.

Listing 6.1: Factorial function in OCaml.

```
let rec fact n =
   if n <= 1 then 1
   else n * fact (n - 1)
```

This is a standard recursive definition of the factorial function, the product of all positive integers less than or equal to n. If the argument is 0 or 1, then the function returns 1. If the argument is bigger than 1, then the function multiplies the argument with all previous results. Now let us see the defunctionalized factorial function.

Listing 6.2: Defunctionalized factorial function in OCaml.

```
type kont =
   | K0
   | K1 of int * kont

let rec main n = fact n K0

and fact n k =
   if n <= 1 then apply k 1
   else fact (n - 1) (K1 (n, k))

and apply k arg = match k with
   | K0 -> let x = arg in x
   | K1(nf, k) -> let n = arg in apply k (n * nf)
```

As we can see in listing 6.2, the defunctionalized factorial function is more verbose and there is a new data type. The new data type represents a continuation, an abstract representation of the control flow of the program. Making the control flow of the program explicit helps the user reason about correctness because the data can be accessed during runtime instead of being hidden by the language. The actual factorial function is now composed of 3 functions. The first function, `main`, is simply a helper function that calls the `fact` function with the starting continuation. The second function is the actual `factorial` function. The function computes the same result but with the help of the `apply` function. Last we have the `apply` function. This function is an explicit form of the OCaml's call stack. It receives the continuation and performs the computation. Notice that the factorial function no longer performs the multiplication. Instead, it just creates continuations. The multiplication is performed by the `apply` function that receives all the continuations made by the `fact` function. Now, the factorial function is in first-order and the user may apply first-order verifiers, making the correctness proof process automatic[38, 2].

We want to automate the defunctionalization technique using PPX. In the following section, we discuss how the defunctionalization technique can be automatically applied to OCaml code using PPX.

6.2 Defunctionalization with PPX

The automatic hash-consing PPX rewriter is a program transformation that is automatically applied to OCaml code and performs defunctionalization to a function. The PPX assumes that the function is written in continuation passing style. To use the PPX, the user must first write an extension node in their OCaml code. The extension node `%defun` must be written at the beginning of a let-definition, right after the `let`, as shown in the example in listing 6.3.

Listing 6.3: Tree height function with extension node.

```
type t = E | N of t * t

let %defun rec main t = h_cps t (fun [@K0] x -> x)

and h_cps (t: t) (k: int -> 'a) =
   match t with
      | E -> k 0
      | N ((l: t), (r: t)) ->
          h_cps l (fun [@K1] (hl: int) ->
          h_cps r (fun [@K2] (hr: int) ->
          k (1 + max hl hr)))
```

In the example of listing 6.3, there is a tree data type and a function to measure the height of a tree. As indicated by the type, a tree can be either empty with value `E` or a node `N` with two trees. The tree height function measures the height of a tree. If the tree is empty, then the function indicates that the tree has a height of zero. If the tree

has one or more nodes, then the function calculates the height of the tree by returning the maximum between the two branches and adding the height of the node itself. The function is written in continuation passing style as the PPX is expecting it. The user must also annotate places where new functions are defined with attributes. This is needed so that the PPX can correctly collect free variables in the code. The attribute names can be chosen by the user. The @K0, @K1, @K2 were just the ones chosen by us. The user must also indicate the type of the arguments of the function. This is because the PPX is applied during a stage of the compilation pipeline where the OCaml AST is still untyped and the PPX needs to know those types in order to create the new continuation data type. The OCaml compilation pipeline was shown in figure 2.2 of chapter 2. Then the user can use the *rewriter* utility of `ppx_tools` mentioned in chapter 3, with the PPX and target program as argument, to transform the program. This lets the user check the changes that were made to the code. The user may also use the PPX during the compilation process with the *-ppx* flag. The resulting code from applying the defunctionalization PPX to the code in listing 6.3 is shown in listing 6.4.

Listing 6.4: Defunctionalized tree height function.

```
1  type t = E | N of t * t
2
3  type kont =
4      | K0
5      | K1 of t * kont
6      | K2 of kont * int
7
8  let rec main t = h_cps t K0
9
10 and h_cps t k =
11     match t with
12         | E -> apply k 0
13         | N ((l : t),(r : t)) -> h_cps l (K1 (r, k))
14
15 and apply k arg =
16     match k with
17         | K0 -> let x = arg in x
18         | K1 (r,k) -> let hl = arg in h_cps r (K2 (k, hl))
19         | K2 (k,hl) -> let hr = arg in apply k (1 + (max hl hr))
```

After the Defunctionalization PPX was applied, a new data type was added to the code and the factorial function was altered. The tree data type was not altered. Then a new type `kont` was added. This type is used to represent continuations which mimic stacks in OCaml's stack call. As it can be seen, the @K0, @K1, @K2 attributes were converted into components of the `kont` type. Each one is a product type and each component of the product type represents a free variable found by the PPX. This data structure is used in the new factorial function. The factorial function is similar to the previous one but now has an auxiliary function `apply` that computes the continuations. The `h_cps` and

`apply` functions recursively accumulate continuations. Once the tree passed as argument has been traversed, the apply function decomposes the accumulated continuations and computes the height of the tree. This means the behaviour of the calling stack is made explicit and now the user can reason about it which helps prove the correctness of the program. Namely, the user can keep track of each subcomputation, check when it is being executed and what the result is. The user may also use a first-order automatic verifier to prove the correctness of the function given that it is in first-order after the PPX has been applied. Additionally, the result of the function is unaffected by the transformation.

As we have said, the new code doesn't have direct performance improvements. The benefit of this transformation lies in turning the correctness proof of the code smoother and faster for the user. Regardless, it is still important to check and analyse the performance of the new code in case the user wants to execute it. For this reason, we have executed the code, checked and analysed its time and space performance. The results are shown in the following section.

6.3 Defunctionalization PPX Performance Evaluation

To understand if the automatic defunctionalization PPX alters the performance of the code, we executed the two versions of the tree height example shown in the last section. We measured the time it took for the function to end its execution and the memory that it consumed during it. The code was executed in a computer with a 1,6 GHz Dual-Core Intel Core i5 processor and a 8 GB 2133 MHz LPDDR3 memory. There were no other programs being executed other than background operating system processes. Each test was performed five times and the median of the results was selected to avoid skewed results.

6.3.1 Execution Time

The following table 6.1 shows the time it took for each version of the height to terminate its execution. The normal version and the defunctionalized version.

Table 6.1: Execution time of both versions of the tree height function.

Tree Height	Height Function (s)	Defunctionalized Tree Height Function (s)
10	0,000002	0,000003
100	0,000006	0,000004
1.000	0,000033	0,000034
10.000	0,000323	0,000290
100.000	0,018536	0,006875
1.000.000	0,318863	0,249314

Overall we can see that the execution time increases with the height of the tree. We can also see that the difference between both versions is negligible. This means that the user can transform their code using the defunctionalization PPX and then execute it

without having to worry about a loss of performance in time. The user may then debug their code by inspecting the explicit call stack, keeping track of the exact point where the execution is. We must also verify if the defunctionalization has any influence with the memory consumption of the code. In the next subsection we show and analyse the influence of the program transformation in the memory consumed by the code.

6.3.2 Memory Consumption

The following table 6.2 shows the memory consumed by each version of the height function during its execution. The normal version and the defunctionalized version.

Table 6.2: Memory consumption of both versions of the tree height function.

Tree Height	Height Function (MB)	Defunctionalized Tree Height Function (MB)
10	0,01	0,01
100	0,02	0,02
1.000	0,08	0,08
10.000	0,80	0,80
100.000	7,35	7,35
1.000.000	69,42	69,42

The memory consumed increases with the height of the tree which is normal and the difference between both versions is negligible. Similarly to the time results, the defunctionalization of the code does not affect the memory performance of the function. The user may execute the code and not worry about the defunctionalized version consuming more memory.

Overall, we've seen how easy it is to defunctionalize a function using PPX. We've also seen that the defunctionalization technique doesn't alter the performance of the code. It doesn't improve the code which is a pity but it can improve code development by saving time for the user. Making the stack explicit helps the user correct the code. Turning a higher-order function into a first-order function lets the user apply first-order verifiers that automatically prove the code correctness[38, 2].

7

Conclusions

In this chapter we reflect about the dissertation as a whole. We discuss the contributions made with it and how future work may develop.

7.1 Contributions

As stated in chapter 1 our goal with this dissertation was to improve the efficiency of declarative programming. We wanted to be able to take advantage of all the benefits of this programming paradigm while avoiding or minimizing its disadvantages. In order to do this, we used the OCaml PPX technology to apply code optimization techniques automatically. We wanted the code optimization to be as seamless as possible for the user. To do this, we created three PPX rewriters that apply different program transformations. These were implemented to improve code efficiency and help automatically prove code correctness.

The Memoization PPX, automatically memoizes a function in OCaml. We were able to greatly reduce the execution time of functions using the technique. The Hash-Consing PPX, automatically applies hash-consing to a data type in OCaml. The results obtained showed to improve memory consumption of the code and the time performance of the *equal* function thanks to maximal sharing. The Defunctionalization PPX, automatically defunctionalizes a function in OCaml. The technique lets users automatically prove the correctness of their code by transforming a higher-order function into a first-order one. Because of this, the code can be used with first-order verifiers and automatically proven correct.

By using PPX, we were able to let the user program use the declarative paradigm and consequently take advantage of all its advantages. The user was also able to optimize their code without having to think of ways to do it themselves. All the cumbersome aspects of creating extra code were shifted from the user to the PPX. Each of the three implemented PPX covers a different group of disadvantages of the paradigm. They were successful in improving the efficiency of declarative code in OCaml.

7.2 Future Work

Our work has covered a wide range of characteristics from declarative programming. Nevertheless, there are always other improvements that can be made.

The first thing that can be done, is to create one collective PPX that is able to apply any of the three created. This ensures the user simply uses a single PPX file but takes advantage of the different optimizations. Another improvement that can be done is to create a PPX that detects flaws in the code. At the moment, the user must know what parts of the code are lacking in performance and know which PPX is able to improve it. A PPX that could make this detection would save even more time for the user. It could even be implemented together with the current PPX created.

7.3 Final Remarks

The project was quite ambitious given that it required getting to learn and know in depth the OCaml AST, learn to write PPX, learn the optimization techniques, and know how to combine all of these together. Overall, we have achieved the goal of this dissertation. Even with the current situation of the world caused by the covid-19 pandemic which hindered the work done. We found a way to use declarative programming and remove the efficiency problems that come with it by using PPX rewriters.

Bibliography

[1] V. Balat, J. Vouillon, and B. Yakobowski. "Experience Report: Ocsigen, a Web Programming Framework". In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. Vol. 44. ICFP '09. Edinburgh, Scotland: Association for Computing Machinery, 2009-08, pp. 311–316. ISBN: 9781605583327. DOI: 10.1145/1596550.1596595 (cit. on p. 3).

[2] P. Barroso, M. Pereira, and A. Ravara. "Animated Logic: Correct Functional Conversion to Conjunctive Normal Form". In: (2019). URL: https://release.di.ubi.pt/factor/pdfs/Animated_Logic_I_CNF.pdf (cit. on pp. 17, 43, 46).

[3] T. Braibant, J.-H. Jourdan, and D. Monniaux. "Implementing and reasoning about hash-consed data structures in Coq". In: *Journal of automated reasoning* 53.3 (2014), pp. 271–304 (cit. on p. 30).

[4] "Camlp4". In: *Practical OCaml*. Berkeley, CA: Apress, 2007, pp. 411–429. ISBN: 978-1-4302-0244-8. DOI: 10.1007/978-1-4302-0244-8_29 (cit. on p. 14).

[5] Camlp4. *Camlp4*. Accessed: 30th November 2021. URL: https://github.com/camlp4/camlp4 (cit. on p. 14).

[6] M. Cimadamore and M. Viroli. "A Prolog-oriented extension of Java programming based on generics and annotations". In: *Proceedings of the 5th international symposium on Principles and practice of programming in Java*. PPPJ '07. Lisboa, Portugal: Association for Computing Machinery, 2007, pp. 197–202. ISBN: 9781595936721. DOI: 10.1145/1294325.1294352 (cit. on p. 7).

[7] P. Cuoq and D. Doligez. "Hashconsing in an Incrementally Garbage-Collected System: A Story of Weak Pointers and Hashconsing in OCaml 3.10.2". In: *Proceedings of the 2008 ACM SIGPLAN workshop on ML*. 2008, pp. 13–22 (cit. on pp. 30, 33).

[8] O. Danvy and L. R. Nielsen. "CPS Transformation of Beta-Redexes". In: *BRICS Report Series* 39 (2004-12) (cit. on p. 16).

[9] O. Danvy and L. R. Nielsen. “Defunctionalization at Work”. In: *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP ’01. Florence, Italy: Association for Computing Machinery, 2001, pp. 162–174. ISBN: 158113388X. DOI: 10.1145/773184.773202 (cit. on pp. 17, 42).

[10] Facebook. *React A JavaScript library for building user interfaces*. Accessed: 30th November 2021. URL: https://reactjs.org (cit. on p. 3).

[11] J.-C. Filliâtre and S. Conchon. “Type-Safe Modular Hash-Consing”. In: *Proceedings of the 2006 Workshop on ML*. ML ’06. Association for Computing Machinery, 2006-09, pp. 12–19. ISBN: 1595934839. DOI: 10.1145/1159876.1159880. URL: http://www.lri.fr/~filliatr/ftp/publis/hash-consing2.pdf (cit. on pp. 20, 32, 33, 38).

[12] T. Haenisch. “A Case Study on Using Functional Programming for Internet of Things Applications”. In: *Athens Journal of Technology & Engineering* 3.1 (2016). DOI: 10.30958/ajte.3-1-2 (cit. on p. 1).

[13] K. Hammond. “Why Parallel Functional Programming Matters: Panel Statement”. In: *International Conference on Reliable Software Technologies - Ada-Europe 2011*. Springer. 2011, pp. 201–205 (cit. on p. 1).

[14] J. Hughes. “Why Functional Programming Matters”. In: *The Computer Journal* 32.2 (1989-01), pp. 98–107. DOI: 10.1093/comjnl/32.2.98 (cit. on p. 1).

[15] S. L. P. Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987-01. ISBN: 013453333X. URL: https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/ (cit. on pp. xii, 18, 19).

[16] S. Joosten, K. V. D. Berg, and G. V. D. Hoeven. “Teaching Functional Programming to First-Year Students”. In: *Journal of Functional Programming* 3.1 (1993), pp. 49–65. DOI: 10.1017/S0956796800000599 (cit. on p. 3).

[17] G. L. S. Jr. *Rabbit: A compiler for Scheme*. Tech. rep. 1978 (cit. on pp. 16, 17).

[18] X. Leroy et al. *The OCaml system release 4.05: Documentation and user’s manual*. 2017-07 (cit. on pp. 11, 35).

[19] J. Lloyd. “Practical Advantages of Declarative Programming”. In: *GULP-PRODE ’94 1994 Joint Conference on Declarative Programming*. 1994, pp. 1, 7 (cit. on p. 1).

[20] J. M. Lourenço. *The NOVAthesis LaTeX Template User’s Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[21] D. J. F. Martins. “JavaCO – Uma variante do Java com um sistema de tipos baseado em covariância”. MA thesis. Faculdade de Ciências e Tecnologia, 2010. URL: https://run.unl.pt/handle/10362/4590 (cit. on p. 7).

[22] B. McAdam. "Y in Practical Programs Extended Abstract". In: (2001) (cit. on p. 19).

[23] D. Michie. ""Memo" Functions and Machine Learning". In: *Nature* 218.5136 (1968), pp. 19–22 (cit. on p. 23).

[24] Y. Minsky, A. Madhavapeddy, and J. Hickey. *Real World OCaml: Functional programming for the masses*. 2nd ed. (in progress). Accessed: 30th November 2021. 2020. URL: http://dev.realworldocaml.org/index.html (cit. on pp. 12, 37, 38).

[25] OCaml. *Functor Weak.Make*. Accessed: 30th November 2021. URL: https://caml.inria.fr/pub/old_caml_site/ocaml/htmlman/libref/Weak.Make.html (cit. on pp. 33, 38).

[26] OCaml. *If Statements, Loops and Recursion*. Accessed: 30th November 2021. URL: https://ocaml.org/learn/tutorials/if_statements_loops_and_recursion.html#Tail-recursion (cit. on p. 17).

[27] OCaml. *What is OCaml?* Accessed: 30th November 2021. URL: https://ocaml.org/learn/description.html (cit. on p. 10).

[28] ocaml-ppx. *Camlp4*. Accessed: 30th November 2021. URL: https://github.com/ocaml-ppx/ppx_tools (cit. on p. 15).

[29] ocaml-ppx. *OCaml-migrate-parsetree*. Accessed: 30th November 2021. URL: https://github.com/ocaml-ppx/ocaml-migrate-parsetree (cit. on p. 16).

[30] ocaml-ppx. *Ppxlib - Meta-programming for OCaml*. Accessed: 30th November 2021. URL: https://github.com/ocaml-ppx/ppxlib (cit. on p. 16).

[31] T. H. Park et al. "Towards a Taxonomy of Errors in HTML and CSS". In: *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*. ICER '13. Association for Computing Machinery, 2013-08, pp. 75–82. ISBN: 9781450322430. DOI: 10.1145/2493394.2493405 (cit. on p. 3).

[32] R. Pawlak et al. "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code". In: *Software: Practice and Experience* 46.9 (2016), pp. 1155–1179. DOI: 10.1002/spe.2346 (cit. on p. 7).

[33] J. Pedersen. "Classification of Programming Errors in Parallel Message Passing Systems". In: *Communicating Process Architectures 2006: WoTUG-29: Proceedings of the 29th WoTUG Technical Meeting, 17-20 September 2006, Napier University, Edinburgh, Scotland*. Vol. 64. IOS Press. 2006, pp. 363–376 (cit. on p. 2).

[34] G. Radanne, J. Vouillon, and V. Balat. "Eliom: A core ML language for Tierless Web programming". In: *APLAS 2016*. 2016-11, pp. 377–397. URL: https://hal.archives-ouvertes.fr/hal-01349774 (cit. on p. 3).

[35] D. Rémy. "Using, Understanding, and Unraveling The OCaml Language From Practice to Theory and vice versa". In: *International Summer School on Applied Semantics*. Springer. 2000, pp. 413–536. ISBN: 3-540-44044-5. URL: http://pauillac.inria.fr/~remy/cours/appsem/ocaml.pdf (cit. on pp. xii, 18, 19).

[36] J. C. Reynolds. “Definitional interpreters for higher-order programming languages”. In: *Higher-Order and Symbolic Computation* 11 (1998), pp. 363–397. DOI: 10.1023/A:1010027404223 (cit. on pp. 17, 42).

[37] P. V. Roy. “Programming Paradigms for Dummies: What Every Programmer Should Know”. In: *New computational paradigms for computer music* 104 (2009), pp. 616–621 (cit. on pp. 8, 9).

[38] T. L. Soares. “A Deductive Verification Framework For Higher Order Programs”. In: *CoRR* abs/2011.14044 (2020). arXiv: 2011.14044. URL: https://arxiv.org/abs/2011.14044 (cit. on pp. 17, 41–43, 46).

[39] G. J. Sussman and G. L. S. Jr. “Scheme: A interpreter for extended lambda calculus”. In: *Higher-Order and Symbolic Computation* 11.4 (1998), pp. 405–439 (cit. on p. 16).

[40] M. Zuckerberg. Accessed: 30th November 2021. 2017-06. URL: https://www.facebook.com/zuck/posts/10103831654565331 (cit. on p. 1).

2021
Efficient Declarative Programming in OCaml
Diogo Silvério
NOVA
NOVA SCHOOL OF
SCIENCE & TECHNOLOGY