Literature survey on implementation techniques for type systems: Inductive data types and pattern matching

What are the different implementation techniques for type systems regarding inductive data types and pattern matching that have been proposed in the literature?

Pau Faraldos

Supervisor(s): Jesper Cockx, Bohdan Liesnikov

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Pau Faraldos
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Bohdan Liesnikov, Annibale Panichella

Abstract

Data types and pattern matching are fundamental concepts in programming. Data types define the structure of data, while pattern matching allows efficient manipulation and extraction of the same data. This text provides an overview of different implementation techniques for type systems regarding data types and pattern matching in the existing literature. Data types considered include inductive, coinductive, and mutually inductive, while the main pattern-matching methods considered are decision trees, backtracking finite state automata, and term decomposition. Though approaches for implementation techniques of data types can be compared more objectively, separate approaches for pattern matching have different benefits and drawbacks, thus, a one-fits-all technique does not exist.

1 Introduction

Type systems serve as an essential tool for programmers to identify various errors during the compilation process. They also provide additional advantages, such as facilitating refactoring operations and making the code self-documenting, enabling understanding through reading alone.

Literature regarding how practical type systems for commonly used languages work often contradicts one another, especially on the topic of pattern matching and data types. Additionally, there is a shortage of up-to-date, relevant sources on the topic. This adds a level of complexity for prospective research to be done in the area, as more time is needed to research these discrepancies. This becomes even more complex for languages with many features that data types in a language support, as this can rapidly complicate the type-checking step of compiling a program. Thus, investigating this subject and documenting the most important points of existing literature would simplify subsequent research into the area. As such, this paper would serve as a basis for relevant knowledge in the area, so that language implementers or researchers can base their further work on it.

Some of the most relevant existing work includes *Compiling pattern matching* [4], which is one of the foundational publications in the field and served as a basis for subsequent research. Many subsequent papers have drawn on the insights presented in this paper to provide extensions, including *Compiling pattern matching to good decision trees* [14], *A Term Pattern-Match Compiler Inspired by Finite Automata Theory* [18] and *Compiling Pattern Matching by Term Decomposition* [19].

The main research question that will be focused on in this paper is *What are the different implementation techniques for type systems regarding inductive data types and pattern matching that have been proposed in the literature?* In order to know how to approach this question, a small set of sub-questions has been posed, which, when all are considered, should give a good basis for a concrete answer for the main focus of the paper. The sub-questions can be seen below:

- Along which dimensions can existing implementation techniques be grouped and compared with each other? What are the advantages and disadvantages of these techniques?
- What criteria can we establish for implementers of new languages or prospective researchers to consider when adding support for inductive data types and pattern matching?

The structure of the paper is as follows. Section 2 presents the method that was used to conduct the research and an overview of how strategies were categorized. Following this, section 3 reflects on ethical considerations that were taken into account throughout the process. In section 4, the implementation techniques and optimizations for both data types and pattern matching are explained. Section 5 focuses on comparing the aforementioned approaches, as well as presenting some advice for future researchers and language implementers. Finally, section 6 provides a summary of the most important findings, accompanied by concluding remarks and future work to be done in the field.

2 Methodology

In order to achieve the goals outlined in the introduction, the research process was broken down into smaller, manageable tasks that are easily measurable and provide guidance for others can conduct the research and obtain similar findings.

The first step was finding a small number of relevant research papers on the topic, namely inductive data types and pattern matching. These were found through a round of preliminary research that did not intend to delve deep into the topic but rather to provide an overview of what research in the area looked like. As these sources were cited and were cited by a large number of other sources, it facilitated the discovery of additional relevant papers.

Often, papers that discuss implementation techniques include a *Related work* section, which explores similar papers consisting of other techniques, providing more background and a theoretical perspective on important elements of the technique, or presenting extensions of the discussed techniques. Additionally, the *References* or *Bibliography* section of papers offers a useful resource for exploring further sources. Additionally, systems such as ConnectedPapers [1], ResearchGate [2] and Google Scholar [3] can be valuable to identify related papers based on citation patterns.

After discovering and understanding a large collection of papers that discuss implementation techniques for inductive data types and pattern matching, the next step involved finding a way to group these papers so that the results can be discussed in a meaningful way. One natural approach consisted of grouping the techniques by their code generation strategy. The major strategies for pattern matching in non-dependently typed languages are decision tree approaches, and backtracking finite state automata. Additionally, there are hybrid techniques combining both approaches and further techniques for

[1] https://www.connectedpapers.com/

[2] https://www.researchgate.net/search

[3] https://scholar.google.com/

languages with specific evaluation methods (e.g. term decomposition for lazy languages).

Once strategies have been organized into the major groups discussed above, it is important to have a method for comparing these. A logical approach would be to consider the features that the type system using this technique would support. For example, whether decision tree approaches support inductive data types in the language. Other considerations include the way the language is typed (strongly vs. weakly and statically vs. dynamically), and other features that the language may have that can affect how pattern matching functions, such as lazy evaluation vs. eager evaluation.

3 Responsible Research

It is essential to study the ethical implications that a study may have, while also ensuring that the steps taken are properly documented to ensure reproducibility.

The ethical considerations of this paper have been considered extensively throughout the research process. Even though this paper focuses on conducting an extensive survey of existing literature in the field and does not involve humans or personal data, it is important to acknowledge the steps through which papers were sourced, how information was extracted, and to provide correct acknowledgments through citations. Furthermore, potential conflicts of interest and biases were carefully recognized and reduced through appropriate steps (such as adopting a systematic approach to source and analyze publications, as outlined in the Methodology, avoiding cherry-picking of studies and choosing a wide variety of sources, as well as reviewing by peers and supervisors) to ensure the research remains as objective as possible.

Additionally, reproducibility is necessary for responsible research. To ensure that the results discussed in the paper can be reproduced by other researchers, all sources used in the survey have been appropriately documented in the *References* section. This was done with the aim of simplifying the process of verifying the findings presented in the paper, as well as facilitating further research in the field. The *Methodology* section discusses in depth the steps that were taken to conduct the research and how the results were found, which is especially helpful for other researchers aiming to verify these findings or to expand on those presented in the paper.

4 Implementation Techniques

As most interesting use-cases of pattern matching consist of being applied to data types, the first topic that will be discussed will be how data types can be represented by the compiler, in subsection 4.1. After having an understanding of how a data type is represented by a programming environment, pattern matching is discussed in subsection 4.2.

4.1 Data Types

Inductive Data Types

An inductive data type is a data type that is defined in terms of itself, allowing recursive structures such as lists and trees. These types are essential tools in functional programming languages, and they also allow programmers to think of recursion in a more natural way. They also allow inductive mathematical proofs to be reasoned about through code (as can be seen in the programming languages of Coq [1], Agda [3], and Lean [5], among many others).

Fundamentally, an inductive data type consists of zero or more base cases, and zero or more inductive cases that refer to a smaller instance of the same type, each of which is represented through a constructor in a programming language.

```
data InductiveType a =
    Base
    | Rec ...
    ...
    | RecN ...
```

Listing 1: Basic representation of an inductive type in Haskell

As can be seen in Listing 1, an inductive data type consists of at least one base case, representing the most simple form of the data type, as well as multiple recursive constructors, where the ellipsis should represent the structure of the recursion.

Though the code is specific to Haskell [2], the idea remains the same for other programming languages that support defining data types recursively, bearing syntax differences.

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

Listing 2: Inductive data type representing a list defined in Haskell

A concrete example of a simple inductive data type can be seen in Listing 2, which describes a singly linked list in the language of Haskell. The most simple form of a list is an empty list, represented by the **Nil** constructor which serves as the base case. Making use of the base case is the Cons constructor, which is defined as having a value of type **a** (a value of the list), as well as a **List** (the tail of the singly linked list), making use of its own definition.

Though the examples above have described how inductive data types can be used by a programmer, it is essential to understand how these types are represented by a compiler and how they are optimized.

Naively, every inductive data type (e.g. **List** in Listing 2) can be represented as a tagged pointer, in which case the tag is used to differentiate between the cases (in this case **Nil** or **Cons**). For base cases that don't store any data, their pointer will be null, usually represented by '0'. On the other hand, the pointer of inductive cases points to an **n-tuple**, where **n** is the number of arguments that the constructor takes (in the case of **Cons** it is 2, one for **a** and one for **List a**) [6].

Though this approach works for representing data types, it can be inefficient, as it would be preferred if rather than storing two pointers for each constructor, we could only store one ('0' in the base case, and the **n-tuple** in the recursive case).

Programming languages use the terms **boxed** and **unboxed** to describe how values are represented in memory. Unboxed

types refer to values that are directly stored in memory (such as the stack frame or registers). These types allow for improved performance as it skips one layer of memory indirection by not having to first find the corresponding data in the heap, however, the compiler must know the size and representation of these types at compile-time, so these are usually used for small primitive types. On the other hand, boxed structures refer to a pointer that points to a place in memory where the data is stored (i.e. in the heap). Boxed types allow for more flexibility in terms of copying these values (as only the pointer is copied), however, it can also lead to performance problems. These types are typically used when the value can have varying sizes, such as is the case for inductive data types [6]. The terms **unboxed** and **boxed** are commonly used in Java (and many other programming languages including C#), where unboxed types refer to primitive values directly, while boxed types are primitive values wrapped in an object type.

Compilers make use of a property **always-boxed**, for further optimizing data types, which is true if all cases of a type are boxed. To give some examples, primitive types such as booleans and characters are not always-boxed (since they are unboxed). The List described in Listing 2 is not always-boxed, as the base case **Nil** is represented by null (in many languages, such as C, this is represented as a pointer to **0**).

Certain optimizations can be made to how data types are represented given the always-boxed property is computed for all cases. Constants (such as Nil in Listing 2) can be represented by an unboxed integer, where different numbers are needed in case there are multiple base cases, which is an improvement over having a tag and a null pointer. For constructors that contain always-boxed data, the tag can be removed and the n-tuple can be represented immediately. However, there are some cases in which this can introduce uncertainty, such as when multiple recursive constructors have the same arguments, in which case, the previous representation of tagged types can be used. This is a standard optimization in the compilers of many commonly used functional languages (including Haskell and ML), which can remove many cases of memory indirection and is not overly complicated.

Coinductive Data Types

Coinductive data types add support to non-terminating, infinite data structures, which can be useful for modeling infinite lists when lazy evaluation is supported.

An example of a coinductive data type is shown in Listing 3, which together with a function that recursively calls itself to add new elements to the stream, could result in a non-terminating list of items of type 'a'.

```
data Stream a = Cons a (Stream a)
```

Listing 3: Coinductive data type representing an infinite stream in Haskell

The largest difference between coinductive data types and inductive data types is that the former can have infinite paths, whereas simple inductive data types are always finite. Many functional languages (such as Standard ML) represent constructors as functions that construct values of the corresponding data types. As such, this approach does not directly support infinite structures or coinductive data types, unless the language uses lazy evaluation, like in the case of Haskell [10]. Nonetheless, there are some techniques for simulating the intended behavior, such as through recursive functions. Regardless, this might not be as straightforward, or flexible as a solution that more modern functional languages use. In OCaml, constructors are not directly represented as functions, rather, the constructors can specify different cases or states of a potentially infinite process [12].

Because of the many similarities between inductive and coinductive data types, it makes sense for them to have a shared framework that supports using constructors and destructors and allows for the use of pattern matching. Jeannin et al. give a thorough explanation of how this can be achieved, while also giving a pseudo-implementation for further aid in their paper [11].

This is a good place to consider the differences between languages that use lazy evaluation compared to eager techniques. In eager evaluation, expressions are computed as soon as they are encountered, and function arguments are evaluated before the function is applied. This has the benefit of being more predictable, however, it can also lead to inefficiencies due to unnecessary computation (e.g. computing all arguments when only one is used, or evaluating an entire list when only the first element of the list is needed). Conversely, in lazy evaluation, expressions are not evaluated until a step of evaluation has been reached that needs a value from the expression. Some benefits of this technique include avoiding unnecessary computation and handling potentially infinite structures, such as coinductive data types. On the other hand, drawbacks include the introduction of overhead from storing the memoized values, as well as less predictable performance as the evaluation of an expression might not be where one expects it to take place.

It is important to note that coinductive data types make sense to be used in combination with lazy evaluation, as this is the most natural and efficient approach, but coinductive data types can still be used in languages that feature eager evaluation. An example of this is OCaml [13], which is an eager language with support for lazy evaluation either through explicit use of memoization and delayed evaluation or by using the lazy module [17].

Mutually Inductive Data Types

Mutually inductive data types are data types that depend on each other's definitions. This can be understood as having a circular dependency between the data types and is widely supported by functional languages such as Haskell, Coq, Standard ML, and OCaml.

An example of two mutually inductive data types can be seen in Listing 4. It introduces a pair of types for even and odd numbers, where each definition refers to its counterpart.

```
data Even = EvenZero | EvenSucc Odd
data Odd = OddSucc Even
```

Listing 4: Mutually inductive data types representing odds and evens in Haskell

Despite this seeming relatively simple, and not largely different than normal inductive data types, there are some important considerations when considering mutual data types in a programming language. Firstly, the compiler must run an algorithm that analyzes the dependencies of the types to determine the order in which the types should be processed, as otherwise, it might be impossible due to mutual dependencies. Following this, depending on how the programming language handles scope and forward declaration, forward declarations or type annotations might be necessary for the compiler to know the basic structure of types before their definitions are reached.

Many functional languages make use of specific keywords to distinguish between mutually inductive data types and simply inductive data types to aid these considerations. For example, in Standard ML, the **and** keyword is used.

```
datatype 'a tree = Empty
    | Node of 'a * 'a forest
and 'a forest = None
    | Tree of 'a tree * 'a forest
```

Listing 5: Mutually inductive data type representing trees and forests in Standard ML

The example in Listing 5 comes from Harper (2011) and describes the definition of trees and forests in a mutually recursive manner [9].

On the other hand, Haskell natively supports mutually inductive data types through the **data** keyword, due to the fact that Haskell's type inference system is powerful enough to not require forward declarations, thus eliminating the need for explicit keywords for defining mutually inductive types.

4.2 Pattern Matching

Pattern matching is a technique, mostly used in functional programming languages, that allow the programmer to match the structure of a value against a pattern (also known as destructing a value), and based on which patterns match the value at hand, a different block of code can be executed.

Patterns can take many forms, but the subset that will be explored in this paper, and that can be used for most applications consist of:

- Literal values (e.g. integers, booleans)
- Identifiers, which act as placeholders by capturing any value and binding it to the identifier
- Wildcards, which also capture any value but don't bind it to any identifier
- Constructors, which are used to destructure and identify the case of the data type that matches the value

Listing 6 gives examples for each of these patterns in pseudo-code.

```
0 | 1 | "hello" | true := Literals

identifier := Variable capture

_ := Wildcard

(Cons head tail) := Constructor
```

Listing 6: Basic Pattern Examples

Many programming languages also add additional features to pattern matching, including or-patterns and boolean guards. Or-patterns provide a way for multiple patterns and corresponding blocks of code to be grouped into a single pattern, reducing repetitive code. Boolean guards are additional conditions that must hold true in addition to the pattern for this match to be successful.

There are two main approaches for how a compiler can represent pattern matching based on the code generation strategy. These techniques are decision trees and backtracking finite state automata, which will be discussed in the following two subsections. Following this will be a discussion of other techniques that can be used for compiling pattern-matching, including approaches that build off on top of decision trees and backtracking finite state automata.

Decision Trees

A decision tree is a data structure that represents a sequence of decisions (represented by the path to take at a specified node), and their possible consequences (represented by their leaves). These structures are a good fit for evaluating pattern matching, as they provide an efficient way to test conditions (e.g. constructors or literal values) against an input value. This structure is a natural approach for compiling pattern-matching as it follows the branching nature of pattern-matching expressions.

The root node of the decision tree corresponds to the starting point in the execution of the pattern-matching expression. It will then have the first pattern/condition to be checked against. Internal nodes in the tree represent further patterns or conditions to check against the input value. These tests can take many forms, including checking against a literal value, checking that constructors match, and that boolean guards are satisfied. Based on the results of these tests, different child nodes will be followed. Finally, leaf nodes represent the end of pattern matching, and, when the entire input value has been matched correctly, indicates the associated block of code to continue execution from.

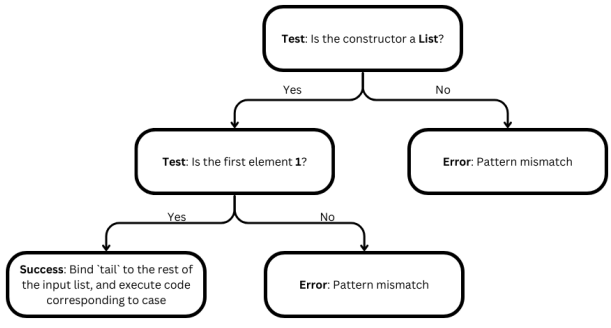

Figure 1: Decision tree example representing pattern: Cons 1 tail

Figure 1 shows a basic example of a pattern ('Cons 1 tail')

being represented by a decision tree, as described by the approach above. Though this is a simple pattern where the order of the tests doesn't have a large impact, the illustration can help clarify how the order matters, as an adequate ordering can reduce the number of tests that must be executed for a majority of cases.

It is important to note that although using decision trees is a simple strategy to represent pattern-matching, there is a major limitation in that the order of the tests is essential for the performance. Baudinet and MacQueen (1985) proposed the goal that compilers should have when it comes to choosing the order of pattern matching tests, named the **Dispatching problem**. It is described as follows: "Given a sequence of patterns $p_1, ..., p_n$ of type τ, find out in which order the subterms of any possible subterm t of type $\tau'(\tau' \leq \tau)$ have to be examined to determine with the minimum number of tests on the subterm s of t, which pattern $p_i(1 \leq i \leq n)$ is the solution to the matching problem defined by $p_1, ..., p_n$ and t." [16]. A proof resulting from reduction from the pruned O-trie space minimization problem described in Comer's and Sethi's papers [7, 8] shows that this problem is NP-complete. To create the most optimal decision tree, at each stage of constructing the data structure, the best test node must be selected. The best test node is one that created a balanced split, meaning that the test node divides the results into approximately equally-sized subsets. This search corresponds to an exponential explosion, so some heuristics for building "good" decision trees must be considered. The original paper by MacQueen and Baudinet (1985) that describes how to construct decision trees for pattern matching considers three heuristics: relevance, branching factor, and arity factor. The relevance heuristic is based on prioritizing the most informative attributes, determined by selecting the patterns with the highest predictive power distinguishing between different patterns. The branching factor heuristic attempts to minimize the number of branches at each decision point, striving to result in shallower trees that require fewer comparisons. Finally, the arity factor heuristic is based on prioritizing constructors with a low arity, as they might require fewer comparisons for arguments. There are many other heuristics that are used in other papers and that are used in practice by compilers, many of which are explained in Scott and Ramsey's report [20].

Though these heuristics worked well enough to be incorporated into an ML compiler at the time, further work has been done in this field to improve the heuristics, as well as to support newer features, such as lazy pattern-matching. Le Fessant and Maranget (2001) discuss new heuristics using maximal sharing (a technique to minimize redundant storage and sharing common substructures among branches of the decision tree) and column heuristics, as well as their performance, which matches the performance of an optimizing compiler to backtracking automata, while being able to perform better in some specific cases [14].

Other optimizations that can be used on decision trees include redundancy elimination, which is the process of discovering operations and statements that are redundant. For example, there might be two patterns that represent the exact same values (imagine one having a variable captured vs. one with an anonymous variable), in which case one can be removed, reducing the tree size and simplifying its structure. There are also cases where patterns can be eliminated due to them being unreachable, either because a similar pattern already exists, or because the information from the type system can be used to find that certain patterns can never match. For example, we know that if a value 'x' is of type List as described in Listing 2, we know that 'x' can never match against '1', so this case could be removed. Similar optimizations can be incorporated depending on how the language is typed, with statistically, strongly typed languages supporting more due to knowing more about the types of values at compile-time than dynamically, weakly typed languages. Similarly, dependently typed languages can be useful in further eliminating redundant or unreachable patterns from the case tree.

Backtracking Finite State Automata

A Finite State Automaton (FSA) is a basic computational model which consists of states and transitions and can be used to model how patterns are evaluated while evaluating a pattern-matching expression. Though there are multiple ways in which a pattern-matching expression is transformed into FSAs, the most basic approach consists of transforming each pattern into its own FSA, and then traversing these automata in order in order to see when an accepting state is reached. Once an accepting state is reached, this would correspond to a pattern being a match and would allow the compiler to continue executing the code in the corresponding block. Other approaches include composing multiple FSAs representing multiple patterns into a larger individual FSA that can be used for more patterns. This has the added benefits of not having to keep track of as many FSAs, as well as being able to have shared states and transitions, optimizing the runtime performance [4].

Patterns are transformed into FSAs in a very similar manner to how patterns were transformed into decision trees, as discussed in the previous section. This technique treats patterns as a regular expression of atomic values, constructors, and variables.

Pettersson (1992) describes the algorithm to generate the automata corresponding to a pattern-matching expression, consisting of four steps. The first consists of renaming patterns to include the path going through the pattern, which is necessary for the third step. After this, the patterns and finite states must be mapped to a Deterministic Finite Automaton (DFA), which corresponds to each state representing a test (such as checking the equality of a literal value, and constructors matching), while the transitions represent the outcomes that the test can have. If the set of constructors and literals is exhaustive (assuming the type system can determine this), the DFA is completed. Otherwise, a default state representing failure is added. Following this, the third step is optimizing the automaton, by merging equivalent states. Final states are considered to be equivalent if they correspond to the same block of code, while test states are equivalent if the same path variable is being tested, and the same outgoing edges exist. The fourth and final step corresponds to creating intermediate code representing the automaton so that the compiler can efficiently traverse the automaton and find the

correct next block of evaluation [18].

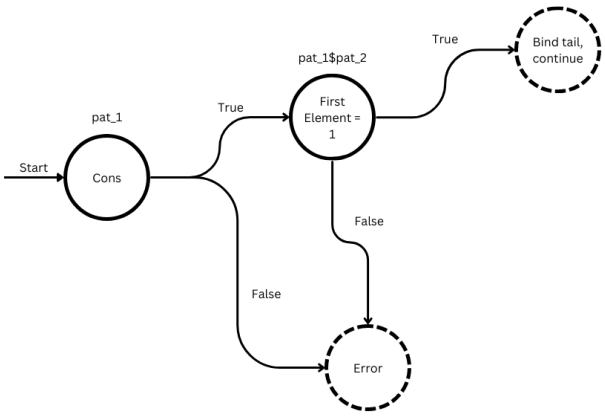

Figure 2: FSA example representing pattern: Cons 1 tail

Figure 2 shows a basic example of how a pattern ('Cons 1 tail') can be represented by a finite state automaton following the approach described above. It is clear that the FSA will be traversed starting at the initial state (Cons), represented by having an incoming "Start" transition. Final states are represented by having a dashed border, namely "Error" and "Bind tail", representing a failure at matching and a successful matching respectively. The example also demonstrates how patterns are renamed to be augmented with their path variable, which can provide additional information about the path taken to reach a state.

This technique can be extended to consider other features previously discussed, such as boolean guards and or-patterns without much complication. Boolean guards can be considered new states with the corresponding outgoing edges (failure, success). Since in FSAs, transitions are triggered only through the input value, and in this case, patterns are being treated as a regular expression of atomic values, constructors, and variables, boolean guards can be appended to this representation of a pattern through a regular expression, corresponding to the final input symbol.

Other Strategies

There are other code-generation strategies that can be used for compiling pattern-matching, though they are not standalone techniques, and instead are built on top of the techniques described previously.

One of these strategies is **term decomposition**, which is particularly suited for languages that employ lazy evaluation, while also providing support for handling complex data structures (such as coinductive data types). Rather than being a standalone technique, it is integrated into other pattern-matching techniques. This technique involves recursively decomposing the term that is pattern-matched against, as well as the patterns, comparing their sub-structures, and ensuring they match. In the case that the outermost layer of the term being matched against is a constant, the technique verifies whether this constant and the corresponding sub-pattern are equal. If this is the case, the remaining sub-terms are checked. On the other hand, if the outermost layer of a term is a constructor, a check is made to ensure that this constructor matches the constructors in the sub-patterns, which is usually done by comparing the constructor names and the number of arguments they have. If the constructors match, all of the arguments are recursively checked through this approach. The most noteworthy benefit of employing term decomposition in a lazy language is that the value being pattern-matched against might not have to be evaluated fully. Instead, it can be partially evaluated to avoid unnecessary computation while still accurately determining which patterns match [19].

```
term_decomposition (Cons 1 tail) = A
term_decomposition _ = ...
```

Listing 7: Compiling Pattern Matching through Term Decomposition in Haskell

Listing 7 demonstrates an example function in Haskell with a simple pattern 'Cons 1 tail'. If this function was called with a variable with a value of '(Cons 1 (Cons 2 (Cons 3 Nil)))' in a language with lazy evaluation, first, the outermost layer of the term and the pattern would be checked to match. Since both the constructor names ('Cons') and the number of arguments match, term decomposition would be applied recursively to each argument. The first argument of the term would then be evaluated, resulting in '1', which would match the sub-pattern. As for the second argument, since it is a variable capture ('tail'), the term does not need to be evaluated further, and hence the compiler can get away without needing to evaluate the term fully. Even though this example presented an example with three elements, it is easy to see how this might lead to significantly better performance for especially large lists (and is necessary for infinite lists).

Many other techniques exist (e.g. compiling constant patterns with hash tables [15]), however, they are mainly incorporated by compilers for specific optimizations based on heuristics, and as such won't be discussed further in the paper.

5 Discussion and Advice

Before discussing specific advice regarding the implementation techniques discussed in the previous section, it is essential that programming language implementers have considered and thoroughly analyzed the contexts in which their languages should be used. Some programming languages must be incredibly performant at run-time due to being used in latency-critical systems, while in others, run-time performance does not matter much, but instead, the speed of compilation is essential (for example to allow programmers to iterate quickly). For some other languages, avoiding failures at run-time is essential (e.g. medical equipment, airplane controllers), while other systems can get away with this due to not causing any real, large consequences. The platforms in which programming languages can be used (and whether they are platform-independent or not) can also influence decisions when it comes to designing a language, as the programming language used to write the compiler must also fit these characteristics.

Once a clear idea of the use cases that a programming language should be developed for, it is important to consider which features the language should contain originally, and how the language might grow in the future. The latter is especially crucial to avoid having to make unnecessary large changes to the compiler and surrounding environment. As a rule of thumb, it is not always better to incorporate more features because "they might be useful for a few users", as this can further complicate adding more relevant features subsequently, might lead to performance issues, or might make the language more confusing to users.

5.1 Data Types

There are many considerations that programming language implementers should be familiar with when it comes to the technical details of implementing complex data types.

Knowing the difference between boxed and unboxed types is essential when it comes to implementing efficient data types. It is essential to also understand when the language should use boxed vs. unboxed types. Even though unboxed types should fit in the stack memory, the specific maximum size a value can have in the stack depends on the operating system and hardware of the target user, unless the code is always executed through a virtual machine (such as Java and Scala). Unboxed types offer better performance as the associated value won't need to be fetched from the heap memory, however, boxed types use pointers to the data stored in the heap. This has the benefit of being able to create more efficient copies, which might be an important consideration for the language. Thus, it is essential to create a clear distinction between which types should be boxed vs. unboxed, considering size and variability.

The most basic representation for data types, tagged pointers is a valid implementation that can be easily extended even for more complex types, however, it can also be inefficient due to storing a tag and an additional pointer for each constructor. A commonly used optimization includes leveraging the "always-boxed" property, which can decrease memory indirection and usage, with the drawback being additional complexity in the compiler. This is another case where it is not always better to make the compiler more efficient blindly, as this should be dependent on the requirements of the language. If run-time performance is important, it is a good idea to apply the optimizations that are discussed in detail in subsection 4.1: Inductive data types.

Though basic inductive data types are an essential feature that multi-purpose functional programming languages should support, more complex inductive data types, such as coinductive data types and mutually inductive data types might not be applicable to the language being designed, so worrying about their implementation might be unnecessary.

Coinductive data types support infinite paths and can be used to model cyclic recursion, and are especially useful when talking in the context of languages with lazy evaluation. Choosing the best evaluation strategy for a language should mainly be depending on the requirements, however, other differences include that eager evaluation computes expressions whenever they are encountered, while lazy evaluation defers computation until the value is needed. Eager evaluation can be more predictable, however, lazy evaluation can be more efficient in run-time if an efficient implementation of memoization is considered and some variables are never used. If the language supports lazy evaluation and should support coinductive data types, the paper by Jeannin et al. provides a detailed explanation and pseudo-implementation that can be helpful for incorporating a shared framework for inductive and coinductive data types [12]. In the case that eager evaluation is used but coinductive data types are still a useful feature in the language, inspiration from OCaml, which incorporates it through explicit memoization or dedicated modules can be a good starting point.

Introducing mutually inductive data types can be relatively simple if the language features a powerful type inference system, as it may allow dependencies to be analyzed without needing forward declarations. This is how Haskell handles mutually inductive data types, which consequently allow them to be defined like standard **data** definitions. On the other hand, functional languages with less expressive type systems need to consider the order of dependencies and processing order for these data types, as well as a specified syntax for the compiler to understand the basic structure of the mutual types.

5.2 Pattern Matching

The two main techniques that programming languages use to compile pattern matching, decision trees and backtracking finite state automata, can both deal with basic patterns that were described in subsection 4.2, however, there are certain advantages and disadvantages to each of these approaches, which can be minimized with further optimizations.

For patterns with a small number of tests and a small number of possible branches at each test node, decision trees are the best solution. They excel when patterns follow a clear hierarchical structure as they never test a subterm more than once, whereas backtracking finite state automata are more suitable for patterns that have non-linear and complex conditions, as they provide more expressive power due to their backtracking nature.

Regarding performance, decision trees can provide more efficient run-time performance when the order of tests is carefully optimized, however, as this is an NP-Complete problem, heuristics, such as those provided by Baudinet and MacQueen (1985) [16] and Maranget (2001) [14], are the best way to create a good order of tests. This makes decision trees (with these optimization heuristics) a better approach for cases when run-time performance is critical. Backtracking FSAs can have slower performance due to the overhead corresponding to keeping track of the state of the automaton.

It is also noteworthy to point out that if the optimization heuristics for decision trees are not applied, there is a possibility of code size explosion (in an exponential form), whereas backtracking FSAs guarantee code that grows in size linearly with respect to the patterns. Code size explosion can lead to many drawbacks, including longer compilation times, and limitations in resource-limited environments (such as in embedded systems, where typically not much storage and memory is available).

Furthermore, some extensions of basic patterns (such as or-patterns and boolean guards) can be implemented more naturally through FSAs, so they may be more appropriate when performance is not the primary consideration, but rather being able to deal with complex patterns without considerable modifications.

As can be seen, different techniques are better for some cases, so, despite using only one of these techniques is sufficient for a working compiler, many programming languages add further optimizations for speeding up compilation and runtime performance. A large number of these optimizations are based on other techniques not discussed in this paper (such as using hash tables when all patterns are strings, as done in OCaml, for example [15]). There is also the possibility of using hybrid approaches (both decision trees and backtracking finite state automata) depending on the patterns at hand. This way, the benefits of both techniques may be appreciated, with the significant drawbacks being an overhead for choosing which technique to use, as well as a much more involved codebase to allow this flexibility. As such, unless there is a clear reason for the programming language to need one of the advantages, or to avoid one of the disadvantages, combining approaches might not be the best choice.

Another strategy that is based on either of the two techniques previously described is term decomposition, which is particularly useful in languages that use lazy evaluation. This is due to the ability to allow only portions of terms that are necessary to be evaluated, meaning that if some sub-patterns are not needed to determine a match, they don't need to be evaluated, potentially saving computation time and resources. Term decomposition is also especially helpful in the context of coinductive data types, as it allows for pattern matching on infinite data structures without needing to fully evaluate the term first.

6 Conclusions and Future Work

To conclude, the question that was treated throughout the paper was *What are the different implementation techniques for type systems regarding inductive data types and pattern matching that have been proposed in the literature?*, with sub-questions dealing with finding advantages and disadvantages of these implementation techniques, as well as criteria that can be established for implementers of new languages to consider when considering adding these features. Throughout the investigation of inductive data types, certain optimizations were discovered to reduce memory indirection when considering tagged pointers. Coinductive data types and mutually inductive data types were introduced, as well as approaches for their compilation. In terms of pattern matching, a basic framework describing basic patterns was first introduced, before discussing concrete implementation techniques. The two major techniques discussed are decision trees, which can be crafted simply but requires further optimizations to aid the run-time and compile-time performance, as well as backtracking finite state automata, which is a more involved approach but also provides more flexibility regarding the patterns and complex types that are supported. Following this is a short discussion into term decomposition, a strategy that builds off on top of other pattern-matching techniques to exploit optimizations possible in languages that use lazy evaluation.

Future work in this area includes a deep dive into commonly used programming languages in order to understand exactly how a modern language makes use of the techniques discussed in this paper for a mass-use, production-ready language. It can also be relevant to discuss how multiple of the techniques described in the paper can be combined to boast the most benefits while minimizing the drawbacks. Furthermore, investigating how pattern matching can be extended to handle more complex data types (such as dependent types and polymorphic types) can be an interesting area of further research.

Acknowledgements

I would like to express my deepest gratitude to my responsible professor, Jesper Cockx, as well as my supervisor, Bohdan Liesnikov, as this endeavor would not have been possible without their help along the way.

References

[1] *Introduction and Contents - Coq Documentation*. https://coq.inria.fr/refman/.

[2] *Haskell 2010 Language Report*, 2010. https://www.haskell.org/documentation/.

[3] *Agda User Manual*, 2023. https://agda.readthedocs.io/en/v2.6.3/.

[4] Lennart Augustsson. Compiling pattern matching. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 368–381, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[5] Jeremy Avigad, Gabriel Ebner, and Sebastian Ullrich. *The Lean Reference Manual*, 2018. https://leanprover.github.io/reference/.

[6] Luca Cardelli. Compiling a functional language. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, page 208–217, New York, NY, USA, 1984. Association for Computing Machinery.

[7] Douglas Comer. Heuristics for trie index minimization. *ACM Trans. Database Syst.*, 4(3):383–395, sep 1979.

[8] Douglas Comer and Ravi Sethi. The complexity of trie index construction. *J. ACM*, 24(3):428–440, jul 1977.

[9] Robert Harper. Programming in standard ml. 07 2011.

[10] Graham Hutton. *Programming in Haskell*, pages 212 – 216. Cambridge University Press, USA, 2nd edition, 2016.

[11] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. Language constructs for non-well-founded computation. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 61–80, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[12] Jean-Baptiste Jeannin, Dexter Kozen, Alexandra Silva, David Baelde, Arnaud Carayol, Ralph Matthes, and Igor Walukiewicz. Cocaml: Functional programming with regular coinductive types. *Fundam. Inf.*, 150(3–4):347–377, jan 2017.

[13] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, KC Sivaramakrishnan, and Jérôme Vouillon. *The OCaml system*, 2022. https://v2.ocaml.org/docs/.

[14] Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, page 35–46, New York, NY, USA, 2008. Association for Computing Machinery.

[15] Luc Maranget. [PATCH] Optimization of pattern-matching on strings. GitHub, 2013. Issue #6269.

[16] Marianne Baudinet and David MacQueen. Tree pattern matching in ML, 1985.

[17] Keiko Nakata. Lazy mixin modules and disciplined effects, 2009.

[18] Mikael Pettersson. A term pattern-match compiler inspired by finite automata theory. In Uwe Kastens and Peter Pfahler, editors, *Compiler Construction*, pages 258–270, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[19] Laurence Puel and Ascander Suarez. Compiling pattern matching by term decomposition. *Journal of Symbolic Computation*, 15(1):1–26, 1993.

[20] Kevin Scott and Norman Ramsey. When do match-compilation heuristics matter? Technical report, USA, 2000.