Pattern Matching in Encore

Gustav Lundin
Abstract

Pattern Matching in Encore

Gustav Lundin

Encore is a young object-oriented programming language. The standard object-oriented model which embraces encapsulation works well for many kinds of problems but not all; it does not perform well when the algorithm inherently demands many objects to be decomposed at once. To address the need for an elegant way to write programs that solve such problems we have extended Encore with a pattern matching construct capable of decomposing objects. This thesis recounts how we designed and implemented pattern matching in Encore and presents some examples of how it can be used. We found that for problems that are not well suited for fully encapsulated objects, pattern matching leads to much more concise and clean code. However, this comes at the cost of additional run-time.
Acknowledgments

I would like to thank my supervisor, Elias Castegren, for all the help and feedback that he has provided me during this project.
Contents

1 Introduction 2
  1.1 Project Goal ........................................... 2
  1.2 Outline ............................................... 2

2 Background 2
  2.1 Encore .................................................. 2
  2.2 Pattern Matching in Functional Languages ............... 5
    2.2.1 Algebraic Datatypes ................................ 5
    2.2.2 Pattern Matching ................................... 5
  2.3 Pattern Matching in Object Oriented Languages ........ 7
    2.3.1 Algebraic Datatypes and Objects ................... 7
    2.3.2 Dynamic Dispatch .................................. 8
    2.3.3 Related Work ...................................... 9

3 Design 11
  3.1 Suggested Syntax ...................................... 12
  3.2 Active Objects ....................................... 12

4 Implementation 14
  4.1 Parsing ................................................ 14
  4.2 Typechecking .......................................... 15
  4.3 Code Generation ....................................... 17

5 Results and Discussion 19
  5.1 Expressive Power ..................................... 19
    5.1.1 Tree Rotations ................................... 19
    5.1.2 Algebraic Simplification ......................... 19
  5.2 Performance ........................................... 23
    5.2.1 Benchmarks ...................................... 23
    5.2.2 Discussion ...................................... 23

6 Conclusion and Future Work 24

7 References 25

A Tuples 27

B The Algebra Code 27
1 Introduction

A programming language that offers a high level of abstraction allows the programmer to focus on solving the actual problem at hand. This generally leads quicker development and code that is easier to understand and maintain. However, these benefits may come at the cost of an increased run-time, but very often this trade off is worth it.

Pattern matching is a programming language feature that allows the programmer to branch both on the value of data as well as its structure while at the same time binding sub-structures to local names. In some sense it is a generalization of the switch statement found in most imperative programming languages. In the functional family of programming languages, pattern matching is common and is known for leading to very concise code in many problem domains.

1.1 Project Goal

Encore (Brandauer et al., 2015) is a young, object-oriented programming language that is being developed by the programming language group at Uppsala University. The goal of this thesis project has been to design and implement a pattern matching construct (figure 1) for Encore. The new construct should be powerful enough to be able to decompose objects and match them against custom patterns.

1.2 Outline

The structure of this thesis is as follows: We will first introduce Encore a little more thoroughly, then we will describe pattern matching as it appears in other languages, and reflect over if and how pattern matching fits into an object-oriented context. After that we will cover the design and implementation of the pattern matching construct in Encore. Finally, we will tie back to the introduction and showcase some Encore programs where pattern matching have lead to increased conciseness and readability. We will also benchmark these programs to better understand the run-time cost of pattern matching in Encore.

2 Background

2.1 Encore

Often the easiest way to make a computer faster, is to add more computing cores to its processor. However, in order for a program to benefit from the extra parallel hardware it has to be designed with parallelism in mind. Most programming...
languages were designed for writing sequential programs. Creating correct and scalable parallel programs in these languages can be tricky.

Encore is an object-oriented, general purpose programming language designed with parallelism in mind already from the start. Encore is built on an active object model for parallelism. An active object is an object that has its own heap and logical thread of control. When you call a method on an active object you instantly get a future back. A future, of type \( T \), is a container that promises to sometime in the future contain data of type \( T \). When you actually need the data, you can wait on the future and block your own thread of control, until the future has been fulfilled by the active object, but as long as you don’t need the data both your thread, and the active object you invoked the method on, can execute in parallel.

In Encore, objects are active per default, but you can still create a regular objects from classes that were defined using the passive keyword. Furthermore, you cannot create an active object from a passive class.

An advanced type system that uses capabilities to control the sharing of passive data between active objects is being worked on for Encore. When this is done, it will turn concurrency problems into type problems, and compiled programs will be guaranteed to be free from data races.
def pop() : Maybe T
match this.top with
    Empty => Nothing
    Link(data) => Just data

Figure 1: An example of pattern matching.

class Counter {
    value : int

    def init(start : int) : void
        this.value = start

    def count() : void
        while this.value > 0 {
            print this.value;
            this.work();
            this.value = this.value - 1
        }

    def work() : int
        for i in [1..1000000]
            1*2*3*2*1
}

class Main
    def main() : void
        for i in [1..2]
            (new Counter(100)).count()

Figure 2: Program demonstrating the concurrent execution of active objects in Encore.
2.2 Pattern Matching in Functional Languages

2.2.1 Algebraic Datatypes

In most functional programming languages, objects do not exist, instead algebraic datatypes are used to define composite data. For example, the code in Figure 3 would define a polymorphic linked list type in some imaginary, but ML-inspired (Milner et al., 1997), functional language:

Here List is the name of the new type, while Empty and Link are names of constructors that can be used to construct new instances of the List type. The Empty constructor takes no arguments, while the Link constructor takes two arguments, of type a and List a. Figure 4 shows how a List containing the letters 'a', 'b' and 'c', in order, would be constructed.

2.2.2 Pattern Matching

In functional programming languages, pattern matching is a conditional construct that changes the control flow based on both the structure and value of data. The idea of pattern matching on composite data originates from a paper by Burstall (1968).

Figure 5 showcases a recursive function that uses pattern matching to compute the factorial of its argument. The factorial function consists of a single match expression and x is called the argument of the match expression. Note that in this case x is also the argument of the function, but this does not have to be the case in general. The match expression also consists of two more lines. Each of these is called a match clause, or just a clause. The left hand side of a match clause is called its pattern, and the right hand side its handler.

The semantics of a match expressions are such that the argument is matched against each pattern in order, until one matches – then the corresponding handler is executed. The match expression as a whole will evaluate to the same value as the chosen handler. However, if no matching pattern was found, then a run-time error will generally occur.

The factorial function in Figure 5 consists of two clauses. The first clause’s pattern is just the number 0. This is called a value pattern. Value patterns match whenever the argument is equal to the value in the pattern.

datatype List 'a = Empty
| Link ('a * List 'a)

Figure 3: An algebraic datatype representing a linked list in some ML-like language.
2.2 Pattern Matching in Functional Languages

\[
l = \text{Link('a', Link('b', Link('c', Empty)))}
\]

Figure 4: A linked list containing the letter 'a', 'b' and 'c'.

```python
def fac(x):
    match x with
    0 => 1
    n => n * fac(n-1)
```

Figure 5: A factorial function.

In the second clause of the factorial function the pattern is a variable, \(n\). This is called a variable pattern. Variable patterns match against anything. Furthermore, when a variable pattern matches, the corresponding handler will execute inside a new environment where the variable in the pattern is bound to whatever it matched against. In the case of the factorial function, this means that whenever \(x\) is something other than 0, the second pattern will match and the second handler will be executed with the name \(n\) bound to the number \(x\).

The true power of pattern matching comes from its ability to take apart complex data structures, so let’s have a look at an example involving an algebraic datatype. Figure 6 shows a function that calculates the sum of a linked list, defined as in Figure 3. Here we see examples of another kind of pattern, the constructor pattern. Remember that \(\text{Empty}\) and \(\text{Link}\) were the names of the constructors that could be used to create data of the \(\text{List}\) type. Constructor patterns deconstruct that which they are matched against to see how it was created. In the example, the \(\text{Empty}\) pattern will only match if \(\text{list}\) was constructed using the \(\text{Empty}\) constructor. Similarly, \(\text{Link}(\text{head}, \text{tail})\) will only match if \(\text{list}\) was constructed using the \(\text{Link}\) constructor. However, unlike the \(\text{Empty}\) constructor, the \(\text{Link}\) constructor took two parameters. If a constructor pattern has arguments, and that which it matched against was created using the correct constructor, then it will also extract the parameters that was given to the constructor and recursively match those against its own arguments.

In the factorial example, we used pattern matching to select a handler based on the value of data. Then, in the sum example, we used pattern matching to select a handler based on the structure of data. Now let’s see how we can combine this and make a choice on both the structure of data, and the value of substructures, at the same time.

Figure 7 defines a new algebraic datatype called \(\text{Tree}\), representing a binary search tree. The function \(\text{insert}\) inserts new data into the tree while maintaining the invariant property that all data in the right sub-tree is larger than the data in the root, and all data in the left sub-tree is smaller than the data in the root.
2.3 Pattern Matching in Object Oriented Languages

2.3.1 Algebraic Datatypes and Objects

Object oriented languages generally do not have algebraic datatypes, instead objects fill their role as containers of data. There are some very important differences between algebraic datatypes and objects. Algebraic datatypes reveals their insides to the outside world. This is what allows functional languages to pattern match on them. On the other hand, objects generally restrict access to their fields. This practice is known as encapsulation and reduces coupling between different parts of the program. Often, this leads to code that is easy to test, maintain and reuse.

At first it seems like pattern matching and encapsulation are at odds with each other; one is about decomposing an object from the outside while the other is about restricting outside access. However, the are precedents within the object-oriented world. When an algorithm demands that we access some sub-structure
of an object from the outside, a *getter method* is generally used to access it. What the getter method returns doesn’t have to be part of the objects actual internal representation; instead it may be calculated on the fly when the method is called. In this way the getter method allows the object to present an interface to the outside world that is *independant of its own internal representation*.

### 2.3.2 Dynamic Dispatch

Pattern matching does not exist in most object oriented programming languages. Instead there are other techniques that attempt to solve the same problems as pattern matching: selecting a handler based on the structure of data. For example, let’s say we have an abstract ListLink super-class, an ActualLink class and a Sentinel class that both inherits from ListLink. If we now call a find method on an object of type ListLink, then this will do different things depending on whether the object was an ActualLink or a Sentinel at run-time. This is called *dynamic dispatch* and is a feature of most object oriented programming languages. Dynamic dispatch is similar to pattern matching on our ListLink object in that both are able to select a handler based on the run-time type of our object.

However, dynamic dispatch is limited to the object that we call the method on (Figure 8. When the problem inherently demands that you decompose multiple objects in order to make some choice, then standard object-oriented methods that rely on dynamic dispatch run into problems. An example of such a problem is algebraic simplification. Let’s say we want to simply the expression $3x + 2x$ to $5x$ and we have the following classes which all implement an Expression interface: Number, Variable, Sum, Product. When we call the simplify method on our Expression object, dynamic dispatch will be able to decompose the initial Expression into a sum of two terms, but then we run into problems. To proceed we need to decompose both terms, but dynamic dispatch cannot do this.

The algebraic simplification problem can still be solved without pattern matching, but we will have to do alot of type testing and type casting.

In a sense pattern matching is similar to *multiple dispatch* (Figure 8), which exists in some languages, such as Common Lisp, but not in mainstream object oriented languages such as Java and C++. However, the analogy is far from perfect. Unlike, multiple dispatch, pattern matching can deconstruct its arguments to an arbitrary depth and it also allows you to take the value of fields into account when selecting the handler.
2.3 Pattern Matching in Object Oriented Languages

```scala
class Animal {
  def greet(other : Animal)
  print "Hi!
}

class Cat : Animal {
  def greet(other : Animal)
  print "Meow!

  def greet(other : Cat)
  print "Grrr!

}

murre = new Cat() : Animal
rger = new Cat() : Animal

murre.greet(tiger)
```

Figure 8: In a language with dynamic dispatch murre will greet tiger with “Meow!” since the version of `greet` that gets invoked is chosen depending on the run-time type of `murre`, but the compile-time type of `tiger`. In a language with multiple dispatch `tiger` would also have his type inspected at run-time so murre would instead greet him with “Grrr!”.

### 2.3.3 Related Work

One early attempt to reconcile pattern matching with object-oriented programming was made in the language Pizza (Odersky and Wadler, 1997). Pizza had case classes, which are regular classes except for that its constructor implicitly defines an algebraic datatype and corresponding constructor pattern. Case classes also appeared much later in Scala (Odersky et al., 2004). See Figure 9 for an example using case classes in Scala.

Scala also has extractors (Emir et al., 2007) which are user defined functions that deconstruct an object into a `Maybe` (called an `Option` in Scala). The extractor function may be called by the compiler to transform the object into a `Maybe` during pattern matching. This makes possible to write patterns that preserve encapsulation, something that isn’t possible with case classes.

Very similar to Scala’s extractors are F#’s active patterns (Syme et al., 2007) which allows for custom patterns through user defined functions that deconstruct algebraic datatypes into `Maybes`. Both extractors and active patterns are based on the concept of views (Wadler, 1987), which are user defined implicit coercions between algebraic datatypes that may be used during pattern matching.

Matchete (Hirzel et al., 2008) is an extension to Java that adds a pattern matching statement to the language. However, instead of returning a `Maybe`,
2.3 Pattern Matching in Object Oriented Languages

```
abstract class TreeNode

case class Branch(key : Int, left : TreeNode, right : TreeNode)
  extends TreeNode

case class Empty extends TreeNode

class Tree {
  ... 
  def insert(root : TreeNode, key : Int) {
    root match {
      case Empty => 
        Branch(key, Empty, Empty)
      case Branch(k, left, right) if key > k => 
        Branch(k, left, insert(right, key))
      case Branch(k, left, right) if key < k => 
        Branch(k, insert(left, key), right)
    }
  }
}
```

Figure 9: Scala’s case classes allow for very concise syntax when creating classes that can be used in pattern matching, but do not preserve encapsulation.

Matchete’s custom patterns use tuples. This means that if the pattern is defined for one class, then it must match all members of that class. Matchete also uses a unified syntax for pattern matching on datatypes, and matching on other forms of patterns, such as regular expressions.

```
class Branch(key : Int, left : TreeNode, right : TreeNode)
  extends TreeNode

object Branch {
  def unapply(treeNode : Branch) : 
    Option[(Int, TreeNode, TreeNode)] = 
    Some((treeNode.key, treeNode.left, treeNode.right))
}
```

Figure 10: This Scala extractor creates a Branch class that can be used in pattern matching, similar to Figure 9.
3 Design

Remember that when some piece of data is matched against a constructor pattern, the pattern checks which constructor that was used to construct the data, and also extracts the parameters that was used to construct that data for further matching. Clearly, a pattern for matching against objects, an object pattern, must also extract some data that represents the object’s substructures if it is to be useful. One natural way to do this, if we do not want to break encapsulation, is to let each class explicitly define its own patterns using code. An object pattern must decide whether it matches or not, and if it does it must return a representation of some substructures for further matching. This is exactly the description of a method whose return type is a Maybe, also known as an Option. A Maybe is a simple algebraic datatype defined as in Figure 11. This suggests that the natural way of adding matching on objects would be to implement a primitive Maybe type and matching on it. Then we reduce the the problem of matching on objects to the simpler problem of matching on algebraic datatypes.

By letting the user define how objects of a given class is to be matched against the valid pattern for that class, the patterns remain independent of the object’s actual implementation. For example, an object representing a 2d vector could match both a Polar(r, φ) pattern and a Cartesian(x, y) pattern, regardless of whether it’s implemented using Cartesian or polar coordinates.

Encore’s object patterns are defined by regular methods that return Maybes. Figure 12 shows an example where a stack has been implemented as a linked list, using pattern matching in the pop method. The example demonstrates why it’s important to allow patterns to fail by returning a Nothing: A Link should not match against an Empty object, but at the same time it must still declare an empty method to satisfy the type-checker. Unless we want to disable the type-checking of object patterns, we have to return a Maybe.

datatype Maybe 'a = Nothing |

Figure 11: The definition of the Maybe type.
3.1 Suggested Syntax

Even though there are good logical and historical reasons for why a method that defines an object pattern should return a `Maybe`, one could still argue that the complex type signature is confusing to new users. The Encore compiler has a desugaring pass which can be used to add syntactic sugar to the language. Figure 13 shows how the Link class from figure 12 could look with some extra syntactic sugar. However, actually adding support for this to the compiler is beyond the scope of this thesis.

3.2 Active Objects

When a class is defined without any extra keywords, it defaults to an active class from which active objects will be created. However, despite programmers being encouraged to use active objects by the language’s syntax, we do not allow decomposing active objects using match expressions.

One reason for this, is that it encourages a programming style that does not handle concurrency well. A method that implements an object pattern is supposed to extract a representation of its object which introduces potential aliasing problems. Of course a responsible programmer might return clones of the objects internals, or a totally different representation of the object, just like an irresponsible programmer may return the active object internals from a regular method.

Another reason why matching on active objects has been disallowed is that the active object may change during matching which may make it impossible to predict which clause of the match expression that will be executed (Figure 14). Similar race conditions can occur when another active object modifies a passive object that is currently being matched on, but those race conditions should be caught by Encore’s future type-system that was mentioned in section 2.1.

Finally, the problems where pattern matching shines are those were multiple objects have to be decomposed before a some important decision can be made. In other words, for these problems, behavior is inherently `separated from data`, while active objects are a way to add concurrency to the standard object-oriented model where behavior is `bundled with data`. This in turn means that, for good pattern matching problems, objects mostly serve as containers of data; they’re inherently passive objects.

Taking all this into account, we conclude that disallowing matching on active objects doesn’t restrict the expressiveness of the language, but it does remove some potential pitfalls when it comes to designing thread-safe concurrent applications.
trait ListLink<t> {
    require link() : Maybe (t, ListLink<t>)
    require empty() : Maybe void
}

passive class Empty<t> : ListLink<t> {
    def link() : Maybe(t, Link<t>)
        Nothing
    
    def empty() : Maybe void
        Just ()
}

passive class Link<t> : ListLink<t> {
    data : t
    next : ListLink<t>

    def init(data : t, next : ListLink<t>) : void {
        this.data = data;
        this.next = next
    }

    def link() : Maybe(t, ListLink<t>)
        Just (this.data, this.next)

    def empty() : Maybe void
        Nothing
}

passive class Stack<t> {
    top : ListLink<t>

    def init() : void
        this.top = new Empty<t>

    def push(data : t) : void
        this.top = new Link<t>(data, this.top)

    def pop() : Maybe t
        match this.top with
            empty(() => Nothing : Maybe t
            link(data, next) => {
                this.top = next;
                Just data
            }
        }
}

Figure 12: A stack implemented in Encore, using an underlying linked list.
4 IMPLEMENTATION

```java
class Link<T> {
    data : T
    next : ListLink<T>

def init(data : T, next : ListLink<T>) : void {
    this.data = data;
    this.next = next
}

pattern link : (T, ListLink<T>) =
    extract (this.data, this.next)

pattern empty : void =
    fail
}
```

Figure 13: How the Link class from Figure 12 could look if some extra syntactic sugar were added to Vncore.

```python
def checkParity(x : IntContainer) : void
    match x with
        Even() => print 'x is even!'
        Odd() => print 'x is odd!'
        _ => print 'This should never be executed'
```

Figure 14: If matching on active objects were possible, some match expressions might not match any clause, despite logically covering each possible case. This is because the active object might change state during the execution of the match expression.

4 Implementation

The Encore compiler is written in Haskell (Marlow, 2010) and generates C code as output which is then piped into Clang (cla, 2010) to generate executable code.

4.1 Parsing

The parser is built on top of Parsec (Leijen, 2001), a combinator parser that allows you to very easily write the actual parser in code. In contrast, most parsers are built using generators where you specify the grammar and then run a parser generator to build the actual parser.

Figure 15 shows the parts of Encore’s grammar that relates to pattern matching. Note that methods that define object patterns are declared using regular method syntax, and they can indeed be called from outside of match expressions – just like regular methods.
4.2 Typechecking

The purpose of the typechecker is to eliminate some execution errors that would otherwise have been possible. These will instead be caught during the compilation.

In Encore, three things must be true of a well-typed match expression. First of all, the right hand side of all MatchClauses must all be of the same type. Second, if a guard is present, it must have a boolean type. Finally, all patterns (the left hand sides of a MatchClause) must be of the same type as the match expression’s argument.
However, the typing rules for patterns are slightly more complicated than for other parts of the program. Remember that one possible pattern is the value pattern which matches only against arguments that are identical to itself. This means that potentially any expression could be a valid pattern, however, the language has been restricted so that global functions are forbidden from being called to generate a value pattern in Encore. This is to avoid confusion with the object patterns. Instead, any function call appearing inside a pattern will be interpreted as an extractor pattern and will be typechecked as one.

An object pattern is well-typed if the argument’s class has a method, with the same name as the object pattern, that has the type () -> Maybe k, where k is the type of the arguments supplied to the object pattern.

Using formal typing rules, as described in (Cardelli, 1997), we can summarize the typing rules of the pattern matching expression as follows:

\[
\begin{align*}
\text{Environment} & \\
\Gamma ::= & \\
(\text{Empty}) & \\
| & \Gamma, x : t \\
\end{align*}
\]

\[
\begin{align*}
\text{Pattern} & \\
p ::= & \\
(\text{Value}) & \\
| & v \\
| & x \\
| & \text{Just } p \\
| & (p_1, \ldots, p_n) \\
| & m(p) \\
| & (\text{Object}) \\
\end{align*}
\]

\[
\begin{align*}
\text{Value} & \\
v ::= & \\
(\text{int}) & \\
| & \text{string} \\
| & \text{bool} \\
| & \text{real} \\
| & \text{Nothing} \\
| & () \\
\end{align*}
\]
4.3 Code Generation

Let \( x \Gamma \vdash e : t \) mean that \( e \) is a well-formed expression of type \( t \) when evaluated in the environment \( \Gamma \). Also, let \( t; p \vdash \Gamma \) mean that \( p \) is a well-formed pattern of type \( t \), producing environment \( \Gamma \), in which the corresponding guard and handler will be evaluated. Finally, \( t' <: t \) means that \( t' \) is a subtype of \( t \).

The following rules show when a pattern is well-formed and how variables are bound to types:

\[
\begin{align*}
\text{Value} & \quad \frac{e \vdash v : t}{t; v \vdash e} \\
\text{Variable} & \quad \frac{t; x \vdash x : t}{t} \\
\text{Tuple} & \quad \frac{\forall i \in [1..n], t_i; p_i \vdash \Gamma_i}{(t_1, ..., t_n); (p_1, ..., p_n) \vdash \bigcup_{i=1..n} \Gamma_i}
\end{align*}
\]

\[
\begin{align*}
\text{Maybe} & \quad \frac{\Gamma \vdash p : t}{t' \vdash p \vdash \Gamma} \\
\text{Object} & \quad \frac{\text{methodLookup}(t, m) = () -> t'}{t; m(p) \vdash \Gamma}
\end{align*}
\]

\[
\begin{align*}
\text{Match} & \quad \frac{\forall i \in [1..n], \Gamma, \Gamma_i \vdash g_i : \text{bool} \quad \forall i \in [1..n], \Gamma, \Gamma_i \vdash h_i : t_i' \quad \forall i \in [1..n], t_i' <: t}{\Gamma \vdash (\text{match } e \text{ with } [p_i \text{ when } g_i = \bowtie h_i]_{i \in [1..n]} : t')}
\end{align*}
\]

4.3 Code Generation

The Encore compiler, encorec, generates C code which is then piped into a C compiler to generate executable code. An Encore match expression is translated into a series of nested C if-statements with one “then-clause” per MatchClause in the original code. The overall structure of the generated code is shown in Figure 17. Do note, however, that this isn’t literally what’s generated.

First comes the declaration of the variable that is intended to hold the value of the entire match-expression, regardless of which MatchClause that is chosen. Next follows the translation of the match-expression’s argument. Then all variables that are used in the first pattern are declared. It is important that the pattern’s free variables are declared before the if-statement as they may be referenced both from the pattern, the guard and the handler and therefore need to be in scope the whole time.

The if-statement’s condition is made up of two parts joined together by an and operator. The second part corresponds to the guard. If the original source has no guard, a guard that is always true will be added in. Adding the redundant guard simplifies encorec’s code and it is assumed that the C compiler that the
some_type valueOfMatchExpression;
some_type arg = //Translation of argument

some_type variableUsedInPattern1;
if ((/* Translation of pattern1 */ &&
    /* Translation of guard1 */) { 
    valueOfMatchExpression = //Translation of handler1
} else {
    some_type variableUsedInPattern2;
    if ((/* Translation of pattern2 */ &&
        /* Translation of guard2 */) { 
        valueOfMatchExpression = //Translation of handler2
    } else {
        // Etc for remaining patterns...
    }
}

Figure 17: The structure of the C code that is generated from a pattern matching expression.

translated code is piped into will be able to optimize this away.

The first part of the if-statement’s condition corresponds to the actual pattern and will be different depending on what pattern that we are dealing with. The translated pattern will have to contain code that both checks whether the pattern matches and binds the pattern’s free variables. Let’s have a look at all the possible patterns:

1. **Value Patterns** are translated into *identity* checks, except for when strings are compared, then *equality* is instead checked.

2. **Variable Patterns** almost always match so they’re translated into code that binds the corresponding variable to the value of the argument and then evaluates to true. The exception to this is when when the same variable occurs many times inside the same pattern. In that case the same rules as for value patterns apply. The translator uses a *set* to keep track of which variables that have already been used inside the current pattern or one of its parent patterns (in the case that the current pattern was nested inside of another pattern).

3. **Maybe Patterns.** In Encore, Maybes are translated into a pointers to structs consisting of a tag and a pointer to an optional value. However, every instance of *Nothing* is translated into a pointer to the same struct. Thus the translation of *Nothing* patterns are actually treated as a *Value Pattern* and this section needs only concern itself with Just patterns. Just patterns are translated into an and expression where the first part checks
that the argument’s tag is \textit{just} and the second part is recursively generated by translating the pattern’s value (matched against the argument’s value).

4. \textbf{Tuple patterns} are translated by recursively translating each component of the tuple pattern and joining the code together with and operators.

5. \textbf{Object patterns} are translated by calling the method that defines the pattern—it will have the same name as the pattern itself—on the \textit{match expression}’s argument, and translating the \textit{object pattern}’s argument (when matched against the method call’s return value).

The then-part of the if-statement contains the translation of that \textit{Match-Clause}’s handler, and the else-part contains the translation of the next \textit{Match-Clause} which can be recursively translated. The else-part of the last if statement throws a run-time error by printing an error message and calling C’s \texttt{exit} function.

5 \hspace{1em} \textbf{Results and Discussion}

5.1 \hspace{1em} \textbf{Expressive Power}

5.1.1 \hspace{1em} \textbf{Tree Rotations}

Figure 18 and figure 19 show two examples of how a method that does a right tree rotation can be implemented in Encore. The example using pattern matching is objectively shorter, and more importantly, \textit{related lines of code occur closer to each other} which makes it easier to get an overview of what the code does; in the pattern matching example we check the structure of the tree and bind names to relevant sub-structures with a single line of code, without pattern matching we have to spend 6 lines of code on the same task. The difference is even bigger in the \texttt{else} case. In the pattern matching example we can write the pattern and the handler immediately following each others. Without pattern matching, the leaf case’s handler, and it’s “pattern” are separated by the branch case’s handler.

5.1.2 \hspace{1em} \textbf{Algebraic Simplification}

Figure 20 show some simple algebraic simplifications of a \texttt{Sum}. Figure 21 shows an alternative version that doesn’t use pattern matching. However, it’s quite the monstrosity. Writing a functional simplification method in valid Encore, without using pattern matching, is not a simple task. The standard way of doing it in object-oriented languages involves type-tests and type-casts. However, casting an
def rotateRight() : void

match this with
  Branch(x, branch(y, leftLeft, leftRight), right) =>
    { this.value = y;
      this.left = leftLeft;
      this.right = branch(x, leftRight, right) }

Leaf() => print "Error, can't rotate a leaf!"
_ => print "Error, can't rotate if left is a leaf!"

Figure 18: Right-rotation using pattern matching in Encore.

def rotateRight() : void
if not this.isLeaf then
  if not this.left.isLeaf then
    let v = this.value
    leftLeft = this.left.left
    leftRight = this.left.right
    right = this.right in
    { this.value = this.left.value;
      this.left = leftLeft;
      this.right = branch(v,leftRight,right) }
  else
    print "Error, can't rotate if left is a leaf!"
else
  print "Error, can't rotate a leaf!"

Figure 19: Right-rotation without using pattern matching in Encore.
5.1 Expressive Power

5 RESULTS AND DISCUSSION

def simplify() : Exp
    match (this.term1, this.term2) with
       (Num(a),Num(b)) => new Num(a+b) : Exp
       (Product(Num(a),Var(x)),Product(Num(b),Var(x))) =>
           new Product(new Num(a+b), new Var(x))
   _ => this

Figure 20: Algebraic simplifications of a Sum using pattern matching.

object from a trait (Exp) to a class that implements that trait (Num) is illegal in Encore. The workaround displayed in Figure 21 circumvents this by forcing the the trait Exp to require methods that explicitly returns the object with a more specific type. Naturally, these methods would have to return null when called on an object of the wrong type. I.e. Product.asNum() = null, but Num.asNum() = this. One might think that using Maybe would be a better idea than null, but currently you need pattern matching to decompose a Maybe in Encore so that’s technically cheating – and it still leads to awful code.

Another possibility would be to rewrite the entire class hierarchy so that there only is a single Exp class, containing boolean fields which indicates which of the original classes that it actually represent. Either way one would have to write code that tries to circumvent Encore’s type system instead of having it work in your favor. We think it’s safe to say that it’s actually impossible to write a good simplifier in Encore without using pattern matching.

However, despite being the by far best solution to the algebraic simplification problem, pattern matching still suffers from some problems. The entire code for the pattern matching version is included in appendix B. There we can see that the pattern matching version still has a lot of notational overhead when defining the classes: Exp, Num, Var, Sum and Product. Each of these classes has a corresponding pattern that it matches, but it also has to define the patterns corresponding to the other classes, which it shouldn’t match. In the general case were we have c classes implementing the same trait and p patterns required by that trait we have to do cp pattern definitions. A fairly common scenario is that you have one pattern per class and each class matches only its own pattern. In this scenario the notational overhead grows quadratically with the number of classes. However, in Encore you will run into the same problem when defining the class specific predicates required by the non-pattern matching version of the code. Furthermore, if Encore was to support method overriding with dynamic dispatch, then the trait could define a default implementation of each pattern that would not match, and each class would only have to define the patterns that it actually matches. With this feature the notational overhead would instead grow at a linear rate which would make a massive difference.
def simplify() : Exp
    if term1.isNum() and
term2.isNum() then
        let a = term1.asNum().getVal()
            b = term2.asNum().getVal()
in
        new Num(a + b)
    else if term1.isProduct() and
term1.asProduct().getFactor1().isNum() and
term1.asProduct().getFactor2().isVar() and
term2.isProduct() and
term2.asProduct().getFactor1().isNum() and
term2.asProduct().getFactor2().isVar() then
        let a = term1.asProduct().getFactor1().asNum().getVal()
            b = term2.asProduct().getFactor1().asNum().getVal()
            v = n1.getFactor2()
in
        new Product(new Num(a + b), v)
    else
        this

Figure 21: The standard object-oriented way of doing algebraic simplification of a Sum
using type-tests and type-casts. Notice how Encore doesn’t support type casts to more
specific types, so these have to explicitly implemented as methods required by the trait
Expr.
5.2 Performance

5.2.1 Benchmarks

The benchmarking code first performed a right-rotation on a tree and then a left rotation on the same tree to return it to its original state. This operation was repeated one million times for each rotation version and the total time was measured. The whole procedure was repeated 10 times and the measurements were averaged. On average, the version using pattern matching took 985ms while the other version took 435ms.

5.2.2 Discussion

That the match expression would slow down the code was expected; longer run-times is generally the cost of being able to code with higher level abstractions. Naturally, the magnitude of the relative difference will depend on how much work that is done on top of the actual matching. If we were doing tree insertions instead of rotations the relative difference would be larger, while if we were doing synchronous network communication it would be insignificant.

To perform the pattern matching in the tree rotation example Encore has to do a lot of overhead. First it calls the Branch method, this method will do different things for branches and leaves and is analogous to the first if-statement in the non-pattern matching example. However, the actual method call is pure overhead.

Second, to actually communicate whether the target was a branch or not to the match expression, the branch method returns a Maybe, this Maybe has to be allocated on the heap which incurs further overhead.

Third, to extract the value, and the left and right sub-trees from the target these have to be stored inside the returned Maybe. However, a Maybe can only contain one value so a tuple of size three has to be created and allocated on the heap. Everything related to this tuple is pure overhead.

Finally, to compare the extracted values to those supplied as arguments to the extractor pattern, a second tuple has to be created to store the pattern’s arguments. This, again, is overhead.

Since Encore translates into C code, we can use a C profiling tool to verify the above argument. Figures 22 and 23 show some of the output generated by gprof when used on the benchmarking example. As can be seen, the pattern matching example spends a lot more time inside the allocation functions. This suggests that the performance of the pattern matching expression could be improved if we could allocate tuples and Maybe on the stack instead of the heap. However, even if we managed to allocate these data structures on the stack, we would still
### Conclusion and Future Work

The goal of this thesis project was to design and implement a pattern matching construct that is capable of matching on objects. We have accomplished this goal, and in section 5 we saw some examples where using pattern matching lead to increased conciseness and readability.

However, throughout this thesis we’ve also noted several improvements that could be made to the pattern matching construct.

In section 3.1 we suggested some syntactic sugar that could make object patterns easier to use for new users.

Then, in section 5.1.2, we saw how combining pattern matching with broad class hierarchies still leads to significant notational overhead when defining patterns. We also noted that this was a symptom of Encore not supporting method

---

<table>
<thead>
<tr>
<th>% cumulative time</th>
<th>self seconds</th>
<th>calls</th>
<th>self ms/call</th>
<th>total ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.15</td>
<td>0.09</td>
<td>0.09</td>
<td></td>
<td></td>
<td>heap_alloc</td>
</tr>
<tr>
<td>14.90</td>
<td>0.16</td>
<td>0.07</td>
<td>1000000</td>
<td>0.00</td>
<td>TreeNode_rotateLeft</td>
</tr>
<tr>
<td>12.77</td>
<td>0.22</td>
<td>0.06</td>
<td>1000000</td>
<td>0.00</td>
<td>TreeNode_rotateRight</td>
</tr>
<tr>
<td>10.64</td>
<td>0.27</td>
<td>0.05</td>
<td>4000000</td>
<td>0.00</td>
<td>TreeNode_branch</td>
</tr>
<tr>
<td>6.38</td>
<td>0.30</td>
<td>0.03</td>
<td>2000000</td>
<td>0.00</td>
<td>fun_branch</td>
</tr>
<tr>
<td>4.26</td>
<td>0.32</td>
<td>0.02</td>
<td>6000011</td>
<td>0.00</td>
<td>TreeNode__init</td>
</tr>
<tr>
<td>4.26</td>
<td>0.34</td>
<td>0.02</td>
<td></td>
<td></td>
<td>encore_alloc</td>
</tr>
</tbody>
</table>

Figure 22: Profiling of tree rotations with pattern matching.

<table>
<thead>
<tr>
<th>% cumulative time</th>
<th>self seconds</th>
<th>calls</th>
<th>self ms/call</th>
<th>total ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.79</td>
<td>0.03</td>
<td>0.03</td>
<td>6000011</td>
<td>0.00</td>
<td>init_TreeNode</td>
</tr>
<tr>
<td>15.79</td>
<td>0.06</td>
<td>0.03</td>
<td>1000000</td>
<td>0.00</td>
<td>TreeNode_rotateRight</td>
</tr>
<tr>
<td>10.53</td>
<td>0.08</td>
<td>0.02</td>
<td>6000011</td>
<td>0.00</td>
<td>TreeNode__init</td>
</tr>
<tr>
<td>10.53</td>
<td>0.10</td>
<td>0.02</td>
<td>2000005</td>
<td>0.00</td>
<td>TreeNode_makeBranch</td>
</tr>
<tr>
<td>10.53</td>
<td>0.12</td>
<td>0.02</td>
<td></td>
<td></td>
<td>encore_alloc</td>
</tr>
<tr>
<td>5.26</td>
<td>0.13</td>
<td>0.01</td>
<td>2000000</td>
<td>0.00</td>
<td>fun_branch</td>
</tr>
<tr>
<td>5.26</td>
<td>0.14</td>
<td>0.01</td>
<td>1000000</td>
<td>0.00</td>
<td>TreeNode_rotateLeft</td>
</tr>
<tr>
<td>5.26</td>
<td>0.15</td>
<td>0.01</td>
<td>1</td>
<td>10.00 130.02</td>
<td>Main_main</td>
</tr>
<tr>
<td>5.26</td>
<td>0.16</td>
<td>0.01</td>
<td></td>
<td></td>
<td>heap_alloc</td>
</tr>
</tbody>
</table>

Figure 23: Profiling of tree rotations without pattern matching.

need to populate them and this also takes a significant amount of time, which is evident from the fact that the pattern matching version also spends more time inside the rotateLeft, rotateRight and branch methods – the last isn’t called at all by the non pattern matching version since it is the method that defines the branch pattern.
overriding.

Finally, in section 5.2.2 we analyzed the run-time performance of pattern matching in Encore and noted how the ability to pass tuples and maybes on the stack could lead to improved performance.

7 References

References


Cardelli, L. (1997). Type systems.


A Tuples

One other contribution that this thesis project has made towards the improvement of Encore is adding tuples to the language. While tuples are not directly related to pattern matching, it turns out that it is hard to do anything meaningful with matching on objects if the functions that define patterns cannot return more than one value. Thus we added tuples to Encore.

Tuples were implemented in C, as a part of Encore’s run-time library. The code below shows how the tuples are represented by a C struct.

```
struct tuple_t {
    size_t size;
    pony_type_t **types;
    encore_arg_t elements[];
};
```

The run-time library includes functions for creating, tracing, and modifying tuples as well as a function for accessing it’s elements. However, at the moment, only the function for creating a tuple has a direct equivalent in Encore proper. If you want to take apart a tuple to inspect its parts, you currently need to use pattern matching.

B The Algebra Code

```
trait Exp {
     require Num() : Maybe int
     require Var() : Maybe String
     require Sum() : Maybe (Exp, Exp)
     require Product() : Maybe (Exp, Exp)

     require show() : void

     require simplify() : Exp
}

passive class Num : Exp {
    num : int

    def init(num : int) : void
        this.num = num

    def Num() : Maybe int
        Just this.num
```
```python
def Var() : Maybe String
    Nothing

def Sum() : Maybe (Exp, Exp)
    Nothing

def Product() : Maybe (Exp, Exp)
    Just (new Num(1), this)

def show() : void
    print("{}", this.num)

def simplify() : Exp
    this : Exp
}

passive class Var : Exp {
    name : String

def init(name : String) : void
    this.name = name

def Num() : Maybe int
    Nothing

def Var() : Maybe String
    Just this.name

def Sum() : Maybe (Exp, Exp)
    Nothing

def Product() : Maybe (Exp, Exp)
    Just (new Num(1), this)

def show() : void
    print("{}", this.name)

def simplify() : Exp
    this : Exp
}

passive class Sum : Exp {
    term1 : Exp
    term2 : Exp

def init(t1 : Exp, t2 : Exp) : void
    this.term1 = t1.simplify();
    this.term2 = t2.simplify();
}
```
def Num() : Maybe int
    Nothing

def Var() : Maybe String
    Nothing

def Sum() : Maybe (Exp, Exp)
    Just (this.term1, this.term2)

def Product() : Maybe (Exp, Exp)
    Nothing

def show() : void {
    this.term1.show();
    print("{}", " + ");
    this.term2.show()
}

def simplify() : Exp
match (this.term1, this.term2) with
    (Num(a), Num(b)) => new Num(a+b) : Exp
    (Product(Num(a), Var(x)), Product(Num(b), Var(x))) =>
        new Product(new Num(a+b), new Var(x))
    _ => this
}

passive class Product : Exp {
    factor1 : Exp
    factor2 : Exp

def init(f1 : Exp, f2 : Exp) : void{
    this.factor1 = f1.simplify();
    this.factor2 = f2.simplify();
}

def Num() : Maybe int
    Nothing

def Var() : Maybe String
    Nothing

def Sum() : Maybe (Exp, Exp)
    Nothing

def Product() : Maybe (Exp, Exp)
    Just (this.factor1, this.factor2)

def show() : void {
    this.factor1.show();
}
print("{}");
this.factor2.show()
}

def simplify() : Exp
    match (this.factor1, this.factor2) with
        (Num(a), Num(b)) => new Num(a*b) : Exp
        - => this
    
}