For-Comprehension
An Encore Compiler Story

Joy Elizabeth Manning
Abstract

For-Comprehension, An Encore Compiler Story

Joy Elizabeth Manning

Encore is an active object programming language that is being developed at Uppsala University. The programming language includes a for-loop construct with limited functionality. This thesis describes the implementation of a more extensive looping construct called for-comprehension, to replace the for-loop. The resulting for-comprehension is a powerful tool for iteration that allows for elegantly written loops and is ripe for further development.
## Contents

1 **Introduction**  
   1.1 Goal, Purpose and Results ............................................. 2

2 **Background**  
   2.1 Encore ................................................................. 3
   2.2 Encore Compiler ....................................................... 3
      2.2.1 Abstract Syntax Tree ............................................ 4
      2.2.2 Parser ............................................................ 4
      2.2.3 Desugarer ....................................................... 5
      2.2.4 Type Checker .................................................. 6
      2.2.5 The Non-Optimizing Optimizer ................................. 6

3 **The New for comprehension, Why and How?**  
   3.1 Reasons for New for comprehension ................................ 7
   3.2 What is the New for comprehension? ................................ 8
   3.3 Correctness ........................................................... 9

4 **Design Challenges** ..................................................... 11
   4.1 Boxing Allows Updates ............................................... 11
   4.2 Break Inside A Method ............................................... 12
   4.3 Ranges - The Unique Collection .................................... 12

5 **Implementation** .......................................................... 13
   5.1 Extending the AST .................................................... 13
   5.2 Parser ............................................................... 14
   5.3 Desugarer ............................................................ 15
   5.4 Type Checker ........................................................ 15
      5.4.1 Type Checking for comprehensions ............................ 15
      5.4.2 Adding typing data to AST node ............................... 16
      5.4.3 Type checking again ........................................... 16
   5.5 Desugaring the Typed AST ......................................... 17
      5.5.1 Desugaring \texttt{For} into map, flatMap and foreach .... 17
      5.5.2 Break: Desugaring \texttt{For} into maybeForeach ............ 19
      5.5.3 Boxing ......................................................... 19

6 **Evaluation of Implementation** ....................................... 23
   6.1 Elegant Source Code ................................................ 23
   6.2 Run Time ............................................................. 24
   6.3 Interface of iteration ............................................... 24
   6.4 Added compilation steps ............................................ 26
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Illustration of a for loop</td>
<td>1</td>
</tr>
<tr>
<td>2.1</td>
<td>Simplified AST representation of a while loop</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Desugaring of an if-then clause</td>
<td>5</td>
</tr>
<tr>
<td>6.1</td>
<td>Performance of for comprehension vs. for loop</td>
<td>25</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The for loop is a looping construct, that repeatedly executes instructions a specific number of times [9]. In the programming language Encore the for loop exists in a limited form, however it has the potential to become a powerful tool [3] for iteration, by expanding it into for comprehension [11]. The for comprehension is similar to the for loop in appearance, but inside the compiler these looping constructs are very different [3]. The current for loop is a collection-based for loop [9]. This means that iteration is always relative to a programmer-specified collection (or a range), where the loop-variable takes on the value of every element in that collection [9]. The limitation of Encore’s for loop is that the only collection data structures it can iterate over are arrays and ranges. Figure 1.1 illustrates the for loop written in Encore of Listing 1.1, where the header contains a loop variable $x$ that takes on the value of each successive element in the array, and executes the loop-body for every iteration.

![Illustration of a for loop](image)

**Figure 1.1: Illustration of a for loop**

```encore
for x <- [2, 5, 6] do
  print("{}", x)
end
```

**Listing 1.1: A simple for loop**

The for comprehension is also a collection based loop, that can take a similar form as the for loop in the surface syntax [11, 9]. The main difference is hidden in the compiler where the for comprehension is desugared\(^1\) into combinations of method calls [3]. These methods are part of an interface and any type that implements this interface is iterable\(^2\) using for comprehension. For comprehension is therefore not

---

\(^1\)Desugaring - to remove syntactic sugar by replacing some advanced syntactic forms with a smaller set of core constructs.

\(^2\)Iterable - In the context of this thesis iterable object can be iterated over using for comprehension.
limited to only ranges and arrays, but can be used on any iterable collection data structure.

1.1 Goal, Purpose and Results

The goal of this thesis is to develop the current for loop into for comprehension that can iterate over more kinds of data structure by desugaring it into method calls, while maintaining all of the old features of the for loop. The purpose behind this implementation is to improve upon the existing looping construct, and thereby improving upon the programming language Encore.

This thesis reports on a substantial refactoring of the central for loop construct in Encore, to a fully-fledged for comprehension. The new for comprehension can handle iterating over multiple collections, can return a collection, and can also iterate over more data structures than only arrays and ranges. The development of the new for comprehension also includes an interface that allow developers to make their own iterable collection objects. The for comprehension came with a cost as it resulted in an expected performance overhead, however, the benefits of a versatile for comprehension far outweigh the cost.

This thesis gives an overview of Encore and the Encore compiler, discusses the current for loop and its limitations and the design and motivation behind the new for comprehension, as well as its implementation and the consequences thereof.
Chapter 2

Background

This chapter covers relevant background information on the Encore language, a description of Encore’s compiler, and relevant terminology.

2.1 Encore

Encore is a programming language currently being developed at Uppsala University in the context of the FET open-X EU project UPSCALE [2]. Encore is an active-object-based parallel programming language. An active object in Encore is similar to an object in a pure object-oriented language, however active objects can run concurrently, or in parallel with each other [2]. For the purposes of this thesis, Encore can be considered an object-oriented language and parallelism will not be further discussed as this implementation only considers sequential computation within a single actor.

Encore is a class-based object-oriented language, which means that objects are instances of classes [2]. It also has a functional flair to it [11], thus it supports functions outside of classes. A valid Encore program must at least have three things: a Main class, a main method that is inside of the Main class, and a body inside the main method [11]. An example of a valid Hello World Encore program is shown in Listing 2.1 which shows how all methods, and bodies of methods are indented and their scope is closed by the keyword end. All methods begin with keyword def, while all functions begin with fun [11].

```encore
active class Main
    def main() : unit
        print("Hello World")
    end
end
```

Listing 2.1: Valid Encore Program

An Encore source file must have the suffix .enc and can be compiled by using the Encore compiler, called encorec, on a machine that has it installed. Further details of how to install the compiler and run an Encore program can be found in Encore’s documentation [11].

2.2 Encore Compiler

A compiler is, roughly speaking, a program that translates code from a source language into a target language [1]. The Encore compiler translates source code written
in Encore into C [2]. The compiler itself is written in the functional programming language Haskell. The implementation will only make changes to the Encore compiler, thus for the purpose of this thesis, only the Encore compiler will be discussed. The compilation process is broken up into several steps and not all of them have been changed for this thesis. Table 2.1 shows what the steps are, and which steps have been changed or added. What follows are summaries of some of the compilation steps of the Encore compiler in its current form [10].

<table>
<thead>
<tr>
<th>Compilation Steps</th>
<th>Changed</th>
<th>Added step</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Parsing</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>2. Desugaring</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>3. Pre-checking</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>4. Type checking</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>5. Capture checking</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>6. Typed Desugaring</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>7. Re-type checking</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>8. Re-capture checking</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>9. Optimizing</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Table of compilation steps

2.2.1 Abstract Syntax Tree

In the first step of compilation, the source code is broken up into smaller segments, and turned into an abstract syntax tree (AST), which is an intermediate representation of the source code [1]. Each segment represents an expression of the source-code, such as a for loop or an assignment, a method- or a function-call (see Appendix A for all the main AST-nodes’ data structures). These AST nodes in turn can have children nodes, which represents an expression’s sub-expressions, such as a method call’s arguments or body. A while loop, for example, has a conditional sub-expression that determines if the body of the while loop should continue its repeated execution, and the body itself is also a sub-expression of the while loop. A simplified AST representation of the Encore while loop in Listing 2.2 can be seen in Figure 2.1, which prints “0 1 2 3 4” to standard output.

```plaintext
while (x == 5) do
    print("{ } ", x)
    x = x + 1
end
```

Listing 2.2: A while loop

2.2.2 Parser

The AST is created in one of the first steps of compilation [1], by Encore’s parser. The parser takes as input source code, and identifies the different segments the code should be broken up into to produce the AST [1]. Consider a program that contains the while loop described in Listing 2.2. The parser recognizes the key word “while”, and enters into the sub-parser specific to a while loop. It begin by attempting to parse a conditional statement, then the keyword “do”, afterwards, the parser expects an indented body of expressions, and finally the keyword “end”. If the keywords are omitted, or misspelled, or if the indentation is incorrect or something unexpected is
parsed, then the parser generates an error \[1\]^10. The parser returns an AST that is then handed over to the next step of compilation, desugaring.

2.2.3 Desugarer

After the parser has converted the source code into an AST \([1]\), it translates AST nodes that are syntactic sugar, to other kinds of AST nodes. Syntactic sugar, a phrase coined by Peter Landin \([8]\), are expressions that are intended to make code easier to read and write without affecting the expressiveness of the programming language \([8]\). The expression \(x++\), for example, is common syntactic sugar for the less concise expression \(x = x + 1\). This translation process is called desugaring. Encore also contains syntactic sugar, which is removed in the desugaring step of compilation. Consider for example an if-then clause, it is syntactic sugar for an if-then-else clause, with an empty then-branch. Therefore, the Encore compiler desugares all if-then clauses into if-then-else clauses, as illustrated in Figure 2.2.

Figure 2.1: Simplified AST representation of a while loop

Figure 2.2: Desugaring of an if-then clause
The current desugarer does not manipulate the For AST nodes \[10\]. It is, however, important to understand the concept of desugaring as for comprehension is syntactic sugar for a combination of method calls that handle the iteration process. The desugaring of for comprehension is handled in an added step of compilation, and is vital to this implementation.

2.2.4 Type Checker

After the AST has been through the desugaring process, it is type checked. In this step, the compiler verifies that the type of an AST node matches the expected type \[1\], this is called static type checking. It checks for example that mathematical operators are only used on numerical arguments, and that functional arguments have the correct type. Certain types of AST nodes may contain different types, so beyond verifying the correct type of nodes, the type checker also adds typing information to AST nodes. Consider for example the while loop in Listing 2.3 that contains a conditional statement and has a body. The type checker verifies that the conditional statement results in a boolean type, and that the boolean statements compares two expressions of the same type. It also infers that the body of the while loop contains an expression of a numerical type and adds that typing data to the while loop’s AST node. \[10\]

```
var x = 5
while (x > 0) do
  x += -1
end
```

Listing 2.3: A while loop

2.2.5 The Non-Optimizing Optimizer

When the AST contains all its typing information, further improvements upon the code can be made. This is called the optimization step, however it is a misnomer. In the Encore compiler, the optimization step contains both optimizations and further manipulations of the AST that are both necessary for compilation, and is only possible after the AST has been given its typing information. In the implementation of for comprehension, an additional step of this kind will be added called Typed Desugaring. It is in this step that the new for comprehension is desugared into nested method calls. This desugaring can only be done when the typing information is available, but must be done before further compilation, as the rest of the compilation steps have no mechanisms to handle the For AST node.
Chapter 3

The New for comprehension, Why and How?

In this chapter, we explain the reasons for replacing the existing for loop with a for comprehension as well as describe how it is desugared into method calls, and its syntax.

3.1 Reasons for New for comprehension

Encore’s current for loop is a collection based loop [9]. This means that the programmer provides the collection, i.e. an array or a range, and provides a loop variable that takes on the value of each element in the collection as it is being iterated over [9]. An example of an Encore for loop can be seen in Listing 3.1; with each iteration the loop variable $x$ takes on the value of the string elements in the array. The Encore compiler translates the Encore for loop into a C for loop. However, unlike in the C for loop [4], the programmer does not have to provide the condition for continuation, a boolean statement that when false terminates the loop. This means that the current design prohibits programmers from making the fence-post-error [8], or the off-by-one error as it is also called, that can occur when programmers have to provide this condition themselves. The for loop is, however, limited. The new implementation will maintain the core of this design, while extending it into something more comprehensive: for comprehension.

```plaintext
for x <- [ "Katt", "Hund", "Host" ] do
  print("/| \n", x)
end
```

Listing 3.1: A for loop

The collections the for loop can iterate over are limited to arrays and ranges [10]. These data structures are as of this thesis, built in data structures [10]. The new for comprehension is intended to rectify this, by extending the current for comprehension into being able to iterate over all objects that have implemented specific methods. This would make the for comprehension more versatile as programmers could implement their own iterable collection objects, or use other data structures than only arrays or ranges from Encore’s standard library [3].
3.2 What is the New for comprehension?

The new implementation desugars the for comprehension into a combination of calls to `map`, `flatMap` and `foreach` [3]. These methods are used to apply some function to elements in collection data structures such as lists and arrays [7]. This desugaring extends the for comprehension to allow it to iterate over all objects that includes an implementation of these methods, making these methods a kind of interface for the for comprehension [3]. In accordance with Encore’s functional flavour [11], the methods `map` and `flatMap` return collections of elements. This means that the new for comprehension does not only support a body of side effects, but can also return a collection. Finally the combination of methods `flatMap` and `map` allow the for comprehension to iterate over multiple collections in one call. Listing 3.2 shows a possible Encore for comprehension, that has a return value stored in the variable `retCollection`, and that iterates over multiple lists in one call. More examples of the kinds of for comprehensions that are supported by the new implementation can be found in section 3.3. What follows are more detailed descriptions of these methods.

```scala
var retCollection = for x <- listX, y <- listY do
  x + y
end
```

Listing 3.2: A for comprehension

The method `map` takes a collection, such as a list or an array, and applies a function to each element and returns a collection with the contents altered by the function. Thus given a collection data structure of type `L`, with inner elements of type `A`, and a function `f` that returns a type `A`, `map` works in the following way [11]:

\[
L[A].map[f: A => A]: L[A].
\]

`L` calls on method `map` that applies the function `f` to every element in `L` and returns a collection data structure of the same type `L`, but now with the elements altered by the function `f`. It is important to note that the structure is preserved. Given a structure of type `L`, it will return a structure `L`. Using `map`, however, on a multiple structures in one call, would return nested structures. Consider for example, that there were two structures of type `L`, with elements of type `A`, if `map` was used, it would return nested structures `L[L[A]]`. A solution to this is to use `flatMap`.

Given multiple data structures of type `L`, with elements of type `A`, `flatMap` flattens the data structures using concatenation to return a single, non-nested data structures `L[A]`. `flatMap` is used on all data structures that are to be iterated over, except the last one, that uses a simple `map` [7]:

\[
L1[A].flatMap(f: A => L2[A].map(f: A => A)): L[A]
\]

In this case `L1` calls on `flatMap`, which contains a function, where `L2` calls on `map`. `map` returns a collection for every element in `L1`. `flatMap` concatenates these collections to in turn, return a single collection. Let us look at an example of a translation from an Encore for comprehension into calls to `map` and `flatMap`. Listing 3.3 is an example of a for comprehension. For every element in `arrayListA`, it iterates over the entire `arrayListB`. The body of the loop is the expression `a + b`, and the resulting collection of this for comprehension is stored in `arrayListC`. For simplicity’s sake, assume that both `arrayListA` and `arrayListB` contains
elements of type \texttt{int}. The equivalent Encore code, using \texttt{map} and \texttt{flatMap} can be seen in Listing 3.4.

```encore
var arrayListC = for a <- arrayListA, b <- arrayListB do
    a + b
end
```

Listing 3.3: A for comprehension

```encore
var arrayListC = arrayListA.flatMap(arrayListB.map(a + b))
```

Listing 3.4: Desugared for comprehension

Here it is important to note, that when iterating over multiple collection data structures using \texttt{map} and \texttt{flatMap}, the collection data structures must be of the same type. The elements they contain can be of different types, but the containers have to be the same type. This is because these are methods, that are part of a class. As such \texttt{flatMap} expects a function that returns a collection of the same type. Therefore a LinkedList collection, for example, that calls on the method \texttt{flatMap}, expects a method that yields another LinkedList, not an array. Another limitation of \texttt{flatMap} and \texttt{map} is that they cannot be used by ranges. This is because these methods return collections, that may or may not have different inner types, than the collections it iterated over. A range of strings, or a range of objects are not possible as a range in Encore only supports int-type elements. For this reason the \texttt{foreach} method is also required.

Not all for comprehensions have a return value that is being used, or can be used. The \texttt{foreach} method works similarly to the \texttt{map} method: a collection data structures calls on \texttt{foreach} to apply some function \texttt{f} on all of its elements. This method is relevant when the only matters of interest are the side effects. As \texttt{foreach} only returns a \texttt{unit} type$^1$ it can be used on ranges, and in cases where the return value is not used.

Finally the new for comprehension does not support any stepping functionality. In the current for loop, it is possible to declare a value that determines how many elements it will skip for every iteration. If, for example, the for loop contains a "by 2", statement, then the for loop will iterate over every other element. Instead of this old stepping functionality, a future version of the for comprehension can support a filtering function so that the body of the for comprehension is only applied to the elements that pass through the filter. That is, however, outside of the scope of this thesis.

### 3.3 Correctness

Here is a sample of some of the for comprehensions that should compile, and are used to test the implementation, a full list can be found in Appendix B. These test that the new for comprehension can handle objects, arrays and ranges, that it can handle a for comprehension as a function call, that the return collection have the correct type and so on.

```encore
for x <- [1 .. 10] do
    linklistInt.append(x)
end

var retList = for x <- ["1", "2", "3"], y <- ["a", "b", "c"] do
    x.concatenate(y)

```

$^1$unit type in Encore is equivalent to C's \texttt{void}
for i <- range do
  for j <- ["y", "x"] do
    print("{}{} ", i, j)
  end
end

Listing 3.5: For comprehensions that should compile and run as expected

However, there are also several scenarios that are not allowed to compile. Listing 3.6 is not a valid for comprehension as 23 is an integer, and not a valid collection that has implemented map, flatMap or foreach.

var range = for x <- 23 do
  x
end

Listing 3.6: Illegal collection

Another illegal use of the for comprehension is one with different kinds of collection data structures in the header, as in Listing 3.7, where the header contains a range and an array. All the collections must be of the same type although they may contains elements of different types.

for x <- [0..10], y <- [1, 2, 3] do
  print("{}", x + y)
end

Listing 3.7: Illegal header
Chapter 4

Design Challenges

The new for comprehension was based on a previous attempt \cite{6} to implement a more comprehensive for comprehension. In that attempt the for comprehension was desugared into a combination of calls to the methods `map`, `flatMap` or `foreach`, which inspired this implementation. This is also similar to the for comprehension in Scala, which also uses this kind of desugaring \cite{7}. These methods are used to apply some function to elements in collection data structures such as lists and arrays. The previous attempt however failed to account for certain features of Encore that do not allow for mutable variables to be updated inside a new closure if they have been declared outside it. In other implementations of for comprehension of this kind, such as in Scala, the for comprehension body may not contain updates of mutable variables \cite{7} at all. The goal of this implementation is to extend the for comprehension such that it does not remove previous functionality. To that end certain specific problems caused by the design of the implementation of the new for comprehension have to be addressed, including this update problem. What follows are descriptions of these problems and an overview of their solution. The specifics of how these solutions are implemented can be found in section 5.5.

4.1 Boxing Allows Updates

A variable may not be updated inside of a new closure. Unless there is a solution, then Listing 4.1 would not be possible, as the variable \texttt{x} would not be allowed to update inside of the for comprehension.

\begin{verbatim}
var x = 0
for y <- [1, 2, 3] do
    x += y
end
\end{verbatim}

Listing 4.1: A for comprehension that requires boxing

A solution to the update problem is called boxing. Boxing is done by creating a box, which is a new instance of a class allocated to the heap, and putting the value of the variable that is being updated inside of it. Instead of updating the stack variable, the variable inside of the box is updated. After the for comprehension has terminated, the boxes are unboxed. This is done by updating the variable outside of the for comprehension, with the value inside of the box. Listing 4.2 illustrates how Listing 4.1 is made possible through the boxing of the variable \texttt{x}.

\begin{verbatim}
var x = 0
var box_x = new Box(x) [int]
\end{verbatim}
for y <- [1, 2, 3] do
    box_x.value += y
end
x = box_x.value

Listing 4.2: For comprehension with boxing

Boxing solves the update problem, however there are more obstacles that the implementation has to overcome.

4.2 Break Inside A Method

break is a built in function in Encore that can be used inside of a loop to force it to terminate [11]. It is however not allowed by the type checker to be used inside of a method or function, for a good reason. Given a function, for example, that is supposed to return a value, if a break is used to halt the execution of the function, then no value is returned, and the typing of the method is violated. This becomes an obstacle for the new for comprehension, which is a looping construct that should be able to handle break, as it is desugared into method calls, which are not.

A solution to this is to add a new kind of method, similar to foreach, called maybeForeach. This method returns the type Maybe unit. If the for comprehension terminates normally, then it returns Just unit, however, if it executes the break instruction, then it returns Nothing. There has to be this distinction between normal termination and break termination, as the new for comprehension supports iteration of multiple collections in a nested fashion. This means that if one level of nested maybeForeach has been returned the value of Nothing, then all other levels must also halt their execution. Therefore desugaring all break inside of a for comprehension into return Nothing combined with the use of maybeForeach solves the break problem.

4.3 Ranges - The Unique Collection

Ranges in Encore are different from other kinds of collection data structure, in that their elements are not inserted, but rather a range is defined in its instantiation. In Encore a range contains three integer values: the starting value, the terminating value, and a step value and are exclusively used to to be iterated over using a for loop [10].

In Encore, a range is also a built in data type, and there is no external library that connects to ranges, that would allow manipulation or additions of functionality [10]. As for comprehensions are desugared into method calls in the new implementation, the method foreach must be available for the desugaring process, but as there is no library for ranges, this is not possible. To circumvent this issue, a new range class can be created that contains an implementation of the foreach method. In the desugaring step of compilation, all ranges can then be desugared into an instance of this new class. This can be done without fear of influencing other parts of the compiler as ranges are exclusively used with for loops [11].
Chapter 5

Implementation

The new for comprehension is being implemented by making changes to the Encore compiler. These changes are made to expand the current for loop into for comprehension, which allows it to iterate over more kinds of collection data structures than only the built-in array and range, yield a return collection, and iterate over multiple collections in one call to the for comprehension. The implementation requires changes to the AST, the parser, the type checker and the addition of new steps to the compiler. What follows are descriptions on how and why these changes were necessary.

5.1 Extending the AST

The Encore compiler uses an intermediate representation of the source code in the form an abstract syntax tree (AST). Each node in the abstract syntax tree represents an expression in the source code, such as an assignment, a method call, or a for comprehension. As such, each expression has a corresponding data structure.

The data structure of the for loop is called For, and Listing 5.1 contains the Haskell data structure of the current For. It has four parts that store different data about the for loop. Meta data about the for loop is stored in emeta, the name of the loop variable representing the current element of a collection being iterated over is stored in name, the number of steps taken with every iteration in the collection is stored in steps, and the collection being iterated over is stored in src, finally the body of the for loop is stored in body. This data structure can contain all the relevant information about the for loop which can only have one loop variable and one collection.

```
For (emeta :: Meta Expr,
    name :: Name,
    step :: Expr,
    src :: Expr,
    body :: Expr)
```

Listing 5.1: Data structure of For AST node

The new for comprehension replaces the old for loop’s For data structure. Because it can iterate over multiple collections, the new data structure must be able to contain multiple loop variables, and containers. In the for comprehension, the number of collections and their corresponding loop variables is not fixed, and are therefore stored in a separate data structure called ForSource and For contains a list of ForSource, whose size is determines at compile time. Listing 5.2 shows
the new data structure of the For AST node in the Encore compiler, written in Haskell. When compared to the current For there are a few key differences: instead of containing one src, it contains a list of ForSource, where the names of the loop variable has changed from name to fsName and from src to collection. Another difference is that the new data structure of For also contains a Maybe type. This is where the type of the loop variable is stored, based on the type of the collection’s elements, and is further discussed in section 5.4 Type Checker.

```
For {
  emeta :: Meta Expr,
  sources :: [ForSource],
  body :: Expr}

ForSource {
  fsName :: Name,
  fsTY :: Maybe Type,
  collection :: Expr}
```

Listing 5.2: Data structure of new For

The new AST data structure For represents the new syntactic possibilities of the for comprehension’s header to contain multiple collections and their corresponding loop variables. This data structure is filled with its data values by the parser, as it parses the source code.

5.2 Parser

The parser translates the source code into an AST representation of the program. The data structure for a for comprehension, For, is returned by the parser when it has completed parsing of the entire for loop. As discussed in the previous section, the new for comprehension is syntactically different from the for loop and the For parser must accommodate these differences. Most notably is that a for loop contains exactly one collection and one loop variable in the header, while a for comprehension possibly contains multiples of these. Therefore the new For parser also has a special ForSource parser that parses all the comma separated loop variables and their respective collections and return these as children AST nodes to For. Listing 5.4 shows the For parser, and which part of the parser parses which parts of the example in Listing 5.3.

```
for = blockedConstruct $ do
  emeta <- buildMeta
  reserved "for"
  sources <- option [] $ (commaSep getForSource)
  reserved "do"
  return $ \body -> For{emeta, sources, body}
where
  getForSource = do
    forVar <- Name <$> identifier
    let forVarType = Nothing
    reservedOp "<-"
    collection <- expression
```

Listing 5.3: For comprehension example
return ForSource {forVar, forVarType, collection}

Listing 5.4: For comprehension’s parser

5.3 Desugarer

In this compilation step syntactic sugar \[8\] is removed from the AST. Encore contains two built in collection data structures: ranges and arrays. For comprehension requires ranges to be manipulated using the method `foreach`, which is not possible to add as there is no standard library for ranges. Thus, included in the changes made to the Encore compiler to allow the new for comprehension to function, all ranges are desugared into calls to the constructor of the class `RRange` \[1\] which is a range class defined in Encres standard library. This makes it possible to iterate over ranges using for comprehension, as `foreach` has been define inside of the `RRange` class.

Desugaring of the for comprehension is done at a latter step, as it requires typing information which is added to the AST in the next compilation step.

5.4 Type Checker

The purpose of the type checker is to ensure that the source code follows all of Encore’s typing rules, and to add typing information to each node in the AST, returning a so called typed AST \[10\]. Type checking the for comprehension needs to do both of those things; it needs to check that none of the typing rules specific to for comprehension are violated and to add the correct typing information to the For AST node. The latter determines what kind of desugaring is done in the following compilation step. The Encore type checker contains a type checking function `doTypecheck` that pattern matches AST nodes with their corresponding type checker, and returns a type checked node.

5.4.1 Type Checking for comprehensions

There are two typing rules specific for the for comprehension. The first is in the case of for comprehensions iterating over more than one collection, all collection types must be the same, modulo element types. It cannot for example handle a for comprehension that iterates over both an array and a range. If any are of a different type then an error is generated stating that "Collections are not of the same type, they must all be a CC collection", where CC is the type of the first collection.

The second rule specific to the for comprehension, is that the collections must be iterable. All arrays and ranges are iterable, but the rest of the objects must be checked to see that they have implemented the trait `Functor` which is the trait that works as an interface for iterability. `Functor` requires a class to have implemented methods `map`, `flatMap` and `foreach`. This is because in the next step of compilation, the For AST node is desugared into method calls to the previously mentioned methods.

Consider for example Listing 5.5 whose header contains two collections of different types, an array and a linked list. This violates the first typing rule, and the type checker catches this by comparing all the collections’ types to the first collection’s type. If there is a miss match, then an error is thrown.

---

1In the Encore compiler, "Range" is a protected keyword. Therefore the range class had to be named with double r’s: RRange
for valA <- [1, 2, 3, 4], valB <- linkedListA do
x = valA + ValB
end

Listing 5.5: For comprehension that generate error

5.4.2 Adding typing data to AST node

Type checking the for comprehension i.e. the AST-node of For, can be split up into three steps: type checking each collection being iterated over, extracting the parametric type of each collection, and type checking the body while adding the extracted typing data to the environment of the body.

The collections being iterated over are also themselves AST nodes that can be type checked. Therefore, in the first step, these are type checked by using the the typing information returned by calling the doTypecheck function on them, and adding that data to the For AST node. In this step the collections traits are also checked, that they have implemented the Functor trait as described in the previous section. This part of type checking is uncomplicated.

The second step is more complicated because there are different ways of extracting the parametric data from arrays that are built in objects, and other kinds of objects. Therefore this step has three different branches for if it is an array, range or other kind of object. In this step the objects are also checked to ensure that they have implemented the trait Functor, which is the interface for all iterable objects. After this check is done the extracted parametric type of the collection, together with the collections corresponding looping variable is returned to be used in the type checking of the body of the loop.

Finally the body of the loop is type checked. Similarly to the collections, the body is its own AST node and can be type checked using the function doTypecheck in a recursive fashion. However, in order for that function to work properly it needs the correct environmental data. To this end the looping variables, and their type, i.e. the parametric type of their corresponding collection, must be added to the environment of the body. Therefore the previously extracted parametric type, together with the correct looping variables are added to the body’s environment, and then the body is type checked using the same method as for the collections described above.

These three steps are used to add the correct typing information to the different parts of the For AST node. At this step of compilation it is not known whether the for comprehension is expected to return a collection or not. In the type checker it is therefore assumed that the for comprehension will, unless it is iterating over ranges, and the return type of the for as a whole is a collection of the same type as those in the header, with elements of the same type as the body of the for comprehension, otherwise the return type is unit which is Encore’s equivalent to C’s void.

5.4.3 Type checking again

In the desugaring process, several new AST nodes are created, such as method calls for example. As these are new they do not contain the correct typing information. For this reason, the entire AST is once again put into the type checker for this information to be added. In a few places the type checker actually makes changes to the AST which results in it failing the second type checking. One such example is the type checking of Abort. Listing 5.6 shows part of Abort’s type checker. It first checks that Abort contains the correct number of arguments. If there are no arguments
or too many, then the type checker throws an error, as seen on lines three and four of Listing 5.6. When the type checked `Abort` is returned, as seen on line 5, `Abort`’s list of arguments is empty. This means that when the AST is type-checked again, `Abort` will contain the wrong number of arguments, and the type checker will throw an error.

```haskell
1 doTypecheck abort@Abort{args} = do
2   let expectedTypes = [sty] 
3   unless (length args == length expectedTypes) $
4   tcError $ WrongNumberOfFunctionArgumentsError
5   return $ abort{args=([],::[Expr]))}
```

Listing 5.6: Part of type checker of `Abort`

The particular problem of `Abort` is an easy one to solve, partly because `Abort`’s type checker is relatively small and uncomplicated, however similar issues exists with type checking a `Match Clause` twice, which is much more complicated. There is currently a goal to add modular type checking to Encore which would relieve this issue, therefore solving this for `Match Clause` is considered outside of the scope of this thesis.

5.5 Desugaring the Typed AST

After the parsing step of compilation, the AST is transformed by the desugaring step of compilation. It is, however, not possible to desugar the for comprehension into method calls at that stage of compilation. This is because different types of for comprehensions are desugared differently, this typing information is only available after the type checking step of compilation. Therefore another desugaring step is added after the type checking step, to the Encore compiler: Typed Desugaring. In this step the `For` AST node is desugared into method calls, the mutable variables that are updated inside of the for comprehension are boxed and any use of break is desugared into returns. The following sections will discuss how this is implemented, and it is recommended that you read chapter 4, because the implementation of the solutions to the problems described there are discussed in this section.

5.5.1 Desugaring `For` into map, flatMap and foreach

The desugaring process of the `For` AST node can be split into three steps: identifying which methods the `For` should be desugared into, building the new method calls’ AST nodes and finally nesting the method calls together, as described in section 3.2.

In the first step of desugaring the `For`, its typing information is used to identify what methods it will be desugared into. For comprehensions fall into two categories, for comprehensions with only side effects, and for comprehensions which returns a collection of elements. For comprehensions that have only side effects are desugared into the method calls to the method `foreach`, as this method applies a function to each element in a collection and returns unit, Encore’s equivalent to C’s `void` [11]. These include for comprehensions whose results are not stored anywhere and all for comprehensions that iterate over ranges, as in Listing 5.7.

```haskell
var range = [0 .. 9]
for x <- range do
  print x
end
```

Listing 5.7: for comprehension without result
for comprehensions that are not ranges and whose return value are stored or used somewhere, are desugared to combinations of map and flatMap. An example of this can be seen in Listing 5.8.

```haskell
var listA = new LinkedList<int[]
var listB = new LinkedList<int[]
var newList = for a <- listA, b <- listB do
    toString(a + b)
end
```

Listing 5.8: For comprehension with result

The second and third step involves building the method calls and nesting them together. In order to understand how a method call is built and nested, three things will be explained: method calls, how the nesting is structured and what a closure is and used for. Just like the For AST node, method calls also have their own AST node, with its own structure, appropriately named MethodCall. This node has four distinct fields, TypeArguments that contains the potential type arguments required, target that contains the name of the object being acted upon, name that contains the name of the method being called and finally args that contains the arguments of the method. Listing 5.9 show the data structure of MethodCall.

```haskell
MethodCall { typeArguments :: [Type],
    target :: Expr,
    name :: Name,
    args :: Arguments}
```

Listing 5.9: Data structure of MethodCall

In the desugaring process, the argument of a method call is either another method call, or the body of the for comprehension. If there are multiple collections being iterated over, then the method calls are nested as argument of each other with the most nested method call having the body of the for comprehension as its nested function as illustrated in Listing 5.10, which is the combinations of map and flatMap that Listing 5.8 is desugared into.

```haskell
var list = listA.flatMap(listB.map(toString(a + b)))
```

Listing 5.10: Use of map and flatMap

The body of the for comprehension can be considered a function with the loop variables its parameters. In order to make a method call, with a nested function, or method as its arguments, the body is put inside of a closure, with the loop variables as arguments of the closure. A closure in Encore also has its own AST node, called Closure. A Closure contains three distinct field, eParams which contains the parameters, mty which contains the return type of the closure, and a body, which in this case contains the nested methods. The data structure of Closure can be seen in Listing 5.11.

```haskell
Closure { eparams :: [ParamDecl],
    mty :: Maybe Type,
    body :: Expr}
```

Listing 5.11: Data structure of Closure

A method call is built by setting the collection being iterated over as the target of the method call, and then putting the loop variable, and the body or the nested method call, inside of the closure, and adding that closure as an argument to the method call being built. Listing 5.12 illustrates which parts of the for comprehension in the source code are part of which parts of the desugared For, note that some parts of the code have been simplified or omitted.
var list = new LinkedList()[int]
for x <- list do
    print x
end

MethodCall {typeArguments :: [Type],
    target = list,
    name = Name "foreach",
    args = [Closure {eparams :: [x],
        mty :: unit,
        body :: print x}]}

Listing 5.12: How for comprehension is desugared

5.5.2 Break: Desugaring For into maybeForeach

There is a third category of methods that for comprehensions can be desugared into, that is a method called maybeForeach. This method exists in order to ensure that the new for comprehension can still perform like the current for loop, in that it can use break. break is a function in Encore that may only appear in loops, and when executed, terminates the loop. The for comprehension is desugared into method calls, where a break is not allowed, therefore a special method is made. A break in a for comprehension is desugared into a return Nothing. The method maybeForeach, works exactly like foreach, except that it returns a type Maybe unit instead of unit, and when it receives a Nothing it immediately halts execution and also returns Nothing, thereby forcing the loop to terminate. Listing 5.13 is an example of an implemented maybeForeach for ranges in Encore. A maybeForeach that receives a Nothing when applying the function, must also return it as it might be one among multiple nested calls to maybeForeach, otherwise the loop will not terminate correctly.

def maybeForeach(f : int -> Maybe unit) : Maybe Unit
    var current = this.start
    var res = Just unit
    while (current <= this.stop) do
        res = f current
        if res == Nothing then
            return Nothing
        end
        current += this.step
    end
    return Just unit
end

Listing 5.13: A maybeForeach implementation

The desugaring implementation has two steps: checking for break in the body of the For, and desugaring them into return Nothing. First the body of the for comprehension is traversed to see if it contains break in order for it to desugar the For into the correct method calls. If the body does contain break, then all break are replaced by return Nothing.

5.5.3 Boxing

Boxing is an answer to the update problem caused by the use of closures. As described above, desugaring For involves closures, and a variable on the stack cannot be updated inside of a closure if has been declared outside of it. An update means that
an already declared variable is assigned a new value, as in Listing 5.14. A more
detailed description can be found in section 4.1

variable = newValue

Listing 5.14: An update

Assignments have their own AST node in the Encore compiler, therefor all variables being updated can be identified by parsing the body of the For and searching for assignment nodes. This can not, however, be done blindly. If a variable is updated that has been declared inside of the For body, then boxing it outside of the For is unnecessary, and could potentially change the behaviour of the for comprehension. The first step of boxing is therefore to get the set of all updated variables that are not local to the for comprehension.

The values of these variables are thereafter boxed, as in Listing 5.15 inside an object called MutBox, which is part of Encore’s standard library, shown in 5.16. The class MutBox contains one field called value, which stores the current variable value. MutBox implements parametric polymorphism allowing it to contains a value of any type, thereby not limiting the for comprehension to only be able to update certain types of mutable variables.

var __box_variable = new MutBox(variable)

Listing 5.15: A variable declaration

local class MutBox[t]
    var value : t
    def init( v : t) : unit
        this.value = v
    end
end

Listing 5.16: Class MutBox

All of the references to variables being updated, inside of the for comprehension, are changed to refer to the value inside of the box, as shown in Listing 5.17. This circumvents the update problem as the value is now boxed on the heap, instead of on the stack. All occurrences of these variables have to be changed in order to preserve the behaviour of the for comprehension, and not only in the variable updating.

__box_variable.value = newValue

Listing 5.17: Reference to value in box

After the for comprehension has terminated, the value inside of the MutBox is un-boxed. This is done by updating the original variable with the new value inside of the MutBox, directly after the for comprehension. Listing lst:simpleBoxing is an example of how a for comprehension, which updates a variable, is boxed using Encore source code.

From:

var variable = 0
for x <- [1, 1, 1] do
    variable = variable + 1
end

Into:

var variable = 0
val __box_variable = new MutBox[int](variable)
for x <- [1, 1, 1] do
    __box_variable.value = __box_variable.value + 1
Listing 5.18: How for comprehension is boxed

In the compiler variable declarations and their scope are represented in \textit{Let-in} clauses. A more accurate example of how boxing is represented in the compiler is therefore in Listing 5.19, where the scope of the variables declared in the \textit{Let-in} clause, referring to the boxes, only encompasses the for comprehension and the un-boxing.

From:

\begin{verbatim}
let
  variable = 0
in
  for x <- [1, 1, 1] do
    variable = variable + 1
  end
end
\end{verbatim}

Into:

\begin{verbatim}
let
  variable = 0
in
  let
    var __box_variable = new MutBox(variable)[int]
  in
    for x <- [1, 1, 1] do
      __box_variable.value = __box_variable.value + 1
    end
  variable = __box_variable.value
end
end
\end{verbatim}

Listing 5.19: Boxing

The boxing as described so far, does not include returning the collection produced by the for comprehension. To do this, a return\_collection variable is declared and is updated by the for comprehension. In Encore there is an in-explicit return at the end of a sequence of instructions. Thus by setting the return\_collection variable as the very last line of the boxing, it is explicitly returned when the for comprehension is the last instruction, or the only instruction. This allows the for comprehension to be passed as an argument for example, as shown in Listing 5.20, where a for comprehension’s return array collection is passed as argument to the function \texttt{foo}, and it contains the side effect of updating a variable.

From:

\begin{verbatim}
var variable = 0
foo {for x <- [1, 2, 3] do
    variable += 1
    x += variable
  x, variable}
\end{verbatim}

Into:

\begin{verbatim}
let
  variable = 0
in
  foo(let
    return\_collection = null
  in
    let

Boxing variables that are being updated inside of the for comprehension’s body allows the for comprehension to update variables that have been declared outside of it. By making the final instruction of the boxing being the return collection, for comprehensions can be used for either pure side effects or to also return a collection, thus making it a more versatile looping tool than a for loop which only produces side effects.
Chapter 6

Evaluation of Implementation

6.1 Elegant Source Code

With the new for comprehension fewer lines of code are required to execute nested loops or to create new collections based on other collections. This is because the header of a for comprehension can contain multiple collections which are iterated in a nested fashion, and that a for comprehension can return a collection. Consider for example copying an array as shown in Listing 6.1. To copy an array using the current for loop requires declaring a new array before the for loop, knowing the length of the array that is being copied and using a range to index the array correctly, without making any off-by-one errors. With the new for comprehension none of this is necessary and copying an array, or any collection data structure for that matter can be done in three lines of code. This is because the for comprehension has an implicit yield, so that each element in the new collection contains the value of the last expression in the body of the for comprehension.

```
-- copying an array using for loop
var array = [1, 2, 3]
var copiedArray = [] |array|
for x <- [0 .. |array|-1] do
  copiedArray(x) = array(x)
end

-- copying an array using for comprehension
var copiedArray = for x <- [1, 2, 3] do
  x
end
```

Listing 6.1: Copying an array

Similarly traversing a matrix is made much simpler by using the for comprehension. Consider Listing 6.2 where printing each element in a matrix using for loops, requires nesting one for loop in another, but this is not necessary to achieve the same result using for comprehension. Traversing a matrix could be made even easier by creating a Matrix data structure that implemented the trait Functor in Encore’s standard library, so that it can be used in a for comprehension, using only one collection in the header, instead of two.

```
-- traversing and printing each element of a matrix using nested for loops
var matrix = [[0, 1], [2, 3]]
for i <- matrix do
  for j <- i do
    print("{}", j)
```

23
with the new for comprehension, expressing certain kinds of looping can be done
elegantly and simpler with fewer lines of code than by using for loops. This is great
for algorithms that require complicated list comprehensions for example.

6.2 Run Time

The new for comprehension may be used to express complicated looping more el-
egantly, but the implementation is more general which does not permit as efficient
compilation. Desugaring for comprehensions into nested closures with method calls,
allocated on the heap, was expected \[3\] to result in a performance overhead when
compared to the current for loop. A future improvement would be to put these al-
location on the stack to improve performance. To measure the overhead, a simple
program was written in Encore that looped over ranges, and first compiled with
the current version of Encore that uses for loops, and then compiled with the im-
plemented for comprehension. The executable files were then run, and their run
time measured with Ubuntu’s built in \texttt{time} command that can measure total CPU
time, and finally their results were compared. The results were not surprising, the
implemented for comprehension was about two times slower than the for loop, as
seen in Figure \[6.1\], a diagram of performance of for comprehension and for loop over
number of iterations. For comprehensions seems to be in the same complexity class
as (O(n)) loops, but with a constant close to two. To see the test program and the
raw data, please see Appendix C.

This result was an expected \[3\] consequence of using closures to nest the method
calls together. The C compiler, which creates an executable file from the compiled
Encore source code, can make certain loop optimization \[3\]. The for loop is translated
into a C for loop and can therefore be further optimized by the C compiler. The new
for comprehension, however, is not translated into a C for loop, but is desugared into
method calls and therefore cannot benefit from typical loop optimisations such as
unrolling. The run-time performance is the main trade-off of the for comprehension’s
implementation.

6.3 Interface of iteration

The new for comprehension allows developers to create their own iterable objects
by implementing the trait \texttt{Functor} from the Encore standard library. It requires
the methods: \texttt{map}, \texttt{flatMap} and \texttt{foreach}. The goal was for the interface to be
simple, which was mostly achieved as the previously mentioned methods are the
only required methods. The interface is, however, made more complicated by the
optional method \texttt{maybeForeach}, which is required to be able to use \texttt{break} inside
of a for comprehension.
The built-in function `break` only works within looping mechanisms, and not within methods. As the new for comprehension is desugared into method calls, the normal `break` function would not compile. The work around described in section 4.2 is to desugar all `break` into `return Nothing`, which requires a new version of the method `foreach`, that returns a `Maybe` type. `maybeForeach` returns a `Maybe Unit`, and requires the implementation to include a branching depending on the return result of applying the function, included as a parameter, to each element in the collection being iterated over. When a `break` is executed the function returns `Nothing`, and the `maybeForeach` should immediately halt execution and also a `return Nothing`. All this is to say that the implementation of these methods have certain requirements in order for for comprehension to work properly.

Therein lies the risk that comes with the for comprehension’s dependency on Encore’s libraries. If these methods, such as the `maybeForeach`, are incorrectly implemented, then the for comprehension could result in unexpected behavior. Because of this, the simplicity of the interface is paramount to help developers use for comprehension to their advantage when they create their own collection data structures.

There is a possibility though that future development of the for comprehension will mean adding required methods to the `Functor` trait, thereby making it harder for developers to implement. One possible feature that could be added for example is a filter feature, a conditional statement used on the collection to filter out which elements it should apply the loop body to. In the previous attempt to implement for comprehension in Encore, this was accomplished by having a `filter` method as part of the interface [6]. Although the future of Encore’s for comprehension may mean a more complicated interface for developers, the relationship between compiler and the standard library means that new features can be added to the for comprehension relatively easily. Therefore this relationship could be considered a feature, rather than a bug.

For this reason I believe that the future development of Encore’s for comprehen-
sion should emphasize developing Encore’s standard library with more collection data structures that implement the trait `Functor`. This would mean that developers could rely more heavily on Encore’s standard library for their for comprehension needs, as the for comprehension gains more features and possibly adds required methods to `Functor`. Thereby developers are less likely to make mistakes, and can reap all the benefits of for comprehension, without having to pay the price of implementing a complicated trait.

6.4 Added compilation steps

As the for comprehension was desugared into method calls, a new kind of AST node, it needed to be type-checked again. This is because the type checking of a for comprehension is different from type checking a closure, a function call or a method call, which it is desugared into. As Encore currently has not implemented modular type checking, where it is possible to type check a single AST node, the whole AST had to be type checked again. The added compilation time was so minuscule that it could not be measured using the timing method of measuring run time. When modular type checking is implemented, this added compilation step can be removed.
Conclusion

The purpose of this thesis has been to extend the Encore programming language’s for loop into for comprehension, and to implement a simple interface Functor for iterable collection. A for comprehension iterates over collection data structures that implement Functor, and can iterate over multiple collections in a nested fashion, as well as return a collection. This is done by changing for comprehensions into combinations of method calls.

The for comprehension allows certain algorithms that require looping to be written more concisely, while maintaining readability when compared to the use of for loops. It does come with a cost though, as the implementation resulted in a performance overhead. The performance cost is reasonable trade-off, when considering that the for comprehension can iterate over all collection data structures that implement Functor, and not only arrays and ranges like the Encore for loop.

Although the trait Functor is simple in its current form, the future development of the for comprehension could mean complicating the trait, by adding more required methods. This does not have to be an obstacle, but can be considered an opportunity to also develop a standard library with more collection data structure that reliably implement Functor, and thereby allowing developers to reap the benefits of for comprehension, without having to pay the cost of complicated implementation.

This thesis’ for comprehension is ripe for future development, but is also a useful and powerful tool to Encore programmers in it own right.
Bibliography


Appendices

Appendix A: Data structure of Encore expressions

Every expression in Encore, weather it is an addition, a variable declaration or a for comprehension, has a unique intermediate representation in the Encore compiler, an AST node. Bellow in Listing 7.1 is the data structure of each node written in the programming language Haskell. Note that this includes the data structure of the new for comprehension including the ForSource data structure, and not the data structure of the for loop.

```haskell
data ForSource =
    ForSource { fsName :: Name,
                fsTy :: Maybe Type,
                collection :: Expr}
    deriving (Eq, Show)

data Expr = Skip {emeta :: Meta Expr}
            | Break {emeta :: Meta Expr}
            | Continue {emeta :: Meta Expr}
            | TypedExpr {emeta :: Meta Expr,
                        body :: Expr,
                        ty :: Type}
            | MethodCall {emeta :: Meta Expr,
                         typeArguments :: [Type],
                         target :: Expr,
                         name :: Name,
                         args :: Arguments}
            | MessageSend {emeta :: Meta Expr,
                          typeArguments :: [Type],
                          target :: Expr,
                          name :: Name,
                          args :: Arguments}
            | Optional {emeta :: Meta Expr,
                           optTag :: OptionalPathComponent}
            | AdtExtractorPattern {emeta :: Meta Expr,
                                   name :: Name,
                                   arg :: Expr,
                                   adtClassDecl :: ClassDecl}
            |ExtractorPattern {emeta :: Meta Expr,
                              name :: Name,
                              arg :: Expr}
            | FunctionCall {emeta :: Meta Expr,
                           typeArguments :: [Type],
                           qname :: QualifiedName,
                           args :: Arguments}
            | FunctionAsValue {emeta :: Meta Expr,
                              typeArgs :: [Type],
                              qname :: QualifiedName}
            | Closure {emeta :: Meta Expr,
```
eparams :: [ParamDecl],
mty :: Maybe Type,
body :: Expr}
| PartySeq {emeta :: Meta Expr,
par :: Expr,
seqfunc :: Expr}
| PartyPar {emeta :: Meta Expr,
pari :: Expr,
parr :: Expr}
| PartyReduce {emeta :: Meta Expr,
seqfun :: Expr,
pinit :: Expr,
par :: Expr,
runassoc :: Bool}
| Async {emeta :: Meta Expr,
body :: Expr}
| Return {emeta :: Meta Expr,
val :: Expr}
| MaybeValue {emeta :: Meta Expr,
mdt :: MaybeContainer }
| Tuple {emeta :: Meta Expr,
args :: [Expr]}
| Let {emeta :: Meta Expr,
mutability :: Mutability,
decls :: [([VarDecl], Expr)],
body :: Expr}
| MiniLet {emeta :: Meta Expr,
mutability :: Mutability,
decl :: [([VarDecl], Expr)]}
| Seq {emeta :: Meta Expr,
eseq :: [Expr]}
| IfThenElse {emeta :: Meta Expr,
cond :: Expr,
thn :: Expr,
els :: Expr}
| IfThen {emeta :: Meta Expr,
cond :: Expr,
thn :: Expr}
| Unless {emeta :: Meta Expr,
cond :: Expr,
thn :: Expr}
| While {emeta :: Meta Expr,
cond :: Expr,
body :: Expr}
| DoWhile {emeta :: Meta Expr,
cond :: Expr,
body :: Expr}
| Repeat {emeta :: Meta Expr,
name :: Name,
times :: Expr,
body :: Expr}
| For {emeta :: Meta Expr,
sources :: [ForSource],
body :: Expr}
| Match {emeta :: Meta Expr,
arg :: Expr,
clauses :: [MatchClause]}
| Borrow {emeta :: Meta Expr,
target :: Expr,
name :: Name,
body :: Expr}
| Get {emeta :: Meta Expr,
val :: Expr

| Forward {emeta :: Meta Expr,
forwardExpr :: Expr}
| Yield {emeta :: Meta Expr,
val :: Expr}
| Eos {emeta :: Meta Expr}
| IsEos {emeta :: Meta Expr,
target :: Expr}
| StreamNext {emeta :: Meta Expr,
target :: Expr}
| Await {emeta :: Meta Expr,
val :: Expr}
| Suspend {emeta :: Meta Expr}
| FutureChain {emeta :: Meta Expr,
future :: Expr,
chain :: Expr}
| FieldAccess {emeta :: Meta Expr,
target :: Expr,
name :: Name}
| ArrayAccess {emeta :: Meta Expr,
target :: Expr,
index :: Expr}
| ArraySize {emeta :: Meta Expr,
target :: Expr}
| ArrayNew {emeta :: Meta Expr,
ty :: Type,
size :: Expr}
| ArrayLiteral {emeta :: Meta Expr,
args :: [Expr]}
| Assign {emeta :: Meta Expr,
  lhs :: Expr,
  rhs :: Expr}
| VarAccess {emeta :: Meta Expr,
  qname :: QualifiedName}
| TupleAccess {emeta :: Meta Expr,
target :: Expr,
  compartment :: Int}
| Consume {emeta :: Meta Expr,
target :: Expr}
| Null {emeta :: Meta Expr}
| BTrue {emeta :: Meta Expr}
| BFalse {emeta :: Meta Expr}
| NewWithInit {emeta :: Meta Expr,
ty :: Type,
  args :: Arguments}
| New {emeta :: Meta Expr,
ty :: Type}
| Print {emeta :: Meta Expr,
  file :: FileDescriptor,
  args :: [Expr]}
| Exit {emeta :: Meta Expr,
  args :: [Expr]}
| Abort {emeta :: Meta Expr,
  args :: [Expr]}
| StringLiteral {emeta :: Meta Expr,
  stringLit :: String}
| CharLiteral {emeta :: Meta Expr,
  charLit :: Char}
| RangeLiteral {emeta :: Meta Expr,
  start :: Expr,
  stop :: Expr,
  step :: Expr}
Listing 7.1: Data structure of AST nodes

Appendix B: for comprehension Tests

In order to ensure the correctness of the implementation, it needs to be tested. Below in Listing 7.2 are all the written tests of the for comprehension. The tests are conducted by running the test file and comparing it to its corresponding expected output file. These test among other things, that the implemented for comprehension can handle:

- Looping in the style of the for loop, with ranges and arrays.
- Looping over collection data structures other than ranges and arrays
- Arrays using flatMap and map correctly
- Returning a collection
- Returning the correct collection
- Updating a variable correctly
- Multiple collections in the header
- Nested for comprehensions
- Passing a for comprehension as an argument to a function

```scala
import Collections.Mutable.LinkedList

given useList(list : LinkedList[String]) : unit
    for elem <- list do
        print("{}", elem)
    end
print("\n")
end

given active class Main
    def main() : unit
        var linklistInt = new LinkedList[int]()
```python
var linklistString = new LinkedList[String]()

var range = [0 .. 10]
var range2 = [0 .. 5]

var array = ["a", "b", "c", "d", "e"]
var array2 = ["1", "2", "3", "4", "5"]

for x <- range do
    linklistInt.append(x)
end
for x <- array do
    linklistString.append(x)
end

-- result in: 0 1 2 3 4 5 6 7 8 9 10
for x <- range do
    var k = 0
    print("{} ", x)
    print(k )
end

print("\n")

-- result in: 0 1 2 3 4 5 6 7 8 9 10
for x <- linklistInt do
    print("{} ", x)
end

print("\n")

-- result in: a b c d e
for x <- array do
    print("{} ", x)
end

print("\n")

-- result in: a b c d e
for x <- linklistString do
    print("{} ", x)
end

var acc = ""
print("\n")

-- result in: a1 a2 a3 a4 a5 b1 b2 b3 b4
-- b5 c1 c2 c3 c4 c5 d1 d2 d3 d4 d5 e1 e2 e3 e4 e5
var retList = for x <- array, y <- array2 do
    acc = x.concatenate(y)
    print("{} ", acc)
    acc
end

-- result in: e5
print("\n\n", acc)

-- result in: a1 a2 a3 a4 a5 b1 b2 b3 b4 b5 c1 c2 c3
-- c4 c5 d1 d2 d3 d4 d5 e1 e2 e3 e4 e5
for x <- retList do
    print("{} ", x)
end
```
print("\n")

for i <- range do
    for j <- ["y", "x"] do
        print("{}{} ", i, j)
    end
end

print("\n")

var acc2 = 0
for i <- range, j <- range2 do
    acc2 += (i + j)
end
print("{}", acc2)
print("\n")

useList(for x <- linklistString do
    x
end)

end

Listing 7.2: For comprehension tests
These tests should result in the output file of Listing 7.3.

Listing 7.3: Tests expected output

Appendix C: Testing Performance Program

The for comprehension was expected to result in a performance overhead. To compare the performance of the for comprehension with the current for loop the program in Listing 7.4 was written. It contains a for comprehension that can also compile as a for loop by the current Encore compiler. The resulting raw data of the tests can be seen in Table 7.1 and Table 7.2.

These tests were run on a Linux machine running Ubuntu 18.04, with an Intel Core i5, 2.80 GHz and 8 GB RAM.

local class Tester
    var num : int
    def init(value : int) : unit
        this.num = value
    end
    def add() : unit
        this.num += 1
active class Main
  def main() : unit
    var size = 2000000000
    var num = new Tester(0)
    for x <- [1 .. size] do
      num.add()
    end
    if (num.get() != size) then
      print("Not done! ", num.get())
    else
      print("Done!")
    end
  end
end

Listing 7.4: Program used to compare performance

<table>
<thead>
<tr>
<th>Iterations</th>
<th>for loop</th>
<th>for comprehension</th>
<th>Loop/Comp</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 000 000</td>
<td>0.21</td>
<td>0.4188</td>
<td>1.994285714</td>
</tr>
<tr>
<td>500 000 000</td>
<td>1.0245</td>
<td>1.9583</td>
<td>1.911469009</td>
</tr>
<tr>
<td>1000 000 000</td>
<td>2.0033</td>
<td>3.9036</td>
<td>1.948584835</td>
</tr>
<tr>
<td>1500 000 000</td>
<td>2.7927</td>
<td>5.0816</td>
<td>1.819601103</td>
</tr>
<tr>
<td>2000 000 000</td>
<td>3.6527</td>
<td>6.7774</td>
<td>1.855449394</td>
</tr>
</tbody>
</table>

Table 7.1: Total average time
### Iterations: 100 000 000

<table>
<thead>
<tr>
<th>For Loop</th>
<th>For Comprehension</th>
</tr>
</thead>
<tbody>
<tr>
<td>In User Mode [s]</td>
<td>In Kernel Mode [s]</td>
</tr>
<tr>
<td>0.201</td>
<td>0.004</td>
</tr>
<tr>
<td>0.201</td>
<td>0.004</td>
</tr>
<tr>
<td>0.21</td>
<td>0</td>
</tr>
<tr>
<td>0.201</td>
<td>0.004</td>
</tr>
<tr>
<td>0.189</td>
<td>0.005</td>
</tr>
<tr>
<td>0.199</td>
<td>0.008</td>
</tr>
<tr>
<td>0.204</td>
<td>0.013</td>
</tr>
<tr>
<td>0.21</td>
<td>0.009</td>
</tr>
<tr>
<td>0.215</td>
<td>0.005</td>
</tr>
<tr>
<td>0.203</td>
<td>0.015</td>
</tr>
</tbody>
</table>

### Iterations: 500 000 000

<table>
<thead>
<tr>
<th>For Loop</th>
<th>For Comprehension</th>
</tr>
</thead>
<tbody>
<tr>
<td>In User Mode [s]</td>
<td>In Kernel Mode [s]</td>
</tr>
<tr>
<td>1</td>
<td>0.004</td>
</tr>
<tr>
<td>1.007</td>
<td>0.012</td>
</tr>
<tr>
<td>1.009</td>
<td>0.016</td>
</tr>
<tr>
<td>1.006</td>
<td>0.012</td>
</tr>
<tr>
<td>0.999</td>
<td>0.024</td>
</tr>
<tr>
<td>1.015</td>
<td>0.008</td>
</tr>
<tr>
<td>1.005</td>
<td>0.024</td>
</tr>
<tr>
<td>1.024</td>
<td>0.032</td>
</tr>
<tr>
<td>1.006</td>
<td>0.025</td>
</tr>
<tr>
<td>0.996</td>
<td>0.021</td>
</tr>
</tbody>
</table>

### Iterations: 1000 000 000

<table>
<thead>
<tr>
<th>For Loop</th>
<th>For Comprehension</th>
</tr>
</thead>
<tbody>
<tr>
<td>In User Mode [s]</td>
<td>In Kernel Mode [s]</td>
</tr>
<tr>
<td>1.987</td>
<td>0.008</td>
</tr>
<tr>
<td>1.978</td>
<td>0.013</td>
</tr>
<tr>
<td>1.953</td>
<td>0.036</td>
</tr>
<tr>
<td>1.98</td>
<td>0.013</td>
</tr>
<tr>
<td>1.971</td>
<td>0.016</td>
</tr>
<tr>
<td>1.969</td>
<td>0.036</td>
</tr>
<tr>
<td>1.998</td>
<td>0.017</td>
</tr>
<tr>
<td>1.984</td>
<td>0.028</td>
</tr>
<tr>
<td>2.022</td>
<td>0.016</td>
</tr>
<tr>
<td>1.98</td>
<td>0.028</td>
</tr>
</tbody>
</table>

38
Iterations: 1500 000 000

<table>
<thead>
<tr>
<th>for loop</th>
<th>for comprehension</th>
</tr>
</thead>
<tbody>
<tr>
<td>In user mode [s]</td>
<td>In kernel mode [s]</td>
</tr>
<tr>
<td>3,097</td>
<td>0,008</td>
</tr>
<tr>
<td>2,901</td>
<td>0,012</td>
</tr>
<tr>
<td>2,712</td>
<td>0,02</td>
</tr>
<tr>
<td>2,7</td>
<td>0,036</td>
</tr>
<tr>
<td>2,687</td>
<td>0,048</td>
</tr>
<tr>
<td>2,732</td>
<td>0</td>
</tr>
<tr>
<td>2,721</td>
<td>0,02</td>
</tr>
<tr>
<td>2,73</td>
<td>0,008</td>
</tr>
<tr>
<td>2,729</td>
<td>0,012</td>
</tr>
<tr>
<td>2,722</td>
<td>0,032</td>
</tr>
</tbody>
</table>

Iterations: 2000 000 000

<table>
<thead>
<tr>
<th>for loop</th>
<th>for comprehension</th>
</tr>
</thead>
<tbody>
<tr>
<td>In user mode [s]</td>
<td>In kernel mode [s]</td>
</tr>
<tr>
<td>3,649</td>
<td>0,004</td>
</tr>
<tr>
<td>3,643</td>
<td>0</td>
</tr>
<tr>
<td>3,639</td>
<td>0,012</td>
</tr>
<tr>
<td>3,631</td>
<td>0,016</td>
</tr>
<tr>
<td>3,656</td>
<td>0,012</td>
</tr>
<tr>
<td>3,64</td>
<td>0,016</td>
</tr>
<tr>
<td>3,639</td>
<td>0,008</td>
</tr>
<tr>
<td>3,64</td>
<td>0,004</td>
</tr>
<tr>
<td>3,65</td>
<td>0,004</td>
</tr>
<tr>
<td>3,656</td>
<td>0,008</td>
</tr>
</tbody>
</table>

Table 7.2: Table of raw data