A Gleaming Tree

Implementing decision trees in the Gleam compiler

by

Harm van Stekelenburg

Student number: 852343871
Course code: IB9902
Thesis committee: Supervisor: dr. Tim Steenvoorden, Open University
Second assessor: dr. Freek Verbeek, Open University

Open Universiteit
de beste! www.ou.nl

CONTENTS

Acronyms

AST Abstract Syntax Tree. 1, 4, 16–21, 23

GHC Glasgow Haskell Compiler. 5

IR Intermediate Representation. 18

ABSTRACT

Gleam is a functional language that uses pattern matches for control flow. It can be com piled to Erlang and JavaScript. Gleam is designed to be a small and predictable language so a premium is placed on generating readable JavaScript and Erlang that closely matches the Gleam code after compilation [The Gleam team 2024].

Decision trees are a way of representing pattern matches inside a compiler. Allowing the compiler to find an efficient tree to determine the result of a pattern match at runtime. Using decision trees can improve the performance of programs that are compiled with decision trees [Maranget 2008]. If using decision trees in the Gleam compiler would significantly improve the performance of Gleam pattern matches after compilation, the decrease in readability of the produced code could be justified.

In this thesis we implement decision trees in Gleam, for a limited set of patterns. Using that implementation we compare the performance and size of code generated using decision trees, using two test cases. A program balancing red black trees and a program using bubble sort to sort a list. Based on the performance measurements we conclude that the performance of produced JavaScript code does not improve, and can even worsen. The produced code has a greater size than the standard Gleam compiler would produce. With these measurements we show that implementing decision trees in the Gleam compiler cannot be justified because of an increase in performance

1

GLEAM, DECISION TREES AND OPTIMIZATION

1.1. INTRODUCTION

Gleam, as a functional language, uses pattern matching for control flow. Its compiler can generate Erlang and JavaScript code. There are few optimizations applied when generating this output code. The Gleam compiler translates pattern matches to straight-forward if-else statements in JavaScript. It could be there are ways to compile to faster JavaScript code. One compiler technique to increase the performance of compiled pattern matches is decision trees.

Decision trees are a way to evaluate each value that can match a pattern only once. Finding the optimal decision tree is not trivial: it is not a decidable problem. There are multiple heuristics for finding trees that are (close to) optimal. Directly transforming decision trees to output code can result in very large output sizes. So decision trees are often combined with a way to merge identical parts of the decision tree [Maranget 2008].

When looking at decision trees in the Gleam compiler we will limit ourselves to a small number of patterns, just enough to get an impression of the impact of decision trees. We will only target JavaScript as an output target. Erlang has its own pattern matching optimizations. To combine Gleam and decision trees we formulate our research question: *What is the impact of implementing decision trees in the Gleam compiler on the performance and size of generated JavaScript code?*

1.2. CONTRIBUTIONS

To see the impact of implementing decision trees in the Gleam compiler we implement them in the compiler. We describe how Gleam's Abstract Syntax Tree (AST) is translated into JavaScript code both directly, with duplicated output, and indirectly, without duplicated output. We implement two different heuristics: `fdb` and `qba`.

We measure the size of the generated JavaScript code and compare the performance using two Gleam programs: balancing red-black trees and sorting lists using bubble sort. We see that the output size increases and the performance does not improve. We present the conclusion that decision trees are a bad fit for the Gleam compiler since they do not significantly improve the performance of generated JavaScript code but do increase its size.

We discuss possible reasons why the performance does not improve but have no definite conclusions why this is the case.

1.3. Structure

To answer the research question we first need some background. Decision trees are introduced in Chapter 2. A short introduction to Gleam is given in Chapter 3. After introducing Gleam, we also will provide some general background on compilers and specific details of the Gleam compiler in Chapter 3.

With this background we explain our implementation of decision trees in the Gleam compiler in Chapter 4. In Chapter 5 we measure the size of the generated JavaScript code and the performance of using decision trees in the Gleam compiler. Finally in Chapter 6 we answer the research question and discuss what the measurements from Chapter 5 mean.

1.4. Related work

Our research is based on the decision trees as described by Maranget [2008], they use this to optimize pattern matching in OCaml and test the performance of different heuristics. They show decision trees can increase the performance compared to the previous optimizations using backtracking automata and so implement decision trees in the OCaml compiler.

Decision trees are further build on by Baudon, Gonnord, and Radanne [2022]. They provide a different algorithm for creating trees, that allows for different memory representations of values and provide for more dynamic values as just constructors. This work could allow languages like Rust to apply optimized pattern matches. They mention two main options besides heuristics for selecting columns from a clause matrix: backtracking automata (a kind of depth first search) and minikanren based search (a kind of breadth first search). Without using different memory representations this thesis uses their algorithm to generate decision trees.

The Glasgow Haskell Compiler (GHC), a Haskell compiler, does pattern match optimizations as well but does it in a series of transformations. One example of a transformation is the *case-of-known-constructor* transformation that removes checks of patterns where it is already known which constructor it will be at compile time and so simplifies the transformed code. Another transformation is *case-elimination* removing cases that can be shown to be redundant [Peyton Jones and Santos 1998; Medeiros Santos 1995]. This series of transformations approach makes it hard for us to compare GHCs pattern matching optimizations to techniques like decision trees or backtracking automata.

Pattern matches in Erlang are also optimized (see Appendix C) but Erlang does not check for exhaustiveness in pattern matches. So implementing decision trees in Erlangs compiler would need extra functionality compared to decision trees as implemented in this thesis to handle match failures.

2

DECISION TREES

When compiling pattern matches there are many ways to structure the resulting (machine) code. In Listing 2.1 we see some code where we introduce a `Color` type that can be either a `Red` or `Black` variant. We then introduce a function `t` that uses a pattern match on three `Colors` and returns a number based on which pattern the `Colors` match.

The code of Listing 2.1 could be translated into the pseudocode of Listing 2.2. There we see `z` is evaluated thrice. The advantage of decision trees is that, when they are optimal, they allow each variable (or when a variable is composed of subterms each subterm) to be tested only once.

In this section we introduce introduce *pattern matching*, *clause matrices* (a concept used to transform pattern matches into decision trees), *decision trees* themselves, how to generate decision trees and ways to optimize decision trees so each subject will be tested only once. Finally we will show techniques for reducing the resulting (machine) code. All information on decision trees and examples used are based on [Maranget 2008] and the refinements of [Baudon, Gonnord, and Radanne 2022].

```
pub type Color {
  Red
  Black
}

pub fn t(x: Color,y: Color,z: Color) {
  case x, y, z {
    _, Red, Black -> 1
    Red, Black, _ -> 2
    _, _, Red -> 3
    _, _, Black -> 4
  }
}
```

Listing 2.1: Gleam example pattern match

```
let t x y z =
  if y == Red
    if z == Black
      return 1
  if x == Red
    if y == Black
      return 2
  if z == Red
    return 3
  if z == Black
    return 4
```

Listing 2.2: Naive translation of pseudocode

2.1. PATTERN MATCHING

We provide an explanation of pattern matching. A pattern match consists of several parts:

Subjects are the variables or expressions that the patterns in the pattern match will be matched against.

Patterns are what the subjects are matched against. What patterns can be depends on the type of the subject. If the subject is an algebraic data type the pattern could be a variant of that type. If the subject is an integer the pattern could be `1` for example. Alternative patterns are when a subject is matched against multiple patterns and if any on of the matches the match succeeds. When an integer is matched against 1 or 2 to match this is written as `1 | 2`.

Wildcards are a subtype of pattern that will match any value the subject has.

Clauses are a collection of patterns that together provide a pattern per subject.

Actions are the results that will occur when all patterns in a clause match.

It is important to note only one action will ever result from a pattern match. This will be the first clause, from the top, that matches the values of the subjects. Even if more clauses would match. Pattern matching is *exhaustive* when for any values the subjects can have a clause will always match. Compilers can guarantee this and the Gleam compiler guarantees the exhaustiveness of all pattern matches.

An example of a pattern match, taken from [Maranget 2008] with minor changes, is Listing 2.1. It shows three subjects: `x`, `y` and `z`. The patterns used are either wild cards (written as `"_"`) or one of the two variants of `Color`. There are four clauses, so there are four associated actions: the pattern match will result in an integer between 1 and 4. It is exhaustive, else it would fail to compile.

2.2. CLAUSE MATRICES

Recall the goal of decision trees is to allow us to test each subject only once. Pattern matches can be turned into decision trees more easily if we represent them as matrices. This allows

x	y	z	Action
_	Red	Black	1
Red	Black	_	2
_	_	Red	3
_	_	Black	4

Table 2.1: Clause matrix for the pattern match in Listing 2.1

us to use columns of patterns when deciding when to test a subject. One way of representing pattern matches as tables are *clause matrices*. Clause matrices have a column per subject and a special column for actions. Each clause and its associated action will result in a row of patterns and one action in the matrix. An example is Table 2.1: one row per clause in the original pattern match. One column per discriminant and a special column for the actions.

2.3. Decision trees

Like all trees decision trees consist of nodes and edges. There are several kinds of nodes:

Unreachable nodes are leaf nodes. They indicate that the pattern match does not cover the path that ends up in this node. When the tree is based on an exhaustive pattern match this will never happen at runtime.

Action nodes are the other kind of leaf nodes in decision trees. They indicate the action to take if a path in the tree ends up in this type of node.

Switch nodes identify which subject, or which subterm of a subject, will be tested. This can also be seen as which column in a clause matrix is picked, since each column identifies a subject.

Or nodes are ignored in this thesis. They are sometimes used to create clearer trees when alternative patterns are used in a clause.

Edges specify *cases*: what path to take depending on the value of the tested subject. In the code from Listing 2.1 the possible cases for a `Color` subject would be `Black`, `Red`, or a *default case*. Default cases indicate what path to take if no other case matches the subjects' value. For an example of a decision tree based on the pseudocode in Listing 2.2 see Figure 2.1. Note how on the path from the root to `2 y` is tested twice. We can do better. Section 2.6 will explain why this is a graph and not strictly a tree.

When evaluating a pattern match we walk through the tree based on the subject, or one of its subterms, as indicated by the nodes and edges. Ending up with an action in one of the leave nodes.

2.4. Generating decision trees

The algorithm that creates decision tree takes as an input a clause matrix and emits nodes and their children. It is a recursive algorithm. When recurring the algorithm will generate modified clause matrices.

We use matrix $\mathscr{P}$ as an input for the algorithm:

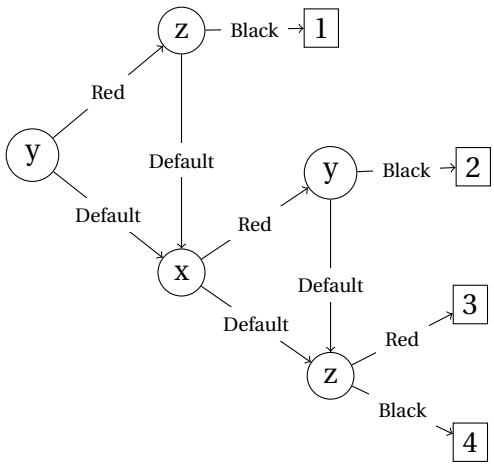

Figure 2.1: Decision tree based on Listing 2.2

1. If $\mathscr{P}$ has no rows emit a *Unreachable* node.

2. If the patterns of the first row consists only of wild cards emit a *Succes* node with the action of that row.

3. If rows contain alternative patterns (like `Red | Black`), add a row for each pattern in place of the original row with just that pattern in the matrix and continue.

4. Construct a *Switch* node.

Constructing a *Switch note* is done in several steps:

1. Select a column, i, of the clause matrix to start from. How to select a column is explained in section 2.5.

2. Determine needed cases. Every non-wild card pattern in the column contributes a case, unless it would be a duplicate. If the cases do not cover every value $subject_i$ can have, a default case is added.

3. For each case build a decision tree based on a modified clause matrix. How to modify the matrix is explained below.

4. Construct the switch node by specify which subject is checked in the node and add a sub tree per case, labeling the edge with the case and using the tree build for it in the previous step.

Modifying a clause matrix based on a case is done in two operations:

Specialization is the first operation. It removes all rows where the pattern in the chosen column i does not match the case. For example if the case is the `Red` variant of `Color` all rows with `Black` in position i would be removed. Wild card patterns are kept. For the default case only rows with wild card patterns in position i are kept.

y	z	Action
Black	_	2

y	z	Action
Red	Black	1
_	Red	3
_	Black	4

Table 2.2: Clause matrix for the `Red` and default case

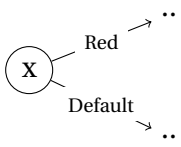

Figure 2.2: Root of the decision tree based on Table 2.1 when picking column 1

Expansion is the second operation. It replaces the subject of column i with its subterms and replaces the patterns in position i with their subpatterns. If there are no subterms or the default case is used the entire column is removed. For example if the tuple `x` is the subject, `Tuple` is the case and the pattern is `#(Red, Black)` (This is how a tuple is written in Gleam). The new subjects would be `x.0` and `x.1` and the new patterns would be `Red` and `Black`. Making all the rows in the new matrix one pattern longer.

In, for example, Table 2.1 we see there are rows, the first row is not only wild cards and there aren't any alternative patterns so we have to construct a switch node. We select the first column. The only pattern we see is `Red` so we add `Red` to the cases. This is not all the values subject `x` can have so we add a default case. That means we need two new matrices: see Table 2.2. Resulting in the partial tree in Figure 2.2.

2.5. Optimizing decision trees

The goal of decision trees is to produce code based on pattern matches where each subject is evaluated once. Depending on what column is picked in the algorithm from Section2.4 different trees are generated. Generating all the trees and selecting the most optimal one could be extremely burdensome for the compiler. That is why we use *heuristics* to select a column.

Each heuristic assigns a *score* to a column. The column with the highest score is picked each time. When composing heuristics any ties produced by the first heuristic are broken by the second heuristic and so on. Some example heuristics are:

First row or `f`. This heuristic only looks at the first row of the clause matrix. If position i in the first row is a wildcard pattern then column i scores zero, else it scores one.

In our example in Table 2.1 column one would get a score of 0 and the other two columns get a score of 1.

Small default or `d`. Scores a column by counting the wild card patterns in a column and negating that count as the score. So the fewer wild cards in a column the higher score it gets.

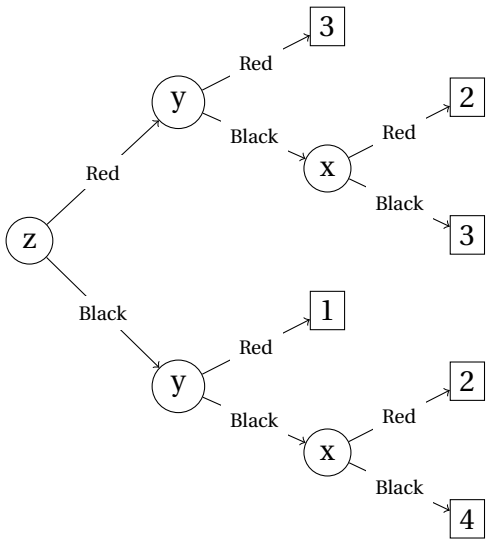

Figure 2.3: Decision tree generated using the `fdb` heuristic based on Listing 2.1

In our example in Table 2.1 column one would get a score of -3, column two gets a score of -2 and column three gets a score of -1.

Small branching factor or `b`. This heuristic counts the number of cases that the patterns in the column create and negates that count. The count is increased by one if the patterns in the column are not exhaustive i.e. there is a default case.

In our example in Table 2.1 each column would get a score of -2. If `Color` would have had three constructors the first column would keep its score of -2 and the other two columns would get a score of -3.

Arity or `a`. This heuristic is the negation of the sum of the arities of the set of constructors present in the patterns in the column. This means columns that will produce less columns in subsequent clause matrices are preferred, since each sub expression in a constructor will create a new column when transforming the matrix.

In our example in Table 2.1 all columns would get a score of 0 since both the `Red` and `Black` constructors have zero arguments.

Constructor prefix or `q`. This heuristic scores a column by counting the number of consecutive non-wild card patterns in a column. The idea being that earlier clauses in a pattern match will have a bigger impact since the first match is the one that will provide the action.

In our example in Table 2.1 column one would get a score of 0, column two a score of 2 and column three a score of 1.

`fdb` is considered a serviceable composition of heuristics to use, with `qba` being considered slightly better but somewhat more difficult to implement [Maranget 2008].

Using our example of 2.1 `fdb` generates the tree of Figure 2.3 we see there is no path in the tree where a subject is tested more than once.

2.6. SHARING SUB TREES

In Figure 2.1 we saw a decision tree that was not a tree but a graph. The procedure for generating trees as explained in Section 2.4 can generate trees with duplicated sub-trees. If we directly translate the generated trees to (machine) code this will lead to repeated code. When comparing code compiled with decision trees versus code compiled with different techniques measuring the output size is an important metric besides just the raw performance. When decision trees are compiled there are multiple techniques for deduplication one example is *hash-consing*. Like we will see in Section 4.2 in this thesis we will use a hash map to keep track of duplicated subtrees.

3

GLEAM AND ITS COMPILER

In this chapter we present a short description of Gleam and some of its notable features. We also need an understanding of compilers so we know how and where to implement decision trees. A high level overview of compilers will be the second section of this chapter.

Figure 3.1: Lucy: the mascot of Gleam

3.1. GLEAM

Gleam is a modern functional programming language designed for building reliable and scalable systems. Gleam has a fairly minimal syntax and aims to keep that syntax modern and familiar. Gleam does not have features such as type classes or currying. Gleam can generate code for both JavaScript and Erlang runtimes. Gleam emerged in 2019, and since then, releasing its 1.0 version on March 4th 2024. The language is actively developed [The Gleam team 2024]. Some examples to showcase the essence of Gleam. They are all based on Gleam contributors [2024].

Immutable Data The values assigned through an expression or statement are immutable, meaning the value contained in a variable cannot be changed. Same goes for lists, when trying to append an item to a list, the original list will not be mutated. Instead a new list will be created with the new item appended to it.

In Listing 3.1, when a Cat instance is created using the create function with the name "John" and a cuteness level of 5, the resulting cat instance is immutable. Subsequently, when the increase_cuteness function is called with cat as an argument, it returns a new Cat instance with the cuteness level incremented by 1. However, the original cat instance remains unchanged, showcasing immutability in Gleam.

```gleam
pub type Cat {
  Cat(name: String, cuteness: Int)
}

pub fn create(name: String, cuteness: Int) -> Cat {
  Cat(name: name, cuteness: cuteness)
}

pub fn increase_cuteness(cat: Cat) -> Cat {
  Cat(name: cat.name, cuteness: cat.cuteness + 1)
}

pub fn main() {
  let cat = create("John", 5);
  // => Cat(name: "John", cuteness: 5)
  let res = increase_cuteness(cat);
  // => Cat(name: "John", cuteness: 6)
  // cat is still: Cat(name: "John", cuteness: 5)
}
```

Listing 3.1: Gleam immutability

Pattern Matching Pattern matching is a Gleam feature for destructuring values and control flow. This is a necessary pre-condition for usable algebraic datatypes, for example where the return value of a function can be a result or an error, which are expressed together in a single return type. An example of destructuring lists is seen in 3.2. Lists can be destructured into a head, the first element, and tail, a list containing all other elements.

```gleam
describeList xs = case xs of
  []        -> "This list is empty"
  [a]       -> "This list has 1 element"
  [a, b]    -> "This list has 2 elements"
  [head, ..tail]   -> "This list has more than 2 elements, \
  \ and 'tail' contains all but the first"
```

Listing 3.2: Gleam pattern matching destructuring

Gleam uses pattern matching for all control flow. In the first pattern in Listing 3.3 we see the only place `ifs` can appear in Gleam, this is called a *guard* and allows for boolean checks that are hard to represent in patterns. The underscore is used to represent a *wildcard*, a pattern that will match any value. Patterns are evaluated top-down and the first match determines the result of a pattern match.

```gleam
fn greet(id: Result(Int, String)) {
  case id {
    Ok(x) if x < 0 -> "Hello negative stranger!"
    Ok(0) -> "Hello, Alice!"
    Ok(1) -> "Hey there, Bob!"
    Error(error) -> string.concat(["Uh looks like a problem: ", error])
    _ -> "Hello, stranger!"
  }
}
```

Listing 3.3: Gleam pattern matching

Function Composition A great asset of functional languages is the construction of more complex functions from simpler one, see Listing 3.4.

```gleam
fn double(x) {
  x * 2
}

fn square(x) {
  x * x
}

fn double_then_square(x) {
  square(double(x))
}
```

Listing 3.4: Gleam function composition

Gleam contains a bit of syntactic sugar to compose functions more elegantly: pipelines. The first argument to a function in a pipeline is always the result of the previous function or value. The code in Listing 3.5 is equivalent to the code in Listing 3.4

```gleam
fn double_then_square(x) {
  x
  |> double
  |> square
}
```

Listing 3.5: Gleam pipelines

Type safety Gleam is a type-safe language, meaning the compiler enforces strict adherence to data types. Variables, functions, and expressions will only operate on values that are of the appropriate type. This strong typing can be inferred by the compiler, without the need of explicit type annotation. In the following example `a` and `b` must be of the same type in order to use the + operator, since it is only defined when the operators are both integers. If we wanted to add two floats we would need to use `+.`.

```gleam
fn add(a, b) {
  a + b
}
```

Listing 3.6: Gleam type inference

Gleam users can define their own types, as seen with the `Cat` type in Listing 3.1.

Multiple compiler targets Gleam compiles to BEAM byte code or JavaScript. This means the language is quite portable, with a standard library that is provided for all targets. It provides interoperability with the ecosystem of libraries of the final runtime with the `@external` annotation.

```gleam
@external(erlang, "rand", "uniform")
pub fn random_float() -> Float

@external(javascript, "./my-handwritten-module.js", "run")
pub fn run() -> Int
```

Listing 3.7: Gleam external function

3.2. COMPILERS

To implement decision trees we need to modify the Gleam compiler. A *compiler* is a piece of software that takes source code and transforms it into a new output format, often machine code for a specific machine. Currently the Gleam compiler, written in Rust, has two output formats, one is JavaScript code and the other is Erlang bytecode (.beam files: run by the BEAM virtual machine).

A compiler can consist of several different parts, often a *backend* and a *frontend*. Where the end product of a compiler frontend is an Abstract Syntax Tree (AST), or syntax tree, the backend transforms this AST and outputs some target-machine code from the source code [Aho et al. 2007]. More extensive subdivisions can be made, see Figure 3.2. All these different phases together are known as a *compiler pipeline*.

The Gleam compiler does not implement all these phases. It transforms a character stream into a token stream, creates a syntax tree based on that, then transforms this syntax tree into a tree where type information is incorporated and finally that tree is transformed into either Erlang or JavaScript code.

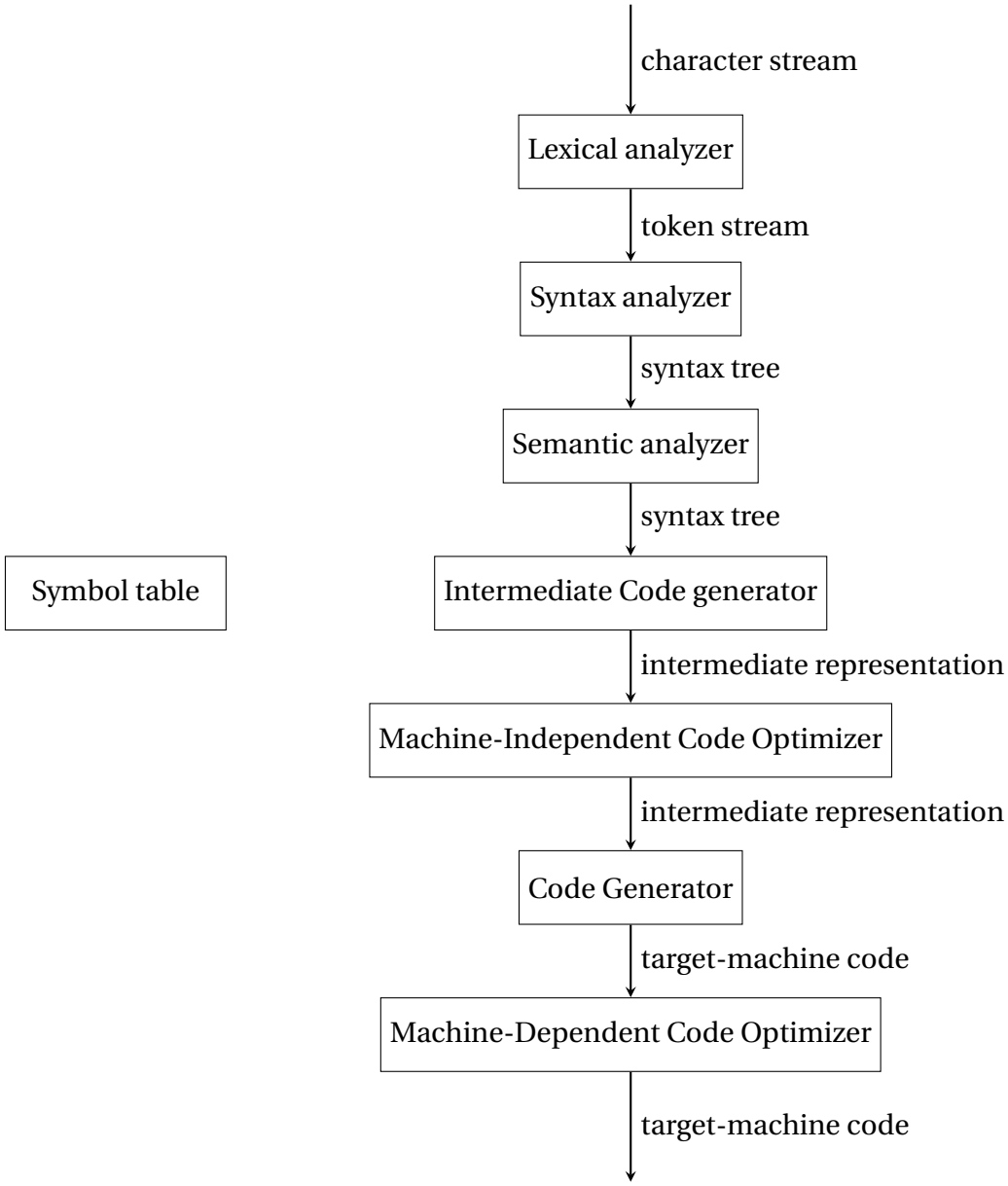

Figure 3.2: Compiler phases [Aho et al. 2007, p. 5]

3.2.1. Abstract syntax trees

To implement decision trees in the Gleam compiler we need to transform the AST in the Gleam compiler. An AST is a representation of the source code as a tree, and is the end product of the syntactic and semantic analysis phases in the compiler, see Figure 3.2. A very simple example would be transforming `1+2*3` into a tree like in Figure 3.3.

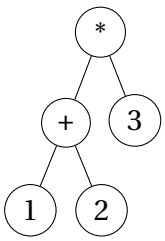

Figure 3.3: An example Abstract Syntax Tree

Compilers generate an AST to have a representation of the source code in a form that closely matches the grammar, or syntax, of the source code [Nystrom 2021, p. 65]. Let's look again at the example of `1+2*3`, a syntax tree abstracts over trivial differences like whitespace: `1 + 2 * 3` will have the same tree difference. When implementing new features in a compiler working with ASTs allows us to ignore these trivial differences.

ASTs also encode the semantics of the language, it allows `1+2*3` to be represented with the correct precedence rules for doing addition and multiplication: `1+(2*3)`. So syntax trees allow us to abstract over the concrete representation of the source code and more directly represent the syntactic or semantic meaning of the source code. The compiler is not limited to a number of transformations, in the running example all nodes from Figure 3.3 could be transformed by the compiler to a single node: 7.

Other uses of ASTs are that they allow for semantic analysis, including type analysis [Aho et al. 2007, p. 8-9]. Type analysis is an important part of semantic analysis in the Gleam compiler, since Gleam is a typed language. *Symbol tables* act in concert with ASTs but provide enriching information. The table can keep track of such things as the the type, value or location of a constant and keeping that information connected to its identifier in the source code [Aho et al. 2007, p.85-86]. There is no explicit symbol table in the Gleam compiler, type information is contained within an AST.

All of this makes ASTs a form of Intermediate Representation (IR). In the context of this thesis this means we can pin down the goal of the resulting artifact as a translation of one type of IR to another; the already existing AST in the Gleam compiler to another tree that includes a representation of decision trees. Then a lightly modified backend can generate the resulting JavaScript code. See Figure 3.4

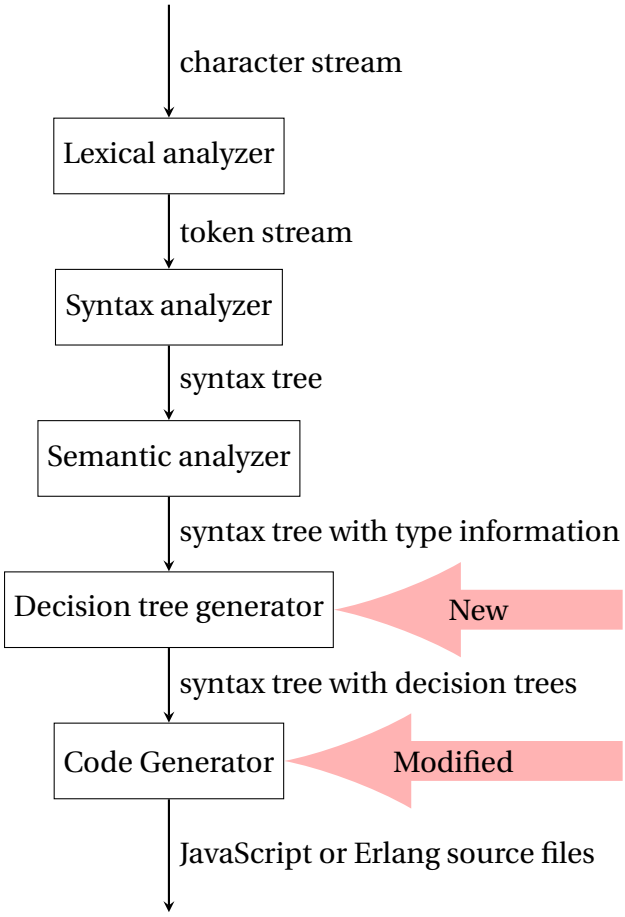

Figure 3.4: Compiler phases in the Gleam compiler. Changes to implement decision trees are indicated.

3.2.2. Specifying ASTs

To describe the implementation of decision trees in the Gleam compiler we need a way to describe possible ASTs, so we can design an AST with decision trees and then implement it in the Gleam compiler. This grammar is a *formal grammar*. An example is seen in Listing 3.8 Formal grammars show how possible expressions (on the left hand side) can be formed by combining other expressions or terminals. Terminals are expressions that need no other expressions to be created. Listing 3.8 shows what the grammar for a language that allows adding and multiplying integers, like in our running example, could look like.

We introduce some special notation for use of grammars. Text on the left of a `::=` specifies an an expression. The right hand side explains how that expression is formed. Alternatives of forming an expression are specified by using `|`. When there are many (or extensive) alternatives these often printed on separate lines. Expressions printed in *italic* require other expressions to be constructed. These other values and their names are specified between parentheses, they provide some clarity as to why expressions need other expressions to be constructed. They closely match the way expressions are implemented in the Gleam compiler: namely as fields of structs or enums in the Rust programming language. A `+` after an expression means one or more repetitions of that expression, `*` means zero or more. `1+2*3` is represented in this grammar as:

```
Calculation (op:  Plus, left_hand_side:  One, right_hand_side:
Calculation (op:  Multiply, left_hand_side:  Two, right_hand_side:
```

Three)).

If we want to implement decision trees in the Gleam compiler we can use a grammar to describe the transformation of one AST to another as we will see in the next chapter.

```
Calculation ::= Integer
              | Calculation (op: Operator,
                left_hand_side: Calculation,
                right_hand_side: Integer)

Integer     ::= Integer (value: Digit+)

Operator    ::= Plus | Multiply

Digit       ::= One | Two | Three
```

Listing 3.8: A basic grammar

4

Implementing decision trees in the Gleam compiler

Decision trees are a way to speed up compiled pattern matches, see Chapter 2 for details. To implement decision trees in the Gleam compiler we need to modify its compiler. The compiler takes the Gleam code and produces JavaScript or Erlang code, as seen in Section 3.2. In this chapter only the JavaScript output is discussed. For why only JavaScript is considered see Section C

We make two changes to the compiler, the first is a transformation of the AST to another AST that contains representations of decision trees. We provide the grammar for this new tree. The second change is a modification of the compilers backend to output the modified AST as JavaScript. We extend this second modification to generate deduplicated decision trees, this reduces outputted code size because each unique decision subtree is only translated into JavaScript once. The implementation can be found on github: `https://github.com/Harmful-Alchemist/gleam/tree/decision_trees_dedup`.

4.1. Transforming the Abstract Syntax Tree

The grammar for pattern matches in Gleam, as found in its compiler, is shown in Listing 4.1. For the notation used see Section 3.2.2. Each expression represented in this grammar is expressed as a enum variant in the Rust compiler code. In this thesis we only look at discard, variable, constructor and list patterns from this grammar. This is enough to get a first impression of the performance of decision trees within Gleam. We leave other types of patterns and clause guards out of scope.

Discard patterns are wildcard patterns. Variable patterns are like discard patterns but bind the value of the discriminant to the variable name assigned so the value can be used in the corresponding clause. An example of constructor patterns is seen in Listing 2.1. There we match on the different constructors of `Color`. Of list patterns we use only a subset, we match an empty list or a list with a head and a tail. See the last pattern in Listing 3.2 for an example. There we also see a variable pattern used to bind the head of the list to `x`.

```
CaseExpr        ::= CaseExpr (type: Type, subjects: Expression+,
                    clauses: Clause+)

Clause          ::= Clause (main: MultiPattern, alt: MultiPattern*,
                    guard: ClauseGuard, then: Expression)

MultiPattern    ::= MultiPattern (patterns: Pattern+)

Pattern         ::= IntPattern
                  | FloatPattern
                  | StringPattern
                  | VariablePattern
                  | VarUsagePattern
                  | AssignPattern
                  | DiscardPattern
                  | ListPattern
                  | ConstructorPattern
                  | TuplePattern
                  | BitArrayPattern
                  | StringPrefixPattern
```

Listing 4.1: Gleam pattern matching syntax

The grammar in Listing 4.2 is a short-hand to represent the implementation in the modified compiler. Based on Baudon, Gonnord, and Radanne [2022] we keep track of not just the actions but also the subjects so when we need a sub-expression we have the intermediate representation of that sub-expression available. We also keep track of the environment so bindings to variables can be generated in the action when generating the outputted code.

```
PatternMatrix       ::= PatternMatrix (subjects: Expression*,
                        pattern_rows: MultiPattern*,
                        actions_with_environment: ActionEnvironment*)

ActionEnvironment   ::= ActionEnvironment (action: Expression*,
                        bindings: Binding*)

Binding             ::= Binding (name: String, value: Expression*)
```

Listing 4.2: Clause matrix syntax

Based on the clause matrices we generate decision trees with the syntax from Listing 4.3. Here again each expression corresponds to a new enum or struct in the compilers code. We do not use `or` nodes in our tree (see Section 2.3), when generating the clause matrix we generate additional rows for alternatives. Columns to create the new clause matrix when generating the tree are selected with the heuristics `fdb` or the heuristic `qba` depending on a feature flag when compiling the modified Gleam compiler.

```
DecisionTree    ::= Unreachable
                  | Success (branch: Expression, bindings: Binding*)
                  | Switch (discriminant: Expression, cases: CaseTree+)

CaseTree        ::= CaseTree (case: Case, tree: DecisionTree)

Case            ::= ConstructorEq
                  | List
                  | EmptyList
                  | Default

ConstructorEq   ::= ConstructorEq (name: String)
```

Listing 4.3: Decision tree syntax

4.2. Translating decision trees to JavaScript

JavaScript has no built-in facilities for pattern matching. The code of Listing 2.1 is straightforwardly translated to Listing 4.4 by the unmodified Gleam compiler. We expect we can improve the performance of the generated code by using decision trees.

```
// generated using: cargo run -- build --target=js
export function t(x, y, z) {
  if (y instanceof Red && z instanceof Black) {
    return 1;
  } else if (x instanceof Red && y instanceof Black) {
    return 2;
  } else if (z instanceof Red) {
    return 3;
  } else {
    return 4;
  }
}
```

Listing 4.4: Standard Gleam compiler output

4.2.1. Generating JavaScript output

With a transformed tree based on the grammar from Section 4.1, we can move to the next step in the compiler pipeline: code generation, the Gleam compilers backend. For more details on the Gleam compiler and the compiler pipeline see Section 3.2.

When generating the JavaScript code for each tree we mostly use the existing functionality of the Gleam compiler. Recall the structure of decision trees from Section 2 and their representation in the AST from Section 4.1. For `Success` nodes we make no changes to the generation of JavaScript. For each `Switch` node we generate if/else-if/else statements for each case.

Taking as our starting point Listing 2.1, the regular unmodified Gleam compiler pro-

duces Listing 4.4. Our modified Gleam compiler (using the `fdb` heuristics) produces Listing 4.5

```
// generated using: cargo run --features=decisiontree -- build \
// --target=js
export function t(x, y, z) {
  if (z instanceof Black) {
    if (y instanceof Red) {
      return 1;
    } else {
      if (x instanceof Red) {
        return 2;
      } else {
        return 4;
      }
    }
  } else {
    if (y instanceof Black) {
      if (x instanceof Red) {
        return 2;
      } else {
        return 3;
      }
    } else {
      return 3;
    }
  }
}
```

Listing 4.5: Output based on a decision tree

This way of translating decision trees to JavaScript does lead to code duplication when the decision tree has duplicated subtrees.

4.2.2. Deduplicating decision trees

The size of the generated JavaScript code can be reduced by outputting the generated code for each unique decision subtree only once. To only keep track of unique decision subtrees we derive or implement the `Hashable` trait for each enum or struct in the Rust code that represents a expression in our grammar. We can then use these expressions as keys in a hashmap. The values will be a unique number, that can be used as a label, for each (sub)tree.

We generate trees with a recursive algorithm. `Success` nodes, see Listing 4.3, will be inserted into the hashmap first and the last node inserted will be the starting node. This way each subtree will only occur in the map once. This approach is inspired by Filliâtre and Conchon [2006]. We then change the way the entire tree is represented in JavaScript instead of nested if-else statements we loop over a switch where each subtree is a case. A

variable outside of the loop(`sub_tree_label`) points to the current node in the tree. The loop label is numbered to account for nested pattern matched in the body of the actions of the original Gleam code. See Listing 4.6 for an example.

This is an attempt at working around JavaScripts lack of a `goto` construct. This way of achieving deduplicated output does lead to unintuitive and somewhat overly verbose output. Since we need `continue` and `break` statements the reduction in output size is limited.

This functionality can be turned off and on by using a feature flag when compiling the modified Gleam compiler.

```
// generated using: cargo run --features=decisiontree_switch -- build \
// --target=js
export function t(x, y, z) {
  let sub_tree_label0 = 9;
  pmloop0: while (sub_tree_label0) {
    switch (sub_tree_label0) {
      case 9: if (z instanceof Black) {
        sub_tree_label0 = 5;
        continue pmloop0;
      } else {
        sub_tree_label0 = 8;
        continue pmloop0;
      }
      case 8: if (y instanceof Black) {
        sub_tree_label0 = 7;
        continue pmloop0;
      } else {
        sub_tree_label0 = 6;
        continue pmloop0;
      }
      case 7: if (x instanceof Red) {
        sub_tree_label0 = 2;
        continue pmloop0;
      } else {
        sub_tree_label0 = 6;
        continue pmloop0;
      }
      case 6:
        {
          return 3;
        }
        break pmloop0;
      case 5: if (y instanceof Red) {
        sub_tree_label0 = 1;
        continue pmloop0;
      } else {
        sub_tree_label0 = 4;
```

```
        continue pmloop0;
      }
      case 4: if (x instanceof Red) {
        sub_tree_label0 = 2;
        continue pmloop0;
      } else {
        sub_tree_label0 = 3;
        continue pmloop0;
      }
      case 3:
        {
          return 4;
        }
        break pmloop0;
      case 2:
        {
          return 2;
        }
        break pmloop0;
      case 1:
        {
          return 1;
        }
        break pmloop0;
    }
  }
}
```

Listing 4.6: Deduplicated decision tree output

5

Performance of decision trees for Gleam

The ultimate goal of using decision trees is code that executes faster, when it contains pattern matches. In this section we verify whether decision trees result in faster code and compare the output size. It can be quite complicated to provide performance measurements. Due to variance between different executions and different operating systems and hardware having an influence [Maranget 2008]. The measurements vary between executions.

To exclude a specific JavaScript engine making the performance difference we test using V8, SpiderMonkey and Bun.

V8 The V8 JavaScript engine is widely used to execute JavaScript, for example in the Chrome browser [Google 2024].

SpiderMonkey The SpiderMonkey JavaScript engine is used in the Firefox browser [Mozilla 2024b]. SpiderMonkey was built using the instructions from Mozilla [2024a] and using its example configuration from an optimized build.

Bun Bun is a JavaScript all-in-one toolkit using the JavaScriptCore JavaScript engine. JavaScriptCore is used in the Safari browser [Bun 2024].

We add the code in Listing 5.1 to the end of the generated JavaScript and execute that file using V8, SpiderMonkey or Bun. `performance.now()` will return the time in milliseconds as a floating point number with up to microsecond precision [MDN 2024]. We execute a single warm-up run, to allow the JavaScript engine to optimize the code and disregard start-up costs.

All tests are run on an intel i7-1165G7 CPU with 4 cores running at 2.80GHz with hyper threading turned off using Ubuntu 24.04.

```
let min = 10000000000000000000000000;
let max = 0;
let avgs = [];
let warm_up;
let result;
for (var i = 0; i < 100; i++) {
  const start = performance.now();
  result = main();
  const end = performance.now();
  const time = end - start;
  if (i===0) {warm_up = time; continue;}
  avgs.push(time);
  if (time < min) min = time;
  if (time > max) max = time;
}
console.log(`${avgs.reduce((acc,x) => acc+x,0)/avgs.length},${min},${max},${warm_u
```

Listing 5.1: Performance measuring code

5.1. Balancing Red-black trees

Based on the example of complicated pattern matching in Baudon, Gonnord, and Radanne [2022], we use the Gleam program in Listing A.1 for testing the performance of the generated code using decision trees. This leads to the result as reported in Figure 5.1. The error bars show the minimal and maximum runtime of the generated code. Table 5.1 shows the same results, notice how the maximum measurements are not shown in the figure.

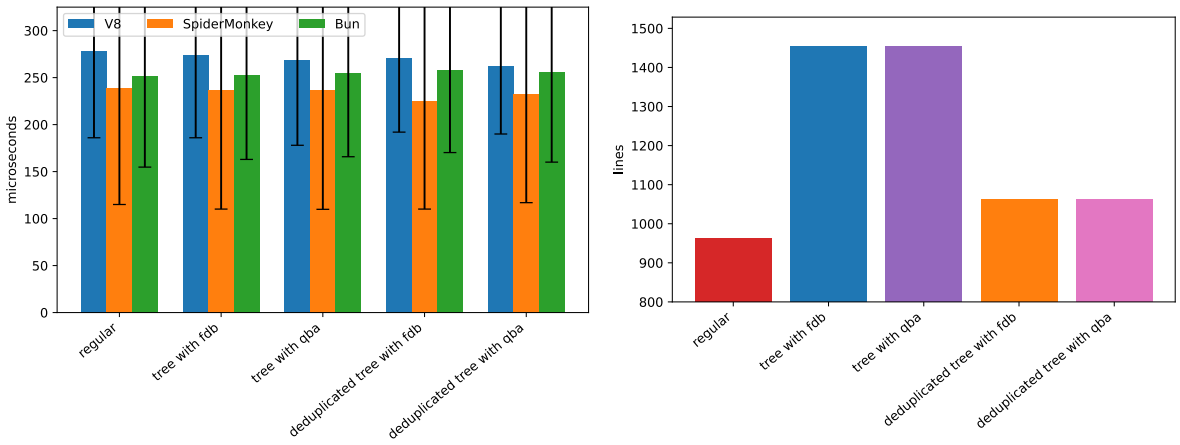

Figure 5.1: Generated JavaScript code performance for V8, SpiderMonkey and Bun and length for balancing red-black trees

5.2. Sorting with bubble sort

The second algorithm we use to test the performance of decision trees is bubble sort. By itself this is not a very efficient algorithm but it can be implemented using a large number

JS engine	Program	Warm-up	Average	Minimum	Maximum
V8	regular	2514.00	278.12	186.00	1024.00
	tree with fdb	2996.00	274.48	186.00	1137.00
	tree with qba	3035.00	269.05	178.00	1086.00
	deduplicated tree with fdb	2468.00	270.89	192.00	892.00
	deduplicated tree with qba	2432.00	262.69	190.00	926.00
SpiderMonkey	regular	1849.85	238.72	114.99	1484.86
	tree with fdb	1985.11	236.80	110.11	1656.98
	tree with qba	1925.05	236.90	109.86	1634.03
	deduplicated tree with fdb	2215.82	224.95	110.11	1182.86
	deduplicated tree with qba	1970.95	232.42	116.94	1311.04
Bun	regular	6364.42	251.23	154.77	715.32
	tree with fdb	6426.61	252.62	162.95	1015.01
	tree with qba	6605.37	254.79	165.82	1114.49
	deduplicated tree with fdb	5934.37	257.87	170.25	1532.75
	deduplicated tree with qba	6089.29	256.36	160.07	578.01

Table 5.1: Balancing red-black trees performance measurements in microseconds

of pattern matches. Therefore bubble sorting is a good candidate when testing the performance of compiled pattern matches. See Listing B.1 for the implementation and Figure 5.2 for the results. Table 5.2 shows the same results. Notice that this example ran in milliseconds as opposed to microseconds in the previous section.

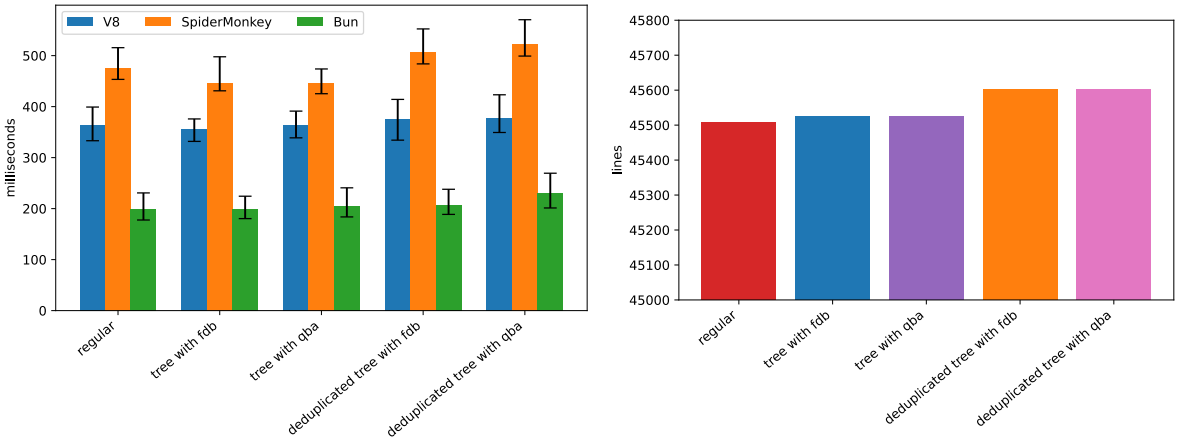

Figure 5.2: Generated JavaScript code performance for V8, SpiderMonkey and Bun and length for bubble sort

JS engine	Program	Warm-up	Average	Minimum	Maximum
V8	regular	398.91	364.01	333.11	399.17
	tree with fdb	385.00	355.77	331.80	375.85
	tree with qba	400.66	364.46	338.79	391.18
	deduplicated tree with fdb	400.94	374.65	334.24	414.15
	deduplicated tree with qba	400.88	376.59	349.26	423.20
SpiderMonkey	regular	496.11	476.03	453.40	515.57
	tree with fdb	450.01	445.90	431.00	497.73
	tree with qba	448.22	445.58	425.31	473.84
	deduplicated tree with fdb	519.40	506.77	483.80	552.36
	deduplicated tree with qba	536.79	523.51	499.11	570.41
Bun	regular	281.66	198.49	177.66	230.77
	tree with fdb	285.23	198.23	180.41	224.32
	tree with qba	286.35	205.47	183.72	240.77
	deduplicated tree with fdb	293.37	207.51	188.66	237.93
	deduplicated tree with qba	315.29	230.05	201.27	269.41

Table 5.2: Bubble sort performance measurements in milliseconds

6

Conclusions

To conclude this thesis we will answer the research question *What is the impact of implementing decision trees in the Gleam compiler on the performance and size of generated JavaScript code?* In all cases decision trees increase the size of the outputted JavaScript code. The attempts at deduplicating the decision trees can counterintuitively increase the size of generated JavaScript code even more as seen in Figure 5.2. The outputted code, on average, never performs significantly faster than the naive translation done by the original Gleam compiler.

On balance we conclude that decision trees are not a good fit for the Gleam compiler. The performance is disappointing. Since there is no general increase in performance for the JavaScript output it seems like there would be no performance improvement when using the technique for Erlang output, since the Erlang already optimizes pattern matches. An increase in performance could justify the work necessary to implement decision trees for clause guards and all pattern matches possible in Gleam, but now this justification is absent.

6.1. Discussion

Unclear is why the implemented decision trees do not significantly increase the speed. There are no obvious deficiencies in the generated JavaScript code. We have manually compared the generated decision trees to what is expected using the generation process seen in Chapter 2. The output of the generated JavaScript program is the same, whether it is compiled using decision trees or not. The lack of a `goto` statement in JavaScript makes reducing the size of the output very difficult, attempts to work around that lack result in a lot of code being generated to replace it.

Flamegraphs generated by the V8 JavaScript engine do not show clear differences when executing the code generated using decision trees versus using the originally generated JavaScript code. See Figure 6.1 for an example of a flamegraph of a single run. There is some variance between runs. The small differences shown are thus not an essential difference. The graphs show a small parsing step and then a call-out to C++ functionality. Unfortunately making it unclear whether the V8 engine uses heuristics that can speed up the original JavaScript code but not the less standard code generated using decision trees.

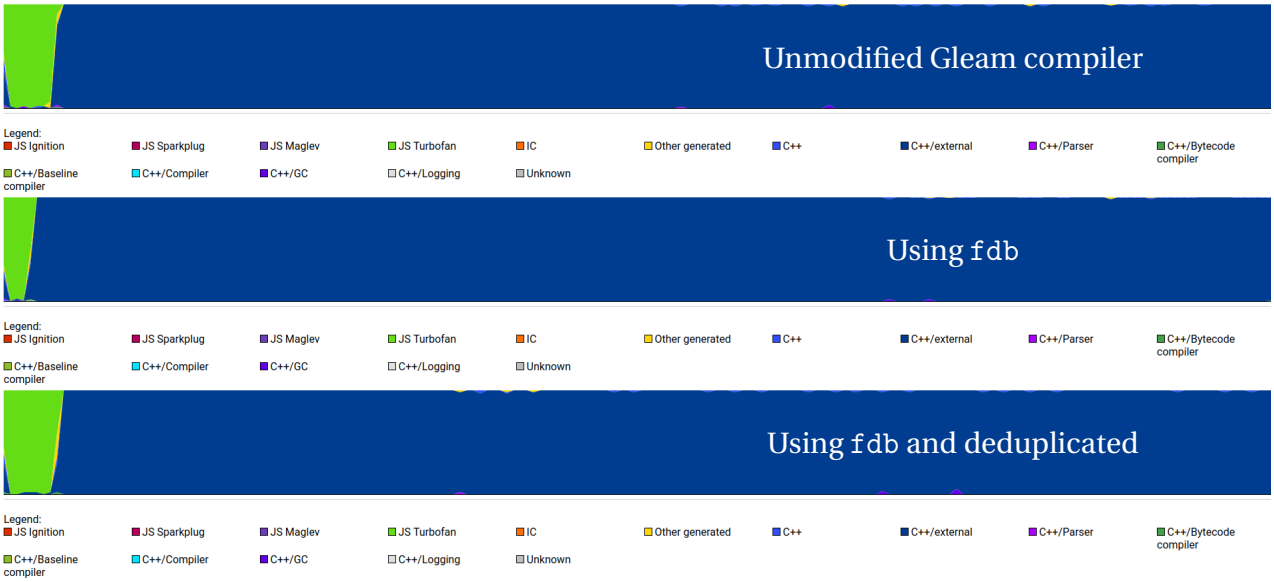

Figure 6.1: Generated JavaScript code performance for the regular Gleam compiler output, decision trees with the `fdb` heuristic and the deduplicated decision trees

The balancing of red black trees was perhaps not the best choice of algorithm for measuring performance. It might be too fast a sample, resulting in very large differences between the average run time and the sometimes seen much slower run times.

SpiderMonkey perhaps uses different optimizations, explaining the slight increase in performance. To really increase the speed of JavaScript code generated by the Gleam compiler a better approach would be researching what optimizations the different JavaScript runtimes make. Comparing that to the JavaScript code the Gleam compiler generates and finding places where optimizable patterns might be generated. Perhaps even targeting specific JavaScript runtimes. The trade-off there would be that the optimizations could change when new JavaScript runtime versions are released. The resulting JavaScript code would become less readable than the non-optimized version and after an optimization change possibly not even faster anymore.

6.2. Future work

We have shown decision trees do, in general, not increase the speed of the JavaScript output of the Gleam compiler. What remains unclear however is why exactly that is. Since this result is somewhat counter-intuitive it would be a good idea to research why this is the case. If a way is found to improve the performance of decision trees in Gleam based on the reason why the trees currently do not perform well it could be worthwhile to increase the number of patterns the current implementation handles. Incorporate clause guards into the trees and research whether the performance of the Erlang output could be improved as well.

ACKNOWLEDGMENTS

Many thanks and credit to Milco Kats for writing Section 3.1 together for an earlier version of what was then our thesis.

A

BALANCING RED-BLACK TREES CODE

Random generation of red black trees as Gleam code is done by a small Python script. All trees have a depth of two, since the the Gleam code only balances the root node. We use a custom `map` function since importing it from Gleam's standard library would mean that standard library code would be compiled with the modified compiler too, and we want our performance tests to be isolated to just our code.

```
pub fn main() {
  let cases = [
    #(
      Red,
      1,
      Node(Black, 1, Node(Black, 1, Empty, Empty), Node(Black, 3, Empty, Empty)),
      Node(Red, 1, Node(Black, 2, Empty, Empty), Node(Red, 3, Empty, Empty)),
    ),
    // .. 50 more randomly generated trees,
  ]

  map(cases, balance_tuple)
}

pub type Color {
  Red
  Black
}

pub type RBT(t) {
  Node(Color, t, RBT(t), RBT(t))
  Empty
}

fn balance(c, v, t1, t2) {
  case c, v, t1, t2 {
    Black, z, Node(Red, y, Node(Red, x, a, b), c), d
```

```
    | Black, z, Node(Red, x, a, Node(Red, y, b, c)), d
    | Black, x, a, Node(Red, z, Node(Red, y, b, c), d)
    | Black, x, a, Node(Red, y, b, Node(Red, z, c, d)) ->
      Node(Red, y, Node(Black, x, a, b), Node(Black, z, c, d))
    a, b, c, d -> Node(a, b, c, d)
  }
}

fn balance_tuple(x: #(Color, t, RBT(t), RBT(t))) {
  balance(x.0, x.1, x.2, x.3)
}

fn map(xs, f) {
  map_acc(xs, f, [])
}

fn map_acc(xs, f, acc) {
  case xs {
    [] -> acc
    [x, ..ys] -> map_acc(ys, f, [f(x), ..acc])
  }
}
```

Listing A.1: Gleam and a complicated pattern match

B

BUBBLE SORT CODE

Random generation of numbers as Gleam code is done by a small Python script. We use our own definition of numbers both to have more executions of pattern matches and since we did implement pattern matches on numbers. This last point is true for boolean pattern matches too, so we implemented our own.

```
pub fn main() {
  sort([
    // 1.000 randomly generated numbers in the range of 0 to 50.
  ])
}

pub type Number {
  Zero
  S(Number)
}

fn smaller(x, y) {
  case x, y {
    Zero, S(_) -> Tr
    S(nx), S(ny) -> smaller(nx, ny)
    _, Zero -> Fa
  }
}

fn lenght(xs) {
  case xs {
    [] -> Zero
    [x, ..ys] -> S(lenght(ys))
  }
}

fn sort(elements) {
  case elements {
```

```
    [] -> []
    [x, ..xs]  -> {
      let len = lenght(xs)
      sort_inner(x, xs, [], len, [])
    }
  }
}

type MyBool {
  Tr
  Fa
}

fn sort_inner(element_to_cmp, elements, acc, len, real_acc) {
  case element_to_cmp, elements, len, acc {
    x, [y, ..ys], _, lacc -> {
      let comp = smaller(x,y)
      case comp {
        Tr -> sort_inner(x, ys, [y, ..lacc], len, real_acc)
        Fa -> sort_inner(y, ys, [x, ..lacc], len, real_acc)
      }
  }
    x, [], S(a), [ac, ..accs] -> {
      sort_inner(ac, accs, [], a, [x, ..real_acc])}
    x, [], _, _ -> [x, ..real_acc]
  }
}
```

Listing B.1: Gleam sorting using pattern matching

C

EXCLUDING ERLANG

In this section we explain why we do not implement decision trees for Erlang output. With just the JavaScript output we can already test whether decision trees are a promising way for increasing the performance of pattern matching in Gleam.

The Erlang compiler generates bytecode for the BEAM virtual machine. The BEAM is a register machine. It already generates code for efficient pattern matching [Armstrong 2007]. We did not discover the precise details of these optimizations, so it is unclear to us whether the Erlang compiler has functionality similar to decision trees and what heuristics (see Section 2.5) it would use. We did not observe the bytecode we would expect if decision trees were used.

We can however show one of these optimizations by taking the Gleam code in Listing C.1, seeing the Erlang code it generates in Listing C.2 and finally looking at the bytecode the Erlang compiler generates in Listing C.3

```
pub type Color {
  Red
  Black
  Purple
  Pink
}

pub fn t(x: Color,y: Color,z: Color) {
  case x, y, z {
    _, Red, Black -> 1
    Red, Black, _ -> 2
    _, _, Red -> 3
    _, _, Black -> 4
    _,_, Purple -> 5
    _, _, Pink -> 6
  }
}
```

Listing C.1: Pattern matching on many constructors in Gleam

Listing C.2 shows the generated Erlang output. Erlang is a dynamically typed language, but programmers can optionally provide type information. We see this in the type `color()`, which has the variants `red`, `black` etc. These variants are known as `atoms` in Erlang. An atom is a constant with a name, also known as a literal [Ericsson AB 2024]. The `t` function takes three colors and returns an integer as seen in its `spec` or specification. The pattern match is almost identical to the one in the Gleam source code, besides some minor syntactic differences like braces surrounding the patterns.

```
-type color() :: red | black | purple | pink.

-spec t(color(), color(), color()) -> integer().
t(X, Y, Z) ->
    case {X, Y, Z} of
        {_, red, black} ->
            1;

        {red, black, _} ->
            2;

        {_, _, red} ->
            3;

        {_, _, black} ->
            4;

        {_, _, purple} ->
            5;

        {_, _, pink} ->
            6
    end.
```

Listing C.2: Erlang code generated by the Gleam compiler

To illustrate some of Erlangs optimizations we need to understand some of the bytecode instructions. In our example we will only use the `x` registers, they are used for temporary data, passing data between functions and do not require a stack frame. Function arguments are placed in the registers and the first argument can be referred to like this: `{x, 0}`, the second as `{x, 1}` and so on. Return values are also placed in `x` registers. We will explain the following instructions:

function
label
f or fail
test
return
move
select_val

The function instruction looks like this: `{function <name>, <arity>, <labelNumber>}`, where `name` is the name of the function, `arity` is the arity of the function and `labelNumber` is a generated entry label for the function. The `{label, <labelNumber>}` instruction defines a label. Using the `{f, <labelNumber>}` instruction we can define a fail label to jump to a label in case of failure. Failures can happen when a test instruction results in a failure.

Test instructions look like this: `{test,<testName>, <failureLabel>, [<value>, <comparator>]}`. A relevant example is `{test,is_eq_exact,f,6,[x,0,atom,red]}`, here if the value in the 0th `x` register, is not exactly equal to the atom `red` the virtual machine will jump to label 6, based on the `f` instruction. The `return` instruction returns from a function. The `move` instruction moves a value to a register.

The `select_val` instruction compares a value to a list of values and jumps to the associated label or if no values match it to the failure label specified. So `{select_val, {x,2}, {f,9}, {list, [{atom,black}, {f,8}, {atom,red}, f,7]}}` will result in a jump to label 8 if the value in register 2 is the atom black, label 7 if the atom is red and to label 9 if it is neither.

```
{function, t, 3, 2}.
  {label,1}.
    {line,[{location,"test_small.erl",10}]}.
    {func_info,{atom,test_small},{atom,t},3}.
  {label,2}.
    {test,is_eq_exact,{f,3},[{x,1},{atom,red}]}.
    {test,is_eq_exact,{f,3},[{x,2},{atom,black}]}.
    {move,{integer,1},{x,0}}.
    return.
  {label,3}.
    {test,is_eq_exact,{f,4},[{x,0},{atom,red}]}.
    {test,is_eq_exact,{f,4},[{x,1},{atom,black}]}.
    {move,{integer,2},{x,0}}.
    return.
  {label,4}.
    {select_val,{x,2},
                {f,9},
                {list,[{atom,black},
                       {f,8},
                       {atom,pink},
                       {f,7},
                       {atom,purple},
                       {f,6},
                       {atom,red},
                       {f,5}]}}.
  {label,5}.
    {move,{integer,3},{x,0}}.
    return.
  {label,6}.
    {move,{integer,5},{x,0}}.
    return.
```

```
{label,7}.
  {move,{integer,6},{x,0}}.
  return.
{label,8}.
  {move,{integer,4},{x,0}}.
  return.
{label,9}.
  {test_heap,4,3}.
  {put_tuple2,{x,0},{list,[{x,0},{x,1},{x,2}]}}.
  {line,[{location,"test_small.erl",11}]}.
  {case_end,{x,0}}.
```

Listing C.3: Produced BEAM bytecode for funtion t

There are also some instructions that are not relevant to understanding the following example, the `line` and `func_info` instructions are used in error handling in the BEAM and not relevant for pattern match optimizations. The final label 9 in Listing C.3 is there for runtime errors, since pattern matches do not have to be exhaustive in Erlang there is always a final case generated in case no clauses match [Högberg 2024].

We can see a small example of Erlangs optimizations by compiling the code from Listing C.1, that produces the assembly shown in Listing C.3. This shows that Erlang uses instructions like `select_val` to avoid repeated tests of the same value. These are control flow constructs we do not have access to in JavaScript. Erlangs `select_val` optimization is at the byte code level and shows what the BEAM will execute. JavaScript does have a `switch` construct, which might execute similarly to the `select_val` statement but this will depend on the JavaScript runtime. The Erlang compiler will also inline values where the `t` function is called with known colors. It is possible the JavaScript runtime will do similar things but there are no guarantees. We have not found any evidence of the Erlang compiler using heuristics similar to the ones found in Section 2.5. Since the Erlang compiler already optimizes pattern matches we do not expect as large a gain in performance using decision trees compared to the JavaScript output where no optimizations are used at all.

BIBLIOGRAPHY

Aho, Alfred V. et al. (2007). *Compilers: principles, techniques, & tools*. 2nd ed. Pearson/Addison Wesley. ISBN: 9780133002140.

Armstrong, Joe (2007). "A history of Erlang". In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pp. 6–1.

Baudon, Thaïs, Laure Gonnord, and Gabriel Radanne (2022). "Knit&Frog: Pattern matching compilation for custom memory representations". PhD thesis. Inria Lyon.

Bun (2024). *Bun website*. URL: https://bun.sh/ (visited on 11/02/2024).

Ericsson AB (2024). *Erlang*. URL: https://www.erlang.org/doc/system/data_types.html (visited on 11/02/2024).

Filliâtre, Jean-Christophe and Sylvain Conchon (2006). "Type-safe modular hash-consing". In: *Proceedings of the 2006 Workshop on ML*, pp. 12–19.

Gleam contributors (2024). *Gleam Language Tour*. URL: https://tour.gleam.run/table-of-contents/ (visited on 04/04/2024).

Google (2024). *V8 website*. URL: https://v8.dev/ (visited on 10/26/2024).

Högberg, John (2024). *A brief introduction to BEAM*. URL: https://www.erlang.org/blog/a-brief-beam-primer/ (visited on 10/26/2024).

Maranget, Luc (2008). "Compiling pattern matching to good decision trees". In: *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pp. 35–46.

MDN (2024). *Performance.now() method*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now (visited on 11/02/2024).

Medeiros Santos, Andre Luis de (1995). *Compilation by transformation in non-strict functional languages*. University of Glasgow (United Kingdom).

Mozilla (2024a). URL: https://firefox-source-docs.mozilla.org/js/build.html (visited on 11/03/2024).

— (2024b). *SpiderMonkey website*. URL: https://spidermonkey.dev/ (visited on 10/26/2024).

Nystrom, Robert (2021). *Crafting interpreters*. Genever Benning.

Peyton Jones, Simon L. and AndréL.M. Santos (1998). "A transformation-based optimiser for Haskell". In: *Science of Computer Programming* 32.1. 6th European Symposium on Programming, pp. 3–47. ISSN: 0167-6423. DOI: https://doi.org/10.1016/S0167-6423(97)00029-4. URL: https://www.sciencedirect.com/science/article/pii/S0167642397000294.

The Gleam team (2024). *Gleam*. URL: https://gleam.run/ (visited on 01/28/2024).