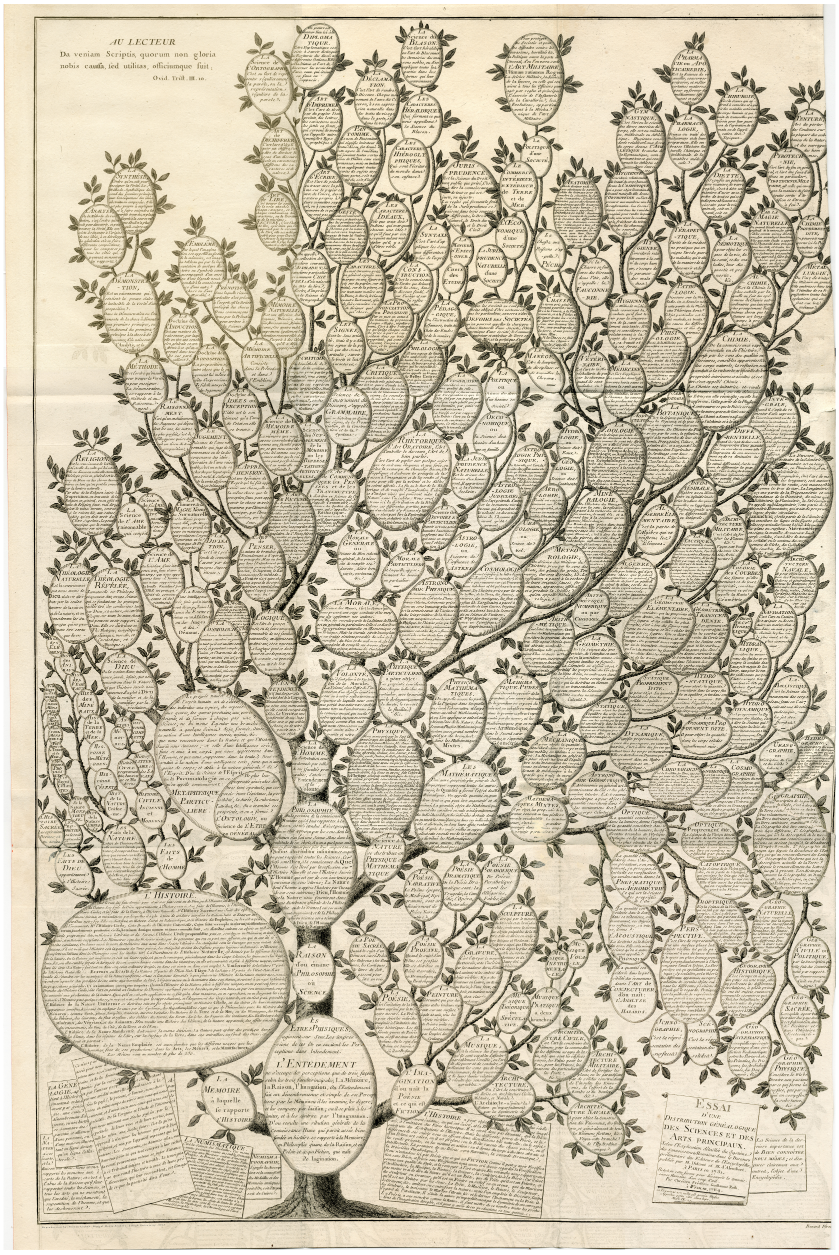
Optimising First-Class Pattern Match Compilation

Version of February 26, 2022



Toine Hartman

Optimising First-Class Pattern Match Compilation

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Toine Hartman
born in Utrecht, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Optimising First-Class Pattern Match Compilation

Author:	Toine Hartman
Student id:	4305655
Email:	`t.j.b.hartman@student.tudelft.nl`

Abstract

Pattern matching is the act of checking if a value is in the set of values described by a pattern. Many programming languages provide constructs to pattern match on program values. Pattern matching constructs appear in different variants. Stratego, a term rewriting language, features first-class pattern matching, which attempts to match a pattern with a value. If the match is successful, variables in the pattern are bound. If the pattern does not match, the matching expression fails. Using choice and sequence operators, one can build expressions that attempt to match multiple patterns until a match succeeds, evaluating the expression that corresponds to that pattern. This is a common way of branching in Stratego programs. The Stratego compiler handles each match attempt independently, disregarding the context.

In this thesis, we research context-aware compilation of first-class matches, aiming to speed up pattern matching operations without requiring changes to programs. We research the resemblance of Stratego pattern matching with *match cases*, a pattern matching construct that is ubiquitous in functional-style languages. Previous research shows that match cases can be compiled efficiently by considering all cases together, instead of a single alternative at the time, during compilation. We develop behaviour-preserving translations from first-class matches to match cases and from match cases to decision tree automata. We efficiently represent these automata in Java, the target language of the Stratego compiler. All these changes integrate in the latest upstream version of the Stratego compiler.

We evaluate the performance of our altered compiler and observe an average speed-up of $4\times$ on programs that rely heavily on pattern matching, at the cost of a 20% increase in compilation time and space.

Preface

With this thesis, I end my 8,5 years at TU Delft. A time, during which I developed myself in many ways, made new friends, and had new experiences.

Although I progressed through my Computer Science studies with success and pleasure, it was not until during my Master's degree that I found my real passion; the research field of programming languages. After attending several courses on languages, compilers and software verification, I decided to pursue a Master's Thesis in this direction. I want to thank Eelco for introducing to me to the topic of pattern match compilation, that ultimately became the subject of this thesis. I thank Jeff for his day-to-day guidance, which provided many fruitful discussions, helpful answers and critical reflections, and Asterios for taking place in my committee.

During the creation of this thesis, which can be a solitary process at times, I had continuous support from friends and family, which I do not want to leave unmentioned. I am grateful for the desk space, coffee, lunch, and company that Christiaan and Jorian provided. I thank Ties for proofreading a draft of this thesis, and for his many useful comments. Jojanneke, my girlfriend, has been an endless source of support and patience, for which I am very thankful. She also had the dubious honour of being my primary rubber duck[1] — I should thank everyone who has been on the receiving end of my rambling about patterns and trees. Finally, I want to thank my parents, who always supported my (career) decisions, and made sure I had a carefree life inside and outside the university.

Toine Hartman
Delft, the Netherlands
February 26, 2022

[1] `https://en.wikipedia.org/wiki/Rubber_duck_debugging`

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Pattern matching is the act of checking if a value is in the set of values described by a pattern. A pattern is a value with wildcards at certain positions, representing a set of values. The semantics of pattern matching may differ between languages. A ubiquitous variant of a pattern match expression is the *case match*, an example of which is in Listing 1.1. A pattern matching construct may list multiple alternatives, in which case the alternatives are tried one-by-one. Upon matching a pattern successfully, the program continues with its right-hand side (RHS) — the match construct succeeds without trying any other patterns. We explain pattern matching in more detail in Chapter 2.

Pattern match compilation aims to represent a pattern match as lower-level instructions, e.g., in the compiler's core or target language. A naive pattern match compiler generates instructions that perform the following actions, starting at the first pattern; try to match the pattern — if it matches, execute the corresponding RHS action, otherwise — repeat for the next pattern. If no patterns remain; fail. The major disadvantage is its run-time; in the case where no pattern matches, *all patterns* need to be visited first in order to find out none matches. This means the worst-case run-time is linear in the number of patterns. In fact, this naive compiler generates code that performs a lot of duplicate work. To mitigate this, researchers discovered techniques to produce an expression of which the runtime is independent of the number of alternative patterns. By matching the pattern's sub-patterns incrementally using an automaton, the factor of the number of patterns can be removed from the runtime equation. Decision trees (Cardelli 1984) and backtracking automata (Augustsson 1984; Augustsson 1985), which we introduce alongside the naive compilation scheme in Chapter 2, are two forms of such an automaton. Compiling pattern match alternatives to a decision tree guarantees that each subpattern is checked at most once, while a backtracking automaton may revisit a subpattern. This gives the former a theoretical run-time advantage over the latter, while

```scala
def matchTest(x: Int): String = x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
}
matchTest(3)  // returns other
matchTest(1)  // returns one
```

Listing 1.1: Pattern match example (in the Scala programming language (Odersky et al. 2004; Scala 2021)). To match `x`, the cases are tried one-by-one — starting at the top — until a pattern is matched successfully. Its corresponding expression (e.g. `"other"`) is then evaluated. The pattern _ is a wildcard, matching anything.

the nature of the backtracking automaton — not duplicating code for a single subpattern — results in better space efficiency than a decision tree approach.

Many concepts in this thesis apply specifically to Stratego (Visser, Benaissa, and Tolmach 1998; Visser 2001; Smits, Konat, and Visser 2020), a declarative language to specify transformations on Annotated Terms (ATerms) (van den Brand, de Jong, et al. 2000). Its main use is the specification of compilers in the Spoofax Language Workbench (Kats and Visser 2010; Spoofax Team 2021). Transformation languages like Stratego use pattern matching in selecting and performing transformations. Stratego, which we introduce in Chapter 3, features first-class pattern matching; matching a single pattern, without an explicit RHS or alternative patterns, is a first-class language construct. Prior to this thesis, Stratego match alternatives compile to naive code, potentially leaving room for performance improvements. Therefore, this thesis will address the following research question:

What are the effects of compiling first-class pattern matches using case pattern match compilation techniques?

In order to answer this question, we present a method to compile first-class matches in Stratego. We introduce a case match intermediate representation (IR) in Stratego, which syntactically and semantically resembles the case match that appears in many functional-style languages and, more notably, literature on pattern match optimisation. Listing 1.1 is an example of a case match. We observe that first-class matches, combined with other Stratego language constructs, match the semantics of the case match. By defining behaviour-preserving transformations from these compositions of first-class matches to case matches, we obtain grouped pattern match alternatives. These case matches may subsequently be compiled using a pattern match compiler from a language without first-class matches. We choose to compile to decision trees because of their inherent efficiency advantage over backtracking automata and apply a method based on the pattern match compilation scheme proposed for Objective Caml (OCaml) (Maranget 2008). In the back-end of the Stratego compiler, we generate efficient Java code emulating the matching automaton. We implement all changes in a recent version of the Stratego 2 compiler (Smits, Konat, and Visser 2020). Throughout this thesis, a Stratego implementation of a simple calculator on natural numbers will be used as an example. The full program is in Listing A.1.

To quantify the effects of our work, we set up benchmarking for Stratego using two corpuses. In highly optimisable, pure rewriting programs, we observe a significant speed-up, especially when optimising the implementation of the generated decision tree. In more idiomatic Stratego, part of a compiler specification, we do not observe a notable difference in run-time performance. For most input programs, our work induces an increased compilation time.

Finally, we conclude and suggest directions for future work.

Contributions In summary, this thesis makes the following scientific contributions.

- A behaviour-preserving transformation from compositions of Stratego first-class matches to the general case match construct (Chapter 4).
- A pattern match compiler for Stratego and other ATerm-based languages (Chapter 5).
- A Stratego compiler back-end extension to handle match automata (Chapter 6).
- A benchmark suite for the incremental Stratego compiler (Smits, Konat, and Visser 2020) (Chapter 7).
- Insight into the performance difference achieved by the pattern match compiler (Chapter 7).

Chapter 2

Background

This chapter provides an introduction to pattern matching, pattern match semantics, and optimisation.

2.1 Pattern Matching

Pattern matching is the act of checking if a value — the *scrutinee* — is an instance of a pattern. Different variants exist, which can be identified by two orthogonal properties; the shape of the values/patterns (Section 2.1.1) and the characteristics of the matching construct (Section 2.1.2).

2.1.1 Pattern Shapes

Patterns can appear in several shapes. This section[1] introduces some of these shapes; tree patterns, sequence patterns and graph patterns.

Tree Patterns Tree patterns represent and match (nested) tree structures. Those trees are n-ary constructors ($n \geqslant 0$), where each argument is a tree pattern. Common practical examples are (inductive) algebraic data types in functional programming, and terms in a term rewriting system (TRS) (Klop 1990; O'Donnell 1985).
A tree pattern can consist of the following elements (Cardelli 1984):

- variables (e.g., `x`);
- wildcard patterns (or *defaults*), often written as `_`. They act as an anonymous variable and match anything;
- data constants/nullary constructor applications (e.g., the literals `54` and `"octopus"` or the empty list constructor `Nil`). They have zero arguments and only match specific values;
- constructor applications with patterns as their arguments (e.g., `Cons hd tl` or `hd:tl`);
- tuples of patterns[2] (e.g., `(Nil, x:xs, _)`).

The Stratego language (see Chapter 3) supports tree patterns. Other variants of patterns exist. Although not essential to this work, this section provides examples of these variants to establish a broader perspective on pattern matching.

[1]This section is an adapted version of a section from my literature survey (Hartman 2021).

[2]Depending on the language internals, a tuple could just be a constructor application. In Haskell, `(Nil, x:xs, _)` is equivalent to `Tuple3 Nil (x:xs) _`.

```scala
val x = 8

val res = x match {
    case _ if x < 0 => "less than zero" // guards checks if x is negative
    case 0 => "zero"
    case 1 => "one"
    case _ => "more than one"
}

println(res) // more than one
```

Listing 2.1: Example of a case match (Scala). For the shown value of `x = 8`, it will print "more than one".

Sequence Patterns Sequence patterns — often called string patterns if the sequence represents text — represent instances of an ordered sequence of values. Matching on sequences often entails finding subsequences at an arbitrary location. The most common contemporary string pattern matching solution is the use of regular expressions, which were introduced by Kleene (1951). Regular expression matching yields zero or more matches, in contrast to tree patterns.

Graph Patterns Graph patterns are generalised tree patterns, appearing in graph rewriting systems (GRSs) (see Chapter 8) and advanced Extensible Markup Language (XML) matching (Cheng et al. 2008). A GRS is a generalised variant of a TRS operating on graphs instead of trees. The translation to graphs enables identifying repeating (shared) subgraphs.

Although graph patterns are more general than tree patterns, the transition from trees to graphs does not significantly influence syntax, semantics, and solutions (Eelco Visser, priv. comm., January 7, 2021).

2.1.2 Matching Constructs

Languages each have specific pattern matching possibilities. The syntax that permits the use of patterns, paired with its desired semantics, is what we call a *matching construct*. In other words; how can the programmer use pattern matching?

Case Match Probably the most well-known and ubiquitous matching construct is one that one could informally name the *case match*. It specifies a scrutinee (possibly implicitly), and a list of match cases. The match cases, in turn, have a pattern, optionally a *guard*, and an accessory right-hand side (RHS). Depending on the exact design of the case match and the semantics of the language, the cases will be evaluated in some order. For the first case that *matches*, the RHS will be evaluated. A *matching* case is one of which the pattern matches and the guard succeeds[3]. Listing 2.1 shows an example of case match syntax and semantics in Scala.

First-Class Match A language with first-class pattern matching allows a single pattern match to appear as an expression, without a guard or RHS, any alternative matches, or defaults. The match might be unsuccessful; if and how the program continues in that case, depends on the language semantics. An example of a first-class match in Stratego is `?Plus(O(), n)`.

[3]The notion of success follows from expression and guard semantics. It may be 'evaluates to a truth value', 'does not fail', or some other set of results.

```scala
abstract sealed class Nat

case class O() extends Nat
case class S(pred: Nat) extends Nat

case class Plus(first: Nat, second: Nat) extends Nat
case class Minus(first: Nat, second: Nat) extends Nat

def calc(exp: Nat): Nat = exp match {
    case O()  => O()  // zero
    case S(n) => S(n) // n + 1

    case Plus(O(), n)  => calc(n)
    case Plus(n, O())  => calc(n)
    case Plus(S(m), n) => S(calc(Plus(calc(m), calc(n))))

    case Minus(O(), O())    => O()
    case Minus(S(n), O())   => S(calc(n))
    case Minus(S(m), S(n)) => calc(Minus(calc(m), calc(n)))
    case Minus(O(), S(n))   => throw new ArithmeticException("Negative result!")

    // If we cannot calculate directly, first calculate left and right expressions
    ↪  and try again.
    case Plus(x, y)  => calc(Plus(calc(x), calc(y)))
    case Minus(x, y) => calc(Minus(calc(x), calc(y)))
}

val one = S(O())
val two = S(one)    // S(S(O()))
val three = S(two)  // S(S(S(O())))
val four = S(three) // S(S(S(S(O()))))
```

Listing 2.2: A simple calculator (addition & subtraction) in Scala.
In this example, **`case class`** `X` may be read as '`X` is a constructor'.

2.2 Pattern Match Compilation

In compiled languages, pattern matching constructs compile to lower-level constructs — while preserving their behaviour. To understand the solution presented in this thesis (Chapters 4 to 6), some knowledge on the compilation of case matches is relevant. We will explore compilation techniques using a simplified[4] version of the Stratego calculator program in Listing A.1, which we implement in Scala (Odersky et al. 2004); this implementation is in Listing 2.2.

2.2.1 Naive Solution

If one would ask a naive programmer to simulate the execution of `calc(Minus(four, two))` — which can be rewritten to `calc(Minus(S(S(S(S(O())))), S(S(O()))))` — given the `calc` function and definitions from Listing 2.2, it would probably go as follows.

[4]This simplified calculator can only perform addition and subtraction.

Algorithm Our programmer is naive, so he checks one `case` at a time. He performs the following steps.

He looks at the outermost constructor of the input value: `Minus/2`, a constructor named 'Minus' with 2 arguments. He goes through the cases to find one that matches this outer constructor.

- The outer constructor of `O()` (`O/0`) is not `Minus/2`. No match. Next.
- The outer constructor of `S(n)` (`S/1`) is not `Minus/2`. No match. Next.
- The outer constructor of `Plus(O(), n)` (`Plus/2`) is not `Minus/2`. No match. Next.
- The outer constructor of `Plus(n, O())` (`Plus/2`) is not `Minus/2`. No match. Next.
- The outer constructor of `Plus(S(m), n)` (`Plus/2`) is not `Minus/2`. No match. Next.
- The outer constructor of `Minus(O(), O())` (`Minus/2`) equals the outer constructor of our value. Now check if the first argument matches. The first argument of our input value is `S(S(S(S(O()))))`, of which the constructor is `S/1`.
  - The constructor of the first argument of `Minus(O(), O())` (`O/0`) is not `S/1`. No match. Next.
- The outer constructor of `Minus(S(n), O())` (`Minus/2`) equals the outer constructor of our value. Now check if the first argument matches. The first argument of our input value is `S(S(S(S(O()))))`, of which the constructor is `S/1`.
  - The constructor of the first argument of `Minus(S(n), O())` (`S/1`) equals `S/1`. Now we check if the second argument matches. The second argument of our input value is `S(S(O()))`, of which the constructor is `S/1`.
    * The constructor of the second argument of `Minus(S(n), O())` (`O/0`) does not equal `S/1`. No match. Next.
- The outer constructor of `Minus(S(m), S(n))` (`Minus/2`) equals the outer constructor of our value. Now check if the first argument matches. The first argument of our input value is `S(S(S(S(O()))))`, of which the constructor is `S/1`.
  - The constructor of the first argument of `Minus(S(n), O())` (`S/1`) equals `S/1`. Now we check if the second argument matches. The second argument of our input value is `S(S(O()))`, of which the constructor is `S/1`.
    * The constructor of the second argument of `Minus(S(m), S(n))` (`S/1`) matches the second argument of our input value. *Successful match!* Now we bind pattern variables (`val m = S(S(S(S(O()))))` and `val n = S(S(O()))`) and evaluate the RHS `calc(Minus(calc(m), calc(n)))`.

The algorithm, performed manually by the programmer, found a match! Although the result is correct, we can do better.

Limitations of the Naive Algorithm In this informal run-through of the naive algorithm, the programmer experienced a lot of unsuccessful checks on the same constructor (`Plus/2`), before ending up at a matching case. Formally, the complexity of this algorithm is linear in the number of cases. If the cases could be grouped by outer constructor, we could skip over the group of `Plus/2` cases or jump into the group of `Minus/2` cases immediately, and save some steps. A generalisation of this will be the driving force behind the improved solutions.

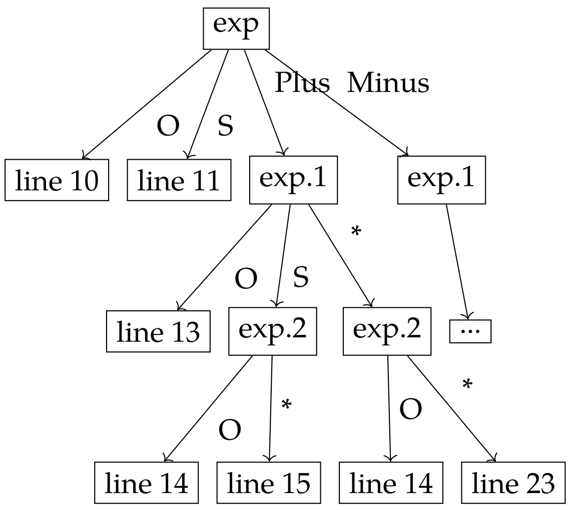


Figure 2.1: Graphic representation of a part of the decision tree for Listing 2.2. The subtree for `Minus(_, _)` patterns has been omitted for brevity.

2.2.2 Improved Solutions

A lot of research has been done on improving the naive algorithm. By grouping the patterns on the equality criteria from the previous example, one can create an automaton of which the matching time is independent of the number of match cases. Informally, that means we check a constructor *once* and subsequently only consider patterns that satisfy this condition. For example, after we confirm that the outer constructor of the scrutinee equals `Plus/2`, we only need to check its arguments to check if it matches one of the following patterns.

```
13  case Plus(O(), n)  => calc(n)
14  case Plus(n, O())  => calc(n)
15  case Plus(S(m), n) => S(calc(Plus(calc(m), calc(n))))

23  case Plus(x, y)  => calc(Plus(calc(x), calc(y)))
```

Authors of previous research mention different techniques to reach this goal. One of the possible techniques is to use a decision tree as the matching automaton. A different solution is to use a backtracking automaton.

Decision Trees The general idea of a decision tree is that it performs a check on each part of the value at most *once*, gaining a runtime advantage over the naive method — which might perform multiple of those checks.

Instead of checking the pattern for each match case separately, a decision tree groups patterns on their properties and distinguishes between them with simple equality checks. It checks the outer constructor, goes into the arguments, and performs the same check on them recursively. The algorithm does not need any global information and is thus straightforward to implement. From this recursive approach, the tree shape follows naturally. Figure 2.1 shows (a part of) the decision tree for our calculator example.

The decision tree approach has one major drawback; in the theoretical worst case, it grows exponentially with the number of constructors that appear in the patterns. This duplication also shows in the example; the decision tree has two leaves referring to a match of the case on line 14.

Backtracking Automata Aiming to mitigate the exponential code size explosion of the decision tree approach, one could consider using a backtracking automaton. The difference is that a backtracking automaton can backtrack, i.e., jump back to a 'higher' state in the automaton. This prevents duplication of states, but as a result, certain positions of the value may be checked more than once.

Although the backtracking approach solves the exponential code explosion that might occur in decision trees, it might evaluate parts of the scrutinee more than once. Therefore, it is not as efficient as the latter method. It is, however, still a significant improvement over the naive method from Section 2.2.1.

Chapter 3

Pattern Matching in Stratego

In this chapter, we explain what constitutes Stratego programs (Sections 3.2 to 3.4), and how the compiler transforms them (Section 3.4). It shows how the program in Listing 3.1 is equivalent to the lower-level program in Listing 3.2, and how this relates to pattern match compilation methodologies from Section 2.2 (Section 3.5).

3.1 Stratego Principles

Stratego (Visser, Benaissa, and Tolmach 1998; Visser 2001; Smits, Konat, and Visser 2020) is a language for defining term rewriting systems (TRSs). It is part of the Spoofax Language Workbench (Kats and Visser 2010), where it can be used to specify program transformations. It is regularly used for declaring desugaring rules, optimisations, compilation, and translation to a different language (transpilation). Stratego features both first-class pattern matching and (labelled) rules — which share properties with match cases.

3.2 Terms

Values in Stratego are *terms*. Stratego builds upon the Annotated Term (ATerm) library (van den Brand, de Jong, et al. 2000), which recognises the following term types.

Constructor Applications A constructor application `X(a, b)` represents the application of constructor `X` to terms `a` and `b`.

Literals Stratego features string (`"octopus"`), integer (`8`) and real (`5.4`) literals.

```
  Calc: Plus(S(m), n)     -> S(Plus(m, n)) // (1 + m) + n => 1 + (m + n)

  Calc: Minus(n, O())     -> n             // n - 0             => n
  Calc: Minus(S(m), S(n)) -> Minus(m, n)   // (1 + m) - (1 + n) => m - n
  Calc: Minus(O(), S(_))  ->
        <fatal-err(|"Negative result!")>   // 0 - (1 + n) => not a nat

strategies
  calculate = innermost(Calc)
```

Listing 3.1: Stratego rules for a single addition/subtraction step. The rules apply when at least one of the operands is a literal natural number.

```
Calc_0_0( | ) =
        {n0:            ?Plus(O( ), n0);      !n0 }
  < id + {n1:           ?Plus(n1, O( ));      !n1 }
  < id + {m0, n2:       ?Plus(S(m0), n2);     !S(Plus(m0, n2){ }){ } }
  < id +                ?Minus(O( ), O( ));   !O( ){ }
  < id + {n3:           ?Minus(S(n3), O( ));  !S(n3){ } }
  < id + {m1, n4:       ?Minus(S(m1), S(n4)); !Minus(m1, n4){ } }
  < id + {arg_m0, where0: ?Minus(O( ), S(_));   ?where0; !"Negative result!"{ };
↪  ?arg_m0; !where0; fatal_err_0_1( |arg_m0) }
```

Listing 3.2: Stratego Core representation of the rules for the simple addition/subtraction calculator (Listing 3.1). Note that variables are normally prefixed with the name of the strategy that they appear in (e.g. `calc_0_0_n0` instead of `n0`). This example omits these prefixes to improve legibility.

Lists Lists of terms can be represented in several ways. `[1, 2, 3]`, `[1 | [2, 3]]` and `[1, 2 | [3]]` are identical notations. The empty list `[]` might also be expressed as `Nil()`. In fact, all notations using square brackets (`[x | xs]`) are alternative syntax for `Cons(x, xs)`.

Tuples Tuples of terms can be written as `(1, 2)`. Internally, tuples are represented by a constructor with the empty string as its name.

Annotations As the name suggests, ATerms have an annotation. In the Stratego syntax, a ⟨*Term*⟩ is a ⟨*PreTerm*⟩ annotated with another ⟨*PreTerm*⟩. ⟨*PreTerm*⟩s are Because this is sparingly used, Stratego provides syntactic sugar for unannotated terms. For now, we ignore the existence of annotations, and consider ⟨*PreTerm*⟩s to be ⟨*Term*⟩s. We revisit this in Section 5.4.

3.3 Core Language

Stratego is built upon the Stratego Core language, which defines *modules* that contain *specifications*. These specifications consist of *strategy definitions*, the body of which is a *strategy expression*. Strategies operate on terms; they take an implicit term (*current term*; the result of the previous strategy) as input, and modify it. The Stratego runtime is capable of backtracking. Backtracking undoes any reversible side effects, like variable bindings. Furthermore, a strategy can fail. Among the core strategies are basic operators (identity, failure), combinators (choice, sequential composition), scope, match and build, as described here.

Identity `id` Return (i.e., do not modify) the current term.

Failure `fail` Explicit failure.

Guarded Left Choice `s1 < s2 + s3` The guarded left choice (GLC) branches conditionally on `s1`. If `s1` succeeds, call `s2` on its result. If `s1` fails, backtrack and call `s3`. The result of the right-hand side (RHS) strategy (`s2` or `s3`) is the result of the GLC.

Scope `{x: s}` Declare `x` as fresh (unbound) variable in `s`. The result of `s` is the result of the scope strategy.

Match `?t` Matches the current term with pattern `t`. Any unbound variables in `t` will be bound to the values that occur at their position in the current term. Fails if the current term does not match the pattern.

Build `!t` Replaces the current term with `t`. Fails if the term contains unbound variables and if `t` is a list of which the tail is not a list (e.g., `![1 | 2]`, which fails, instead of `![1 | [2]]`).

Let `let def = s1 in s2 end` Defines local strategy definition `def` in strategy expression `s2` and evaluates `s2`.

Writing meaningful programs in Stratego Core alone is quite cumbersome. Luckily, Core just provides the building blocks for Stratego Sugar.

3.4 Sugar

Stratego provides many constructs that are in fact syntactic sugar for compositions of Core constructs.

Assign The strategy `t1 := t2` assigns term `t2` to term `t1`, performing pattern matching if `t1` is not a variable. It desugars to `!t2; ?t1`.

Apply Match The strategy `s => t` assigns the result of strategy `s` to term `t`. It desugars to `s; ?t`.

Build Apply A strategy `s` can be applied to a term `t`: `<s> t`. It desugars to `!t; s`.

Left Choice `s1 <+ s2` is sugar for `s1 < id + s2`.

Where The strategy `where(s)` uses strategy `s` as a condition, but discards its change to the current term. Instead, the current term from before `where(s)` is restored afterwards. Its success/failure follows immediately from the success/failure of the wrapped strategy `s`. `where(s)` desugars to `{x: ?x; s; !x}`.

Rule A rewrite rule for the addition $0+n$ could look like this: `Calc: Plus(0(), n) -> n` . A rewrite rule desugars to a strategy which encompasses a scoped composition of a match and a build. For this example, the Core equivalent is `Calc = {n: ?Plus(0(), n); !n}`. A rule can have a where-condition; when it fails, this rule does not apply. Instead, Stratego backtracks. For example, we can simulate the behaviour of the previous rule with this rule: `Calc: Plus(m, n) -> n where 0() := m`. A rule `p -> t where c` desugars to `?p; where(c); !t`.

Pattern Arguments Rules and strategies can have strategy and term arguments, which are separated by a vertical bar in definitions and references. An example is the standard library strategy `list-fold(s|acc)`, which accepts a binary strategy `s` to apply to an element and the accumulator, and an initial accumulator `acc`, which is a term. If a strategy only accepts strategy arguments, the | separator is optional. The declaration of `map(s)` is an example of this.

Term arguments can be variables or terms; in the latter case the argument will be pattern matched. Only if the argument matches, the definition applies. A rule definition `r(|p): t1 -> t2` (where `p` is not a variable) desugars to `r(|x): t1 -> t2 where x := p`.

Multiple Definitions Often, Stratego programs use multiple strategy definitions with the same label to group multiple rules that serve the same goal. Extending the calculator example with rules that perform a single subtraction step, also labelled `Calc`, Listing 3.1 is the result. Generally, multiple strategy definitions with the same label desugar to a single strategy definition which combines the original definitions with left choices (`<+`).

```
strategies
  s = s1
  s = s2
  s = s3
```

The above strategy definitions desugar to the following.

```
strategies
  s = s1 <+ s2 <+ s3
```

Arguments in Definition A strategy might take strategy and term arguments. Explicitly defining an empty argument list is optional. The strategy `s` desugars to `s(|)`.

Strategy Arity During desugaring, the arity of a strategy will be added to its name. The strategy call/definition `s(s1, s2 | t)` desugars to `s_2_1(s1, s2 | t)`.

Variable Names Desugaring modifies variable names in various ways in order to make them unique in the Java implementation of the program. Each fresh variable becomes unique by suffixing it with a number. Furthermore, they are prefixed with the name and arity of the enclosing strategy definition.

3.5 Naive Match Execution

Applying the desugaring steps from Section 3.4 to the calculator rules in Listing 3.1 yields the core strategy in Listing 3.2. This demonstrates that rules translate to alternatives, which are tried in turn. If an alternative does not match, Stratego backtracks and tries the next alternative. This is exactly the naive method described in Section 2.2.1, which leads to believe Stratego pattern matching performance could benefit from an improved pattern match compiler.

The following chapters explain the steps which constitute our solution to efficient and informed compilation of Stratego first-class pattern matches. The Stratego compiler is bootstrapped, i.e., written in Stratego. Therefore, we use an idiomatic Stratego compilation technique; compilation by transformation. We compile the first-class pattern matches to several intermediate representations (IRs) in consecutive orthogonal transformations.

In order to be able to reason about a collection of patterns, which we need for smart compilation, we translate strategies containing first-class matches to match cases (Chapter 4). Match cases can be translated to an efficient automaton (Chapter 5), which in turn can be encoded in the target language of the compiler (Chapter 6).

Chapter 4

Translating First-Class Matches to Match Cases

We implement our pattern match compilation scheme in several layers. We use behaviour-preserving transformations, which we keep as small as possible. This principle, called 'compilation by transformation', is established in the literature (Bacon, Graham, and Sharp 1994; Aho et al. 2007; Peyton Jones 1996).

To be able to translate first-class matches to match cases, the first step is to add a representation of match cases to the Stratego compiler.

4.1 The Case Match Construct

We introduce the case match, a new Stratego Core syntax construct. Listing 4.1 shows a concrete example of its use; the grammar is in Listings B.1 and B.2 (Appendix B). It resembles the case match constructs in other languages, for example Scala's (see Listing 2.1). It does not take an explicit scrutinee, as it matches on the implicit current term. The case match specifies an explicit match order, which denotes in which order the match cases should be attempted. Currently, the compiler implements the sequential order (top-to-bottom), but more intricate orders could be addressed in future work (see Chapter 9). If the match cases have been exhausted without a successful match, the case match fails.

Match Case The match case is a scoping construct; `case x, y | p: s` declares variables `x` and `y` to be unbound in the scope in which the match pattern `p` and right-hand side (RHS) `s` are evaluated. Like a first-class match `?p`, the match case pattern may bind unbound variables. If the pattern matches, the RHS `s` is executed in the same scope. It is semantically equivalent to `{x, y: ?p; s}`.

A variant with a guard also exists; `case x, y | p when s1: s2`. The guard `s1` is executed immediately after the pattern match — if it matches — in the same scope. It may alter the current term. If this is not intended, and the guard should only function as a

```
length =
  match sequential
    case       | []: !0
    case x, xs | [x | xs]: <add> (1, <length> xs)
    case       | _ when not(is-list): fatal-err(|"Not a list!")
  end
```

Listing 4.1: Strategy to calculate the length of a list, implemented using the new case match.

Boolean check, it could be wrapped in a `where`-clause; `where(s1)`. If the guard `s1` fails, the encompassing case match backtracks to the next case. Failure of the RHS `s2` results in immediate failure of the complete case match strategy, instead of backtracking.

4.2 Translating Compositions

The goal of our optimisation is to speed up existing programs, without requiring changes to these programs. To achieve that, we identify common core strategy patterns that appear in idiomatic Stratego and translate them to semantically equivalent strategies involving case matches. The goal is to gather as much match cases together as possible, in order to optimise as effectively as possible. These translations apply to desugared (i.e., Core) Stratego.

We define four classes of transformations: `choice-to-casematch`, `extract-matches`, `extract-righthandsides`, and `lift-defaults`. Each class comprises behaviour-preserving transformations with mutual properties. Each transformation will be implemented as a strategy in the compiler, labelled with the class name.

Choices to Case Matches First, we describe transformations that rewrite (specific) guarded left choice (GLC)-based strategy expressions to a case match. The compiler applies these transformations repeatedly until none apply. This means that the input to one of these transformations is often the result of another.

Listing 4.2 described the transformation of a specific GLC expression, which has a case match as its guard, `id` as its success strategy, and another case match as the alternative. The semantics of the GLC dictate that, if the first case match succeeds, the result of the GLC is the application of the success strategy to the result of the guard. If the first case match fails, the second one will be executed. This is semantically equivalent to a case match with a concatenation of the cases from both original case matches.

```
  match sequential
    case x1* | p1 when sg1: sa1
    case x2* | p2 when sg2: sa2
  end
< id +
  match sequential
    case y1* | q1 when sg3: sa3
    case y2* | q2 when sg4: sa4
  end
```

$\Longrightarrow$

```
match sequential
  case x1* | p1 when sg1: sa1
  case x2* | p2 when sg2: sa2
  case y1* | q1 when sg3: sa3
  case y2* | q2 when sg4: sa4
end
```

Listing 4.2: `ChoiceToCaseMatch` transformation. Optimise a GLC with no middle strategy and case matches in both branches, to a single case match with the concatenation of the cases of the original case matches in the arms.

The transformation in Listing 4.3 is a generalised variant of the one in Listing 4.2. If a case match appears as the guard of a GLC with arbitrary branches, the case match can be 'lifted', as it will be executed nonetheless. Compose the RHSs of the cases sequentially with the success branch of the GLC and add a case with the failure branch of the GLC as its guard. This last case does not need a specific pattern (use a wildcard) nor RHS.

We might rewrite any GLC other to a case match as in Listing 4.4. In the context of pattern match compilation, this is only useful if `s1` and `s3` start with matches — otherwise, we are just generating a cumbersome GLC (and later, a meaningless deterministic finite automaton (DFA), because there are no pattern matches). Therefore, we attempt to extract pattern matches from the case guards and substitute the wildcards `_` for patterns.

```
1  match sequential
2    case x* | p when sg: sr
3    case y* | q when sg': sr'
4  end < s2 + s3
```

$\Longrightarrow$

```
1  let
2    s = s2
3  in
4    match sequential
5      case x* | p when sg: sr; s
6      case y* | q when sg': sr'; s
7      case    | _ when s3: id
8    end
9  end
```

Listing 4.3: `ChoiceToCaseMatch` transformation. Optimise a GLC with a case match as its guard to a case match. Modify the cases; their RHS is the sequential composition of their original RHS and `s2`. Then, add an extra case (line 8) with a wildcard pattern and `s3` as its guard, to match the behaviour of the original GLC.

```
s1 < s2 + s3
```

$\Longrightarrow$

```
match sequential
  case | _ when s1: s2
  case | _ when s3: id
end
```

Listing 4.4: `ChoiceToCaseMatch` transformation. Any GLC can be rewritten to an equivalent case match with this translation. However, it is only useful if we can extract matches from the guards and replace the wildcards with specific patterns.

```
case x* | _ when ?q: s2               ⟹   case x* | q when id: s2
case x* | _ when ?q; s1: s2           ⟹   case x* | q when s1: s2
case x* | _ when {y*: ?q}: s2         ⟹   case z* | q when id: s2
case x* | _ when {y*: ?q; s1}: s2     ⟹   case z* | q when s1: s2
```

Listing 4.5: `ExtractMatch` transformations. The match case has a wildcard pattern and the guard contains a match. The match in the guard position is moved to the match case pattern position by extracting the pattern. If the guard is scoped, the fresh variables are instead introduced in the match case scope position; $z^* := x^* \cup y^*$.

Extracting Matches Match cases may have wildcard patterns, and perform a match in the guard strategy. In this case, we can open up the match case for match optimisation by extracting the match pattern from the guard and moving it replacing the wildcard. The transformations in Listing 4.5 serve this purpose. At most one of these transformations applies to each match case. Chapter 5 will explain how the extracted patterns are relevant in the optimisation.

Extracting RHSs Because guarded operations imply backtracking, they require the generation of particular backtracking code. This goes for GLCs as well as the guards in the DFA intermediate representation (IR). However, if the guard is just `id`, it does not do anything and always succeeds. In this instance, simpler code can be generated, because no backtracking can occur.

If the compiler is sure that the strategy in the guard will succeed, it may be moved to the RHS without changing semantics, but allowing for simpler code generation. For this, we introduce a success/failure analysis for a subset of Stratego Core. Before this analysis, the existing bound/unbound variable of the Stratego compiler analysis is invoked, as success or failure of a strategy may depend on the state of used variables. Take `?x` as an example; it is guaranteed to succeed if `x` is unbound, but not if `x` is bound. We include the full analysis, to be invoked on a Core abstract syntax tree (AST), in Appendix C

The transformation which moves a succeeding guard to the RHS is in Listing 4.6.

```
case x* | p when s1: s2                ⟹    case x* | p when id: s1; s2
```

Listing 4.6: `ExtractRHS` transformation. Only valid if `s1` is guaranteed to succeed.

Lifting Defaults We identify another common composition; where the *last* match case houses a case match in the guard of the RHS position, we can lift the cases from the inner case match to the outer one, as to optimise them together, possibly increasing the efficiency of the pattern match optimisation. The resulting case match has one level less nesting in its guard positions. The transformations in Listings 4.7 and 4.8 perform this optimisation.

```
match sequential
  /* other cases */
  case xs | _ when                             match sequential
    match sequential                             /* other cases */
      case ys | q when s1: s2          ⟹         case xs2 | q when s1; s2: s
      case zs | r when s3: s4                    case ys2 | r when s3; s4: s
    end: s                                     end
end
```

Listing 4.7: `LiftDefault` transformation. A single match case with a case match as its guard, can be translated to multiple match cases using this transformation. The example with two nested cases generalises naturally to any other number. It is only valid for the last match case in a case match.

```
match sequential
  /* other cases */
  case xs | _ when s:                          match sequential
    match sequential                             /* other cases */
      case ys | q when s1: s2          ⟹         case ys2 | q when s; s1: s2
      case zs | r when s3: s4                    case zs2 | r when s; s3: s4
    end                                        end
end
```

Listing 4.8: `LiftDefault` transformation. A single match case with a case match as its RHS, cam be translated to multiple match cases using this transformation. The example with two nested cases generalises naturally to any other number. It is only valid for the last match case in a case match.

A Note on Inlining In idiomatic Stratego, pattern match alternatives may often hide behind strategy definitions. Consider the following example, from the Stratego implementation of the compilation scheme that we introduce in Chapter 5.

```
module match-to-dfa
strategies
  CC = NoRows
    <+ WildcardsFirstRow
    <+ SelectColumn

  NoRows: Matrix(_, []) -> Fail()
  WildcardsFirstRow: Matrix(_, [Row(<all-wildcards>, _, _, _) | _]) -> some_dfa
  SelectColumn = !some_other_dfa
```

If the compiler inlines the calls to `NoRows`, `WildcardsFirstRow` and `SelectColumn` in `CC`, we obtain the following desugared strategy expression for `CC`.

```
   ?Matrix(_, []); !Fail()
<+ ?Matrix(_, [Row(<all-wildcards>, _, _, _) | _]); !some_dfa
<+ !some_other_dfa
```

Because some of these inlined strategies perform a match as their first operation, this strategy naturally translates to match cases.

```
match sequential
  case | Matrix(_, []): !Fail()
  case | Matrix(_, [Row(<all-wildcards>, _, _, _) | _]): !some_dfa
  case | _: !some_other_dfa
end
```

A precondition to the described optimisation is that calls to strategy definitions can be inlined. In the context of incremental compilers, this is not always possible, as a definition in one module may be extended in another module. Inlining is performed on the Core AST of a single module. Therefore, the incremental Stratego compiler is not aware of the complete definition of strategies (Smits, Konat, and Visser 2020), because definitions may reside in other modules. In this situation, calls cannot be inlined safely. The compiler thus does not perform this kind of inlining.

Combining Transformations The transformations in this chapter can be combined to effectively and efficiently find and transform eligible Core ASTs by applying them to the AST using a `downup` traversal strategy. This strategy first traverses the AST in a top-down manner. It applies the `ChoiceToCaseMatch`, `ExtractMatch` and `ExtractRHS` transformations. After visiting the AST leaves, it traverses the tree bottom-up, applying the `LiftDefault` transformation. After this transformation, we have a Core AST with case matches.

Integrating in the Existing Compiler We extend the Stratego compiler with the case match Core construct and implement the translations as an optional optimisation. Because we extend the Core language, we need to make changes to existing machinery in the compiler. The match cases are scoping constructs, so we add variable binding rules and dead variable elimination rules. We also extend the gradual type checker (Smits and Visser 2020). We type-check a match case by type-checking the match and guard, and checking the type of the RHS strategy in this context. The type of a case match is the least upper bound (LUB) of the types of its match cases.

Summary In this chapter, we developed a transformation from strategy expressions with first-class matches, to case matches. By grouping related match alternatives together, we can compile them together. In the next chapter, we develop a method to compile Stratego case matches.

Chapter 5

Translating a Case Match to a DFA

At this stage, the abstract syntax tree (AST) contains case matches, each of which includes a collection of patterns. Each case match can independently be translated to a lower-level construct — an automaton. Instead of translating each pattern match independently, the case match allows us to compile a collection of matches. Concretely, we can improve compilation of a single match case by using knowledge on which patterns have (not) been (partially) matched before this case.

We compile a case match to a decision tree, or deterministic finite automaton (DFA). As will become clear during this section, a decision tree will never inspect a certain (sub)term more than once, making it an efficient automaton (contrary to backtracking automata; see Section 8.3). This solution builds upon Maranget (2008), which describes a pattern match compilation approach for an abstract subset of Objective Caml (OCaml) (Milner, Tofte, and Harper 1990; Owens 2008) — a source language that is less intricate than Stratego regarding pattern syntax and matching semantics.

Before discussing the compilation algorithm, some concepts need to be introduced.

5.1 Prerequisites

The concepts stem from Maranget (2008), but need to be extended to apply to Stratego. Many intermediate representation (IR) constructs will be introduced in the following sections. Some will be accompanied by a concrete syntax. This syntax *only relates to concrete examples* in this thesis. The concrete syntax cannot be used in Stratego programs.

Patterns Maranget (2008) uses a simplified definition of patterns, considering them to be either a wildcard (i.e., anonymous variable), an application of an n-ary constructor to n patterns ($n \geqslant 0$), or an *or-pattern*; a conjunction of two patterns.

The definition of Stratego patterns (which are just terms) is not a contribution of this thesis, but dictated by the syntax of first-class match patterns. For now, we will only consider unannotated ⟨*Term*⟩s (Listing 5.1); those go by the name of ⟨*PreTerm*⟩.

```
⟨Term⟩  ::= ⟨PreTerm⟩              ⟨PreTerm⟩ ::= ⟨Var⟩
                                              | ⟨Wld⟩
⟨Var⟩   ::= ⟨Id⟩                              | ⟨Id⟩((⟨Term⟩ ,)*)

⟨Wld⟩   ::= _
```

Listing 5.1: Core ⟨*PreTerm*⟩ grammar. This concrete syntax definition is part of the Stratego grammar; it is available in the surface language.

$$
\begin{array}{lcll}
\langle \mathit{Path} \rangle & ::= & \langle \mathit{TermPath} \rangle & \\[1ex]
\langle \mathit{TermPath} \rangle & ::= & \texttt{current} & \text{Current term} \\
 & | & \langle \mathit{TermPath} \rangle \texttt{.} \langle \mathit{int} \rangle & \text{Subterm access (0-based)}
\end{array}
$$

Listing 5.2: Path grammar. The type of an ⟨*TermPath*⟩ is a ⟨*Term*⟩. This abstract syntax is part of the DFA IR.

$$
\begin{array}{lcl}
\langle \mathit{ClauseMatrix} \rangle & ::= & \langle \mathit{Path} \rangle^* \ \langle \mathit{Row} \rangle^* \\[1ex]
\langle \mathit{Row} \rangle & ::= & \langle \mathit{Term} \rangle^* \ \langle \mathit{ID} \rangle^* \ \langle \mathit{Strategy} \rangle \ \langle \mathit{Strategy} \rangle
\end{array}
$$

Listing 5.3: Clause matrix grammar. The ⟨*ClauseMatrix*⟩ features lists of paths and rows. A ⟨*Row*⟩ holds a list of zero or more (*) ⟨*Term*⟩ patterns, scoped variable identifiers (IDs), a guard strategy, and a RHS strategy, which is the action to perform if the pattern matches.

In order to reason about the identity of patterns, we define the notion of *head constructor*; $\mathcal{H}(p)$ gives the set of head constructors of pattern p. The head constructor of a pattern uniquely identifies the shape of its outermost constructor application (including arity, but excluding arguments). Each pattern has at most one head constructor. The definition of $\mathcal{H}(p)$ for the patterns from Listing 5.1 is in Equations (5.1a) to (5.1c).

$$
\begin{aligned}
&\mathcal{H}(\_) &&:= \varnothing && && (5.1a)\\
&\mathcal{H}(v) &&:= \varnothing && (v \text{ is a variable}) && (5.1b)\\
&\mathcal{H}(c(p_1, \ldots, p_a)) &&:= c_a && (a \geqslant 0) && (5.1c)
\end{aligned}
$$

Paths In its basic form, a path is a sequence of integers that describes the path to a subterm (see (Maranget 2008, p. 2), which refers to this concept as 'occurrence'). This basic notion covers a language where all terms are applications of constructors to terms. Listing 5.2 shows the grammar of paths.

5.2 The Clause Matrix

To reason about match cases in an algorithm, an abstraction of the case match construct is useful. We introduce the clause matrix, an abstraction of a collection of match cases (Maranget 2008, p. 2). In its basic form, it relates patterns to corresponding actions. The action is the right-hand side (RHS), a strategy in the case of Stratego, which will be executed once the pattern matches. Besides abstracting over semantics details that are not relevant in examples, it helps by enabling us to formulate several operations in terms of matrix rows/columns. A formal description of the clause matrix can be found in Listing 5.3.

The translation from a concrete case match to a clause matrix is trivial. Each match case maps to a matrix row. As an example, consider the match from Listing 4.1, replacing the patterns by term variables and the strategies by strategy calls (Listing 5.4). These cases result in the matrix in Listing 5.5, in Annotated Term (ATerm) format. Note that in this example, the matrix contains some sugared terms and strategies for brevity. In reality, this would be more involuted. The first argument of the matrix is a list of paths; one for each column of patterns. Initially, each row has exactly one pattern. Consequently, the matrix has a single column. The single path '`current`' (abstract syntax `CurP()`) signifies that the patterns should match the current term — as semantics dictate. To be able to process them uniformly in the following steps, each row has the same signature and carries a guard. This example shows that, if the

```
match sequential
  case        | p1        : a1
  case x, xs  | p2        : a2
  case        | _  when g3: a3
end
```

Listing 5.4: Case match example.

```
Matrix(
  [CurP()]
, [
    Row([Var("p1")], []          , Id()                 , CallNoArgs(SVar("a1")))
  , Row([Var("p2")], ["x", "xs"], Id()                 , CallNoArgs(SVar("a2")))
  , Row([Wld()]    , []          , CallNoArgs(SVar("g3")), CallNoArgs(SVar("a3")))
  ]
)
```

Listing 5.5: ATerm matrix representation of match cases in Listing 5.4.

$$\overline{\pi} := \begin{vmatrix} \text{current} \end{vmatrix} \tag{5.2a}$$

$$M := \begin{vmatrix} \text{p1} & \rightarrow \text{a1} \\ \text{p2} & \rightarrow \text{a2} \\ \_ & \rightarrow \text{a3} \end{vmatrix} \tag{5.2b}$$

Listing 5.6: Mathematical representation of match cases in Listing 5.4.

match case has no guards, the guard `id` — doing nothing — will be used. Mathematically, we write this clause matrix as Equation (5.2b). We omit guards and scoped variables. We also store the paths in a separate vector (Equation (5.2a)).

Matrix Decomposition Maranget (2008) builds the algorithm upon matrix decomposition operations. We define two operations on clause matrices, which represent how a subterm match influences the necessary remaining matches. Both operations filter rows. In addition to filtering a row, it is transformed.

The first operation, *specialisation* by constructor c, describes which patterns from the clause matrix $P \rightarrow A$ match a certain head constructor. It can be written as a function $S_m(c, P \rightarrow A)$. Specialisation removes the rows that do not admit the given constructor. Furthermore, it unwraps the head constructor, revealing any arguments that might need to be matched subsequently. The equations in Figure 5.1 define this formally. The equations in Figure 5.2 define the corresponding specialisation of paths; it defines at which paths the 'new' patterns resulting from matrix specialisation reside.

As an example, given the clause matrix representing the patterns in Listing 4.1 with variables simplified to wildcard patterns and RHSs simplified to unique IDs:

$$M := \begin{vmatrix} \text{Nil}() & \rightarrow & 1 \\ \text{Cons}(\_, \_) & \rightarrow & 2 \\ \_ & \rightarrow & 3 \end{vmatrix} \tag{5.3}$$

Specialising this matrix by each of the constructors that occur in its first column yields the following matrices.

$$\mathcal{S}_m(\text{Nil}(), M) := \begin{vmatrix} \rightarrow & 1 \\ \rightarrow & 3 \end{vmatrix} \tag{5.4a}$$

$$\mathcal{S}_m(c_a, \begin{vmatrix} p_1^1 & p_2^1 \dots p_n^1 & \rightarrow & A^1 \\ \vdots & \vdots & & \vdots \\ p_1^m & p_2^m \dots p_n^m & \rightarrow & A^m \end{vmatrix}) \coloneqq \begin{vmatrix} \mathcal{S}'_m(c_a, p_1^1 & p_2^1 \dots p_n^1 & \rightarrow & A^1) \\ \vdots & \vdots & & \vdots \\ \mathcal{S}'_m(c_a, p_1^m & p_2^m \dots p_n^m & \rightarrow & A^m) \end{vmatrix} \tag{5.5a}$$

$$\mathcal{S}'_m(c_a, c(q_1, \dots, q_a) \quad p_2^j \dots p_n^j \rightarrow A^j) \coloneqq q_1 \dots q_a \quad p_2^j \dots p_n^j \rightarrow A^j \tag{5.5b}$$

$$\mathcal{S}'_m(c_a, c'(q_1, \dots, q_{a'}) \quad p_2^j \dots p_n^j \rightarrow A^j) \quad \text{if } (c' \neq c \vee a' \neq a) \coloneqq \text{no row} \tag{5.5c}$$

$$\mathcal{S}'_m(c_a, \_ \quad p_2^j \dots p_n^j \rightarrow A^j) \coloneqq \_ \times a \quad p_2^j \dots p_n^j \rightarrow A^j \tag{5.5d}$$

Figure 5.1: Definition of clause matrix specialisation $S_m(c, P \rightarrow A)$.

$$\mathcal{S}_\pi(c_a, \pi_1, \pi_2 \dots \pi_n) \coloneqq \mathcal{S}'_\pi(c_a, \pi_1)\ \pi_2 \dots \pi_n \tag{5.6a}$$

$$\mathcal{S}'_\pi(c_a, \pi_1) \coloneqq \pi_1 \cdot 1 \dots \pi_1 \cdot a \tag{5.6b}$$

Figure 5.2: Definition of path vector specialisation $\mathcal{S}_\pi(c, \overline{\pi})$.

$$D(\begin{vmatrix} p_1^1 & p_2^1 \dots p_n^1 & \rightarrow & A^1 \\ \vdots & \vdots & & \vdots \\ p_1^m & p_2^m \dots p_n^m & \rightarrow & A^m \end{vmatrix}) \coloneqq \begin{vmatrix} D'(p_1^1 & p_2^1 \dots p_n^1 & \rightarrow & A^1) \\ \vdots & \vdots & & \vdots \\ D'(p_1^m & p_2^m \dots p_n^m & \rightarrow & A^m) \end{vmatrix} \tag{5.7a}$$

$$D'(c(q_1, \dots, q_a) \quad p_2^j \dots p_n^j \rightarrow A^j) \coloneqq \text{no row} \tag{5.7b}$$

$$D'(\_ \quad p_2^j \dots p_n^j \rightarrow A^j) \coloneqq p_2^j \dots p_n^j \tag{5.7c}$$

Figure 5.3: Definition of clause matrix default $D(P \rightarrow A)$.

$$\mathcal{S}_m(\text{Cons}(\_,\_), M) \coloneqq \begin{vmatrix} \_ & \_ & \rightarrow & 2 \\ \_ & \_ & \rightarrow & 3 \end{vmatrix} \tag{5.4b}$$

Informally, this reads as follows: if we establish that the outer constructor of the scrutinee is `Nil()` (the empty list constructor), we have no more patterns to match, and our set of finally matching RHSs is `{1, 3}` (Equation (5.4a)). Specialising by `Cons(_, _)` (the list element constructor) instead yields a matrix with two pattern columns (Equation (5.4b)); those might still need to be matched, in order to know which RHSs to perform.

The second matrix decomposition operation that Maranget (2008) defines is the default matrix computation. The default $D(P \rightarrow A)$ of a matrix $P \rightarrow A$ is the matrix of patterns that remain after matching a constructor that is *not* in the first column of the matrix. In other words; the patterns from the matrix whose first patterns match all constructors that are not in the first column of patterns. These simply are the rows that start with a wildcard pattern. The formal default matrix computation can be found in Figure 5.3.

Note that both decomposition operations consider only the first column of patterns. In order to be able to reason about matrix column i ($i \neq 1$), we define a column swap operation.

Column Swap The $\text{Swap}_i(\overline{v})$ operation changes the order of the elements in $\overline{v}$, which might be a vector or matrix. In the case of a matrix, swap is a column-wise operation, i.e., it considers a matrix as a vector of column elements. Particularly, it removes element i and inserts it at the

⟨*Strategy*⟩	::=	...	
	|	⟨*DFA*⟩	(A DFA is a Core strategy)
⟨*DFA*⟩	::=	`fail`	Failure
	|	`switch` ⟨*Path*⟩ (⟨*Alt*⟩ `\n`)* `default:` ⟨*Strategy*⟩ `end`	Multi-way switch on subterm constructor
⟨*Alt*⟩	::=	`alt` ⟨*Pat*⟩`:` ⟨*Strategy*⟩	Subterm pattern match alternative
⟨*Pat*⟩	::=	⟨*Id*⟩`/`⟨*Int*⟩	Constructor application
	|	⟨*String*⟩	String literal
	|	⟨*Int*⟩	Integer literal
	|	⟨*Real*⟩	Real literal
	|	`()/`⟨*Int*⟩	Tuple

Listing 5.7: DFA grammar.

first position.

$$\text{Swap}(\begin{vmatrix} v_1 \dots v_{i-1} & v_i & v_{i+1} \dots v_n \end{vmatrix}) \coloneqq \begin{vmatrix} v_i & v_1 \dots v_{i-1} & v_{i+1} \dots v_n \end{vmatrix} \tag{5.8}$$

Having the definition of the clause matrix, a method of generating a matrix for a case match construct, and definitions for matrix decomposition operations, we can continue to define the compilation algorithm.

5.3 Building a DFA from the Clause Matrix

In this step, which is the centre of gravity of the pattern match compilation algorithm, a DFA — decision tree — is generated from a clause matrix. The idea for this algorithm, as the preliminaries from previous sections, comes from Maranget (2008).

Target Language The target language of our pattern patch compiler is the DFA, of which the grammar is in Listing 5.7. As with the preliminary constructs, the concrete syntax only exists to support development and examples in this thesis, and will not be adopted in the Stratego Core syntax.

Compilation The compilation algorithm is a recursive function $\mathcal{CC}$, which takes a vector of subterm paths $\overline{\pi}$ and a clause matrix $P \to A$ as its arguments. It is defined by several cases, each of which covers a specific state of the matching automaton based on simple conditions on the patterns in the clause matrix. These conditions are not mutually exclusive. Therefore, the cases should be evaluated in this order. For these conditions, consider P to be the matrix of patterns in the clause matrix, which has m rows and n columns.

$$\begin{vmatrix} p_1^1 \dots p_n^1 & \to & A^1 \\ \vdots & & \vdots \\ p_1^m \dots p_n^m & \to & A^m \end{vmatrix} \coloneqq \begin{vmatrix} P & \to & A \end{vmatrix} \tag{5.9}$$

P **has no rows** $(m = 0)$**.** In this case, matching fails, as no row matches.

$$\mathcal{CC}(\overline{\pi}) \coloneqq \text{'}\texttt{fail}\text{'} \tag{5.10}$$

All patterns in the first row of P are wildcard patterns ($m > 0$). As a wildcard matches any subterm, matching will succeed on the first row, yielding the respective RHS strategy. Note that this case applies when there are multiple empty rows.

$$\mathcal{CC}(\overline{\pi}, \left|\begin{matrix} \_\cdots\_ & \to & A^1 \\ p_1^2 \cdots p_n^2 & \to & A^2 \\ \vdots & & \vdots \\ p_1^m \cdots p_n^m & \to & A^m \end{matrix}\right|) := A^1 \tag{5.11}$$

Any other case ($m > 0 \wedge n > 0$). In this case, there exists at least one column of which at least one pattern is not a wildcard (if all columns only contain wildcards, so do the rows, in which case the previous compilation case applies). Matching a column with only wildcards is redundant; it does not provide any information about the scrutinee. Instead, we select a column i which has at least one non-wildcard pattern. For now, this selection method will be opaque and non-deterministic; we revisit this later. Depending on the value of i, one of these cases applies.

- If $i \neq 1$, the selected column is not the first column. As we define the decomposition operations w.r.t. the patterns in the first column, it is necessary to make the selected column the first column. In this case, we define the following.

  $$\mathcal{CC}(\overline{\pi}, P \to A) := \mathcal{CC}(\text{Swap}_i(\overline{\pi}), \text{Swap}_i(P) \to A) \tag{5.12}$$

  We swap column i to position 1 and recursively apply the algorithm $\mathcal{CC}$ to the resulting clause matrix.

- Else, $i = 1$; compilation should yield a (partial) decision tree that will match a certain subterm. The compiler considers the first column of patterns and constructs an appropriate switch to decide between these cases. It needs a single alternative for every possible head constructor. Therefore, compute the *set* of head constructors C.

  $$C := \bigcup_{j=1}^{m} \mathcal{H}(p_1^j) \tag{5.13}$$

  The function $\alpha(c)$ defines the construction of a switch alternative for constructor c (see the grammar in Listing 5.7.) An alternative's RHSs is determined by a recursive call to $\mathcal{CC}$ with a derived path vector and matrix, both specialised to c. Matrix specialisation on c defines the matches that need to be matched if the current subterm matches c, while path specialisation defines their respective subterm positions — exactly reflecting the matches that the tree at the RHS of the alternative should perform.

  $$\alpha(c) := \text{Alt}(c, \mathcal{CC}(\mathcal{S}_\pi(\overline{\pi}), \mathcal{S}_m(c, P \to A))) \tag{5.14}$$

  Using this definition of $\alpha(c)$, the set of alternatives $\mathcal{A}$ is defined.

  $$\mathcal{A} := \{\, \alpha(c_k) \mid c_k \in C \,\} \tag{5.15}$$

  The subterm being matched might admit none of the head constructors at runtime. In that case, none of the alternatives succeeds. In this situation, the switch falls back on default δ.

  $$\delta := \mathcal{CC}(\pi_2 \ldots \pi_n, \mathcal{D}(P \to A)) \tag{5.16}$$

  With the set of switch alternatives $\mathcal{A}$ and the default alternative δ, the result of this call to $\mathcal{CC}$ can be defined as the following switch, from the DFA grammar (Listing 5.7).

  $$\mathcal{CC}(\overline{\pi}, P \to A) := \text{Switch}(\pi_1, \mathcal{A}, \delta) \tag{5.17}$$

This compilation scheme closely resembles Maranget's. The reader interested in referencing the original paper should be informed of the following changes.

```
⟨Term⟩ ::= ⟨Var⟩                          ⟨PreTerm⟩ ::= ⟨Var⟩
         | ⟨Wld⟩                                    | ⟨Wld⟩
         | ⟨PreTerm⟩{^⟨PreTerm⟩}                    | ⟨Int⟩
         | ⟨Var⟩@⟨Term⟩                             | ⟨Real⟩
         | ⟨TermPath⟩                               | ⟨Str⟩
                                                    | ⟨Id⟩((⟨Term⟩ ,)*)
⟨Var⟩  ::= ⟨Id⟩                                     | ⟨Term⟩#⟨Term⟩
                                                    | ⟨Var⟩@⟨PreTerm⟩
⟨Wld⟩  ::= _
```

Listing 5.8: Core ⟨*Term*⟩ grammar. This concrete syntax definition is part of the Stratego grammar; it is available in the surface language. A pattern is a term. A ⟨*PreTerm*⟩ is a ⟨*Term*⟩ without annotations, which are terms.

Omitting swap nodes. Instead, we use a recursive application of the compilation algorithm to a path vector and clause matrix, after swapping their columns.

Every switch has a default value. Because the Stratego type system builds on an open-world assumption, it cannot know all constructors of a certain type at compile time. Thus, it can not be determined at compile time whether a set of constructors fully covers a signature, and a switch should always have a default case to catch unexpected constructors.

Simplifying leaf nodes to their contents. Instead of using $\mathrm{Leaf}(A)$ to construct an action strategy in the DFA, we use A in Equation (5.11). Explicitly wrapping leaf nodes only convolute the compiler, without adding functionality, and are therefore omitted.

The presented algorithm assumes simplifications and source/target languages which do not fully apply to Stratego. Section 5.4 presents extensions that adapt the to apply to Stratego specifically.

5.4 Extending the Basic Compilation Scheme

The previous section introduced the idea for a basic compilation scheme, but it does not apply to all features of Stratego. This section presents extensions to apply the pattern match compilation algorithm to Stratego Core. Some extensions pertain to the wider range of Stratego Core patterns, others to intricacies of the Stratego internals.

Annotations on Patterns Listing 5.1 lists only unannotated terms, preterms, as possible patterns. In Stratego Sugar, terms may optionally be annotated, but in desugared Core, each term is explicitly annotated. An annotation is a preterm. Therefore, the real grammar of terms (and thus patterns) is as in Listing 5.8. Effectively, this modified grammar only adds a rule for an annotated term `pt1{^pt2}`. During desugaring, existing compiler machinery desugars any unannotated pattern (match term) `t` to `t{^_}`; a wildcard pattern `_` becomes the explicit annotation. Pre-existing annotations that are not a list, like `Var(x){^"bound"}`, are transformed to a list.

In order to adapt the algorithm to this extended grammar, extend head constructor definition $\mathcal{H}$.

$$\mathcal{H}(pt1\{^\wedge pt2\}) := \text{`}anno' \tag{5.18}$$

Furthermore, extend the definitions for specialisation.

$$\mathcal{S}'_m(\text{‘anno’}, pt1\{^\wedge pt2\}\, p^j_2 \dots p^j_n) := pt1\, pt2\, p^j_2 \dots p^j_n \tag{5.19}$$

$$\mathcal{S}'_\pi(\text{‘anno’}, \pi_1) := \pi_1\, \pi_1 \cdot \text{‘anno’} \tag{5.20}$$

$\langle \textit{TermPath} \rangle$ ::= ...
  | $\langle \textit{TermPath} \rangle$`.anno`

Listing 5.9: Path grammar extension: annotations. π.anno accesses the annotation of π.

$\langle \textit{Path} \rangle$ ::= ...
  | $\langle \textit{FieldPath} \rangle$

$\langle \textit{TermFieldPath} \rangle$ ::= $\langle \textit{TermPath} \rangle$
  | $\langle \textit{FieldPath} \rangle$

$\langle \textit{FieldPath} \rangle$ ::= ...
  | $\langle \textit{TermPath} \rangle$`.int`
  | $\langle \textit{TermPath} \rangle$`.real`
  | $\langle \textit{TermPath} \rangle$`.str`

Listing 5.10: Path grammar extension: literals.

$$\mathcal{H}(x) \coloneqq \text{string} \qquad (x \text{ is a string literal}) \tag{5.21a}$$
$$\mathcal{H}(n) \coloneqq \text{int} \qquad (n \text{ is an integer literal}) \tag{5.21b}$$
$$\mathcal{H}(r) \coloneqq \text{real} \qquad (r \text{ is a real literal}) \tag{5.21c}$$

Figure 5.4: Definition of head constructor for literals.

These three added rules allow the compiler to effectively handle the annotation term as any other term. Equation (5.20) uses `anno`; this path element serves to query the annotations of a term. Listing 5.9 shows this extension to the path grammar.

Term Types As stated in Section 3.2, Stratego terms are more flexible than just variables and constructor applications. The Stratego back-end uses the ATerm library (van den Brand, de Jong, et al. 2000), and its corresponding term types. The back-end of the compiler (which is addressed in Chapter 6) handles different term types differently; a distinction that needs to be made in the DFA generation as well. This extension addresses the special handling of literals, lists and tuples.

Literals The pattern grammar (Listing 5.8) lists $\langle \textit{Str} \rangle$, $\langle \textit{Int} \rangle$ and $\langle \textit{Real} \rangle$ terms. Although they are technically not related, this thesis refers to them as 'literals'. Literals are terminal patterns; they do not accept term arguments. To match a literal pattern, only the literal value needs to be checked. This literal value is not a term (refer to Chapter 6 for details). Instead, it will be referred to as a 'field' — a literal term is merely a box or wrapper around its field.

To explicitly access these fields in the DFA, the path grammar needs an additional path element for each literal, as in Listing 5.10. Figures 5.4 and 5.5 show the necessary head constructor and specialisation definitions.

A DFA switch node only switched on constructors before this change. Now, it should also be able to switch on a literal value. To distinguish between these cases, introduce a function $\mathcal{T}$ that determines the type of its pattern argument (Figure 5.6). Now, the information about the type of the current subterm can be used in an adaptation of the definition of the switch. The new function $\Pi(\pi, \tau)$ appends the path π with the appropriate path element to switch on to match a term of type τ. For example, to match a constructor application, the DFA matches on its constructor (before recursively matching its subterms).

$$\Pi(\pi, \text{Appl}) \coloneqq \pi \cdot \text{constr} \tag{5.25a}$$
$$\Pi(\pi, \text{String}) \coloneqq \pi \cdot \text{str} \tag{5.25b}$$
$$\Pi(\pi, \text{Int}) \coloneqq \pi \cdot \text{int} \tag{5.25c}$$
$$\Pi(\pi, \text{Real}) \coloneqq \pi \cdot \text{real} \tag{5.25d}$$

$$\mathcal{S}'_m(\text{'string'}, \quad x \quad p^j_2 \dots p^j_n \to A^j) := p^j_2 \dots p^j_n \to A^j \qquad (x \text{ is a string literal}) \tag{5.22a}$$
$$\mathcal{S}'_m(\text{'int'}, \quad n \quad p^j_2 \dots p^j_n \to A^j) := p^j_2 \dots p^j_n \to A^j \qquad (n \text{ is an integer literal}) \tag{5.22b}$$
$$\mathcal{S}'_m(\text{'real'}, \quad r \quad p^j_2 \dots p^j_n \to A^j) := p^j_2 \dots p^j_n \to A^j \qquad (r \text{ is a real literal}) \tag{5.22c}$$

$$\mathcal{S}'_\pi(\text{'string'}, \pi_1) := \varnothing \tag{5.23a}$$
$$\mathcal{S}'_\pi(\text{'int'}, \pi_1) := \varnothing \tag{5.23b}$$
$$\mathcal{S}'_\pi(\text{'real'}, \pi_1) := \varnothing \tag{5.23c}$$

Figure 5.5: Definition of specialisation for literals. Because literals have no pattern arguments, clause matrix row specialisation (S'_m) does not unwrap any patterns. This is also reflected in S_π; it does not add any paths.

$$\mathcal{T}(c(p_1, \dots, p_a)) := \text{Appl} \tag{5.24a}$$
$$\mathcal{T}(x) := \text{String} \tag{5.24b}$$
$$\mathcal{T}(n) := \text{Int} \tag{5.24c}$$
$$\mathcal{T}(r) := \text{Real} \tag{5.24d}$$

Figure 5.6: Definition of the pattern type function $\mathcal{T}(p)$ determines the term type of pattern p. The term types correspond to the types as defined by the ATerm library.

```
⟨TermPath⟩ ::= ...
            |  ⟨TermPath⟩.size
```

Listing 5.11: Path grammar extension: tuples. π.size accesses the arity of a tuple.

Subsequently, Equation (5.17) can be redefined to first obtain the type τ of the subterm, and use $\Pi(\pi, \tau)$ to generate a path for the switch.

$$\tau := \mathcal{T}(p^1_1) \tag{5.26a}$$
$$\mathcal{CC}(\overline{\pi}, P \to A) := \text{Switch}(\Pi(\pi_1, \tau), \mathcal{A}, \delta) \tag{5.26b}$$

Equation (5.26a) defines the type of the switch as the type of the first pattern in the column. Effectively, this means that all patterns in the column should have the same type. In a strictly typed language, this is realistic, but in Stratego, it is not. For now, we assume this is a reasonable simplification. The proper solution to this is in Section 5.4.

Tuples Another type that the Stratego implementation handles separately is the tuple. In the abstract syntax however, it is just an application of a special constructor with a unique name ("", the empty string) to an arbitrary number ($a \geqslant 0$) of arguments. Therefore, to support tuples, there is no need for separate definitions of head constructor $\mathcal{H}$, matrix specialisations $\mathcal{S}'_m$ and $\mathcal{S}'_\pi$, and default $\mathcal{D}'$; Equations (5.1c), (5.5b), (5.6b) and (5.7b), although initially defined for constructor applications, naturally extend to tuples.

A new term field path (Listing 5.11), and corresponding type-related definitions, are needed to check the size of tuples — tuples with different arities do never match. If multiple tuple patterns with different switch arms exist in the pattern column, each switch arm will represent an arity.

```
⟨TermPath⟩ ::= ...
             | ⟨TermPath⟩.head
             | ⟨TermPath⟩.tail
```

Listing 5.12: Path grammar extension: lists. These path elements access the respective parts of a list.

$$\mathcal{T}((p_1, \ldots, p_a)) := \text{Tuple} \tag{5.27a}$$
$$\Pi(\pi, \text{Tuple}) := \pi \cdot \text{size} \tag{5.27b}$$

Lists The last pattern type that Stratego features, are lists. Possible list constructors are `Cons/2` (often written as `[x | xs]`, where `x` is an element and `xs` a list), which reflects prepending a single element (head) to a list (tail) and `Nil/0` (written as `[]`), the empty list. Lists have no static size; it can be computed at runtime by recursing into the tail of the list until encountering an empty list. A switch thus does not distinguish lists on their size property, but on their constructor (`Cons` or `Nil`). In the Stratego implementation, this constructor is implicit. However, we use the explicit `lcon` path suffix to denote switching on the list constructor. This is reflected in the definitions for lists.

$$\mathcal{S}'_\pi(\text{'Nil'}, \pi_1) := \varnothing \tag{5.28a}$$
$$\mathcal{S}'_\pi(\text{'Cons'}, \pi_1) := \pi_1 \cdot \text{head} \quad \pi_1 \cdot \text{tail} \tag{5.28b}$$
$$\mathcal{T}(\text{Nil}()) := \text{List} \tag{5.28c}$$
$$\mathcal{T}(\text{Cons}(\text{p1}, \text{p2})) := \text{List} \tag{5.28d}$$
$$\Pi(\pi, \text{List}) := \pi \cdot \text{lcon} \tag{5.28e}$$

There is no need to extend to definition of H; Equation (5.1c) applies. The definitions of $\mathcal{S}'_m$ for constructor applications (Equation (5.5b)) and $\mathcal{D}$ (Equation (5.7b)) also apply to lists.

Other ATerm types The ATerm library supports additional types; constructors, references, blobs, and placeholders. Patterns of these types do no exist in the Stratego syntax. Therefore, the pattern match compiler does not handle these types.

Type Switches Because of how Stratego is implemented, the term types are more important than presented until now. Some path elements only apply to certain term types (e.g., `int` is only valid for integer terms). Using a path ending in an inappropriate path element might lead to runtime errors on certain terms. Therefore, while matching, a type check precedes the equality check on the current term path in a switch alternative. As this results in a lot of checks for patterns of the same type, it makes sense to first switch on the type, and only after that, switch on applicable patterns for that type. To this end, we implement a path element that accesses a term's ATerm type. The full path grammar is in Listing 5.13. We also extend our earlier DFA target language (Listing 5.7) to incorporate type switches. The new grammar, which nests value switches (the ones we considered in the basic algorithm) inside type switches. The new grammar is in Listing 5.14. Listing 6.3 shows an example of a DFA with type switches. With this extension, the compilation scheme becomes a 2-stage algorithm; the first stage generates a type switch, and for each type alternative, calls a type-dependant function to create the value switch. Each value alternative represents a constructor/literal as

```
⟨Path⟩           ::= ⟨TermPath⟩
                   | ⟨TypePath⟩
                   | ⟨FieldPath⟩

⟨TermPath⟩       ::= current
                   | ⟨TermPath⟩.⟨int⟩
                   | ⟨TermPath⟩.anno
                   | ⟨TermPath⟩.head
                   | ⟨TermPath⟩.tail

⟨TermFieldPath⟩  ::= ⟨TermPath⟩
                   | ⟨FieldPath⟩

⟨TypePath⟩       ::= ⟨TermPath⟩.type

⟨FieldPath⟩      ::= ⟨TermPath⟩.size
                   | ⟨TermPath⟩.con
                   | ⟨TermPath⟩.int
                   | ⟨TermPath⟩.real
                   | ⟨TermPath⟩.str
                   | ⟨TermPath⟩.lcon
```

Listing 5.13: Extended path grammar. The various ⟨*Path⟩ sorts serve to be explicit about the type of the path. The type of an ⟨*TermPath*⟩ is a ⟨*Term*⟩, ⟨*TypePath*⟩ returns a ⟨*TermType*⟩, and ⟨*FieldPath*⟩ returns an internal field of ⟨*Term*⟩. The purpose of the latter two will become clear in Chapter 6. This abstract syntax is part of the DFA IR.

```
⟨Strategy⟩ ::= ...
             | ⟨DFA⟩                                                        A DFA is a Core strategy

⟨DFA⟩      ::= ⟨SwitchT⟩

⟨SwitchT⟩  ::= switch ⟨TypePath⟩ (⟨AltT⟩ \n)* default: ⟨Strategy⟩ end                 Type switch

⟨AltT⟩     ::= alt ⟨PatT⟩: ⟨SwitchV⟩                                    Subterm type alternative

⟨SwitchV⟩  ::= switch ⟨TermFieldPath⟩ (⟨AltV⟩ \n)* default: ⟨Strategy⟩ end          Value switch

⟨AltV⟩     ::= alt ⟨PatV⟩: ⟨Strategy⟩                                  Subterm value alternative

⟨PatT⟩     ::= APPL                                               Constructor application type
             | LIST                                                                  List type
             | INT                                                                Integer type
             | REAL                                                                  Real type
             | STRING                                                              String type
             | TUPLE                                                                Tuple type

⟨PatV⟩     ::= ⟨Id⟩/⟨Int⟩                                              Constructor application
             | ⟨String⟩                                                          String literal
             | ⟨Int⟩                                                            Integer literal
             | ⟨Real⟩                                                              Real literal
             | ()/⟨Int⟩                                                                   Tuple
```

Listing 5.14: DFA grammar (extended).

before.

$$\mathcal{A}_v(\tau) := \{\, \alpha(c_k) \mid c_k \in C \wedge \mathcal{T}(c_k) = \tau \,\} \tag{5.29a}$$
$$\alpha_t(\tau) := \mathrm{Alt}_t(\tau, \mathrm{Switch}_v(\Pi(\pi_1, \tau), \mathcal{A}_v(\tau), \delta)) \tag{5.29b}$$
$$\mathcal{A}_t := \{\, \mathcal{T}(c_k) \mid c_k \in C \,\} \tag{5.29c}$$
$$\mathcal{CC}(\overline{\pi}, P \to A) := \mathrm{Switch}_t(\pi_1 \cdot \mathrm{type}, \mathcal{A}_\tau, \delta) \tag{5.29d}$$

$\mathcal{A}_v(\tau)$ defines the set of values alternatives for type τ. $\alpha_t(\tau)$ is a function to construct a type alternative for type τ. $\mathcal{A}_\tau$ defines the set of types for the current matrix column. These equations reuse definitions of C, $\mathcal{T}(c)$, $\Pi(\pi, \tau)$, and δ as constructed in this chapter.

Guards Until here, the compiler ignored the existence of guards in the clause matrix rows. To incorporate guards in the matrix, make A a two-column matrix holding guards and actions.

$$A := \begin{vmatrix} g^1 & a^1 \\ \vdots & \vdots \\ g^m & a^m \end{vmatrix} \tag{5.30}$$

The presence of guards does not influence the majority of definitions for the algorithm; only when pattern rows fully match — the DFA would produce a leaf; Equation (5.11) — the guard should be evaluated. To support guards, we can redefine the leaf rule as follows (Pettersson 1999, p. 106).

$$\mathcal{CC}(\overline{\pi}, \begin{vmatrix} \_ \cdots \_ & \to & g^1 & a^1 \\ p_1^2 \cdots p_n^2 & \to & g^2 & a^2 \\ \vdots & & \vdots & \\ p_1^m \cdots p_n^m & \to & g^m & a^m \end{vmatrix}) := \text{if}\, g^1 \text{ then}\, a^1 \text{ else}\, \mathcal{CC}(\overline{\pi}, \begin{vmatrix} p_1^2 \cdots p_n^2 & \to & g^2 & a^2 \\ \vdots & & \vdots & \\ p_1^m \cdots p_n^m & \to & g^m & a^m \end{vmatrix}) \tag{5.31}$$

Informally, this means that we compile a guard to a branching construct. If the guard succeeds, execute the corresponding action. Otherwise, try potential other patterns that are still matchable at this point. If no patterns remain, fail.

Exploded Patterns Stratego features *exploded patterns*; a construct to access constructor names and term arguments as values. The strategy `?c#(xs)` binds the constructor name to `c` and the list of arguments to `xs`; `c#(xs)` is an exploded term. Although the semantics of exploded terms for constructor application are fairly straightforward, this is not the case for other exploded terms (e.g., lists). Therefore, processing them properly in the pattern match compilation scheme is not trivial. However, because exploded terms appear sparingly in matches and often just serve to bind variables in a pattern that always matches, processing them may not lead to a noticeable performance difference in many cases. Instead of incorporating the intricate semantics in the match compilation scheme, we handle exploded terms differently. Once one appears in a subpattern position, it is replaced by a variable. This leaves the match of the exploded term with the value bound to the variable to be done, in order to match the row. To that extend, prepend this check, a match of the variable on the exploded term pattern, to the existing row guard.

Concretely, by rewriting rows with Equation (5.32) before determining head constructors, exploded patterns can be correctly handled without added complexity.

$$\begin{array}{llll} p_1 \ \dots \ p_j \ \dots \ p_n \to \texttt{g} & a & \text{where}\, p_j \text{ is an exploded term} & \Longrightarrow \\ p_1 \ \dots \ x \ \dots \ p_n \to \texttt{where(p}_\texttt{j}\texttt{ := x); g} & a & \text{where}\, x \text{ is a fresh variable} & \end{array} \tag{5.32}$$

This solution is only suboptimal in rare cases, where multiple non-trivial exploded pattern matches appear in a case match.

```
match sequential
  case | ([], _): !1
  case | (_, []): !2
  case | ([x | xs], [y | ys]): !3
end
```

Listing 5.15: List merge, using a case match (Stratego).

As-Patterns As-patterns, of the form `x@t`, bind `x` to `t` in a match. By separating the binding from the pattern, the compilation scheme trivially extends to as-patterns. The binding can be moved to the guard, while preserving semantics (Equation (5.33)).

$$p_1 \ \ldots \ x@p_j \ \ldots \ p_n \rightarrow \texttt{g} \quad a \quad \Longrightarrow \quad p_1 \ \ldots \ p_j \ \ldots \ p_n \rightarrow \texttt{where(x := }\pi_\texttt{j}\texttt{); g} \quad a \tag{5.33}$$

Nested as-patterns (e.g., `x1@x2@p`) can be normalised by repeating this rewrite.

Variable Patterns The basic compilation scheme omits variable patterns; it only considers wildcard patterns. Extending to variables means that, if the match is successful, the variable should be bound to the subterm at its position. In Stratego, a variable in a pattern might already be bound to a value. In that case, the match becomes an equality check. If the variable value does not equal the subterm at its position, the match fails.

The described semantics come from the semantics of the first-class match, but should also apply to the case match. The separation of matching and binding suggests a natural translation to as-patterns (Section 5.4). A simple preprocessing step (Equation (5.34)) indeed employs this similarity.

$$p_1 \ \ldots \ x \ \ldots \ p_n \rightarrow g \quad a \quad \Longrightarrow \quad p_1 \ \ldots \ x@\_ \ \ldots \ p_n \rightarrow g \quad a \tag{5.34}$$

Heuristic Column Selection In its basic, abstract form, the compilation scheme does not specify *how* to select column i. In passing, Maranget defines 'naive compilation' to select the minimal i such that at least one pattern p_i^j is not a wildcard. This is not necessarily optimal.

Optimally minimising a decision tree is an NP-complete problem (Sekar, Ramesh, and Ramakrishnan 1995). Therefore, the compiler employs heuristics to *approximate* an optimal (w.r.t. some metric) DFA. Selecting columns heuristically can reduce code size (e.g., by preventing duplication) and may improve performance (by matching common subterms first). A lot of heuristics have been documented and compared in literature (see Section 8.4). Baudinet and Macqueen (1985) suggests a combination of heuristics where subsequent heuristics break ties from earlier ones.

First row This heuristic favours columns that have a non-wildcard pattern at the top.

Small branching factor This heuristic favours columns with the smallest number of head constructors.

Arity This heuristic favours columns with the smallest sum of arities of their patterns.

To make an effort to maintain reasonable code size, the extended scheme incorporates this combination of heuristics.

Example Call The extended compilation scheme handles all Stratego patterns and generates a DFA that matches the internal representation of terms. As an example, we apply our compilation scheme to the case match in Listing 5.15. This results in the Stratego IR in Listing 5.16.

```
switch current.type
  alt TUPLE:
    switch current.size
      alt ()/2:
        let
          comp_0_0_dfa1( | ) = switch current.1.type
                                 alt LIST:
                                   switch current.1.lcon
                                     alt Nil/0: !2{ }
                                     default: fail
                                   end
                                 default: fail
                               end
        in
          switch current.0.type
            alt LIST:
              switch current.0.lcon
                alt Nil/0: !1{ }
                alt Cons/2:
                  switch current.1.type
                    alt LIST:
                      switch current.1.lcon
                        alt Nil/0: !2{ }
                        alt Cons/2:
                          {comp_0_0_x0, comp_0_0_xs0, comp_0_0_y0, comp_0_0_ys0,
                           where49, where50, where51, where52:
                             ?where49; !current.1.tail; ?comp_0_0_ys0; !where49
                           ; ?where50; !current.1.head; ?comp_0_0_y0; !where50
                           ; ?where51; !current.0.tail; ?comp_0_0_xs0; !where51
                           ; ?where52; !current.0.head; ?comp_0_0_x0; !where52
                           ; !3{ } }
                        default: fail
                      end
                    default: fail
                  end
                default: comp_0_0_dfa1
              end
            default: comp_0_0_dfa1
          end
        end
      default: fail
    end
  default: fail
end
```

Listing 5.16: The result of compiling the pattern match in Listing 5.15.

```
switch current.type
  alt INT:
    switch current.int
      alt 1: s1
      alt 2: s2
      // alternatives 3..999998
      alt 999999: s999999
      alt 1000000: s1000000
      default: s_default
    end
end
```

```
switch current.type
  alt INT:
    switch current.int
      alt 1: s1
      // alternatives 2..499999
      alt 500000: s500000
      default:
        let
          s_split =
            switch current.type
              alt INT:
                switch current.int
                  alt 500001: s500001
                  /* alternatives
                     500002..999999 */
                  alt 1000000: s1000000
                  default: s_default
                end
            end
        in
          s_split
        end
    end
end
```

Listing 5.17: Method to split up a large value switch. Left: the large value switch, wrapped in a type switch alternative. Right: the split strategy. The let-defined strategy `s_split` compiles to a separate Java method, reducing the size of individual methods.

Integrating in the Existing Compiler With the introduction of a new Core construct, comes the need for changes in the existing compiler. In particular, we need to address splitting large strategies into smaller, equivalent ones.

The Stratego compiler targets Java code. Java limits the size of methods to 64 kB of instructions; if a method is larger, compilation fails. The compiler may split up large strategies by lifting part of that strategy to a local definition and substituting a call. We explicitly implement that for our DFA IR. The DFA mostly consists of list structures (lists of alternatives). To reduce the size of a strategy, we split this list in half. If the strategy is still too large, we recursively apply this method to both list halves.

Consider a value switch with many alternatives, contained in a type switch. This can be split by splitting the list of value alternatives in half. Move half of the alternatives and any default to a new value switch, and wrap it in a type switch. Define this as a local strategy and call this strategy in the default of the original switch. The local strategy compiles to a separate Java method, which the original method calls. This method splits up a large strategy into to strategies of roughly half the size. Listing 5.17 shows an example of this method.

The compiler can split up type switches with a similar technique; split the list of alternatives in half, and move one half (including any default) to a new type switch. Define this as a local strategy and call it in the default of the original type switch.

Summary We developed a match compilation algorithm based on Maranget (2008), which generates an efficient, type-checking DFA IR. In the next chapter, we move on to transforming this DFA to the compiler's target language.

Chapter 6

Encoding a DFA in an Imperative Language

In the previous chapters, we recognised program structures to translate to case matches, and developed an algorithm to compile these case matches to deterministic finite automata (DFAs). The last stage of our pattern match compilation scheme is generating efficient code for the DFA. For our case, where we modify the Stratego compiler, we need to extend the existing code generation rules to accept the new DFA abstract syntax trees (ASTs). This code generation back-end targets Java code, which is subsequently compiled a Java compiler, for example `javac` (Oracle 2020) or ECJ (Eclipse Foundation 2021).

To give some context for the translations described in this chapter, we first show how the compiler translates some Stratego Core constructs that are not directly related to our work. We take the liberty to slightly simplify the generated code and abstract over details irrelevant to this thesis.

Strategy Definition The Stratego Java framework exposes terms as interfaces, and makes use of labelled blocks to achieve backtracking and other control flow. A strategy definition translates to a class with a single `invoke` function. Any parameters to the strategy would be represented in the signature of this function. Listing 6.1 shows an example of the translation of the strategy `s(|)`, which accepts no arguments. A call to a strategy returns the current term after evaluating the strategy body. A block labelled Fail redirects control in case the strategy fails. After generating all Java code, the compiler post-processes it. At this stage, it performs some optimisations. It also renames the labelled blocks in such a way, that a labelled **`break`** always breaks to the nearest surrounding block.

Backtracking To demonstrate how the compiler generates breaks and labelled blocks to perform backtracking, we display the code generated for a guarded left choice (GLC); `s1 < s2 + s3`. Listing 6.2 shows how a break out of a `Success` block can skip certain statements. The `Fail` block controls failure of the guard strategy; if it fails, the GLC should backtrack and evaluate the right branch. However, the compiler has to deal with other possible failures in the context of the GLC; what if `s2` or `s3` fails (thus: breaks out of the nearest `Fail` block)? Failure of `s3` will simply be handled by the strategy surrounding the GLC.

Failure of `s2` requires more attention; because `s2` in inside the `Fail` block, it will break that block, not leading to failure of the whole GLC, contrary to what the semantics describe. Therefore, the GLC contains an additional `OuterFail` block. During post-processing, this block ensures that any failure inside `stm_s2;` correctly breaks out of the `Fail` block surrounding the GLC — the same behaviour as for a failure in `stm_s3;`.

With this knowledge, we can implement code generation for the DFA constructs. We suggest three different implementations, and evaluate them in Chapter 7. In particular, we

```java
public static class $S extends Strategy {
  public static $S instance = new $S();

  @Override
  public IStrategoTerm invoke(Context context, IStrategoTerm term) {
    // This term factory build terms. We will see it in action later.
    ITermFactory termFactory = context.getFactory();

    Fail: { // If the body fails, it will break out of this block.
      // Java statements, corresponding to definition body.
      stm_body1;
      stm_body2;
      // ... and so on

      return term; // Return the current term when the body has been executed.
    };
    return null; // Return null if the body failed.
  }
}
```

Listing 6.1: Java translation of a strategy definition.

```java
IStrategoTerm x_termcopy = term; // Copy current term to restore later.

Success: { // Block to break out of when strategy in left branch succeeds.
  Fail: { // Block to break out of when guard strategy fails.
    stm_s1; // Guard
    OuterFail: { // Redirects failure to outer Fail block.
      stm_s2; // Left branch.

      // Left branch succeeds; redirect control flow to skip right branch.
      break Success;
    }
  }
  // Restore current term when guard has failed, undoing any changes by the guard
  ↪  strategy before executing the right branch.
  term = x_termcopy;

  stm_s3; // Right branch.
}
```

Listing 6.2: Java translation of a guarded left choice. `stm_s1;` is the translation of the strategy `s1`, ditto for `s2` and `s3`.

```
switch current.type
  alt APPL:
    switch current.con end // Some switch on constructor applications
  alt INT:
    switch current.int end // Some switch on integers
  alt STRING:
    switch current.str end // Some switch on strings
  default:
    s // A default strategy
end
```

Listing 6.3: An example type switch, building block of the DFA. This is translated to the Java code in Listing 6.4. For syntactic correctness, we include empty value switches in the switch arms. In real applications, these are never empty.

```java
switch (term.getType()) {
  case APPL: {
    // Translation of a value switch on constructor applications
    break;
  }
  case INT: {
    // Translation of a value switch on integers
    break;
  }
  case STRING: {
    // Translation of a value switch switch on strings
    break;
  }
  default: {
    // Translation of a default strategy
    break;
  }
}
```

Listing 6.4: Java translation of the type switch in Listing 6.3.

address how to implement testing alternatives and choosing the right switch arm.

6.1 Type Switch

This switch branches between different term types, which are implemented as a Java `enum`. That means we can efficiently discriminate them using switch cases. Each type alternative of the type switch translates to a switch case in Java. The body of each switch case is the translation of the corresponding value switch. The default case in Java is the translation of the default case from the type switch, or the translation of `fail` if none is present. Each Java switch case ends with a `break`. For example, the type switch in Listing 6.3 translates to the Java implementation in Listing 6.4.

```
switch current.type
  alt APPL:
    switch current.con
      alt Foo/2: !1
      alt Bar/3: !2
      alt Baz/3: !3
    end
end
```

Listing 6.5: Example of a value switch with three alternatives. The value switch only appears inside a type switch alternative, which is listed for completeness.

```
IStrategoAppl appl = (IStrategoAppl) term;

if (appl.getConstructor() == termFactory.makeConstructor("Foo", 2)) {
  term = termFactory.makeInt(1);
} else if (appl.getConstructor() == termFactory.makeConstructor("Bar", 3)) {
  term = termFactory.makeInt(2);
} else if (appl.getConstructor() == termFactory.makeConstructor("Baz", 3)) {
  term = termFactory.makeInt(3);
} else {
  break Fail;
}
```

Listing 6.6: Implementation of a value switch using **`if`**-statements. This is the result of generating code for the value switch in Listing 6.5.

6.2 Value Switch: If-Else

The first implementation we introduce uses if statements to discriminate the different value switch alternatives. To translate a list of alternatives using this method, we translate each alternative to an **`if`**/**`else if`** condition. The corresponding branch contains the code generated from the alternative's right-hand side (RHS), or implements guarded alternatives using the same code structure as that GLC (Listing 6.2). The final **`else`** branch houses the translation of the switch's default strategy, or the translation of **`fail`** if none is present. As an example, we consider the value switch in Listing 6.5. Using a translation to if-statements, it compiles to Listing 6.6.

Intuition The motivation for this implementation is the simplicity of the equality checks; constructor equality can be determined using simple and fast pointer comparison. String equality is the only type where a method call (to `String.equals(other)`) is required; values of all other Annotated Term (ATerm) types can also be compared using fast `==` comparison of primitive values. In theory, all these checks should be very efficient.

However, although determining if an arbitrary alternative matches is efficient, finding a matching alternative in a list is not; in the worst case, we have to unsuccessfully check equality for all alternatives before the very last check succeeds, yielding a subpattern match. This property motivates researching the implementation.

```java
IStrategoAppl appl = (IStrategoAppl) term;
IStrategoConstructor cons = appl.getConstructor();

switch(cons.getArity()) {
  case 2: { // Constructors with arity 2
    switch (cons.getName()) {
      case "Foo": {
        term = termFactory.makeInt(1);
        break;
      }

      default: break Fail;
    }
    break;
  }

  case 3: { // Constructors with arity 3
    switch (cons.getName()) {
      case "Bar": {
        term = termFactory.makeInt(2);
        break;
      }
      case "Baz": {
        term = termFactory.makeInt(3);
        break;
      }

      default: break Fail;
    }
    break;
  }

  default: break Fail;
}
```

Listing 6.7: Implementation of a value switch using nested `switch`-statements on arity and name. This is the result of generating code for the value switch in Listing 6.5.

6.3 Value Switch: Arity & Name Switches

This implementation employs Java switch cases, which may compile more efficiently than equivalent if-else statements. However, switches are limited to certain types; they cannot be used for arbitrary objects, like constructors. They can be used to determine primitive properties of constructors instead. By checking both its name and arity, we can identify any constructor. We can use nested switches; use a switch to discriminate on arity, and another one to discriminate on name. Generating Java code for the value switch in Listing 6.5 using this method yields the code in Listing 6.7.

Intuition This code generation framework for value switches stems from researching the shortcomings of the previous method, which uses if-statements. Instead, by using nested switches, the code may immediately jump to the right alternative. However, this implementation performs two checks; both on arity and name. Generating multiple checks might have a negative influence on performance and result in more than necessary code. In an attempt to mitigate these drawbacks, we research a third code generation target.

6.4 Value Switch: Constructor Hash Switches

This implementation uses constructors hash codes to determine equality. By switching on the hash code of the constructor, it selects the correct case. For this to work correctly, we need to take some things into consideration.

In order to generate the constant for the switch case, the compiler needs access to the same hash code function as the runtime. We extend the standard library with a strategy that accepts a term and returns the hash code of its constructor.

The contract of the Java `hashCode()` method does not require it to be stable, i.e., return the same value during different runs of the same application. However, we do need this property, because the generated code relies on comparison of hash codes that are generated during different runs of the Java Virtual Machine (JVM), namely Stratego compilation and Java execution. The existing Java hash code implementation for terms relies on `String.hashCode()`, which indeed does not guarantee stable results between different Java versions or implementations. Therefore, we use our own hash code implementation for the constructor name, and use that in the hash code calculation for constructors.

Another challenge stems from the nature of hash codes; they map constructors to integers, possibly introducing hash collisions. A collision occurs when multiple constructors have the same hash code. In our implementation, two kinds of collisions might occur, which we need to check for. In the first case, the constructor from the scrutinee has the same hash code as the constructor from the switch alternative, but they are not the same constructor. To prevent this, we explicitly check equality in a hash code switch case, and fall back to the default if the check fails. A collision might also occur between switch alternative constructors. If multiple have the same hash code, we group them under the same switch case and distinguish them using an extra check.

As an example, we apply this code generation technique to Listing 6.5 again. The result is in Listing 6.8. For the purpose of this example, we assume that constructors |Foo/2| and |Bax/3| have the same hash code. Note that this is not true in practice. In fact, we have not been able to artificially cause hash code collisions in our experiments.

Intuition The motivation for this representation of value switches is the need of a single, fast switch to determine constructor equality. Because of the nature of hashing, which might lead to collisions, an additional check is necessary. If implemented properly, hashing should rarely lead to collisions. Therefore, we can expect this additional check to succeed more often than not, preventing the long chains of failing `else if` conditions that we saw in the first implementation.

Summary In this chapter, we developed three different Java implementations for the DFA intermediate representation (IR), based on knowledge of the use of pattern matching in programs, and properties of Java constructs. In the next chapter, we evaluate the performance of our complete compiler, and compare the three implementations.

```
IStrategoAppl appl = (IStrategoAppl) term;
IStrategoConstructor cons = appl.getConstructor();

switch (cons.hashCode()) {
  case 1897: { // Hash code for (Foo, 2) and (Bax, 3)
    if(cons == termFactory.makeConstructor("Foo", 2)) {
      term = termFactory.makeInt(1);
    } else if(cons == termFactory.makeConstructor("Bax", 3)) {
      term = termFactory.makeInt(3);
    } else {
      break Fail;
    }
    break;
  }

  case 1954: { // Hash code for (Bar, 3)
    if(cons == termFactory.makeConstructor("Bar", 3)) {
      term = termFactory.makeInt(2);
    } else {
      break Fail;
    }
    break;
  }

  default: break Fail;
}
```

Listing 6.8: Implementation of a value switch using `switch`-statements on constructor hash code. This is the result of generating code for the value switch in Listing 6.5. Hash codes in this example are imaginary for demonstration purposes.

Chapter 7

Evaluation

In this chapter we evaluate the impact of the work from Chapters 4 to 6. For this, we set up tests and a benchmarking environment. In this chapter, we discuss the benchmarking approach and results.

7.1 Correctness

To establish the correctness of our compiler, we extend the existing Stratego test suite with several tests to directly test the behaviour of the case match. Furthermore, we verify that the original tests cover situations that are affected by our pattern match compiler. We run this test suite on the compiler with and without our changes and compare the results. Throughout the development of the compiler, this approach helped in finding and fixing several bugs. Both compilers have the same results for each test (i.e., some tests failed both for the original and modified compiler).

7.2 Benchmarking Environment

In order to reproduce our benchmarks, we describe the environment in which they run. We use a server machine, which is not heavily used otherwise, to produce stable, reproducible results. The machine has 2 AMD EPYC 7502 processors (2.5GHz, 32 cores each) and 256 GB (16×16 GB) of DDR4-2933 RAM. We use a Docker image, based on the Spoofax2 benchmark image[1], to virtualise, isolate and conveniently reproduce our benchmarking setup for specific commits in the Spoofax and Stratego repository histories.

In the generated container, we run performance benchmarks using Java Microbenchmark Harness (JMH) (see Section 7.2.2). JMH exports raw results, including separate measurements from every single run, in a JSON format. Besides JMH, we use a Java program to measure the sizes of compiled artefacts. We use Python scripts (in Jupyter Lab[2], using Pandas[3] and Seaborn[4]) to combine and process these results and generate visualisations in the form of various plots and tables, as they appear in this chapter and Appendix D.

7.2.1 Subject

We use two different benchmark subjects, which represent specific uses of the Stratego language.

[1]`https://gitlab.ewi.tudelft.nl/spoofax/docker-images/-/tree/096374d/spoofax2-bench`

[2]`https://jupyter.org/`

[3]`https://pandas.pydata.org/`

[4]`https://seaborn.pydata.org/`

Algorithmic rewriting problems from the Rewriting Engines Competition (REC) (Durán et al. 2009). This is a set of Stratego implementations of common 'algorithmic' problems; sorting (bubble sort, merge sort, quick sort), numeric (Fibonacci, factorial, prime sieve of Eratosthenes), and others. These benchmarks represent synthetic, uniform, highly optimisable, pure rewriting problems. We extended the existing programs to produce results for more input sizes.

A Spoofax language implementation of ChocoPy (Padhye, Sen, and Hilfinger 2019). The ChocoPy language — a subset of Python — is developed to teach a compiler construction course at UC Berkeley and is adopted for a similar course at TU Delft, for which this implementation is the grading reference. This represents idiomatic use of Stratego for desugaring and code generation. It also features Stratego code for pretty-printing a ChocoPy abstract syntax tree (AST), which is auto-generated by Spoofax.

7.2.2 Data Collection

We set up an automated benchmark suite in Java using JMH (Oracle 2022). Using Java allows for convenient use of Java Spoofax libraries. We measure elapsed compilation time, elapsed run time of compiled programs, and compiled program size. JMH takes care of standard benchmarking practices such as warming up the Java Virtual Machine (JVM), preparing benchmarks, repeating benchmark runs with different parameters, and aggregating results from multiple runs. It generates results by running 2 forks (i.e., JVM instances) for each benchmark. Each forked instance initiates 5 unmeasured runs to warm up the just-in-time (JIT) compiler and dynamically-loaded Spoofax components. After these runs, the benchmark times stabilise, so we consider the JVM as warmed up. When the JVM is warmed up, JMH performs 5 measured runs. After each run, JMH invokes the garbage collector. All benchmark results in this chapter are a mean of 10 runs (2 forks, 5 runs each) unless stated otherwise. Errors, indicated by bars (e.g., Figure 7.6) or bands (e.g., Figure 7.1), display the corresponding 95% confidence interval (CI).

In addition to timed JMH benchmarks, we perform non-time measurements. This may be any other number that we can derive from a Stratego program or its compilation artefacts. The result of measurements is deterministic, so they are only performed once.

7.3 Results

This section shows a selection of the results of our benchmarks. The full results can be found in Appendix D.

Rewriting Engines Competition The REC problems consist almost entirely of rules with a single transformation label. The main evaluation strategy is the repeated innermost application of this rule to an input. The rules have overlap in their match patterns and have few `where`-clauses.

Figure 7.1 shows a selection of the run-time results. These results show an improvement of the execution time with the optimised compiler, for any of the back-end implementations. The hash switch back-end shows the largest speed-up (up to a factor of 10 w.r.t. execution time without pattern match compilation) for most configurations. Table D.8 shows the run times for different back-ends.

Figure 7.2 shows the execution times for two other programs. These results show a significant speed-up for both the nested-switch and hash-switch back-ends. They do not show a convincing bias towards one of them; the performance of both back-ends are comparable in most configurations.

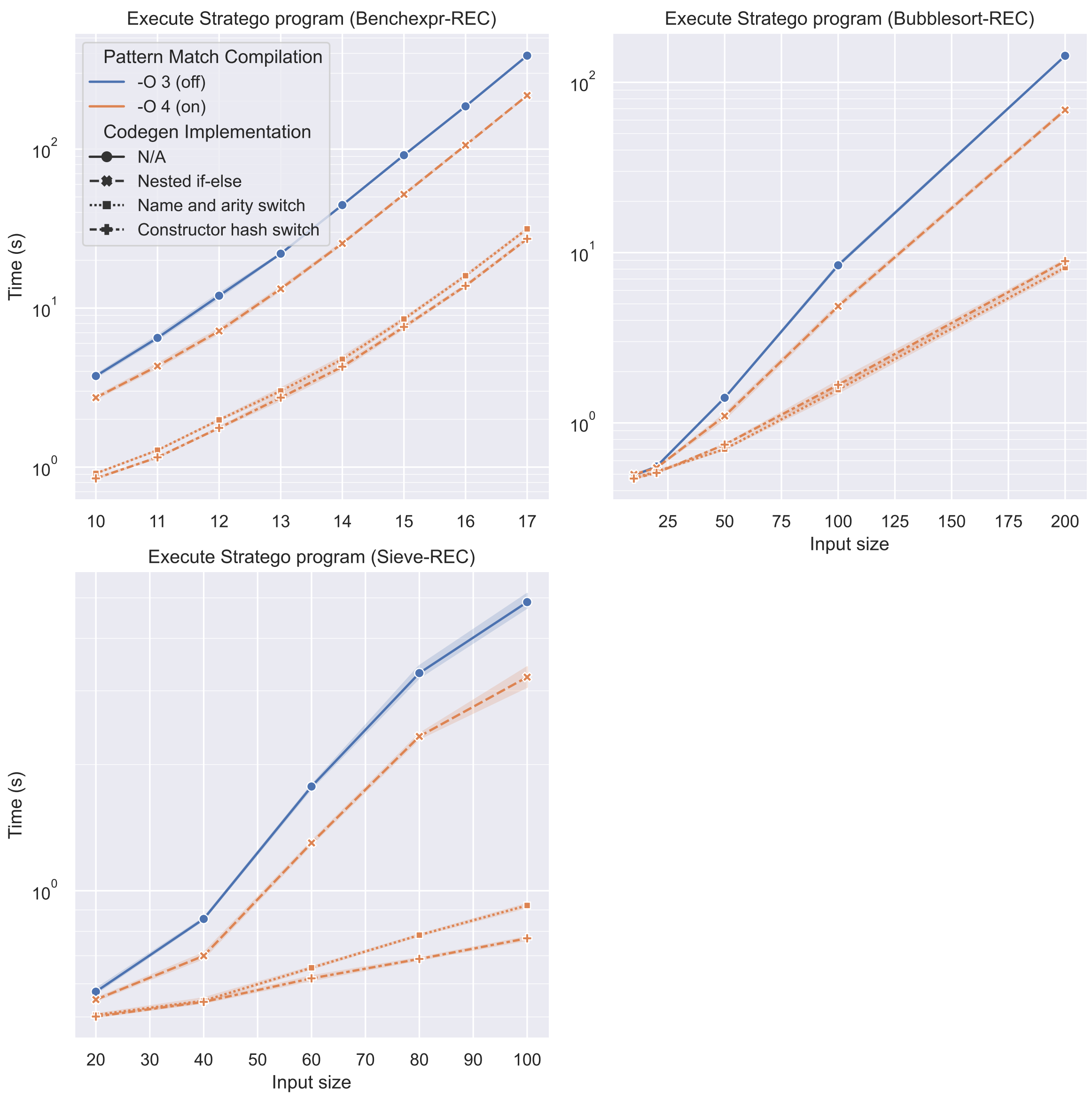


Figure 7.1: Benchexpr, Bubblesort and Sieve benchmark results. Execution time with and without pattern match compilation, for different deterministic finite automaton back-ends.

Besides evaluating the execution, we also investigate possible effects on the compilation of programs. This is two-fold; we benchmark compilation time, and measure the size of compiled programs.

We benchmark compilation time and visualise the distribution as box plots. Figure 7.3 shows the distributions for two sets of problems. The results for the rest of the benchmarks is in Appendix D. Figure 7.4 shows the Stratego compile times for all REC programs. It also lists the number of rewrite rules in the program, and the ratio of compile times with and without pattern match compilation enabled.

In addition to timed benchmarks, we measure the size of compiled Stratego programs, i.e., Java source and compiled class files (Table 7.5). The sizes are the summation of the sizes of all compiled files.

The full set of results obtained from the REC benchmarks is in Appendix D.

ChocoPy Figure 7.6 shows the results from the ChocoPy benchmarks. It shows execution times for code generation and pretty-printing, for compilers with and without pattern match

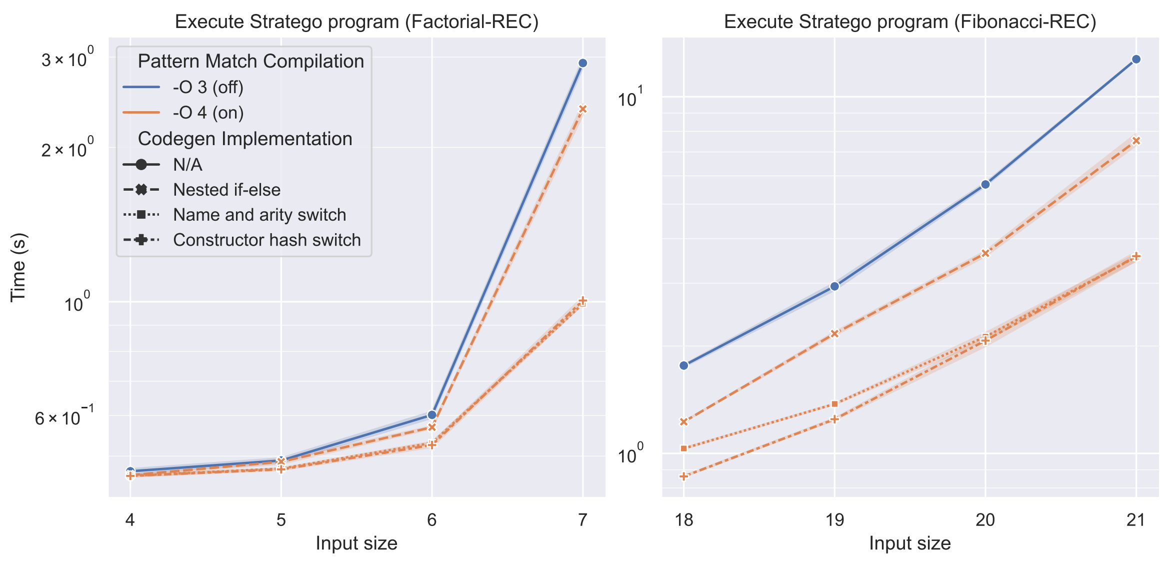
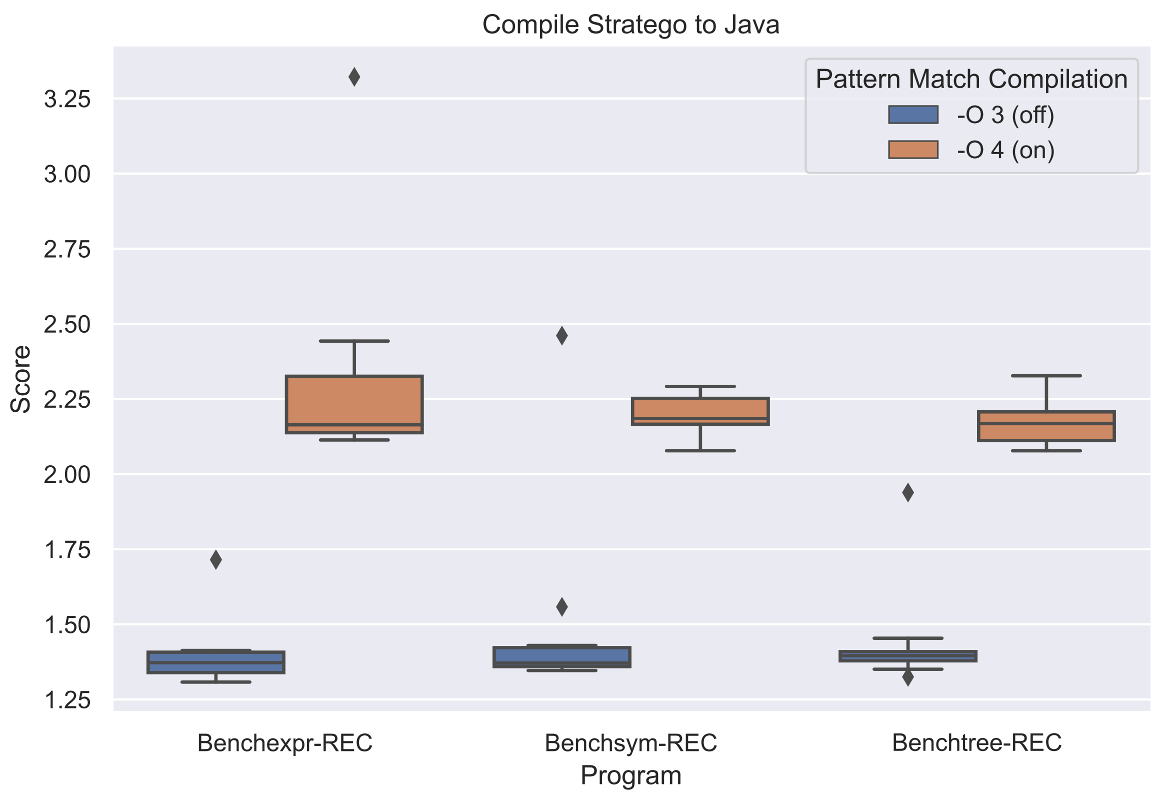


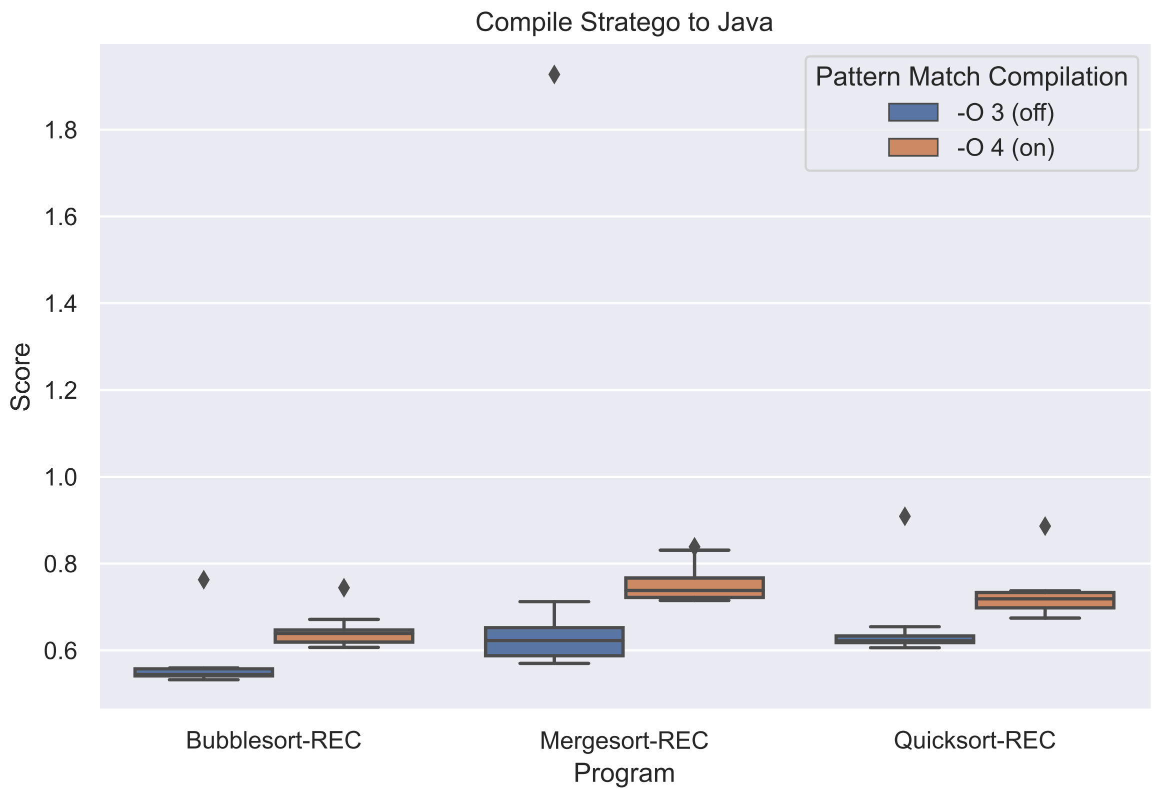
Figure 7.2: Factorial and Fibonacci benchmark results. Execution time with and without pattern match compilation, for different deterministic finite automaton back-ends.

				Size (bytes)	Increase (on vs. off)
		Pattern Match Compilation	-O 3 (off)	-O 4 (on)	
Problem	Number of rules	Compilation artefact			
Benchexpr-REC	155	Java source	237 000	267 000	1.12
		Bytecode	209 000	247 000	1.18
Benchsym-REC	155	Java source	237 000	267 000	1.12
		Bytecode	209 000	247 000	1.18
Benchtree-REC	155	Java source	237 000	267 000	1.12
		Bytecode	209 000	247 000	1.18
Bubblesort-REC	16	Java source	40 400	49 800	1.23
		Bytecode	46 000	54 900	1.19
Calls-REC	9	Java source	30 300	31 700	1.05
		Bytecode	36 400	38 500	1.06
Factorial-REC	6	Java source	21 100	25 400	1.20
		Bytecode	27 300	31 800	1.17
Fibonacci-REC	5	Java source	19 100	23 000	1.20
		Bytecode	24 800	29 200	1.18
GarbageCollection-REC	11	Java source	31 000	37 900	1.22
		Bytecode	31 500	38 200	1.21
Hanoi-REC	31	Java source	60 100	72 300	1.20
		Bytecode	67 200	79 400	1.18
Mergesort-REC	23	Java source	52 600	71 900	1.37
		Bytecode	57 400	75 000	1.31
Quicksort-REC	23	Java source	54 400	68 500	1.26
		Bytecode	61 700	74 300	1.20
Sieve-REC	39	Java source	93 800	115 000	1.23
		Bytecode	103 000	122 000	1.19

Table 7.5: Stratego and Java compilation sizes for different optimisation levels (REC benchmark suite). The rightmost column shows the ratio of the compiled size with pattern match compilation “on” vs. “off+inlining”.



(a) Distribution of compile times for the `benchexpr`, `benchsym` and `benchtree` benchmarks. All programs have 155 rules.



(b) Distribution of compile times for the `bubblesort`, `mergesort` and `quicksort` benchmarks. The programs have 16–23 rules.

Figure 7.3: Distribution of Stratego compile times for several problems from the REC corpus. The whiskers extend to 1.5 times the interquartile range (size of the box) outside the box borders (first and third quartile). Any values outside that range we consider outliers. Outliers appear as diamonds.

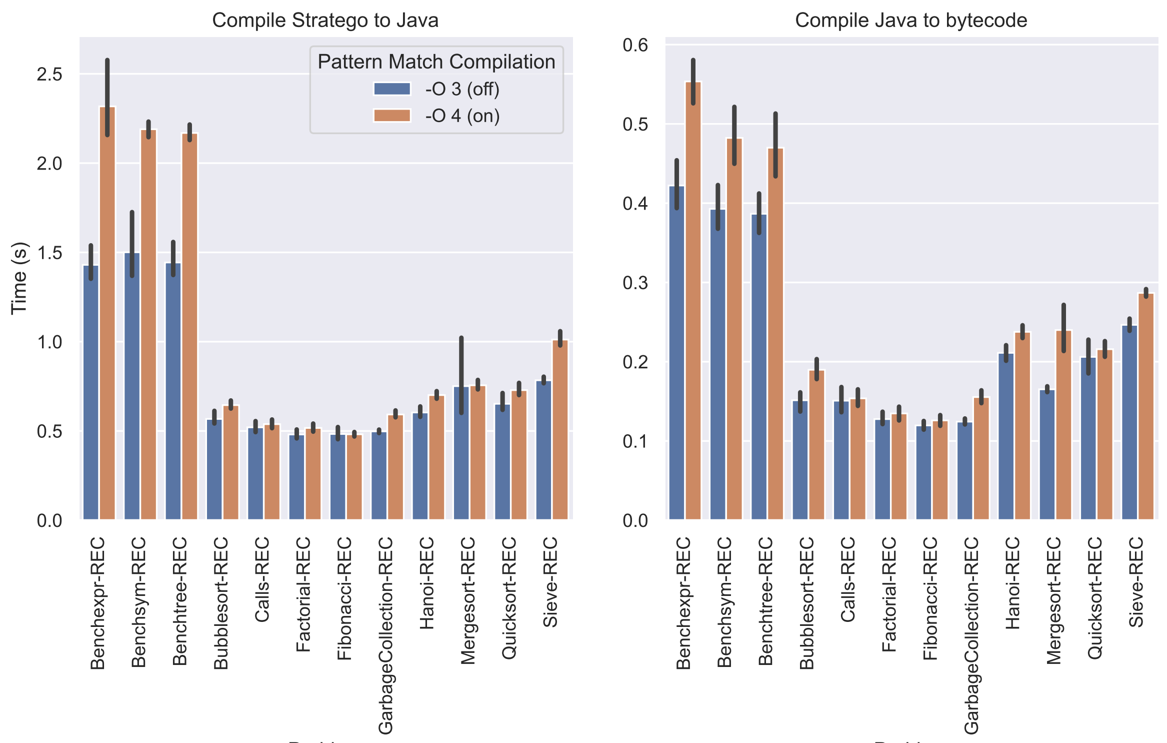


Figure 7.4: Compilation times of all REC problems. Table D.9 shows the same results in a different format.

compilation enabled. These benchmarks use the hash switch backend. Although some benchmarks show outliers in either direction, ChocoPy compile times do not seem to structurally differ between compiler configurations.

7.4 Discussion

We explicitly observe patterns in and draw conclusions from the presented results.

In general, we do not observe a slowdown in programs compiled with our optimisation. Most programs from the REC collection show a significant speed-up. This comes at the cost of increased compilation time and space. If speed benefit outweighs the increased costs, may depend on the available space on the target platform, how often a program is re-compiled, how susceptible a program is to this specific optimisation, and other factors.

When enabling the pattern match compiler, the hash switch implementation is the fastest in most cases. In Figure 7.2, the factorial and Fibonacci problems do not show a clear advantage for the hash switch back-end. We find that these programs have a relatively low number of rules compared to other benchmarks; in this case, the overhead of hashing a constructor may lead to the nested switches being a faster implementation than hash switches (see Sections 6.2 to 6.4 for more on this intuition).

In Figure 7.6, we observe no clear performance difference between ChocoPy compilers that have been built with and without pattern match compilation enabled. Inspecting the Stratego implementation of ChocoPy, we find several rules that only have a single instance, i.e., no overloading and therefore no pattern match alternatives. Other rules have shallow match patterns which do not overlap. Only a fraction of the implementation employs pattern matching on overlapping rules. In the absence of match alternatives, or in the case of non-overlapping alternatives, the performance of our improved compiler only marginally differs from the naive compiler, as no knowledge from partial matches can be reused. This may

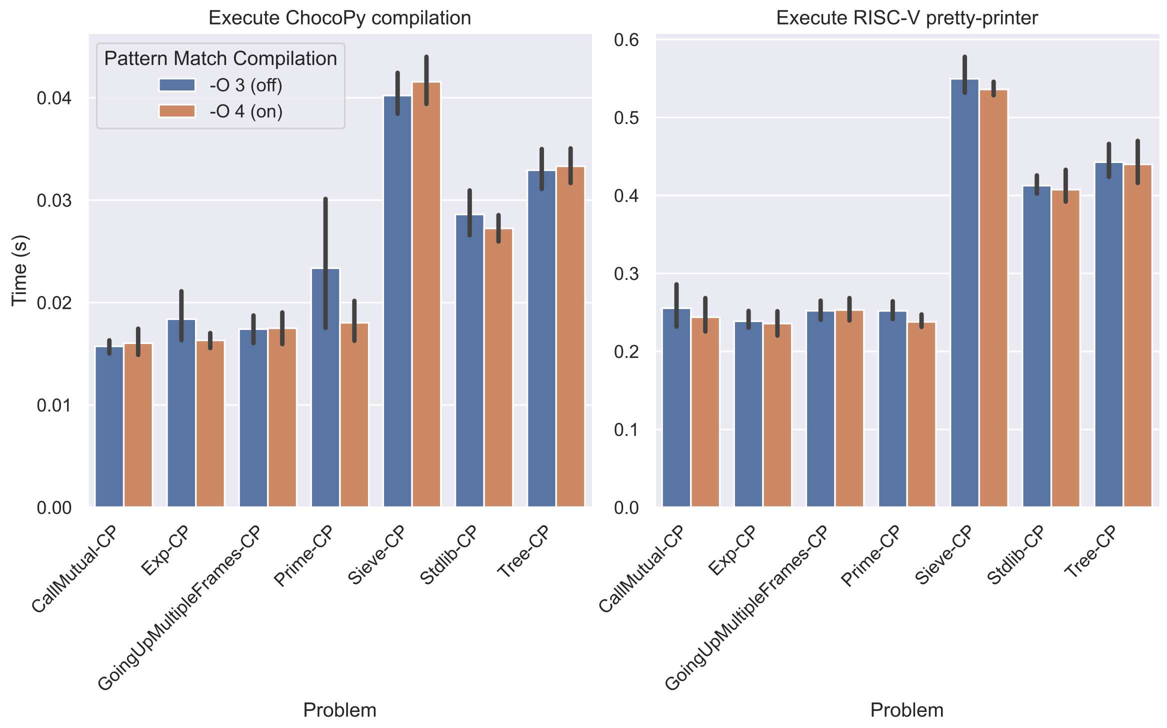


Figure 7.6: Benchmark results for several ChocoPy programs. Running the compiler and pretty-printer.

be a factor in the absence of a perceivable improvement in execution time for the ChocoPy benchmarks.

On the other hand, this does indicate that execution time does not generally deteriorate when applying pattern match compilation to non-optimisable pattern matches. This might indicate that using our compiler results in programs that are at least as fast as naively compiled programs.

7.5 Threats to Validity

We discuss the threats to validity of this benchmarking approach. We address the generalisability of the results (external validity), factors that allow for alternative explanations of the results (internal validity), and the suitability of our approach for our evaluation goals.

7.5.1 External Validity

A sampling bias in the input programs to our benchmarks may form a threat to the generalisability of our benchmark results. As it is not feasible to evaluate our compiler on every single Stratego program, we operate on a subset of programs, which may not be representative. We used two corpuses, with different origins, structures, and code styles. With the use of two entirely different suites, we aim to cover a range of programs; the REC programs are synthetic but suitable for showing the full potential of optimisations, while the ChocoPy specification is a more idiomatic use of Stratego, closer to real-world applications.

7.5.2 Internal Validity

A factor that allows to explain our results alternatively is miscompilation, where the program compiles differently than expected. We experienced this during the development of our compiler; we perceived unexpected results from the ChocoPy benchmarks. Due to misconfiguration of the ChocoPy language project, it did not use our optimised compiler, although we assumed it to. We found out this was the case by inspecting the compiled language specification, decompiling the class files and looking for code that was generated by our optimisation.

Another factor could be the inclusion of other aspects of the compilation pipeline in our results than intended. We choose to measure compilation time separately, reducing any overhead where possible. For example, we initialise the incremental build system (Smits, Konat, and Visser 2020) in an unmeasured setup phase, *before* the timed benchmark run. However, we do measure the elapsed time of a full compilation task in PIE, including some aspects that might induce overhead; reading the program from disk, parsing it, and compiling imported modules, for instance.

7.5.3 Construct Validity

This section evaluates the relevance of our performance metrics for our evaluation goals. We measure performance in several ways. Our main goal, to speed up pattern matching, is reflected by the execution time benchmarks. Other results — compilation time and space — quantify possible side effects of our solution. By enforcing warm-up iterations, we control variance in JIT compilation. Aggregating results from 10 runs helps with eliminating background noise by stabilising results.

Chapter 8

Related Work

8.1 First-Class Pattern Matching

We are not aware of any research on first-class pattern matching, outside of publications on Stratego. There is work on first-class *patterns*, which is a fundamentally different concept (Tullsen 2000; Jay and Kesner 2009; Pope and Yorgey 2019).

8.2 Compiling to a DFA

The choice of deterministic finite automata (DFAs) as the target language for pattern match compilation appears in a plethora of published work. A similar approach to Maranget (2008), using clause matrices, has been documented widely (Cardelli 1984; Baudinet and Macqueen 1985; Schnoebelen 1988; Maranget 1994; Pettersson 1999). A different method, using prefix unification, is proposed by Gräf (1991). Other work proposes methods based on partial evaluation (Jørgensen 1991; Sestoft 1996).

DFAs support the implementation of pattern matching in Standard ML of New Jersey (SML of NJ) (Appel and MacQueen 1987), the New Jersey machine code toolkit (Ramsey and Fernandez 1995) and the Algebraic Specification Formalism + Syntax Definition Formalism (Asf+Sdf) (van den Brand, Heering, et al. 2002), amongst others.

DFAs can also be used as a pattern match implementation for languages with lazy semantics. In this case, the evaluation order of sub-patterns must follow lazy semantics. Maranget (1992) applies a similar method to the one in this work to lazy Meta Language (ML). Peyton Jones (1987) uses a DFA-like target language with lambda abstractions and choice operators.

Compiling to a DAG In an attempt to reduce code size, one could consider compiling to a directed acyclic graph (DAG) instead of a decision tree. Pettersson (1999) describes a method to deduplicate subpattern match tests as well as right-hand sides (RHSs), creating a DAG, after the DFA has been generated. They note that the DAG might also be created without first generating the DFA. Sekar, Ramesh, and Ramakrishnan (1995) and Nedjah, Walter, and Eldridge (1997) propose a similar solution to generate a DAG directly.

8.3 Compiling to a Backtracking Automaton

Augustsson was the first to introduce a pattern match compilation scheme that targets a backtracking automaton, but it had some correctness issues (Augustsson 1984; Augustsson 1985). Maranget (1994) shows a method to generate a backtracking automaton from a clause matrix, and additionally proves it is semantically correct for lazy languages. Le Fessant and Maranget (2001) further optimised this approach.

8.4 Heuristics

Baudinet and Macqueen (1985) introduces an ordered combination of heuristics and claims it produces optimal decision trees in almost all cases. We use these heuristics in this work (Section 5.4). Scott and Ramsey (1999) collect and formalise heuristics from other work and present a comparison of their expected performance. They take heuristics published in earlier work, like Baudinet and Macqueen (1985). Schnoebelen (1988) suggests a single heuristic that prefers a column which is not matching w.r.t. the greatest possible number of RHSs, while Cardelli (1984) proposes the 'Large Branching Factor' heuristic, presuming that will lead to a shallower DFA. Nedjah and de Macedo Mourelle (2001) explicitly mentions the use of heuristics to improve space and time requirements of DAGs. They refer to heuristics as 'position selection strategies', and seem to suggest a strategy that is similar to 'Small branching factor'.

8.5 Code Generation

Our compiler targets Java code. With the goal of realising an efficient implementation of DFA decision nodes, we generate `switch` cases. We delegate the generation of low-level code to the Java compiler, believing that a `switch` will be as efficient as possible. Depending on its cases, it may be compiled in several ways. Bernstein (1985) and Kannan and Proebsting (1994) research the topic of generating good low-level code for `switch` statements.

Chapter 9

Conclusion

This thesis presents an approach for efficient compilation of first-class pattern matches and answers the following research question;

What are the effects of compiling first-class pattern matches using case-based pattern match compilation techniques?

To answer this question, this thesis presents a pattern match compilation scheme for first-class patterns. We develop a behaviour-preserving translation from compositions of first-class matches to match cases. These match cases can be compiled with a DFA-based pattern match compilation technique, which we extend to support ATerms, variables, guards and exploded patterns. We develop an efficient implementation of such a DFA in Java. We implement all facets of the pattern match compiler in the context of Stratego, and study the effects of our work on the performance of the compiler and compiled programs.

Stratego Core features strategy expressions that share semantics with match cases, but are not compiled as efficiently as match cases. By translating these expressions to match cases, we obtain an explicit list of pattern match alternatives, which can be optimised by the compiler. This optimisation is most effective when patterns overlap, i.e., share common outermost constructors. On a handwritten language specification, written in Stratego and featuring only small amounts of shallow patterns, we do not realise a speed-up, but perform as good as the original compiler. On synthetic benchmark programs with deep, overlapping patterns, we obtain an average speed-up of $4\times$, ranging up to $15\times$ in extreme cases. This comes at the cost of 20% longer compile times and 20% larger compiled programs.

Future Work

We suggest some directions for future work.

Deduplicating the DFA A downside of the DFA that we generate is its size; it grows exponentially with the number of constructors in the patterns. In our evaluation, we experimentally show this increase in practice. To mitigate this growth, while keeping the performance benefits of a deterministic finite automaton, the compiler could target a directed acyclic graph (see Section 8.2). This requires keeping track of generated type switches and instead of generating duplicates, substituting a call to the local definition matching that subtree. These changes should be orthogonal to the formal rules defining the behaviour of the pattern match compiler.

Exhaustive Matches on Closed Types Stratego is built on an open-world assumption, meaning that the compiler does not know the set of constructors that constitute a type at compile time (Visser 2001). For this reason, the compiler can not determine exhaustiveness of a set of patterns, which obligates a default case in every switch (see Chapter 5). Recent work on a type system for Stratego (Smits and Visser 2020) could open doors for the introduction of *closed types*, which cannot be extended. For pattern matches on those types, the compiler can deduce exhaustiveness. Exhaustive discriminations on these types can therefore omit defaults, thereby reducing generated code size and complexity, in addition to guaranteeing the match succeeds.

Quantifying the Effects of Heuristics Our pattern match compiler operates on a clause matrix, where sub-patterns are represented by columns. The compiler selects the column to discriminate on heuristically. Depending on the chosen heuristics, the compiler may produce a deep or shallow decision tree, one that has small default cases, or has other desirable characteristics. The goal is to minimise code size or run-time.

For our compiler, we adopted a combination of heuristics suggested by Baudinet and Macqueen (1985). They deem these heuristics to be a good choice for their compiler, which is an implementation of Meta Language (ML) with slightly simplified patterns. We did not measure the effects of heuristics on the performance of our compiler. In typical usage of Stratego, as a component of the Spoofax Language Workbench, most constructors originate from imported signatures that Syntax Definition Formalism 3 (SDF3) automatically generates from syntax definitions (Visser 1999; de Souza Amorim and Visser 2020). This may have an effect on the argument order of constructors, compared to manually-defined constructors. Furthermore, Stratego's versatility allows for divergent code styles. Researching on and experimentation with pattern match compilation heuristics under different parameters may lead to insight in the effects of heuristics on the Stratego pattern match compiler.

Evaluating Back-End Properties Our experiments show varying results for the DFA implementations; it is not straightforward to determine which one is clearly the fastest. One could design benchmarks that build upon the intuitions in Sections 6.2 to 6.4 and try to gain insight in the exact factors that influence the performance of different implementations. Using this information, the compiler could heuristically choose which back-end implementation to use at compile time, depending on the characteristics of the case match.

Match Orders Currently, the case match intermediate representation explicitly states that the alternative matches are attempted in 'sequential' order; earlier cases take priority over later ones. This has been designed to allow extension with other match orders. An example is the specificity ordering, which prefers specific patterns over more general ones (Kennaway 1990). This helps with equational reasoning, which is not always valid for sequential rules.

Inlining Global Definitions Chapter 4 notes how the incremental nature of the Stratego compiler limits inlining of strategy calls (Smits, Konat, and Visser 2020). If the incremental compilation scheme could be adapted to allow inlining in these situations, more possibilities for pattern match compilation might arise, increasing the potential speed-up realised by the optimisation.

Bibliography

Aho, Alfred V. et al. (2007). *Compilers: Principles, Techniques, & Tools*. 2nd ed. Boston: Pearson-/Addison Wesley. 1009 pp. ISBN: 978-0-321-48681-3.

Appel, Andrew W. and David B. MacQueen (1987). "A Standard ML Compiler". In: *Functional Programming Languages and Computer Architecture*. Ed. by Gilles Kahn. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 301–324. ISBN: 978-3-540-47879-9. DOI: 10.1007/3-540-18317-5_17.

Augustsson, Lennart (Aug. 6, 1984). "A Compiler for Lazy ML". In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP '84. New York, NY, USA: Association for Computing Machinery, pp. 218–227. ISBN: 978-0-89791-142-9. DOI: 10.1145/800055.802038. URL: http://doi.org/10.1145/800055.802038 (visited on 10/21/2020).

— (1985). "Compiling Pattern Matching". In: *Functional Programming Languages and Computer Architecture*. Ed. by Jean-Pierre Jouannaud. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 368–381. ISBN: 978-3-540-39677-2. DOI: 10.1007/3-540-15975-4_48.

Backus, J. W. et al. (May 1960). "Report on the Algorithmic Language ALGOL 60". In: *Communications of the ACM* 3.5. Ed. by Peter Naur, pp. 299–314. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/367236.367262. URL: https://dl.acm.org/doi/10.1145/367236.367262 (visited on 02/10/2022).

Bacon, David F., Susan L. Graham, and Oliver J. Sharp (Dec. 1, 1994). "Compiler Transformations for High-Performance Computing". In: *ACM Computing Surveys* 26.4, pp. 345–420. ISSN: 0360-0300. DOI: 10.1145/197405.197406. URL: http://doi.org/10.1145/197405.197406 (visited on 02/13/2022).

Baudinet, Marianne and David Macqueen (Dec. 6, 1985). *Tree Pattern Matching for ML (Extended Abstract)*. Stanford University.

Bernstein, Robert L. (1985). "Producing Good Code for the Case Statement". In: *Software: Practice and Experience* 15.10, pp. 1021–1024. ISSN: 1097-024X. DOI: 10.1002/spe.4380151009. URL: http://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380151009 (visited on 01/17/2022).

Cardelli, Luca (1984). "Compiling a Functional Language". In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming - LFP '84*. The 1984 ACM Symposium. Austin, Texas: ACM Press, pp. 208–217. ISBN: 978-0-89791-142-9. DOI: 10.1145/800055.802037. URL: http://portal.acm.org/citation.cfm?doid=800055.802037 (visited on 09/08/2020).

Cheng, Jiefeng et al. (Apr. 2008). "Fast Graph Pattern Matching". In: *2008 IEEE 24th International Conference on Data Engineering*. 2008 IEEE 24th International Conference on Data Engineering, pp. 913–922. DOI: 10.1109/ICDE.2008.4497500.

De Souza Amorim, L. E. and Eelco Visser (2020). "Multi-Purpose Syntax Definition with SDF3". In: *Software Engineering and Formal Methods* 12310. ISSN: 0302-9743. DOI: 10.1007/978-

3-030-58768-0_1. URL: https://repository.tudelft.nl/islandora/object/uuid%3A9ca9558a-cf08-4f46-8d2b-634d15c98a31 (visited on 01/12/2022).

Durán, Francisco et al. (June 2009). “The Second Rewrite Engines Competition”. In: *Electronic Notes in Theoretical Computer Science* 238.3, pp. 281–291. ISSN: 15710661. DOI: 10.1016/j.entcs.2009.05.025. URL: https://linkinghub.elsevier.com/retrieve/pii/S1571066109001479 (visited on 09/13/2021).

Eclipse Foundation (Mar. 1, 2021). *JDT Core Programmer Guide/ECJ - Eclipsepedia*. URL: https://wiki.eclipse.org/JDT_Core_Programmer_Guide/ECJ (visited on 01/27/2022).

Gräf, Albert (1991). “Left-to-Right Tree Pattern Matching”. In: *Rewriting Techniques and Applications*. Ed. by Ronald V. Book. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 323–334. ISBN: 978-3-540-46383-2. DOI: 10.1007/3-540-53904-2_107.

Hartman, Toine Joep Bart (Feb. 19, 2021). “Classifying Tree Pattern Matching: A Survey on Pattern Match Compilation in Languages”. Literature Study. Delft: University of Technology. 27 pp.

Jay, Barry and Delia Kesner (Mar. 2009). “First-Class Patterns”. In: *Journal of Functional Programming* 19.2, pp. 191–225. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796808007144. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/abs/firstclass-patterns/968C982CA9B727A2C04D216EEF4E6CFC# (visited on 10/21/2021).

Jørgensen, Jesper (1991). “Generating a Pattern Matching Compiler by Partial Evaluation”. In: *Functional Programming, Glasgow 1990*. Ed. by Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst. Red. by C.J. van Rijsbergen. Workshops in Computing. London: Springer, pp. 177–195. ISBN: 978-1-4471-3810-5. DOI: 10.1007/978-1-4471-3810-5_15.

Kannan, Sampath and Todd A. Proebsting (1994). “Correction to ‘Producing Good Code for the Case Statement’”. In: *Software: Practice and Experience* 24.2, pp. 233–233. ISSN: 1097-024X. DOI: 10.1002/spe.4380240206. URL: http://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380240206 (visited on 01/17/2022).

Kats, Lennart C.L. and Eelco Visser (Oct. 17, 2010). “The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs”. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’10. New York, NY, USA: Association for Computing Machinery, pp. 444–463. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869497. URL: http://doi.org/10.1145/1869459.1869497 (visited on 10/28/2021).

Kennaway, Richard (1990). “The Specificity Rule for Lazy Pattern-Matching in Ambiguous Term Rewrite Systems”. In: *ESOP ’90*. Ed. by Neil Jones. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 256–270. ISBN: 978-3-540-47045-8. DOI: 10.1007/3-540-52592-0_68.

Kleene, S. C. (1951). “Representation of Events in Nerve Nets and Finite Automata”. In: p. 41. DOI: 10.1515/9781400882618-002.

Klop, Jan Willem (1990). “Term Rewriting Systems”. In: *Universal Algebra and Applications in Theoretical Computer Science*. Centrum voor Wiskunde en Informatica.

Le Fessant, Fabrice and Luc Maranget (Oct. 1, 2001). “Optimizing Pattern Matching”. In: *ACM SIGPLAN Notices* 36.10, pp. 26–37. ISSN: 0362-1340. DOI: 10.1145/507669.507641. URL: https://doi.org/10.1145/507669.507641 (visited on 09/02/2020).

Maranget, Luc (Jan. 1992). “Compiling Lazy Pattern Matching”. In: *ACM SIGPLAN Lisp Pointers* V.1, pp. 21–31. ISSN: 1045-3563. DOI: 10.1145/141478.141499. URL: https://dl.acm.org/doi/10.1145/141478.141499 (visited on 09/09/2020).

— (Oct. 1994). *Two Techniques for Compiling Lazy Pattern Matching*. Research Report. Rocquencourt: INRIA. URL: https://hal.inria.fr/inria-00074292 (visited on 09/09/2020).

— (Sept. 21, 2008). “Compiling Pattern Matching to Good Decision Trees”. In: *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*. ML ’08. New York, NY, USA: Association for Computing Machinery, pp. 35–46. ISBN: 978-1-60558-062-3. DOI: 10.1145/1411304.1411311. URL: https://doi.org/10.1145/1411304.1411311 (visited on 09/14/2020).

Milner, Robin, Mads Tofte, and Robert Harper (1990). *The Definition of Standard ML*. MIT Press. 119 pp.

Nedjah, Nadia and Luiza de Macedo Mourelle (2001). "Improving Space, Time, and Termination in Rewriting-Based Programming". In: *Engineering of Intelligent Systems*. Ed. by László Monostori, József Váncza, and Moonis Ali. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 880–890. ISBN: 978-3-540-45517-2. DOI: `10.1007/3-540-45517-5_97`.

Nedjah, Nadia, Colin D. Walter, and Stephen E. Eldridge (1997). "Optimal Left-to-Right Pattern-Matching Automata". In: *Algebraic and Logic Programming*. Ed. by Michael Hanus, Jan Heering, and Karl Meinke. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 273–286. ISBN: 978-3-540-69555-4. DOI: `10.1007/BFb0027016`.

O'Donnell, Michael J. (1985). "Equational Logic as a Programming Language". In: *Workshop on Logic of Programs*. Baltimore, MD: Springer, pp. 255–255.

Odersky, Martin et al. (2004). *The Scala Language Specification*. Citeseer.

Oracle (2020). *Javac - Java Programming Language Compiler*. URL: `https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html` (visited on 01/27/2022).

— (2022). *OpenJDK: Jmh*. URL: `https://openjdk.java.net/projects/code-tools/jmh/` (visited on 01/16/2022).

Owens, Scott (2008). "A Sound Semantics for OCamllight". In: *Programming Languages and Systems*. Ed. by Sophia Drossopoulou. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 1–15. ISBN: 978-3-540-78739-6. DOI: `10.1007/978-3-540-78739-6_1`.

Padhye, Rohan, Koushik Sen, and Paul N. Hilfinger (Oct. 25, 2019). "ChocoPy: A Programming Language for Compilers Courses". In: *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. SPLASH-E 2019. New York, NY, USA: Association for Computing Machinery, pp. 41–45. ISBN: 978-1-4503-6989-3. DOI: `10.1145/3358711.3361627`. URL: `http://doi.org/10.1145/3358711.3361627` (visited on 01/10/2022).

Pettersson, Mikael (1999). "Compiling Pattern Matching". In: *Compiling Natural Semantics*. Ed. by Mikael Pettersson. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 85–109. ISBN: 978-3-540-48823-1. DOI: `10.1007/10693148_7`. URL: `https://doi.org/10.1007/10693148_7` (visited on 01/05/2021).

Peyton Jones, Simon L. (Jan. 1, 1987). "The Implementation of Functional Programming Languages". In: URL: `https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/` (visited on 11/12/2020).

— (1996). "Compiling Haskell by Program Transformation: A Report from the Trenches". In: *Programming Languages and Systems — ESOP '96*. Ed. by Hanne Riis Nielson. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 18–44. ISBN: 978-3-540-49942-8. DOI: `10.1007/3-540-61055-3_27`.

Pope, Reiner and Brent Yorgey (Oct. 1, 2019). *First-Class-Patterns*. first-class-patterns: First class patterns and pattern matching, using type families. URL: `//hackage.haskell.org/package/first-class-patterns` (visited on 09/13/2021).

Ramsey, Norman and Mary F. Fernandez (Jan. 16, 1995). "The New Jersey Machine-Code Toolkit". In: *Proceedings of the USENIX 1995 Technical Conference Proceedings*. TCON'95. USA: USENIX Association, p. 24.

Scala (Sept. 30, 2021). *Pattern Matching*. Tour of Scala. URL: `https://docs.scala-lang.org/tour/pattern-matching.html` (visited on 01/12/2022).

Schnoebelen, Ph. (Dec. 1, 1988). "Refined Compiliation of Pattern-Matching for Functional Languages". In: *Science of Computer Programming* 11.2, pp. 133–159. ISSN: 0167-6423. DOI: `10.1016/0167-6423(88)90002-0`. URL: `http://www.sciencedirect.com/science/article/pii/0167642388900020` (visited on 11/10/2020).

Scott, Kevin and Norman Ramsey (Mar. 12, 1999). *When Do Match-compilation Heuristics Matter?* Report. University of Virginia, Department of Computer Science. URL: `https://libraopen.lib.virginia.edu/public_view/d791sg16q` (visited on 09/09/2020).

Sekar, R. C., R. Ramesh, and I. V. Ramakrishnan (Dec. 1, 1995). "Adaptive Pattern Matching". In: *SIAM Journal on Computing* 24.6, pp. 1207–1234. issn: 0097-5397. doi: `10.1137/S0097539793246252`. url: `https://epubs.siam.org/doi/abs/10.1137/S0097539793246252` (visited on 09/09/2020).

Sestoft, Peter (1996). "ML Pattern Match Compilation and Partial Evaluation". In: *Partial Evaluation*. Ed. by Olivier Danvy, Robert Glück, and Peter Thiemann. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 446–464. isbn: 978-3-540-70589-5. doi: `10.1007/3-540-61580-6_22`.

Smits, Jeff, Gabriël D. P. Konat, and Eelco Visser (Feb. 14, 2020). "Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System". In: *The Art, Science, and Engineering of Programming* 4.3, p. 16. issn: 2473-7321. doi: `10.22152/programming-journal.org/2020/4/16`. arXiv: `2002.06183`. url: `http://arxiv.org/abs/2002.06183` (visited on 09/09/2020).

Smits, Jeff and Eelco Visser (Nov. 16, 2020). "Gradually Typing Strategies". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. SPLASH '20: Conference on Systems, Programming, Languages, and Applications, Software for Humanity. Virtual USA: ACM, pp. 1–15. isbn: 978-1-4503-8176-5. doi: `10.1145/3426425.3426928`. url: `https://dl.acm.org/doi/10.1145/3426425.3426928` (visited on 01/26/2022).

Spoofax Team (Dec. 20, 2021). *Spoofax - Spoofax: The Language Designer's Workbench*. url: `https://www.spoofax.dev/` (visited on 01/11/2022).

Tullsen, Mark (2000). "First Class Patterns?" In: *Practical Aspects of Declarative Languages*. Ed. by Enrico Pontelli and Vítor Santos Costa. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 1–15. isbn: 978-3-540-46584-3. doi: `10.1007/3-540-46584-7_1`.

Van den Brand, M. G. J., H. A. de Jong, et al. (2000). "Efficient Annotated Terms". In: *Software: Practice and Experience* 30.3, pp. 259–291. issn: 1097-024X. doi: `10.1002/(SICI)1097-024X(200003)30:3<259::AID-SPE298>3.0.CO;2-Y`. url: `http://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291097-024X%28200003%2930%3A3%3C259%3A%3AAID-SPE298%3E3.0.CO%3B2-Y` (visited on 11/23/2021).

Van den Brand, M. G. J., J. Heering, et al. (July 1, 2002). "Compiling Language Definitions: The ASF+SDF Compiler". In: *ACM Transactions on Programming Languages and Systems* 24.4, pp. 334–368. issn: 0164-0925. doi: `10.1145/567097.567099`. url: `http://doi.org/10.1145/567097.567099` (visited on 10/26/2020).

Visser, Eelco (Dec. 4, 1999). "A Family of Syntax Definition Formalisms". In.

— (2001). "Stratego: A Language for Program Transformation Based on Rewriting Strategies – System Description of Stratego 0.5". In: *Rewriting Techniques and Applications*. Ed. by Aart Middeldorp. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 357–361. isbn: 978-3-540-45127-3. doi: `10.1007/3-540-45127-7_27`.

Visser, Eelco, Zine-el-Abidine Benaissa, and Andrew Tolmach (Sept. 29, 1998). "Building Program Optimizers with Rewriting Strategies". In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ICFP '98. New York, NY, USA: Association for Computing Machinery, pp. 13–26. isbn: 978-1-58113-024-9. doi: `10.1145/289423.289425`. url: `http://doi.org/10.1145/289423.289425` (visited on 10/28/2021).

Vollebregt, Tobi, Lennart C.L. Kats, and Eelco Visser (2012). "Declarative Specification of Template-Based Textual Editors: Twelfth Workshop on Language Descriptions, Tools, and Applications (LDTA'12)". In: *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications (LDTA'12)*. Ed. by A Sloane and S Andova, 8:1–8:7. issn: 978-1-4503-1536-4. doi: `10.1145/2427048.2427056`.

Wirth, Niklaus (1996). "Extended Backus-Naur Form (EBNF)". In: *Iso/Iec* 14977.2996, pp. 2–21.

Acronyms

Asf+Sdf the Algebraic Specification Formalism + Syntax Definition Formalism

ATerm Annotated Term

AST abstract syntax tree

CI confidence interval

DAG directed acyclic graph

DFA deterministic finite automaton

ID identifier

IR intermediate representation

JIT just-in-time

JMH Java Microbenchmark Harness

JVM Java Virtual Machine

GLC guarded left choice

GRS graph rewriting system

LUB least upper bound

ML Meta Language

OCaml Objective Caml

REC Rewriting Engines Competition

RHS right-hand side

SDF3 Syntax Definition Formalism 3

SML of NJ Standard ML of New Jersey

TRS term rewriting system

XML Extensible Markup Language

Appendix A

Stratego Calculator Program

```
module calc
imports libstratego-lib

signature
  sorts Nat                     // natural numbers: 0, 1, 2, ...
  constructors
    O : Nat                     // 0
    S : Nat -> Nat              // 1 + n

    Plus :  Nat * Nat -> Nat // m + n
    Minus : Nat * Nat -> Nat // m - n
    // Ex.: Minus(S(S(O())), S(O()))

rules
  Calc: Plus(O(), n)      -> n              // 0 + n       => n
  Calc: Plus(n, O())      -> n              // n + 0       => n
  Calc: Plus(S(m), n)     -> S(Plus(m, n)) // (1 + m) + n => 1 + (m + n)

  Calc: Minus(n, O())     -> n              // n - 0             => n
  Calc: Minus(S(m), S(n)) -> Minus(m, n)    // (1 + m) - (1 + n) => m - n
  Calc: Minus(O(), S(_))  ->
        <fatal-err(|"Negative result!")>    // 0 - (1 + n) => not a nat

strategies
  calculate = innermost(Calc)
  main = <calculate> Minus(S(S(O())), S(O())) // calculates 2 - 1
```

Listing A.1: A simple calculator, implemented in Stratego using rewrite rules.

Appendix B

Case Match Grammar

```
context-free sorts Strategy MatchOrder MatchCase
context-free syntax
  Strategy.CaseMatch = <match <MatchOrder>
                          <{MatchCase "\n"}*>
                        end>

  MatchOrder.Sequential = <sequential>

  MatchCase.ScopedGuardedMatchCase =
    <case <{ID ", "}*> | <Term> when <Strategy>: <Strategy>>
  MatchCase.ScopedMatchCase =
    <case <{ID ", "}*> | <Term>: <Strategy>>
```

Listing B.1: Syntax Definition Formalism 3 (SDF3) (Vollebregt, Kats, and Visser 2012; de Souza Amorim and Visser 2020) grammar for the case match.

```
strategy = "id" | "fail" | "..." (* previously existing strategies *)
         | "match", white space, matchorder, white space,
             { matchcase, "\n", white space },
           "end";
matchorder = "sequential";
matchcase =
(* ScopedMatchCase *)
    "case", { identifier, ", " }, "|", term, ":", white space, strategy
(* ScopedGuardedMatchCase *)
  | "case", { identifier, ", " }, "|", term, white space, "where", white space,
  ↪  strategy, ":", white space, strategy;
```

Listing B.2: EBNF (Backus et al. 1960; Wirth 1996) grammar for the case match.

Appendix C

Success Analysis

```
// Determines whether a strategy always succeeds (without side effects).
always-succeeds(|[_ | depth]) =
  ?Id()
+ ?Build(_) < Build(
    ?Path(CurP())
  + not(oncebu(fail
      + ?Var(_){"unbound"} // No (potentially) unbound vars in build
      + ?Var(_){"(un)bound"}
      + ?Op("Cons", _) // No list builds
      + ?Explode(_, _) // No exploded term builds
    ))
  )
+ ?Match(_) < Match(?Wld() + ?Var(_){"unbound"})
+ ?Seq(_, _) < Seq(always-succeeds(|depth), always-succeeds(|depth))
+ ?GuardedLChoice(_, _, _) <
     (GuardedLChoice(always-succeeds(|depth), always-succeeds(|depth), id)
    + GuardedLChoice(always-fails(|depth), id, always-succeeds(|depth))
    + GuardedLChoice(id, always-succeeds(|depth), always-succeeds(|depth)))
+ Scope(id, always-succeeds(|depth))

// Determines whether a strategy always fails (without side effects).
always-fails(|[_ | depth]) =
  ?Fail()
+ ?Build(_) < where(oncebu(?Var(_){"unbound"}))
+ ?Seq(_, _) < or(Seq(always-fails(|depth), id), Seq(id, always-fails(|depth)))
+ ?GuardedLChoice(_, _, _) <
     (GuardedLChoice(id, always-fails(|depth), always-fails(|depth))
    + GuardedLChoice(always-succeeds(|depth), always-fails(|depth), id)
    + GuardedLChoice(always-fails(|depth), id, always-fails(|depth)))
+ Scope(id, always-fails(|depth))
```

Listing C.1: Success/failure analysis for Core abstract syntax trees (ASTs). Because the analysis is expensive, we use a depth-bounded implementation.

Appendix D

Evaluation Results

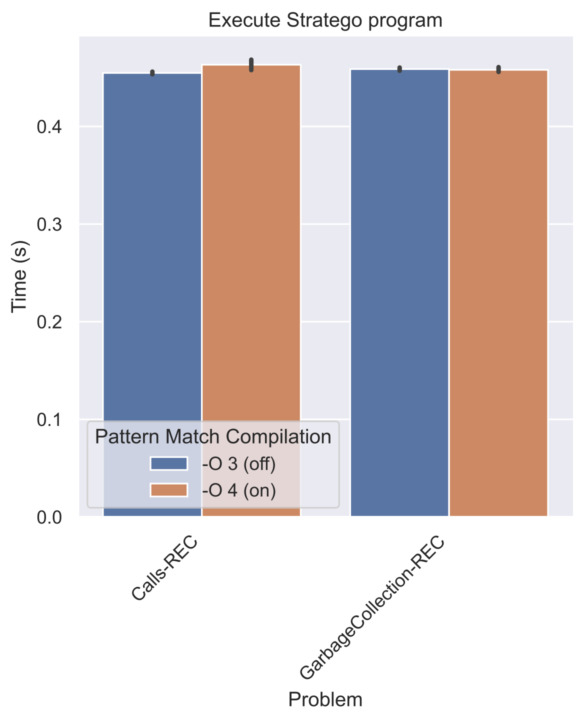


Figure D.1: Run times for all problems with a fixed input from the Rewriting Engines Competition (REC) collection.

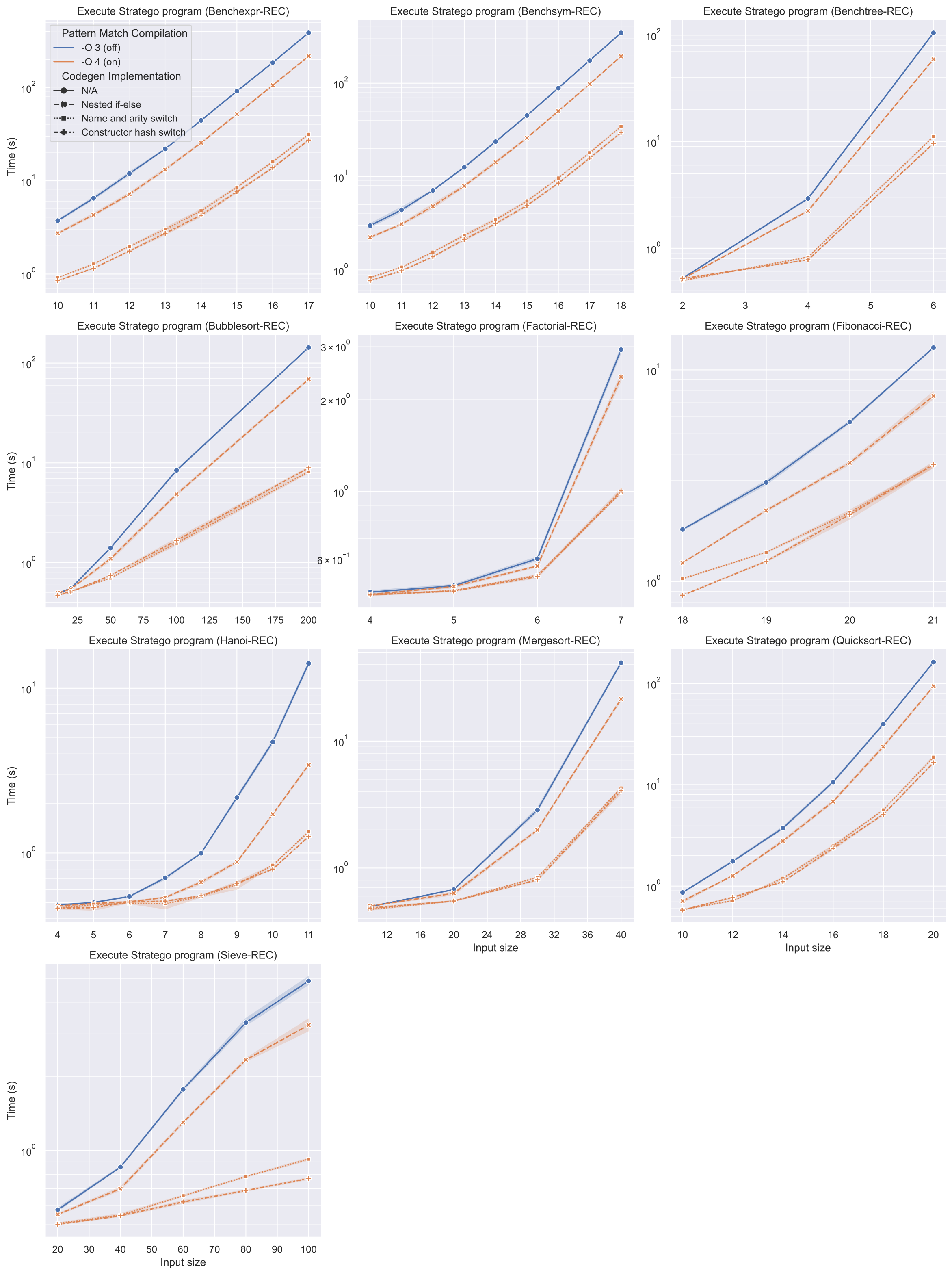
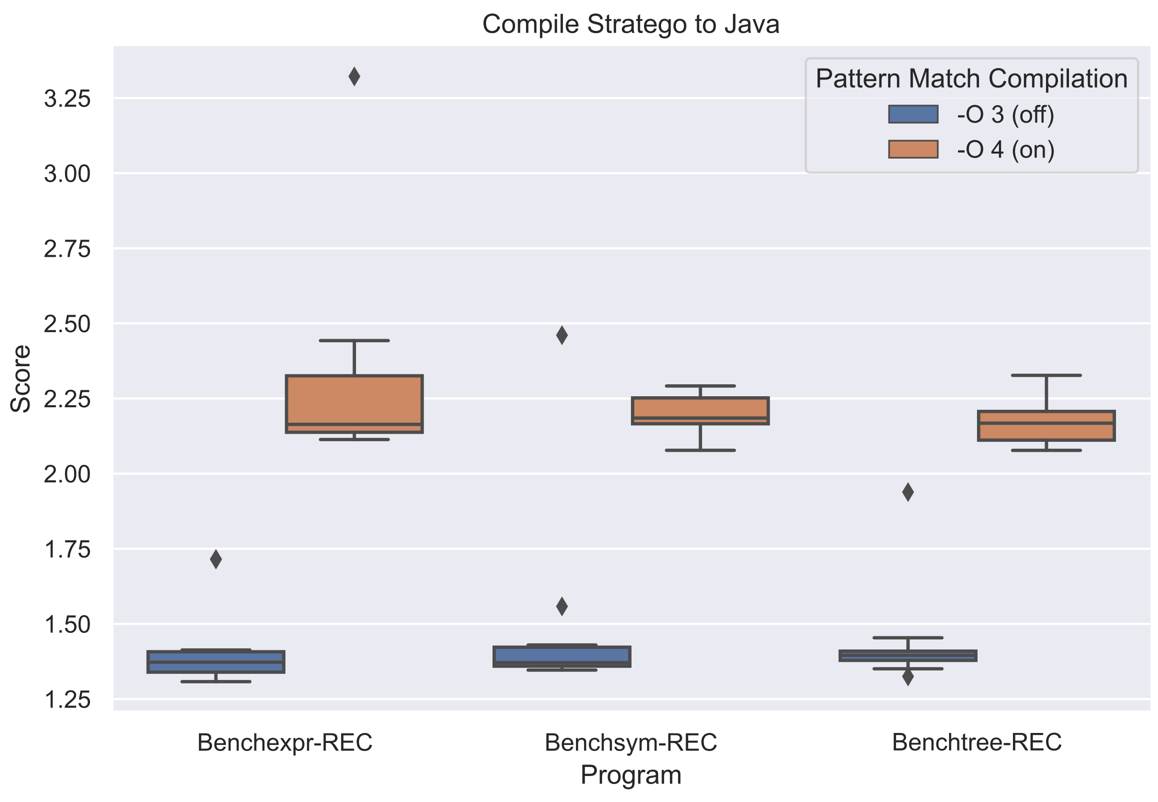


Figure D.2: Run times for all problems with a variable input size from the REC collection.



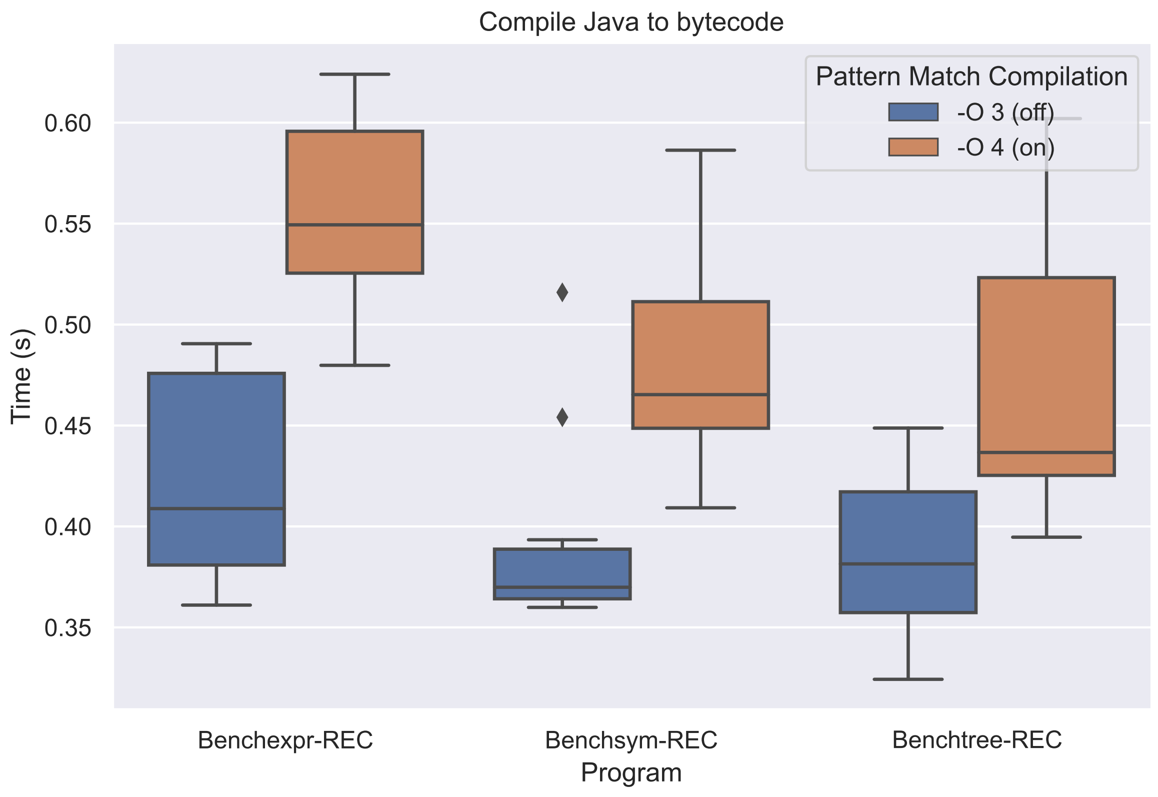


Figure D.3: Stratego & Java compilation times for the Benchexpr, Benchsym and Benchtree problems (REC).

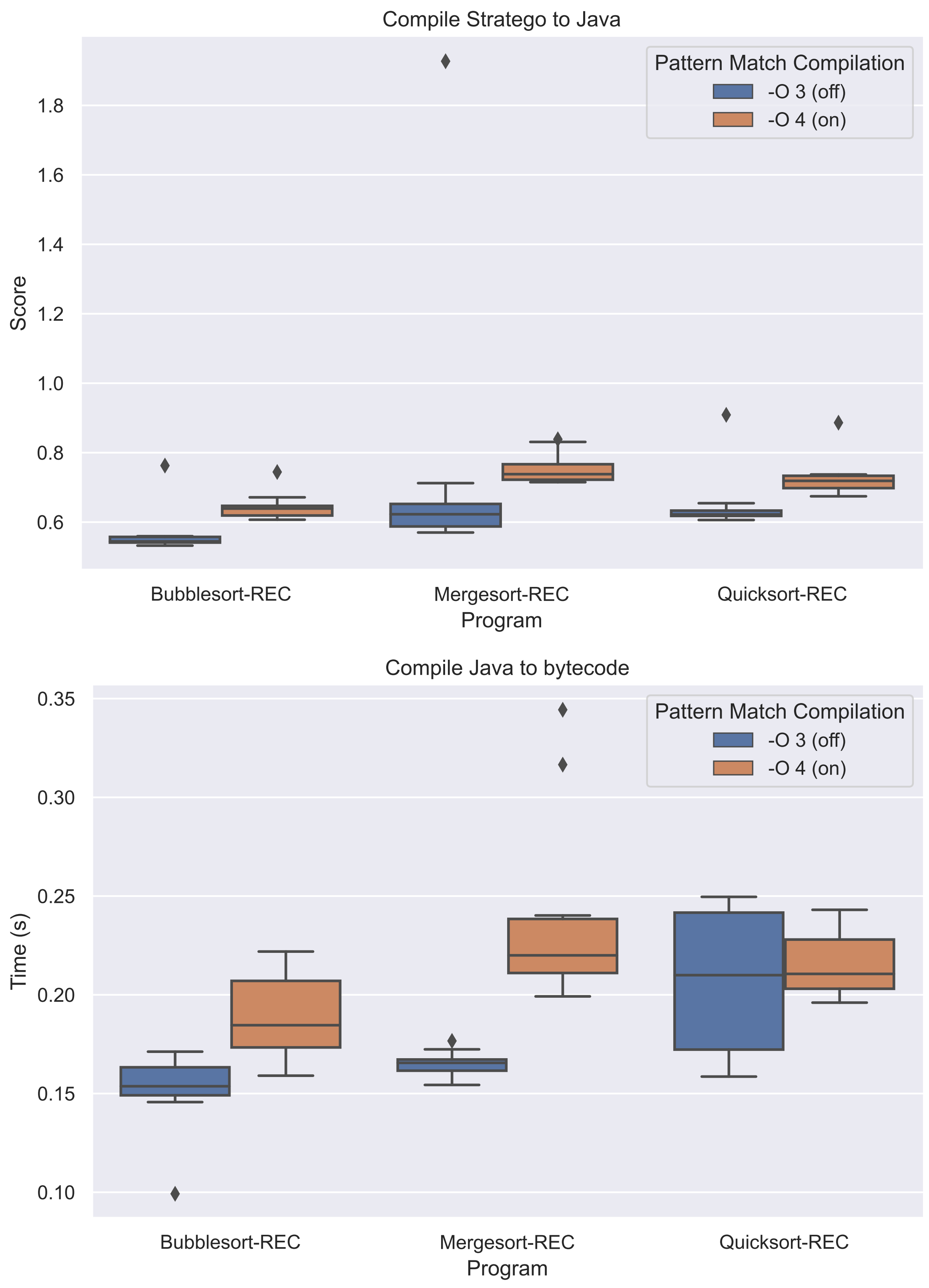
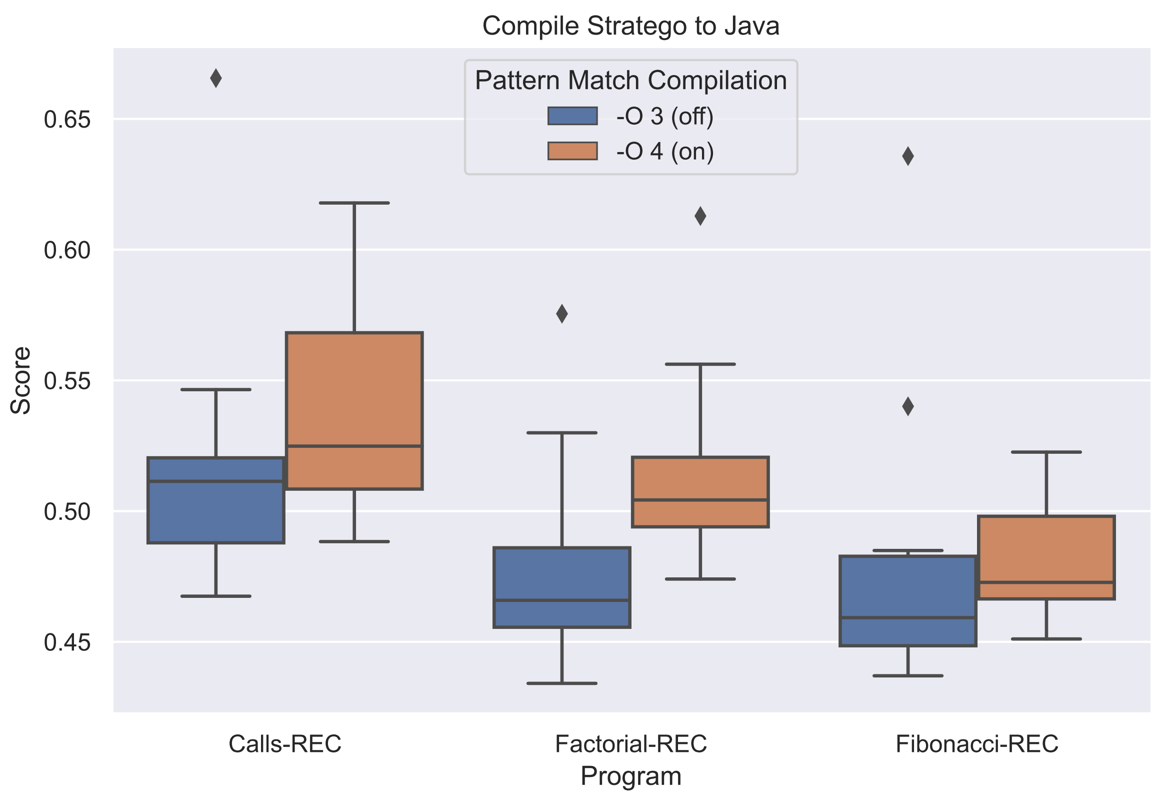


Figure D.4: Stratego & Java compilation times for the Bubblesort, Mergesort and Quicksort problems (REC).



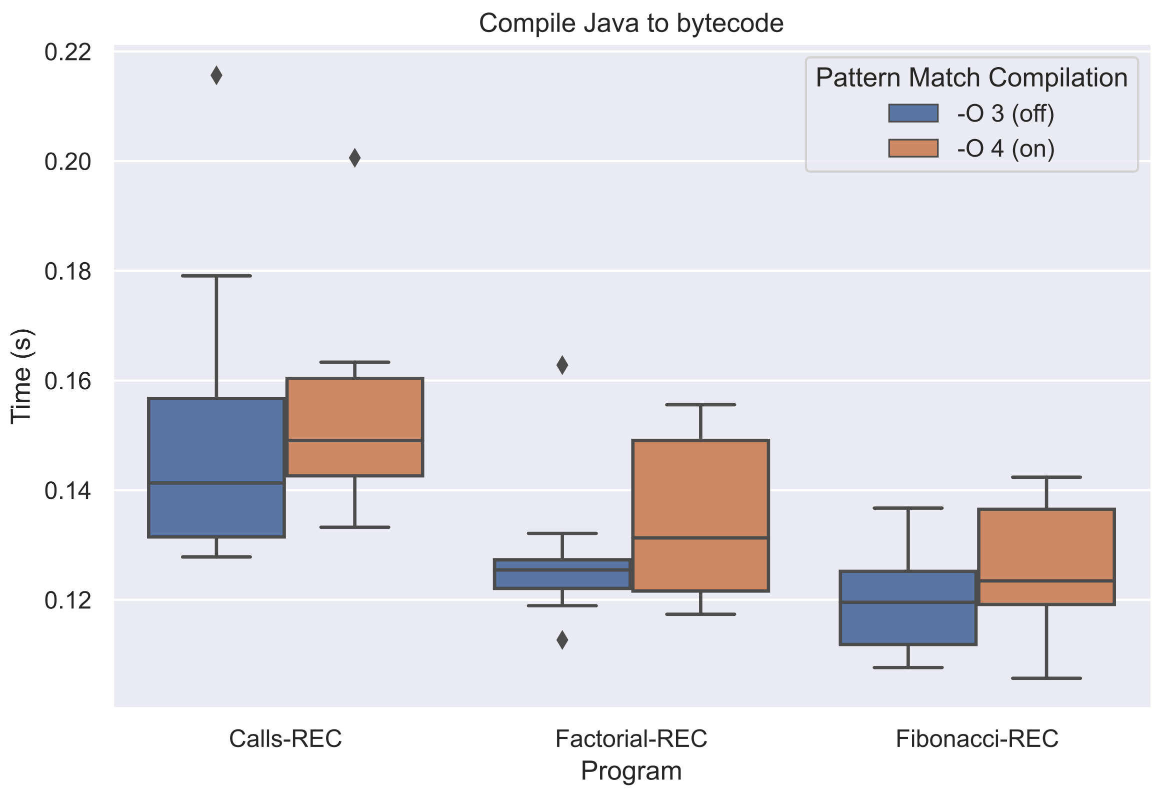
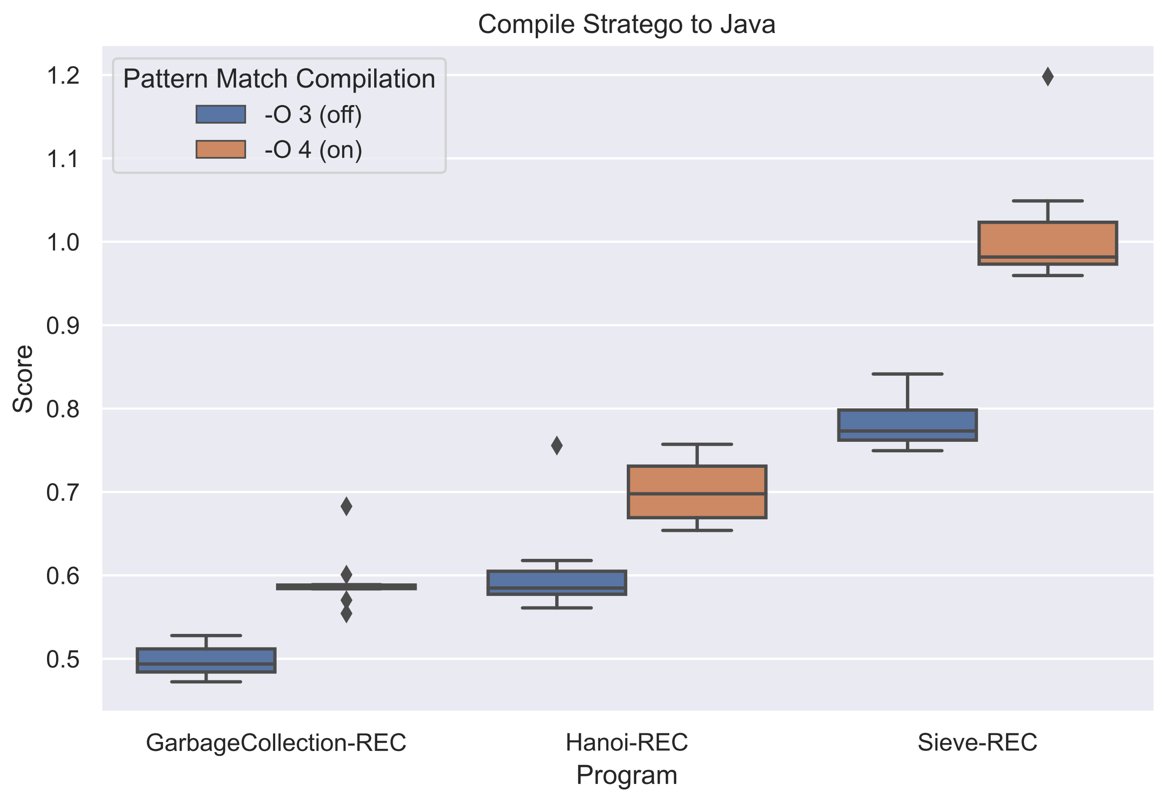


Figure D.5: Stratego & Java compilation times for the Calls, Factorial and Fibonacci problems (REC).



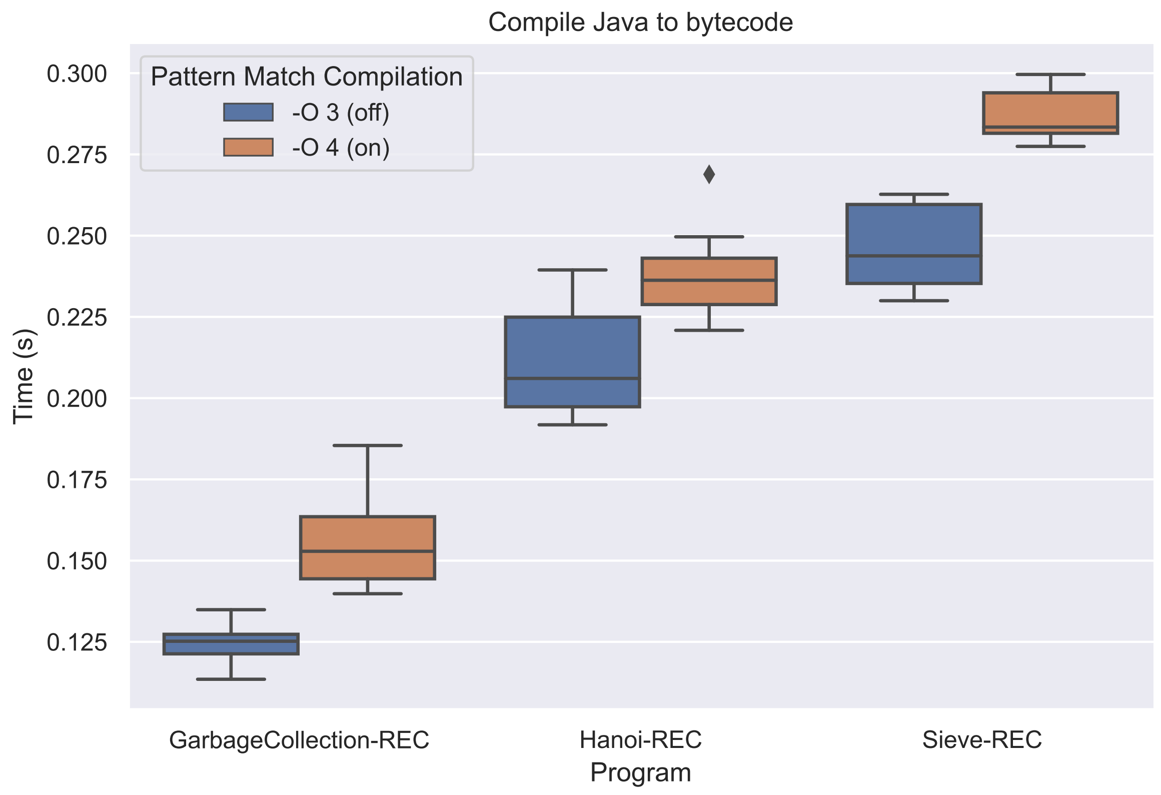


Figure D.6: Stratego & Java compilation times for the GarbageCollection, Hanoi and Sieve problems (REC).

Table D.7: Stratego execution times (REC problems) for different optimisation levels. Runs with pattern match compilation enabled use the hash switch back-end. The rightmost column shows the increase in compilation time with pattern match compilation "on" vs. "off+inlining".

		Pattern Match Compilation	Execution time (s)		Speed-up
			-O 3 (off)	-O 4 (on)	
Problem	Number of rules	Problem Size			
Benchexpr-REC	155	10	3.74	0.849	4.40
		11	6.50	1.15	5.64
		12	12.0	1.76	6.78
		13	22.0	2.74	8.04
		14	44.5	4.28	10.4
		15	91.6	7.61	12.0
		16	186	13.8	13.5
		17	387	27.3	14.2
Benchsym-REC	155	10	2.98	0.766	3.89
		11	4.40	0.977	4.50
		12	7.11	1.38	5.14
		13	12.6	2.12	5.93
		14	23.6	3.13	7.56
		15	45.1	4.89	9.22
		16	88.6	8.50	10.4
		17	175	15.7	11.1
		18	346	29.5	11.7
Benchtree-REC	155	2	0.522	0.519	1.01
		4	2.93	0.778	3.76
		6	105	9.61	10.9
Bubblesort-REC	16	10	0.491	0.471	1.04
		20	0.555	0.509	1.09
		50	1.40	0.746	1.88
		100	8.43	1.67	5.04

Continued on next page

Table D.7: Stratego execution times (REC problems) for different optimisation levels. Runs with pattern match compilation enabled use the hash switch back-end. The rightmost column shows the increase in compilation time with pattern match compilation “on” vs. “off+inlining”.

			Execution time (s)		Speed-up
		Pattern Match Compilation	-O 3 (off)	-O 4 (on)	
Problem	Number of rules	Problem Size			
		200	143	8.91	16.1
Calls-REC	9	<NA>	0.455	0.463	0.982
Factorial-REC	6	4	0.467	0.458	1.02
		5	0.490	0.471	1.04
		6	0.602	0.525	1.15
		7	2.92	1.01	2.90
Fibonacci-REC	5	18	1.76	0.862	2.05
		19	2.94	1.25	2.36
		20	5.68	2.07	2.74
		21	12.7	3.57	3.57
GarbageCollection-REC	11	<NA>	0.459	0.458	1.00
Hanoi-REC	31	4	0.484	0.463	1.04
		5	0.501	0.466	1.07
		6	0.545	0.504	1.08
		7	0.707	0.511	1.38
		8	0.997	0.550	1.81
		9	2.17	0.654	3.32
		10	4.72	0.799	5.91
		11	14.1	1.26	11.3
Mergesort-REC	23	10	0.496	0.484	1.03
		20	0.677	0.550	1.23
		30	2.86	0.807	3.55
		40	41.4	4.07	10.2
		10	0.863	0.580	1.49

Continued on next page

Quicksort-REC	23				

Table D.7: Stratego execution times (REC problems) for different optimisation levels. Runs with pattern match compilation enabled use the hash switch back-end. The rightmost column shows the increase in compilation time with pattern match compilation “on” vs. “off+inlining”.

			Execution time (s)		Speed-up
		Pattern Match Compilation	-O 3 (off)	-O 4 (on)	
Problem	Number of rules	Problem Size			
		12	1.75	0.774	2.27
		14	3.73	1.10	3.38
		16	10.6	2.36	4.50
		18	39.6	5.10	7.77
		20	162	16.5	9.85
		20	0.574	0.501	1.15
		40	0.857	0.543	1.58
Sieve-REC	39	60	1.77	0.618	2.87
		80	3.31	0.688	4.81
		100	4.88	0.770	6.34

Table D.8: Stratego execution times (REC problems) for different codegen back-ends.

			Execution time (s)		
		Codegen Implementation	Constructor hash switch	Name and arity switch	Nested if-else
Problem	Number of rules	Problem Size			
Benchexpr-REC	155	10	0.849	0.913	2.73
		11	1.15	1.28	4.32
		12	1.76	1.98	7.17
		13	2.74	3.01	13.2
		14	4.28	4.76	25.5
		15	7.61	8.54	51.9
		16	13.8	16.0	106
		17	27.3	31.5	217
Benchsym-REC	155	10	0.766	0.826	2.23
		11	0.977	1.07	3.09
		12	1.38	1.55	4.81
		13	2.12	2.34	7.92
		14	3.13	3.45	14.2
		15	4.89	5.44	26.0
		16	8.50	9.64	50.2
		17	15.7	18.0	97.9
		18	29.5	34.4	194
Benchtree-REC	155	2	0.519	0.498	0.515
		4	0.778	0.825	2.24
		6	9.61	11.2	59.3
Bubblesort-REC	16	10	0.471	0.475	0.498
		20	0.509	0.518	0.545
		50	0.746	0.702	1.10
		100	1.67	1.57	4.84
		200	8.91	8.17	68.8

Continued on next page

Table D.8: Stratego execution times (REC problems) for different codegen back-ends.

Problem	Number of rules	Codegen Implementation Problem Size	Constructor hash switch	Name and arity switch	Execution time (s) Nested if-else
Calls-REC	9	<NA>	0.463	0.458	0.461
Factorial-REC	6	4	0.458	0.458	0.459
		5	0.471	0.473	0.488
		6	0.525	0.531	0.569
		7	1.01	0.991	2.38
Fibonacci-REC	5	18	0.862	1.03	1.23
		19	1.25	1.38	2.17
		20	2.07	2.13	3.64
		21	3.57	3.55	7.53
GarbageCollection-REC	11	<NA>	0.458	0.459	0.469
Hanoi-REC	31	4	0.463	0.464	0.477
		5	0.466	0.485	0.494
		6	0.504	0.495	0.506
		7	0.511	0.493	0.537
		8	0.550	0.545	0.665
		9	0.654	0.643	0.882
		10	0.799	0.843	1.72
		11	1.26	1.34	3.42
Mergesort-REC	23	10	0.484	0.473	0.502
		20	0.550	0.550	0.633
		30	0.807	0.844	2.00
		40	4.07	4.27	21.4
Quicksort-REC	23	10	0.580	0.586	0.711
		12	0.774	0.717	1.26
		14	1.10	1.19	2.78

Continued on next page

Table D.8: Stratego execution times (REC problems) for different codegen back-ends.

			Execution time (s)		
		Codegen Implementation	Constructor hash switch	Name and arity switch	Nested if-else
Problem	Number of rules	Problem Size			
		16	2.36	2.48	6.83
		18	5.10	5.64	23.8
		20	16.5	18.7	93.7
		20	0.501	0.506	0.550
		40	0.543	0.547	0.699
Sieve-REC	39	60	0.618	0.655	1.30
		80	0.688	0.784	2.33
		100	0.770	0.923	3.23

		Compilation time (s)		Slowdown
	Pattern Match Compilation	-O 3 (off)	-O 4 (on)	
Problem	Number of rules			
Benchexpr-REC	155	1.43	2.32	1.62
Benchsym-REC	155	1.50	2.19	1.46
Benchtree-REC	155	1.44	2.17	1.50
Bubblesort-REC	16	0.567	0.645	1.14
Calls-REC	9	0.520	0.538	1.03
Factorial-REC	6	0.480	0.517	1.08
Fibonacci-REC	5	0.483	0.482	0.997
GarbageCollection-REC	11	0.498	0.592	1.19
Hanoi-REC	31	0.604	0.701	1.16
Mergesort-REC	23	0.751	0.756	1.01
Quicksort-REC	23	0.652	0.729	1.12
Sieve-REC	39	0.784	1.01	1.29

Table D.9: Stratego compile times (REC problems) for different optimisation levels. The rightmost column shows the increase in compilation time with pattern match compilation "on" vs. "off+inlining".