POLITECNICO DI MILANO
MSc in Computer Science and Engineering
School of Industrial and Information Engineering

Toward a more expressive pattern matching in Haskell

Supervisor: Prof. Matteo Pradella

Master thesis by:
Giacomo Servadei, ID 854819

Academic year 2016-2017

Abstract

Pattern matching has become a very important feature in functional programming. Many languages in this category heavily depend on it, especially when it comes to the definition of functions. In this case, pattern matching acts like a dispatcher that executes the right portion of code depending on the structure of the arguments. Coding with this pattern-action paradigm leads to less error-prone programs. But how powerful can patterns be, so that always more logic is expressed with them rather than in the code that will be executed? Many languages try to make patterns more expressive and powerful. One example is F# that makes the pattern system extensible with its active patterns.

In this thesis, I analyze pattern matching in Haskell, a purely functional programming language, finding its strengths and weaknesses. Once identified those, I describe an extension that takes advantage of the power of its pattern matching and tries to overcome some of its limits. This extension is defined as a superset of the Haskell syntax, this means that every program written in the extended form can be compiled down to Haskell.

Sommario

Il Pattern matching è diventato molto importante nella programmazione funzionale. Molti linguaggi in questa categoria ne dipendono pesantemente, soprattutto per quanto riguarda la definizione di funzioni. In questo caso il pattern matching assume il ruolo di dispatcher ed esegue la giusta porzione di codice in base alla struttura dei parametri. Programmare con questo paradigma pattern-action, porta ad ottenere programmi meno soggetti a errori. Ma quanto potenti possono essere i pattern in modo che sempre più logica sia espressa al loro interno piuttosto che nel codice che verrà eseguito? Molti linguaggi cercano di rendere i pattern più espressivi e potenti. Ne è un esempio F# che rende i pattern estensibili con gli active patterns.

In questa tesi analizzo il pattern matching di Haskell, un linguaggio puramente funzionale, cercando i suoi punti di forza e debolezza. Una volta identificati, descrivo un'estensione che sfrutta la potenza del suo pattern matching, e ne cerca di superare alcuni limiti. L'estensione è definita come un superset della sintassi di Haskell, in questo modo ogni programma scritto nella forma estesa può essere compilato in Haskell.

Contents

Abstract

Sommario

1 Introduction **1**
1.1 Objectives 1
1.2 Thesis structure 1

2 Background **3**
2.1 Pattern Matching in Programming Languages 3
2.2 Primitive Patterns 4
2.3 Data Structure Patterns 4
2.3.1 Lists 4
2.3.2 Tuples 5
2.4 Algebraic Data Types Pattern 5
2.4.1 Algebraic Data Types 5
2.4.2 Pattern 6
2.5 History 6
2.6 Pattern Matching examples 7
2.6.1 ML 7
2.6.2 F# 7
2.6.3 CDuce 8

3 Pattern Matching in Haskell **10**
3.1 Haskell Syntactic Sugar 10
3.1.1 Curried Functions 10
3.1.2 Function definition 10
3.2 Haskell Pattern Matching 11
3.2.1 Informal semantics 12
3.2.2 Guards 13

3.2.3 As pattern 13
3.3 Pattern evaluation 14
3.4 Limits 15
3.4.1 Linear patterns 15
3.4.2 Non-exhaustive patterns 16

4 Extended Pattern Matching in Haskell **18**
4.1 Non-linear patterns 19
4.2 Disjunctive patterns 20
4.3 Assign pattern 22
4.4 Implementation 24
4.4.1 Architecture 24
4.4.2 Usage 24

5 Conclusion and Future Work **26**

Bibliography **28**

Chapter 1

Introduction

1.1 Objectives

Pattern matching has become a main pillar in Functional programming. It forces the programmer to write code in a more declarative manner and it improves readability. Nevertheless, declaring functions by specifying the patterns to which parameters has to match, leads to less error-prone programs.

Haskell, being one of the purest functional programming language, has its own implementation of pattern matching. It is powerful and quite expressive but like everything in computer science, it has some limits. One of the strictest one, is that patterns have to be linear, meaning that one variable cannot appear more than once inside the same pattern.

The main objective of this thesis, is to first analyze Haskell pattern matching and then to extend it in order to improve both its power and expressiveness.

1.2 Thesis structure

In chapter 2 Pattern Matching is introduced in general. There is a theoretical description followed by a bit of history and a general overview of it's implementation in some functional programming languages. After this theoretical introduction, in chapter 3, Haskell pattern matching will be analyzed more deeply showing its characteristics, as well as finding its strengths and its limits. Chapter 4 will then propose a possible extension of Haskell pattern matching from a design point of view, whose implementation will be presented in chapter 5. Finally conclusions and future works will be pointed

up in the last chapter, chapter 6. Every coding example (besides in the second part of chapter 2 where some programming languages will be presented) will be written in Haskell syntax.

Chapter 2

Background

2.1 Pattern Matching in Programming Languages

$PatternMatching$ is a technique based on defining a pattern and finding its presence within a sequence of tokens. The most famous tool based on this technique is $RegularExpressions$, which are mainly used to perform ”find” or ”find and replace” operations on strings.

As a construct in programming languages, it allows the programmer to write patterns to which data should conform. In the matching phase, the value is deconstructed and matched against the pattern. When this succeeds, eventually, the variables in the pattern are bound to the corresponding values. Patterns are different from normal expressions because they do not get executed, but rather they describe the characteristics, or the structure, that the value should have.

Pattern Matching is present in many functional programming languages, in which it is usually adopted to define functions. In this context, it can be seen as a more powerful version of the switch statement. In fact, pattern matching acts as a dispatcher mechanism, choosing which variant of a function is the correct one to call based on its parameter. Defining a function by means of patterns is less error-prone and it improves readability. Also, from a static analysis / diagnosis point of view of the program, patterns make it easy to do exhaustiveness, equivalence and emptiness checking. The reason why they are more common in functional programming is that they force the programmer to think in a more declarative way, rather than imperative.

2.2 Primitive Patterns

The simplest pattern that one can write, is a constant. For example (all the code example of this thesis will be written in Haskell):

```
fact 0 = 1
```

Here, the constant 0 is the pattern. The function *f* returns 1 only when it is called with 0 as parameter. So every time there is a constant in the pattern, the corresponding value has to be exactly equal to it. In order to make the pattern more general, it is possible to introduce a variable.

```
fact n = n * fact (n - 1)
```

Now any value will match the pattern n, and it will also be bound to the variable. In this case, the two bodies of *f* makes a really simple recursive definition of the factorial function. When defining functions with patterns it is important to leave the more general patterns at the end of the definition. The reason for this is to avoid non-terminating program. For example, in the definition of the factorial, if the variable pattern came first, it would match any value including 0 and all negative numbers, leading to an infinite loop.

A small variation of the variable pattern is the wildcard. This also matches any value, but without performing any binding. One should use it instead of the variable, every time the value it is not needed in the body of the function.

2.3 Data Structure Patterns

2.3.1 Lists

Patterns can also be more expressive than the basic constant and variable ones. Some languages have built-in capabilities to describe the internal structure of a data structure within a pattern. The most common example is the *cons* operator. The cons operator takes two values, or pointers, and combine them together in a pair. A cascade of cons operators can be used to create a list, where the second value of the innermost pair contains an empty list. Although this is how lists work in most functional programming languages, the mechanism tends to be hidden behind syntactic sugar.

```
f (1:2:3:[]) = True
```

```
f [1,2,3] = True
```

The two patterns above express the same concept: a list of exactly 3 elements which, in order, are 1,2 and 3. In the first case, the logic is explicit, with colon being the cons operator and the open and close square brackets being the empty list. In the second, instead, the logic is hidden. It is possible to combine data structure patterns and variable patterns in order to make even more expressive patterns.

```
length [] = 0
length (x:xs) = 1 + length xs
```

In this case, the two function bodies define the length function, which returns the length of a list. In the first pattern we try to match an empty list, in case this succeeds, the answer is 0 (the length of an empty list is indeed 0). Otherwise, we try to match a list with at least one element and we leave the second parameter of the cons operator generic, binding it to a variable. Doing so lists of any length will be matched (including the empty one) and the function is then recursive called on this second part.

2.3.2 Tuples

Another common pattern that describes a data structure is the tuple. Tuples, like lists, are used to store a sequence of values, with the difference being that they contain only a fixed amount of them. Usually, they are denoted with parenthesis and their values are separated by commas.

```
fst (x,_) = x
snd (_,x) = x
```

2.4 Algebraic Data Types Pattern

2.4.1 Algebraic Data Types

Algebraic Data Types is a very common concept in functional programming. An algebraic data type is a possibly recursive sum type of product types. A sum type is a disjoint union of a set of classes of values, called variants. Each variant has its own constructor. A product type, instead, is a Cartesian product of a set of classes of values, called fields.

```
data Bool = False | True
data Int = -2147483648 | -2147483647 |
   ... | -1 | 0 | 1 | 2 | ... | 2147483647
data Point = Point Int Int
```

In the example above, Bool and Int are two sum types while Point is a product type. The first has only two variants: False and True; the second instead has all the possible 2^{32} values of a signed 32-bit integer. Point on the other hand, has only one variant, which is the Cartesian product of two fields of type Int.

As stated in its definition, algebraic data types can also be recursive. This means that one of the variants of a data type can be the data type itself.

```
data List a = Nil | Cons a (List a)
```

Here, the same logic of the cons operator is used to define recursively a list data type. With the Nil variant being the empty list, and the product type Cons being the pairing between an element with the rest of the list.

2.4.2 Pattern

Algebraic Data Types patterns allow matching a value against a data type. In the example below, there is the same implementation seen previously in this chapter of the length function, but this time using the List data type just defined.

```
length Nil = 0
length (Cons _ xs) = 1 + length xs
```

As we can see this kind of patterns acts like a kind of type checker. It dispatches the correct definition of the function, by matching it with the type of the parameter. While doing so, it also deconstructs the parameter and binds its values to variables.

2.5 History

Pattern matching started to appear in programming languages after regular expressions became popular. The first application of regular expressions

(and so of pattern matching) in a program was in 1968 when Ken Thompson extended the seeking and replacing features of the QED editor to accept them. The first languages to implement pattern matching were SNOBOL from 1962 [1], Refal from 1968 [2], SASL from 1976 [3], NPL from 1977, and KRC from 1981.

2.6 Pattern Matching examples

2.6.1 ML

ML has a pretty standard implementation of pattern matching. Patterns can be expressed in both function definitions and case expressions.

```
fun fact 0 = 1
  | fact n = n * fact (n - 1)
fun length [] = 0
  | length (_::xs) = 1 + length (xs)
```

In the two examples above we see how constant, variable and data structure pattern are available in the patterns.

2.6.2 F#

The F# language, being in the ML family, has also a standard implementation of pattern matching. What makes F# an interesting case study is the presence of "Active Patterns" [4], a tool to extend the pattern system. The main idea behind active patterns is to be able to decompose data into a number of sub-cases without references to an algebraic data type. An active pattern, in fact, is just a regular function that is defined using a new syntactic element called a structured name and that can be used inside a pattern.

```
let (|Even|Odd|) inputNumber =
  if inputNumber % 2 = 0 then Even else Odd

let TestNumber number =
match number with
  | Even -> printfn "%d is even" number
  | Odd -> printfn "%d is odd" number
```

Here we declare a multi-case active pattern which divides the input into two sub-cases: Even and Odd. The expression part of this special let statement is a function that assigns the input to the right sub-case. Once the new pattern is declared, it is possible to refer to it from inside a function definition, or from any other place in which pattern can be used. The TestNumber function, in fact, matches on the sub-cases as if they were data types. Active patterns in F# are divided into two main categories: total and partial. The one seen before is total, meaning that the input is decomposed into a complete set of sub-cases, while a partial one partitions only a part of the input space. Partial active patterns are declared like totals, but one of the sub-case has to be the wildcard.

```
let (|Fizz|_|) input = if input % 3 = 0 then Some() else None
let (|Buzz|_|) input = if input % 5 = 0 then Some() else None

let whatToPrint number =
  match number with
  | Fizz & Buzz -> printfn "FizzBuzz"
  | Fizz -> printfn "Fizz"
  | Buzz -> printfn "Buzz"
  | _ -> printfn ""
```

In this example, we see that the two partial patterns Fizz and Buzz matches only partially the input.

2.6.3 CDuce

CDuce [5] is a functional language whose design is targeted to XML applications. Its pattern matching is inspired by ML's one, but is much more powerful and is heavily oriented to type checking.

In CDuce, every type can be used as a pattern, which semantics is to only accept values of that type without any binding. This is often combined to a variable pattern with the conjunction operator & for patterns. The conjunction operator matches only if both its left side and its right side match. Therefore by combining these two patterns together, we obtain a pattern which matches and binds to a variable only values of a specific type. In addition to the conjunction operator, there is also the disjunction one. It is expressed with a | symbol and it matches the first side which matches. This is often used with the default value pattern (x:=c). This always matches and binds a variable to a constant value. The semantics obtained by this

composition is to use the default pattern value in case the previous pattern didn't match.

Chapter 3

Pattern Matching in Haskell

3.1 Haskell Syntactic Sugar

Haskell is a lambda calculus based programming language with three main characteristics: it is purely functional, statically typed, and lazy. Every detail of the language definition is contained in the language report of 2010 [6]. Many of the constructs found in the language are just syntactic sugar. This allows Haskell to have a relatively small core.

3.1.1 Curried Functions

One of the main examples of sugar is curried functions. One of the peculiarity of Haskell is that every function takes just one argument. The process of curring allows transforming a multi-parameter function, into many single parameter lambda functions, each of them returning another function. In the next example, there is a function that sums two numbers before and after currying.

```
add x y = x + y
add' = \x -> \y -> x + y
```

3.1.2 Function definition

The dispatching mechanism introduced with pattern matching is also just syntactic sugar. Every function definition can be easily transformed into a case expression where each of the bodies becomes a case alternative. This goes along with the claim that pattern matching can be seen as a more

powerful version of the switch statement. In fact, in Haskell, that's exactly what it is.

```
fact 0 = 1
fact n = n * fact (n - 1)

fact' x = case x of
  0 -> 1
  n -> n * fact' (n - 1)
```

3.2 Haskell Pattern Matching

Pattern matching in this language is powerful and expressive. It appears in lambda functions, function definitions, do expressions, list comprehension, and case expression. As in the language report, this is the pattern grammar:

```
pat   =>  lpat qconop pat            (infix constructor)
      |   lpat

lpat  =>  apat
      |   - (integer | float)        (negative literal)
      |   gcon apat1 ... apatk       (arity gcon = k, k >= 1)

apat  =>  var [ @ apat]              (as pattern)
      |   gcon                       (arity gcon = 0)
      |   qcon { fpat1 ... fpatk }   (labeled pattern, k >= 0)
      |   literal
      |   _                          (wildcard)
      |   ( pat )                    (parenthesized pattern)
      |   ( pat1 ... patk )          (tuple pattern, k >= 2)
      |   [ pat1 ... patk ]          (list pattern, k >= 1)
      |   ~ apat                     (irrefutable pattern)

fpat  =>  qvar = pat
```

Patterns in Haskell can contain constants, variables, and wildcards. They can also deconstruct and match the two built-in data structures: lists and tuples. As far as algebra data types are concerned, they can be deconstructed by just typing the name of the constructor followed by as many patterns

as its arity. One of the first thing that we find which is not common is the fact that infix constructors are also supported. The colon cons operator is an example of a built-in one. Furthermore, since in Haskell it is possible to add names to the various fields of a data type, patterns can match on them with the so-called labeled pattern.

3.2.1 Informal semantics

Attempting to match a pattern can have one of three results: it fails, it succeeds, returning a binding for each variable in the pattern, or it is undefined. Pattern matching proceeds from left to right, and outside to inside, according to the following rules:

- Matching the variable pattern *var* against a value v always succeeds and binds *var* to v.
- Matching the lazy pattern ~*apat* against a value v always succeeds. The actual matching is done on ~*apat* only if and when a variable in *apat* is used. At that point, the entire pattern is matched against the value, and if the match fails or diverges, so does the overall computation (in lazy patterns binding does not imply evaluation, there will be a subsection explaining this mechanism).
- Matching the wildcard pattern _ against any value always succeeds, and no binding is done.
- Matching the constructor pattern *con pat* against a value, where *con* is a constructor defined by *newtype*, depends on the value:
  - If the value is of the form *con* v, then *pat* is matched against v.
  - If the value is undefined, then *pat* is matched against $undefined$.

  That is, constructors associated with newtype serve only to change the type of a value.
- Matching the constructor pattern *con pat*1 ... *patn* against a value, where *con* is a constructor defined by *data*, depends on the value:
  - If the value is of the form *con* $v1$ ... vn, sub-patterns are matched left-to-right against the components of the data value. If all matches succeed, the overall match succeeds, the first to fail or diverge causes the overall match to fail or diverge, respectively.

  - If the value is of the form *con′ v*1 ... *vm*, where *con* is a different constructor to *con′*, the match fails.
  - If the value is undefined, the match diverges.

- Matching against a constructor using labeled fields is the same as matching ordinary constructor patterns except that the fields are matched in the order they are named in the field list. All fields listed must be declared by the constructor; fields may not be named more than once. Fields not named by the pattern are ignored (matched against _).

- Matching a numeric, character, or string literal pattern k against a value v succeeds if $v == k$, where $==$ is overloaded based on the type of the pattern. The match diverges if this test diverges.

- Matching an as-pattern *var*@*apat* against a value v is the result of matching *apat* against v, augmented with the binding of *var* to v. If the match of *apat* against v fails or diverges, then so does the overall match.

3.2.2 Guards

Besides all these common features, Haskell has quite a few interesting additions in terms of pattern matching. For example patterns in Haskell can not only assure that a value has a certain structure, but they can also assure that some of their properties hold. The construct that allows this is called Guard. A guard is just a boolean expression that operates on the variables of the pattern. Every function definition can have any number of guards. The variant of the function that gets executed is the first one that has the guard returning true. There is also the *otherwise* keyword that gets executed in case none of the other guards did.

```
fact n
  | n < 2     = 1
  | otherwise = n * fact (n - 1)
```

As shown in the example, guards are introduced with the pipe sign, and they are more indented than the pattern.

3.2.3 As pattern

As patterns add the possibility to keep a reference to the content of the value being currently deconstructed and matched. It is introduced by a variable

name, an at sign, and the pattern itself. The variable will be bound to the value only if the pattern succeeds.

```
firstChar "" = "Empty String"
firstChar all @ (x:_) = "The first char of "
                        ++ all ++ " is " ++ x
```

3.3 Pattern evaluation

Haskell evaluates programs with the so-called lazy evaluation methods. This means that expressions are not evaluated when they are bound to variables, but their evaluation is deferred until their results are needed by other computations. In consequence, arguments are not evaluated before they are passed to a function, but only when their values are actually used. If Haskell didn't use this method, operations on infinite lists wouldn't terminate. For example. here we are trying to increment by one all the elements of an infinite list by using a map. Afterward, we take the first 5 elements of the resulting list. Without lazy evaluation, the map over the infinite list would go on forever and never terminate. Instead, we see the expected result [2,3,4,5,6].

```
> take 5 $ map (+1) [1..]
[2,3,4,5,6]
```

When it comes to pattern matching, their evaluation can be either lazy or strict depending on the context in which the pattern is used. In let expressions, patterns are evaluated lazily, while in case expressions, therefore in function definitions, they are evaluated strictly.

```
> f (Just x) = 1
> f undefined
*** Exception: Prelude.undefined
> let Just x = undefined in 1
> 1
```

In this example, the right-hand side of function f doesn't need the actual value of x. Regardless, when undefined is passed to the function, we get an exception. This tells us that the evaluation of the Just constructor pattern happens before the execution of the function. On the other hand, in the let expression, even if we try to match undefined with the Just constructor

pattern, we still get 1 as result. The evaluation of the pattern in the case expressions can be forced to be lazy. The so-called lazy patterns (or irrefutable pattern) are introduced with the ~ operator. In the following example we take full advantage of this new operator and when we call f with undefined, we get 1 as result.

```
> f ~(Just x) = 1
> f undefined
1
```

3.4 Limits

3.4.1 Linear patterns

The biggest limit of patterns in Haskell is that they have to be linear. This means that they cannot contain the same variable more than once. Having the same variable appearing more than once inside a pattern would implicitly mean that the content of the corresponding values has to be equal to each others. To understand this idea better let's see two examples of non-linear patterns in Haskell.

```
testEqual x x = True
testEqual _ _ = False

isSingletonOf x [x] = True
isSingletonOf _ _ = False
```

The testEqual function would return True only if the two values were equal. While the second one, given a value and a list, would test whether the list contained only that value and only once. Just to be clear this kind of pattern is not allowed in Haskell.

This design choice of the language has been the point of many debates inside the Haskell community. Some users would love to have this feature while others don't. The most recurring reason is that a non-linear pattern would require the function to have an Eq type, in order to be able to perform the equality test. Also, the pattern would need to be evaluated before the execution of the equality test, this means that it wouldn't be possible for it to be irrefutable. These two points, in my opinion, are rather weak. Eq constraint, in fact, is already necessary every time in the pattern there is

a constant. Without it, it would be impossible to check whether the value is equal to the constant. So having this kind of constraint automatically appear in non-linear patterns wouldn't be much of a problem.

```
> f 0 = 1
> :t f
f :: (Eq a, Num a, Num p) => a -> p
```

3.4.2 Non-exhaustive patterns

As we said before in this thesis, pattern matching is also a dispatcher when multiple patterns are used to declare different variants of a function. The set of patterns of a function can be partial, meaning that it is possible that the function is not defined for some input. When said input is passed to the function, a runtime error for non-exhaustive pattern gets risen. The fact that Haskell doesn't force the programmer to write total functions, can lead to errors more easily. Let's see an example:

```
toUpper :: [Char] -> [Char]
toUpper = map up where
  up 'a' = 'A'
  up 'b' = 'B'
  up 'c' = 'C'
  ...
  up 'x' = 'X'
  up 'y' = 'Y'
  up 'z' = 'Z'

> :l toUpper.hs
> toUpper "hello"
"HELLO"
> toUpper "Hi Giacomo!"
*** Exception: Non-exhaustive patterns in function up
```

The goal here is to write a function that converts all the characters of a string to their upper-case correspondent. This is implemented by mapping the string with a function that converts a character to its upper-case correspondent. This inner function, as we see from the code, is not defined for every possible character, therefore when it gets a character that doesn't

match any of the patterns, we get a non-exhaustive pattern exception. Obviously, function can be easily fixed by adding one final catch-all variable pattern which returns the same input it received. Who wrote the function is the one to blame here, but the point is that Haskell allows this. To mitigate this problem, GHC [7] has a static type-checker which raises a warning when in the code there is a non-exhaustive pattern. This warning, however, is not raised by default, but it has to be manually activated with the flag *f-fwarn-incomplete-patterns*. Also, there are some research in this area that tries to solve the problem [8]. While these tentative solutions exist, it is a good habit to always define our functions for all the possible inputs they can accept.

Chapter 4

Extended Pattern Matching in Haskell

In the last chapter, we saw how pattern matching in Haskell is top-notch and allows the programmer to write quite powerful patterns. One of the most powerful features is guards, with which we can add checks to the content of the values being matched in addition to the structure. Being guards boolean functions, they can be completely arbitrary, as long as they use the same variables contained in the pattern. Besides being this good, pattern matching in Haskell has also some limits: every pattern must be linear, meaning that each variable can appear only once inside the same pattern; the set of patterns used when defining a function must be total, otherwise, it can lead to non-exhaustive pattern exceptions.

Here we will describe an extension to Haskell, both in term of syntax and semantic, that tries to improve the expressiveness of patterns and to mitigate their limits. The main idea behind this extension is that every new construct or semantic added to the language can be also written in plain Haskell. By doing this way, the core language remains untouched and the extension can be implemented without changing GHC, but just with a preprocessor. This preprocessor takes the source code written in the new extended version of Haskell and compiles it down to the non-extended version.

The three extensions will be non-linear patterns, disjunctive patterns, and assign patterns. For each of these three, we will see both a formal and informal definition, how it will be compiled to plain Haskell and also some examples.

4.1 Non-linear patterns

Let's start by introducing the first extension which is the possibility to have non-linear patterns. This allows the programmer to use the same variable more than once in the same pattern. The meaning of having the same variable repeated within the same pattern is that the values must be equals along all the matches. That value will also be bound to the variable.

The simplest example is this one:

```
areEqual x x = True
areEqual _ _ = False

areEqualThree x x x = True
areEqualThree _ _ _ = False
```

The *areEqual* function tests whether the two arguments it accepts are equal by taking advantage of non-linear patterns. It simply use the same variable x twice in the pattern. The exact same reasoning is behind function *areEqualThree* where instead of checking the equality between two arguments, it is checked between three. Let's see another slightly more complex example:

```
checkCoupledList [] = True
checkCoupledList (x:x:xs) = checkCoupledList xs
checkCoupledList _ = False

> checkCoupledList [1,2,3]
False
> checkCoupledList [1,1,2,2,3,3]
True
> checkCoupledList $ concatMap (replicate 2) [1..10000]
True
```

Here we want to check whether a list is composed of couples of equal adjacent objects. This can be achieved without writing a single equality test operator but just by taking full advantage of pattern matching. The idea behind the implementation is: as long as I have a list composed of two equal elements and something after them, I discard the elements and I keep checking; as soon as I reach the end of the list it means that the list as the desired structure and I return *True*; in any other case (the current sublist has only

one element or the first two elements are different) I return $False$.

This first extension doesn't change the syntax of the language, nor it adds new construct. It just adds the semantic of having two variables in the same pattern. The corresponding Haskell code for non-linear patterns is really simple. The goal is to ensure that two variables contained in the pattern have the same content. As we saw in the last chapter Haskell has a really cool feature that allows adding checks on the content of values being matched which is guards. Therefore, every time a variable get repeated more than once within the same pattern, new variables must be inserted in the pattern replacing the repeated once, and the guard must add the check that the old variable is equal to the new one. It is as simple as that. Let's see how the three functions $areEqual$, $areEqualThree$, and $checkCoupledList$ that we introduced before can be translated in Haskell:

```
areEqual x y | x == y = True
areEqual _ _ = False

areEqualThree x y z | x == y && x == z = True
areEqualThree _ _ _ = False

checkCoupledList [] = True
checkCoupledList (x:y:xs) | x == y = checkCoupledList xs
checkCoupledList _ = False
```

As we can see the second and third time the same variable x appeared it got replaced by the new variables y and z, and the guard ensures that these variables are all equal.

When introducing the equality test in the guard, we are forcing the type of the arguments to be a member of the Eq class because only the member of this class can be tested for equality. In fact, every member of this class must define the behavior of the $==$ operator. This new constraint may seem a bit odd, but it also gets added every time a constant pattern is used because the only way Haskell can know whether a value matches a constant is by using the equality test.

4.2 Disjunctive patterns

The second feature that we see now is the possibility to write disjunctive patterns (also called or patterns). As we saw so far pattern matching in

Haskell doesn't give any flexibility on the values being matched, meaning that either we don't care at all of the value or this must match exactly the pattern. With disjunctive pattern this is relaxed and it is possible to add more possible patterns to the same variant of a function or alternative of a case expression.

Disjunctive patterns are introduced with the || (double pipe) operator. The double pipe operator separates the possible variants, which can be more than two. In order for the whole pattern to be matched, at least one of its variants have to match. If none matches than the whole pattern will fail. The variants are tested from left to right and the first one which would eventually match the value is the one used for the binding of the variables. All the variants must have the same set of variables.

```
isBetweenTwoAndFive 2 || 3 || 4 || 5 = True
isBetweenTwoAndFive _ = False

hasZero :: (Int, Int) -> Bool
hasZero (0,_) || (_,0) = True
hasZero _ = False
```

The toy function $isBetweenTwoAndFive$ in the example above returns $True$ only if the value is in between 2 and 5. In this case in the disjunctive pattern, all the number from 2 to 5 are listed so that if the number is any of those ones the function will return True. $hasZero$ instead checks whether at least one of the element of a pair is equal to 0. The two variants match the two cases in which 0 is the first element or the second.

Disjunctions can also be used deeper in a pattern, for example in a data structure or in an algebraic data type constructor. The rule is that the double pipe operator has the highest precedence so it will be reduced first as long as it has valid patterns next to it. Let's see an example of its use inside a data structure.

```
countIfZero :: [(Int, Int)] -> Int
countIfZero [] = 0
countIfZero ((0,_) || (_,0) : xs) = 1 + countIfZero xs
countIfZero (_:xs) = countIfZero xs
```

$countIfZero$ given a list of pairs it counts the number of them that have at least one element equal to zero. As we can see the disjunction happens inside the infix constructor pattern ":" that builds a pair.

The corresponding Haskell version of a disjunctive pattern is very simple. Every time there is one, the current definition gets repeated once for each variant. In each repetition, the pattern is replaced by one of the variants of the initial pattern. The order in which they appear in the pattern will also be the order in which they will be replaced in the repetitions. Let's see how the functions we introduced in this section will be translated in Haskell.

```
isBetweenTwoAndFive 2 = True
isBetweenTwoAndFive 3 = True
isBetweenTwoAndFive 4 = True
isBetweenTwoAndFive 5 = True
isBetweenTwoAndFive _ = False

hasZero :: (Int, Int) -> Bool
hasZero (0,_) = True
hasZero (_,0) = True
hasZero _ = False

countIfZero :: [(Int, Int)] -> Int
countIfZero [] = 0
countIfZero ((0,_) : xs) = 1 + countIfZero xs
countIfZero ((_,0) : xs) = 1 + countIfZero xs
countIfZero (_:xs) = countIfZero xs
```

4.3 Assign pattern

The last feature added in this extension is called assign pattern (or default pattern). It's a combination of the variable and the constant patterns. Basically, it allows writing a pattern that matches any value and binds a variable to a constant decided at compile time. It consists of a variable, the := operator, and a constant. A simple example would be:

```
alwaysFive x := 5 = x
```

The function *alwaysFive*, as the name suggests, always returns 5 no matter what its argument is.

This doesn't sound much useful like this, but its main use is inside a disjunctive pattern. At the end of the last chapter, we saw how the non-exhaustiveness of a pattern it's a big problem because it is not always iden-

tified at compile time and it can raise exceptions at runtime. The default pattern can act like a catch-all last variant where it is possible to assign a certain value to the variable, in case none of the first variants matched. Thinking of it in this way can help the programmer to avoid making non-exhaustive patterns. Let's see an example of this combination.

```
data Fruit = Apple Int | Pear Int | Peach Int

weightOnlyApple :: [Fruit] -> Int
weightOnlyApple x = sum $ map singleWeight x

singleWeight (Apple x) || x := 0 = x
```

Here we have a new type called $Fruit$. Each constructor identifies one kind of fruit and its weight. Given a list of fruits, we want to sum the weight of only the apples. The $singleWeight$ function returns the weight of the fruit only in case it is an apple, otherwise, it returns 0. Notice that the body of the function is always x, but this variable gets either bound to the weight in case the first pattern matches, or it is bound to 0 in case the first doesn't match.

The implementation of this pattern consists of replacing it with a wildcard, then adding to the *where* clause of the current scope, the assignment of the constant to the variable. Below the two examples seen before, translated in Haskell.

```
alwaysFive _ = x where x = 5

data Fruit = Apple Int | Pear Int | Peach Int

weightOnlyApple :: [Fruit] -> Int
weightOnlyApple x = sum $ map singleWeight x

singleWeight (Apple x) = x
singleWeight _ = x where x = 0
```

4.4 Implementation

4.4.1 Architecture

The first proof of concept of this extension has been implemented as a preprocessor that takes as input a file containing extended Haskell source code, and it outputs normal Haskell code. The technology chosen for the implementation is Java.

The preprocessor work-flow is composed of the following phases:

- the source code is cleaned by removing all comments
- all the strings are replaced by an identifying character sequence so that they don't interfere with the parser
- look for usages of the new constructs; in case one is found it gets desugarized according to its semantics. The order in which constructs are processed is
  - Disjunctive patterns
  - Assign patterns
  - Non-linear patterns
- the original strings are reintroduced in the source code

The phases described above are executed by two type of workers: the *Parser* and the *Processors*. The *Parser* is the one that recognizes every part of the source code. It takes the source code as a plain string and it generates a *Source* object which contains all the elements of the code. The *Processors*, instead, are one for each kind of transformation that the code needs to go through. These transformations are the removal of the comments, escaping and de-escaping all the strings, and desugaring of all the new constructs. They go through the various elements of the *Source* object, and transform them accordingly.

4.4.2 Usage

Since the technology chosen for the implementation is Java, the preprocessor is contained in a Jar file. This can be executed on a JVM with the -jar argument and has only a CLI interface. The jar needs an argument which is a path to the file that has to be processed, and it returns the same file with "_output" as a suffix of the file name in case everything worked. Let's see an example of its usage.

```
$ cat source.hs
checkCoupledList [] = True
checkCoupledList (x:x:xs) = checkCoupledList xs
checkCoupledList _ = False
$ java -jar preprocessor.jar source.hs
$ cat source_output.hs
checkCoupledList [] = True
checkCoupledList (x:varPreproc0:xs) | x == varPreproc0 =
  checkCoupledList xs
checkCoupledList _ = False
$ ghci source_output.hs
GHCi, version 8.2.1: http://www.haskell.org/ghc/ :? for help
[1 of 1] Compiling Main     ( source_output.hs, interpreted )
Ok, 1 module loaded.
*Main> checkCoupledList $ concatMap (replicate 2) [1..10000]
True
```

Here we can see how smooth the whole process of using this extension is. Starting form the extended source code, we run the preprocessor on it and then we load the output directly into GHC.

Chapter 5

Conclusion and Future Work

Haskell's implementation of pattern matching is powerful and has many constructs which allow the programmer to write really expressive patterns. Although being this good it has also some limits, for example, patterns are forced to be linear and a set of patterns can be non-exhaustive. In this thesis, I described and implemented a new extension of Haskell which improves pattern matching by allowing non-linear patterns and also adding two new constructs, disjunctive patterns and assign patterns.

A non-linear pattern has the possibility to have the same variable more than once in it. The meaning of having this is that all the values that should be bound to the same variable have actually to be equal to each other, otherwise the pattern fails. Disjunctive patterns, instead, allow writing different variants of a pattern. The pattern will match only when at least one of the variants also matches. Finally, an assign pattern always matches and binds a variable to a constant decided at compile time.

These additions are defined as syntactic sugar on top of normal Haskell. This means that the implementation of the extended version can be simply done with a preprocessor that desugarize the extended source code into Haskell code. Also, any Haskell source code is also compatible with the extended version.

This extended version of Haskell already offers many new possibilities for writing more powerful patterns, but other ideas can be studied and implemented in order to do even further steps. One example that could be studied is the implementation of regular expression constructs inside patterns, like the number of times a pattern has to match inside a data structure. Also, it would be interesting to see if there can be the possibility to define what a value can't be, rather than the opposite. All these possible additions share

the common problem and difficulty of being defined as syntactic sugar over Haskell so that the implementation of the extension can be done without changing GHC.

Bibliography

[1] D. J. Farber, R. E. Griswold, and I. P. Polonsky. Snobol , a string manipulation language. *J. ACM*, 11(1):21–30, January 1964.

[2] Turchin. Recursive functions algorithmical language - refal. 1968.

[3] DA Turner. Sasl language manual, st. *Andrews University, Fife, Scotland*, 1976.

[4] Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*. Association for Computing Machinery, Inc., October 2007.

[5] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. Cduce: An xml-centric general-purpose language. *SIGPLAN Not.*, 38(9):51–63, August 2003.

[6] Simon Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[7] The GHC Team. The glorious glasgow haskell compilation system user's guide. Version 8.4.1.

[8] Neil Mitchell and Colin Runciman. Unfailing haskell: A static checker for pattern matching. In *TFP'05: The 6th Symposium on Trends in Functional Programming*, pages 313–328, 2005.