Optimising First-Class Pattern Matching

Smits, J.; Hartman, Toine ; Cockx, Jesper

DOI
10.1145/3567512.3567519

Publication date
2022

Document Version
Final published version

Published in
SLE 2022: Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering

Optimising First-Class Pattern Matching

Jeff Smits*
Delft University of Technology
Delft, The Netherlands
j.smits-1@tudelft.nl

Toine Hartman*
Independent
Delft, The Netherlands
tjbhartman@gmail.com

Jesper Cockx
Delft University of Technology
Delft, The Netherlands
j.g.h.cockx@tudelft.nl

Abstract

Pattern matching is a high-level notation for programs to analyse the shape of data, and can be optimised to efficient low-level instructions. The Stratego language uses *first-class pattern matching*, a powerful form of pattern matching that traditional optimisation techniques do not apply to directly.

In this paper, we investigate how to optimise programs that use first-class pattern matching. Concretely, we show how to map first-class pattern matching to a form close to traditional pattern matching, on which standard optimisations can be applied.

Through benchmarks, we demonstrate the positive effect of these optimisations on the run-time performance of Stratego programs. We conclude that the expressive power of first-class pattern matching does not hamper the optimisation potential of a language that features it.

CCS Concepts: • **Software and its engineering** → **Patterns**; Translator writing systems and compiler generators; • **Theory of computation** → *Pattern matching*.

Keywords: pattern matching, optimisation, strategic programming

ACM Reference Format:
Jeff Smits, Toine Hartman, and Jesper Cockx. 2022. Optimising First-Class Pattern Matching. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22), December 06–07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3567512.3567519

1 Introduction

Pattern matching is a fundamental tool for expressing programs by case analysis, and is used in functional programming, term rewriting, logic programming, and more.

In many languages, pattern matching is expressed through *case expressions*, which combine matching, variable binding, and control flow in a single construct. In contrast, *first-class pattern matching* [39] breaks apart these concepts into three separate constructs.

For example, Figure 1 shows a case expression in OCaml, and the same expression in Stratego [4, 5, 38] using first-class pattern matching. The *match expression* `?pattern` matches the pattern against an implicit scrutinee, the 'current term', and either continues with pattern variables bound or fails. *Scoping* $\{\overline{n} : S\}$ binds the variables $\overline{n}$ locally for use in patterns and expressions. Finally, the *choice operator* `fst <+ alt` runs the first argument and tries the alternative in case it fails. Because Stratego has these separate concepts, backtracking from a failed pattern match is a core feature of the semantics.

Programs that use case expressions can be compiled into a decision tree [2, 22] or an automaton [12, 13, 21, 24], which can be optimized by reordering branches and arguments. However, these techniques do not apply directly to first-class pattern matching, as matches are spread out in the program. As a consequence, they are not used in Stratego, hurting the run-time performance of Stratego programs.

Our main contributions are the following:

- We demonstrate the mismatch between common pattern match optimisations and first-class pattern matching by example (Section 3).
- We describe an intermediate representation (IR) for Stratego that resembles case expressions, and show how to transform Stratego programs to this IR (Section 4).
- We evaluate our IR and optimisation with a prototype implementation in the Stratego compiler, benchmarking to two different workloads (Section 5).

We starts with an introduction to Stratego in Section 2.

```
let calc n = match n with
  | Plus  (S m, n)   -> S (Plus (m, n))
  | Minus (n,   O)   -> n
  | Minus (S m, S n) -> Minus (m, n)
  | Minus (O,   S _) -> failwith "oops";;
```

```
Calc = {m, n: ?Plus( S(m), n   ); !S(Plus(m, n)) }
    <+ {n   : ?Minus(n   , O() ); !n             }
    <+ {m, n: ?Minus(S(m), S(n)); !Minus(m, n)   }
    <+        ?Minus(O() , S(_)); fatal-err(|"oops")
```

Figure 1. A case expression in OCaml (top) and Stratego (bottom) with four branches, demonstrating matching, scoping, and choice operators in Stratego.

*Joint first authors

SLE '22, December 06–07, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9919-7/22/12.
https://doi.org/10.1145/3567512.3567519

$$
\begin{array}{rcll}
S & ::= & \texttt{id} \mid \texttt{fail} & \text{identity and failure} \\
 & \mid & S\ ;\ S & \text{sequence} \\
 & \mid & \{\ \overline{n}\colon S\ \} & \text{scope} \\
 & \mid & ?T \mid\ !T & \text{match and build} \\
 & \mid & S < S + S & \text{guarded choice} \\
T & ::= & n & \text{variable} \\
 & \mid & l & \text{literal} \\
 & \mid & n(\overline{T}) & \text{constructor}
\end{array}
$$

Figure 2. Stratego core grammar.

2 A Short Introduction to Stratego

Stratego [40] is a gradually typed [33] term rewriting language with programmable rewrite strategies. The terms that are rewritten are ATerms [36], whose sorts and constructors can be defined in Stratego. There is an open world assumption, i.e. Stratego assumes that there may be more constructors and sorts than are defined.

Stratego has syntactic sugar over a core language that strongly resembles System S [39], a core calculus for rewriting and strategies. The Stratego core grammar is shown in Figure 2 and defines identity and failure strategies, sequences, lexical scopes for term variables, match and build, and guarded choice. Matching and building is done on term patterns, which includes variables, literals, and constructors and their arguments.

The semantics of these strategies and terms is defined in Figure 3. For a full explanation of these rules see the System S paper [39]. We will only go over the highlighted rules that pertain to first-class pattern matching. Each rule takes a pair of the store and the 'current term', where the store keeps local variable bindings to values. The rule applies the strategy on the arrow to produce another pair of store and term. The result on the right-hand side of the arrow may also be failure (the ↑).

The first two highlighted rules describe scoping behaviour. A list of fresh variables of a scope are removed from the store for execution of the body of the scope, then bindings of those variables from before the scope ($St|\overline{x}$) are added again, while preserving other bindings from the body of the scope. Failure in the scope body is propagated.

The next three rules are part of the semantics for matching, in particular matching a variable. An unbound variable is bound in the store. A *bound* variable's binding is compared against the current term, failing if they differ. This provides for non-linear pattern matching semantics in Stratego.

The final four rules are for the guarded choice: Whether the guard (first strategy) fails decides which of the second and third strategy is evaluated. This can result in local backtracking of variables: In the last rule, the failed guard s_1 causes s_3 to be evaluated on the original store St, without bindings from the partial execution of s_1.

Our main point here in looking at the semantics of the core of Stratego, is how pervasive first-class pattern matching to

Strategy S applied to store St and term T $\quad \boxed{St; T \xrightarrow{S} St; T}$

$$St; t \xrightarrow{\textbf{id}} St; t \qquad St; t \xrightarrow{\textbf{fail}} \uparrow$$

$$\frac{St; t \xrightarrow{s_1} St'; t' \quad St'; t' \xrightarrow{s_2} St''; t''}{St; t \xrightarrow{s_1; s_2} St''; t''}$$

$$\frac{St; t \xrightarrow{s_1} St'; t' \quad St'; t' \xrightarrow{s_2} \uparrow}{St; t \xrightarrow{s_1; s_2} \uparrow} \qquad \frac{St; t \xrightarrow{s_1} \uparrow}{St; t \xrightarrow{s_1; s_2} \uparrow}$$

$$\frac{St \setminus \overline{x}; t \xrightarrow{s} St'; t'}{St; t \xrightarrow{\{\overline{x}\,:\,s\}} (St' \setminus \overline{x}) \cup (St \mid \overline{x}); t'} \qquad \frac{St \setminus \overline{x}; t \xrightarrow{s} \uparrow}{St; t \xrightarrow{\{\overline{x}\,:\,s\}} \uparrow}$$

$$\frac{x \notin Dom(St)}{St; t \xrightarrow{?x} St \cup \{x \mapsto t\}; t} \qquad \frac{St(x) = t}{St; t \xrightarrow{?x} St; t} \qquad \frac{St(x) \neq t}{St; t \xrightarrow{?x} \uparrow}$$

$$\frac{St_0; t_1 \xrightarrow{?t'_1} St_1; t_1 \ldots St_{n-1}; t_n \xrightarrow{?t'_n} St_n; t_n}{St_0; f(t_1, \ldots, t_n) \xrightarrow{?f(t'_1, \ldots, t'_n)} St_n; f(t_1, \ldots, t_n)}$$

$$\frac{f \neq g \vee m \neq n}{St_0; g(t_1, \ldots, t_m) \xrightarrow{?f(t'_1, \ldots, t'_n)} \uparrow} \qquad \frac{t = l}{St; t \xrightarrow{?l} St; t} \qquad \frac{t \neq l}{St; t \xrightarrow{?l} \uparrow}$$

$$\frac{\mathrm{vars}(t_2) \subseteq Dom(St)}{St; t_1 \xrightarrow{!t_2} St; St(t_2)} \qquad \frac{\mathrm{vars}(t_2) \nsubseteq Dom(St)}{St; t_1 \xrightarrow{!t_2} \uparrow}$$

$$\frac{St; t \xrightarrow{s_1} \uparrow \quad St; t \xrightarrow{s_3} \uparrow}{St; t \xrightarrow{s_1 < s_2 + s_3} \uparrow} \qquad \frac{St; t \xrightarrow{s_1} St'; t' \quad St'; t' \xrightarrow{s_2} St''; t''}{St; t \xrightarrow{s_1 < s_2 + s_3} St''; t''}$$

$$\frac{St; t \xrightarrow{s_1} St'; t' \quad St'; t' \xrightarrow{s_2} \uparrow}{St; t \xrightarrow{s_1 < s_2 + s_3} \uparrow} \qquad \frac{St; t \xrightarrow{s_1} \uparrow \quad St; t \xrightarrow{s_3} St'; t'}{St; t \xrightarrow{s_1 < s_2 + s_3} St'; t'}$$

Figure 3. Stratego core operational semantics. ↑ is failure. Some rules for first-class pattern matching are highlighted.

a language design. Because of the use of backtracking, every construct of the language needs to take 'failure' into account. At the same time backtracking provides the opportunity for a different code style in Stratego than in a typical functional programming language. An alternative result ('fail') is always available, without the requirement to explicitly propagate that failure, as it freely bubbles up to the nearest point where a choice catches it.

Separate matching and scoping have their own benefits. A typical use case is found in the Stratego standard library strategy `fetch-elem(s)`. This strategy returns the element in a list for which the strategy parameter `s` matches. It can be implemented as follows:

```
fetch-elem(s) =
  is-list; one(s; ?x); !x
```

What this code does is (1) test that the input term is a list, (2) use a generic traversal primitive[1] to visit the list elements, attempting to apply the strategy parameter until it succeeds on one of the list elements, and (3) build variable x to return its binding as the result of **fetch-elem**. Notably, the strategy parameter to the generic traversal attempts to apply s and if it succeeds it binds the result to x. The resulting list is ignored as we have found what we wanted during the traversal and bound it to a variable that is scoped outside of the traversal by **fetch-elem**. This trick can be applied to much deeper traversals than a single list, allowing easy extraction of information without result tuples everywhere.

3 Pattern Matching Optimisation and First-Class Pattern Matching

The naive way to execute pattern matching of a case expression would be to attempt each branch of the case expression in isolation. For example, when executing the OCaml function from Figure 1 on , the naive method would: (1) fail to match the first pattern because the outermost constructor does not match, then (2) attempt to match the second pattern, match the outermost constructor, but fail to match on the second argument, then (3) attempt to match the third pattern, matching the outermost constructor *again*, which already demonstrates the naivety. Each pattern is tested in isolation, even when we know the outmost constructor from the previous branch. Pattern matching optimisation techniques leverage the information gathered by previously tried branches to make pattern matching faster.

In Stratego the execution method for the code in Figure 1 is the naive method described just now. Even if we wrote more high-level rewrite rules that looked more like the OCaml code, it would still desugar to the Stratego code in Figure 1. And the desugared code definitely tests each pattern in isolation during a separate first-class pattern matching operation.

Note that first-class pattern matching is not an intermediate representation used by compilers or publications for the description of Stratego's semantics, this is part of the language and actually used by the end user. A common pattern in Stratego is to have a ruleset where some rewrite rules match a different pattern but have otherwise the same logic and result. To remove this code duplication, we can use the core operations like match, choice, and build to write a single strategy that matches both patterns. For example, if we compute the pessimistic time complexity of a language with normal and parallel for loops we see some code duplication:

```
max-complexity: For(i, lo, hi, b) -> <subtS> (hi, lo)
max-complexity: ForPar(i, lo, hi, b) ->
                  <subtS> (hi, lo)
```

[1]This is a central feature of Stratego for its programmable rewrite strategies, for more on this, see Visser and Benaissa [39, §2.3].

```
Calc = match sequential
  case m, n | Plus(S(m), n): S(Plus(m, n))
  case n    | Minus(n, O()): n
  case m, n | Minus(S(m), S(n)): Minus(m, m)
  case      | Minus(O(), S(_))
       when fatal-err(|"Negative result"): id
end
```

Figure 4. A case expression version of Figure 1.

We can remove this code duplication using first-class pattern matching:

```
max-complexity =
    (?For(i, lo, hi, b) <+ ?ForPar(i, lo, hi, b))
  ; <subtS> (hi, lo)
```

In general, we cannot expect all Stratego code to match case expression style code as closely as the example in Figure 1. In other words, the information that is close together in a match case expression, can be farther apart in Stratego code, and not readily available to fuel optimisation. In this paper, we will tackle the general problem of optimising first-class pattern matching.

4 A Stratego Compatible Case Expression

Pattern match optimisation is a well researched problem in the context of functional programming and case expressions [2, 3, 7, 13, 22, 24, 28, 30]. The key idea of our work to find a comparable construct that interacts well with Stratego's backtracking semantics, and translate first-class pattern matching to that construct. Once we have a clear list of branches, we can reuse previous pattern match optimisation ideas, adapting them to Stratego's execution model with local backtracking.

We first look at Stratego code that we can easily translate, to find out what minimum requirements there are for our case expression construct in Stratego. Figure 1 shows an already desugared form of rewrite rules. This core Stratego code is a regular structure of a chain of choices, where each guard is a scoped expression starting with a match. These matches should become the left-hand sides of our case expression branches. Our construct also needs to be able to introduce variables like the scopes, have right-hand sides for the builds, and guards in case anything fails and other (overlapping) patterns need to be tried. An example of the case expression that complies with these requirement is displayed in Figure 4.

Each branch has some local variables to scope, a pattern, a right-hand side, and optionally a guard. As you can see, the right-hand side of the last case statement is the identity strategy. All the logic is put into the guard of the match, where, if anything fails, we can backtrack to another case. This is a Stratego interpretation of the concept of guards, where failure rather than a boolean value governs the behaviour. In later sections we will see how these guards play a key role

in translating first-class patterns to case expressions despite Stratego's backtracking semantics.

The naive semantics of our case expression can be shown through a translation back into Stratego core shown in Figure 5. A pattern match on term `t_0` and guard `s_1` decide whether we evaluate RHS `s_2`. It is naive because evaluation on e.g. `Plus(O(), O())` would still try each pattern and fail every, whereas the ideally we would fail to match in just two steps, matching the outer `Plus` constructor and finding the `O()` as the first child. For the optimisation of case expression patterns, we can exploit overlap and mutually exclusive patterns to get this behaviour. This can be visualised as a *matching automaton*, exemplified in Figure 6. Each intermediate node is labeled with the path into the term that is matched. The Λ is the empty path denoting the root term, and $p.n$ is the (0-indexed) n^{th} child of the term denoted by the path p. Each edge is labeled with the constructor or value matched against. The final nodes of the automaton refer to the n^{th} branch of the case expression.

```
match sequential                          {xs: t_0; s_1 < s_2 +
  case xs | t_0                           match sequential
      when s_1: s_2          ⇒              // more cases
  // more cases                           end }
end
match sequential
  // empty                   ⇒           fail
end
```

Figure 5. Naive semantics for case expressions by transformation to Stratego core.

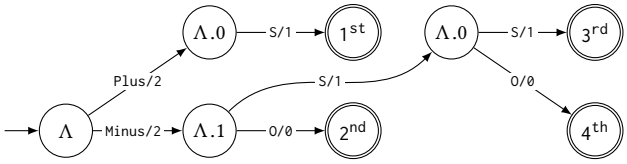

Figure 6. Optimised match automaton for the running example.

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

Figure 7. Transformation of guarded choice to case expression.

```
case xs | _ when {ys: ?p; s1}: s2
                    ⇓
case xs, ys | p when s1: s2
```

Figure 8. Extracting a match from a case expression guard.

4.1 Translation into Case Expressions

In order to have a small set of transformations, we defined our translation on Stratego core. While our examples are edited for legibility, Stratego core as generated by the compiler through desugaring makes every variable globally unique and explicitly scoped. We will therefore not concern ourselves with name conflicts and capture-preserving substitution for this translation.

Choices to Cases. The simplest transformation of guarded choices to case expressions is given in Figure 7. Without knowing anything about the strategies used in the choice, we can safely put `s1` into the guard for the first branch and `s2` into the right-hand side. `s3` ends up in the second branch, where either the guard or the right-hand side would work. We choose the guard here because it interacts better with the next transformations.

Match Extraction. In order to take advantage of optimisations for case expressions, we need to have patterns in each branch of course. So once we have case expressions, we can start extracting patterns from the guards, as shown in Figure 8. Here we have the most general case where a branch already scopes variables `xs`, and the guard scopes variables `ys`, which are combined in the result. As long as the guard then starts with a match, while the branch pattern is a wildcard, we can move the pattern from guard to branch. If anything occurs after that in the guard (`s1`), that stays. And the RHS (`s2`) stays as well, it is not affected by this transformation.

RHS Extraction. Apart from identifying the pattern match in the guard, we can see if the guard is redundant. If we can statically guarantee that the guard always succeeds, it might as well be put into the RHS. A simple local analysis of the guard is enough to find the typical simple cases, such as the build of a term where all variables in the term are guaranteed to be bound.

Flattening. Since guarded choices are a nested structure, our case expressions are also nested in the guards of case expressions at this point. In order to flatten this structure, we define a transformation that flattens a case expression in the guard of another case expression in Figure 9. The local strategy `s` that is introduced provides some amount of code sharing, at the expense of creating and calling a closure. This can be avoided by treating these local strategies specially as part of the shared tail in the matching automaton during optimisation of the case expression, although our current prototype does not do so.

5 Evaluation

To evaluate our pattern match optimisation, we have implemented a prototype optimisation and included it into a fork of the Stratego compiler. This prototype is limited to

generating decision trees rather than automata, and therefore generates duplicate code. But it should still result in improved run time performance. To evaluate this optimisation, we benchmark some Stratego program executions with and without the optimisation. Before we test the performance, we first test the implementation to be behaviour preserving. At the end of this section we consider threats to the validity of our results. We aim to answer the following questions:

RQ1. Does the optimisation improve run time performance of Stratego programs?
RQ2. Is there overhead for small matches?
RQ3. Is the optimisation effective in practice on "normal" code bases?
RQ4. What is the relative compile time cost of the prototype optimisation?

5.1 Correctness

For behaviour preservation testing, first we use the compiler test suite and make sure all the tests succeed with optimisation on. This compiler test suite numbers 159 tests and was also used in the past to migrate the compiler back-end from C to Java without breaking any edge cases in program behaviour. Then, we added 14 tests that explicitly cover situations that are affected by our optimisation. These all succeed with our current prototype.

We also run the modified Stratego compiler with and without the optimisation on, on the set of benchmark programs used for performance evaluation below, comparing the computed result.

5.2 Performance

For our performance evaluation we describe our benchmark setup, and the subjects on which we run our benchmark before discussing the results for each of the subjects.

Environment. We ran our experiments on a Macbook Pro (Early 2013) with an Intel Core i7 2.8 GHz CPU, 16 GB 1600 MHz DDR3 RAM, and an SSD. The machine is running Mac OS 10.14.5, Java OpenJDK 1.8.0_212, and Docker 4.9.1 (81317).

Subjects. We use algorithmic rewriting problems of the Rewriting Engine Competition (REC) [10]. These problems are generated from the REC language, and can be translated into up to 18 different languages for comparison of rewriting engines. There are problems for sorting (bubble sort, merge sort, quick sort), numeric functions (Fibonacci, factorial, prime sieve of Erastosthenes), and other kinds of problems. The 'algorithmic' problems are expressed as abstract syntax of programs and rewrite rules that do a small-step interpretation of that abstract syntax.

The REC programs are not typical programs as would be written in Stratego by hand, which is acknowledged in

```
match sequential
  /* other branches */
  case xs | _ when
    match sequential
      case ys | p1 when s1: s2
      case zs | p2 when s3: s4
      /* more branches */
    end: s5
end

                    ⇓

let s = s5
in
  match sequential
    /* other branches */
    case xs, ys | p1 when s1; s2: s
    case xs, zs | p2 when s3; s4: s
    /* more branches with extended guard and RHS s */
  end
end
```

Figure 9. Transformation of nested case expressions to a flat case expression.

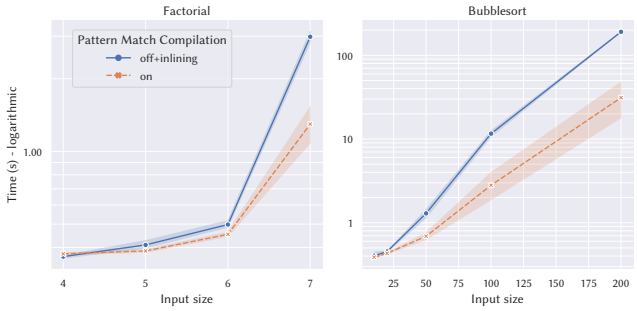

Figure 10. Benchmark of REC problems Bubblesort and Factorial.

Durán et al. [10, §4.1]. Therefore, like Durán et al., we also use a program-transformation problem set based on the Tiny Imperative Language (TIL) [9].

Data Collection. We use the Java Microbenchmark Harness (JMH) to manage the warm-up of the Java Virtual Machine (JVM), preparing the benchmarks, and repeating runs with different parameters. We used *Single shot time* as the benchmark mode, with 5 warm-up iterations and 5 measurement iterations in two forks of the virtual machine.

REC. Most programs from the REC subject show a significant speed-up, while none slow down due to our optimisation. We show two examples in Figure 10, both of which show that we can answer RQ1: our optimisation *does* improve run time performance of Stratego programs. The factorial program is interesting because it has a small match of six different branches, and yet there is a clearly visible run time

improvement. Therefore we conclude that, as expected, there is no overhead for small matches (RQ2).

TIL. We included TIL for RQ3 to see what impact our optimisation has on a more typical Stratego program that does program-transformation. Figure 11 shows that whether we run Stratego code that optimises TIL programs (left) or executes them through big-step transformations (right), there is a small but clear speed-up from our pattern match compiler. This is visible due to our generated large TIL input programs that read one integer, add one to it and put it in a new variable, repeated some number of times, then write the result:

```
var n0; n0 := readint();
var n1; n1 := n0 + 1; // repeated...
write(n500); // ... e.g. 500 times
```

Compilation Time. Figure 12 shows that compilation times (RQ4) for REC problems show that our prototype optimisation increases the compilation time of the Stratego compiler, which makes sense as these programs largely consist of rules that the optimisation works on. For the total compile time of the TIL language project, the impact is milder due to smaller rulesets and a larger amount of other compilation work, between 9.68 % and 11.97 % of the median of the baseline. We think that at the current increase of compilation time, the optimisation is worthwhile to include in the main Stratego compiler, with the option to turn it off.

5.3 Threats to Validity

We consider generalisability of the results (external validity), factors that allow for alternative explanations (internal validity), and suitability of metrics for the evaluation's goals (construct validity).

External Validity. The results of our evaluation are specific to our implementation for the Stratego language. They are merely a datapoint for the general argument that pattern match optimisation can be used on first-class pattern matching. Within Stratego, we have attempted to provide a good range of styles of Stratego programs by using not only the algorithmic problems from REC but also a more typical use of Stratego with TIL.

Internal Validity. A typical alternative explanation for why an optimisation performs well is that it actually removes relevant code and breaks semantics preservation. We have already addressed this concern in Section 5.1.

Another explanation would be that we misconfigured our measurements, which we did at one point: we got entirely similar results for our benchmarks because the optimisation was not actually applied in either measurement. We were able to manually inspect the compilation results and notice

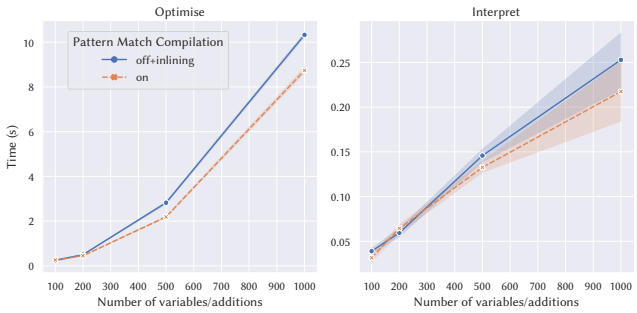

Figure 11. Benchmark of TIL optimisation and big-step interpretation on a simple addition program.

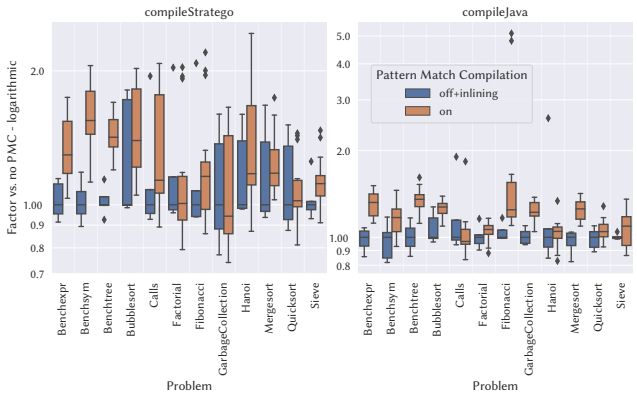

Figure 12. Compilation times for REC benchmark problems.

this though, so we checked that in situations where results are similar.

We used the newest Stratego compiler for our measurements, and our prototype is built on this new compiler. This compiler was designed to be highly incremental, and to achieve this, inlining of strategies was removed. For first-class pattern matching this can be a limitation as a strategy might be defined by choices between differently labeled rewrite rules, but these are optimised separately when not inlined. In our benchmarks we partially mitigated this by manually inlining these code patterns.

Construct Validity. As the experiments are performed in a virtualised environment (Docker), the absolute numbers of our measurements may include some virtualisation overhead. However, the overhead is in all measurements, where we compare between two measurements rather than interpret the absolute numbers. And this allows us to provide a virtual machine image that can be reused to reproduce our experiments.

We used JMH which contains many best practices for benchmarking on the JVM in its default options. We let it run warm-up iterations, checked for stabilisation of the times, and aggregated results to counter noise from background tasks we could not eliminate on the benchmark machine.

6 Related Work

Pattern matching has existed since the early ages of computer science, with matching for tree-shaped data being first described in 1972 by Karp et al. [19], with performance being a key consideration from the beginning. It was introduced to the world of programming languages by SASL [35] and Hope [6]. Since then, it has been one of the staples of functional programming languages like ML and Haskell, and more recently it has been introduced to other mainstream languages such as Python, Swift, Ruby, and Rust.

6.1 First-Class Pattern Matching

First-class pattern matching is a feature that so far only exists in the Stratego language [5, 32, 38, 40]. The core language of Stratego is System S [39], which first explored the idea of separating pattern matching into operators *match*, *scope*, and *choice*. Visser [37] describes this unique form of pattern matching in more detail. First-class patterns[2] are a similarly named, but fundamentally different concept to first-class pattern matching.

Cirstea et al. [8] describe a translation from first-class pattern matching to a regular term rewriting system, with the goal of proving termination. This technique could in theory be applied to compile first-class pattern matching by first transforming it into a traditional rewrite system and then applying standard compilation techniques. However, this would likely not lead to good results as the translation has not been designed for this purpose and the encoding might introduce an extra overhead. The technique also seems to require a closed world assumption, while Stratego has an open world assumption.

6.2 First-Class Patterns

We consider pattern matching a first-class *expression* in Stratego because it is a primitive operation in the (strategy) expression language. While it sounds similar, first-class patterns are a very different concept, namely that patterns are *values* that can be manipulated in a programming language. This idea shows up in both the (pure) object-oriented programming world [16] and the functional programming world [17, 34].

While this is a powerful idea, it is not directly related to first-class pattern matching as seen this paper. Indeed, the path polymorphism of Jay and Kesner [17] provides something more similar to the generic traversal capabilities of Stratego, a feature we did not focus on this paper.

Depending on the way that first-class patterns are created and used, it may be possible to optimise the pattern match at run time. If at run time the information of the different patterns is not co-located enough for a just-in-time optimisation based on traditional pattern matching optimisation, the ideas from this paper can be reused to bring such information together. This would be particularly relevant for any programming language with first-class patterns and backtracking semantics.

[2] https://hackage.haskell.org/package/first-class-patterns

6.3 Pattern Match Compilation

There are two main techniques that are used for compiling pattern matching to efficient code: on the one hand, compiling to a *discrimination tree* (also known as a *decision tree* or *case tree*) as pioneered by Overmars and van Leeuwen [25] and Hoffmann and O'Donnell [15], and similar work on *deterministic* automata by Gräf [13], Pettersson [26] and Nedjah et al. [24], and on the other hand compiling to a *backtracking automaton* as proposed by Augustsson [1, 2] and further developed by Maranget [21] and Fessant and Maranget [12]. A backtracking automaton has the advantage that it avoids the code duplication that can occur in the construction of a case tree. However, to avoid this duplication it might need to inspect the same term more than once as a result. In theory, this seems like a classic tradeoff between code size and run-time performance. But in practice the difference is less clear, as noted by Fessant and Maranget [12]:

> However, sophisticated compilation techniques exist that minimise the drawbacks of both approaches. [...] In the absence of a practical comparison of full-fledged algorithms, choosing one technique or the other reflects one's commitment to guaranteed code size or guaranteed run-time performance.

A third approach introduced by Jørgensen [18] is to use *partial evaluation* to compile definitions by pattern matching in a way that avoids examining or decomposing arguments multiple times, and eliminates code duplication. Sestoft [31] further develops this idea and specialises the partial evaluator to a more traditional pattern match compiler that produces a discrimination tree. The paper notes that this tree can be further optimised by eliminating duplicate trees through hash-consing and by replacing equality checks with switch statements. In our prototype we use switch statements, while eliminating duplicate trees is still left to do.

Strict vs. Lazy Evaluation. In a language that uses strict evaluation such as Stratego, we are free to inspect the arguments of a function in any order, so there is a lot of room for possible optimisation. This is a major advantage over lazy languages, where changing the order of evaluation might influence the termination of our program and hence room for optimisation is more limited [20, 21].

Pattern Match Compilation in Practice. Many programming languages compile pattern matching into a discrimination tree, following the example of the ML compiler [3, 7]. Other languages compile pattern matching to a backtracking automaton, for example RML [27, chapter 7] and

OCaml. OCaml also applies several optimisations to the control flow of the generated automata to mitigate the performance impact of backtracking [12]. Finally, yet other implementations of (mostly lazy) languages see pattern matching as mere syntactic sugar for nested case expression, for example Lazy ML [1, 2] and GHC. To deal with overlapping cases, Lazy ML uses a **default** construct that triggers backtracking. In contrast, GHC expands catchall cases and uses *join points* in its core language to avoid duplication of terms [11]. In the current Stratego implementation, closures and their calls are computationally expensive, so a direct adaptation of this approach would result in poor performance.

6.4 Heuristics

Over the years, a large number of different heuristics have been proposed for producing better discrimination trees or matching automata, depending on whether one wishes to optimise for code size or run-time performance. Detailed comparisons between these different heuristics can be found in the studies by Scott and Ramsey [29] and Maranget [22]. Here we list some of the most common ones, with an indication of whether they optimise for run-time or code size:

Relevance (run-time and code size) Pick an argument position that is matched on in a higher priority clause [3].

Necessity (run-time) Pick an argument position that must be inspected by *any* matching algorithm [22, 23, 30].

Large branching factor (run-time) Pick an argument position with the largest number of distinct constructors [7].

Small branching factor (code size) Pick an argument position with the smallest number of distinct constructors [3, 23].

Small arity factor (code size) Pick an argument position where the total arity of all constructors is lowest [3].

Small default (code size) Pick an argument position with the smallest number of wildcard patterns [3].

Scott and Ramsey [29] find little difference in performance between these heuristics for most practical examples, with a few notable exceptions. Maranget [22] recommends a combination of necessity, small branching factor, and arity, but also notes that the choice depends on the specifics of the language and the expected kinds of pattern matching code. Hence, it would be interesting to experiment with what combination of heuristics is the most suitable for compiling idiomatic Stratego code.

7 Conclusion

We have introduced the problem of optimising *first-class pattern matching* as seen in the Stratego programming language, where pattern matching is broken apart into three constructs: match, scope, and choice. To solve this problem, we have developed a behaviour-preserving transformation that combines these three constructs into an intermediate representation that resembles *case expressions* from functional programming, while still adhering to the local backtracking semantics of Stratego. This allowed us to optimise first-class pattern matching with the same techniques used for 'regular' pattern matching.

We still plan to make some practical improvements to our prototype implementation, investigate which heuristics for the discrimination trees work best for typical Stratego code, and investigate the addition of closed types to Stratego and what performance effect this may have on our optimisation.

Nevertheless, our benchmarks have demonstrated that our current prototype can already have a positive effect on the run-time performance of Stratego programs. With a speed-up that increases with the input size, we expect Stratego users will be willing to pay a 10% increase in compile time.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments and suggestions. This research was supported by a gift from the Oracle Corporation.

Division of Labour. The primary work for this publication was done by Toine Hartman for his master's thesis [14], under the direct supervision of Jeff Smits. Minor further development, primarily the benchmarks for TIL, were done by Jeff Smits. Writing of this article was primarily done by Jeff Smits with the help of Jesper Cockx.

In Memoriam. This work was partially developed under the guidance of the late Eelco Visser. We miss him dearly.

References

[1] Lennart Augustsson. 1984. A Compiler for Lazy ML. In *Proceedings of the ACM Symposium on LISP and Functional Programming (LFP '84)*. Association for Computing Machinery, New York, NY, USA, 218–227. https://doi.org/10.1145/800055.802038

[2] Lennart Augustsson. 1985. Compiling Pattern Matching. In *Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science, Vol. 201)*. Springer-Verlag, Nancy, France, 368–381. https://doi.org/10.1007/3-540-15975-4_48

[3] Marianne Baudinet and David MacQueen. 1985. *Tree Pattern Matching for ML (Extended Abstract)*. Technical Report.

[4] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2006. Stratego/XT 0.16: components for transformation systems. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2006, Charleston, South Carolina, USA, January 9-10, 2006*, John Hatcliff and Frank Tip (Eds.). ACM, 95–99. https://doi.org/10.1145/1111542.1111558

[5] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming* 72, 1-2 (2008), 52–70. https://doi.org/10.1016/j.scico.2007.11.003

[6] Rod M. Burstall, David B. MacQueen, and Donald Sannella. 1980. HOPE: An Experimental Applicative Language. In *LISP Conference*. 136–143. https://doi.org/10.1145/800087.802799

[7] Luca Cardelli. 1984. Compiling a Functional Language. In *Proceedings of the ACM Symposium on LISP and Functional Programming (LFP '84)*. Association for Computing Machinery, New York, NY, USA, 208–217. https://doi.org/10.1145/800055.802037

[8] Horatiu Cirstea, Sergueï Lenglet, and Pierre-Etienne Moreau. 2015. A faithful encoding of programmable strategies into term rewriting systems. In *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland (LIPIcs, Vol. 36)*, Maribel Fernández (Ed.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 74–88. https://doi.org/10.4230/LIPIcs.RTA.2015.74

[9] James Cordy. 2009. *TILChairmarks*. https://www.program-transformation.org/Sts/TILChairmarks.html (visited on 2022-07-26).

[10] Francisco Durán, Manuel Roldán, Jean-Christophe Bach, Emilie Balland, Mark G. J. van den Brand, James R. Cordy, Steven Eker, Luc Engelen, Maartje de Jonge, Karl Trygve Kalleberg, Lennart C. L. Kats, Pierre-Etienne Moreau, and Eelco Visser. 2010. The Third Rewrite Engines Competition. In *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6381)*, Peter Csaba Ölveczky (Ed.). Springer, 243–261. https://doi.org/10.1007/978-3-642-16310-4_16

[11] Richard Eisenberg and GHC development team. 2020. *System FC, as implemented in GHC*. Technical Report. https://gitlab.haskell.org/ghc/ghc/-/blob/97655ad88c42003bc5eeb5c026754b005229800c/docs/core-spec/core-spec.pdf

[12] Fabrice Le Fessant and Luc Maranget. 2001. Optimizing Pattern Matching. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*. Association for Computing Machinery, New York, NY, USA, 26–37. https://doi.org/10.1145/507635.507641

[13] Albert Gräf. 1991. Left-to-Right Tree Pattern Matching. In *Rewriting Techniques and Applications (Lecture Notes in Computer Science, Vol. 488)*, Ronald V. Book (Ed.). Springer, Como, Italy, 323–334. https://doi.org/10.1007/3-540-53904-2_107

[14] Toine Hartman. 2022. *Optimising First-Class Pattern Match Compilation*. Master's thesis. Delft University of Technology. Advisor(s) Jeff Smits. http://resolver.tudelft.nl/uuid:414026ac-b08e-49f3-8aca-1367766161bb

[15] Christoph Martin Hoffmann and Michael James O'Donnell. 1982. Pattern Matching in Trees. *J. ACM* 29, 1 (jan 1982), 68–95. https://dl.acm.org/doi/10.1145/322290.322295

[16] Michael Homer, James Noble, Kim B. Bruce, Andrew P. Black, and David J. Pearce. 2012. Patterns as objects in grace. In *Proceedings of the 8th Symposium on Dynamic Languages, DLS '12, Tucson, AZ, USA, October 22, 2012*, Alessandro Warth (Ed.). ACM, 17–28. https://doi.org/10.1145/2384577.2384581

[17] C. Barry Jay and Delia Kesner. 2009. First-class patterns. *Journal of Functional Programming* 19, 2 (2009), 191–225. https://doi.org/10.1017/S0956796808007144

[18] Jesper Jørgensen. 1990. Generating a Pattern Matching Compiler by Partial Evaluation. In *Proceedings of the 1990 Glasgow Workshop on Functional Programming (Workshops in Computing)*, Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst (Eds.). Springer, 177–195. https://doi.org/10.1007/978-1-4471-3810-5_15

[19] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. 1972. Rapid Identification of Repeated Patterns in Strings, Trees and Arrays. In *Conference Record, Fourth Annual ACM Symposium on Theory of Computing, 1-3 May 1972, Denver, Colorado, USA*. ACM, 125–136. https://doi.org/10.1145/800152.804905

[20] Luc Maranget. 1992. Compiling Lazy Pattern Matching. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming (LFP '92)*. Association for Computing Machinery, New York, NY, USA, 21–31. https://doi.org/10.1145/141471.141499

[21] Luc Maranget. 1994. *Two Techniques for Compiling Lazy Pattern Matching*. Research Report RR-2385. INRIA. https://hal.inria.fr/inria-00074292 Projet PARA.

[22] Luc Maranget. 2008. Compiling pattern matching to good decision trees. In *Proceedings of the ACM SIGPLAN Workshop on ML (ML'08)*, Eijiro Sumii (Ed.). Association for Computing Machinery, New York, NY, USA, 35–46. https://doi.org/10.1145/1411304.1411311

[23] Nadia Nedjah and Luiza de Macedo Mourelle. 2001. Improving Space, Time, and Termination in Rewriting-Based Programming. In *Engineering of Intelligent Systems (Lecture Notes in Computer Science, Vol. 2070)*, László Monostori, József Váncza, and Moonis Ali (Eds.). Springer, 880–890. https://doi.org/10.1007/3-540-45517-5_97

[24] Nadia Nedjah, Colin D. Walter, and Stephen E. Eldridge. 1997. Optimal Left-to-Right Pattern-Matching Automata. In *Algebraic and Logic Programming, 6th International Joint Conference, ALP 97 - HOA 97, Southampton, U.K., Spetember 3-5, 1997, Proceedings (Lecture Notes in Computer Science, Vol. 1298)*, Michael Hanus, Jan Heering, and Karl Meinke (Eds.). Springer, 273–286. https://doi.org/10.1007/BFb0027016

[25] Mark H Overmars and Jan van Leeuwen. 1979. Rapid subtree identification revisited. (1979). Technical report.

[26] Mikael Pettersson. 1992. A Term Pattern-Match Compiler Inspired by Finite Automata Theory. In *Compiler Construction, 4th International Conference on Compiler Construction, CC 92, Paderborn, Germany, October 5-7, 1992, Proceedings (Lecture Notes in Computer Science, Vol. 641)*, Uwe Kastens and Peter Pfahler (Eds.). Springer, 258–270. https://doi.org/10.1007/3-540-55984-1_24

[27] Mikael Pettersson. 1999. *Compiling Natural Semantics*. Lecture Notes in Computer Science, Vol. 1549. Springer. https://doi.org/10.1007/b71652

[28] Philippe Schnoebelen. 1988. Refined Compilation of Pattern-Matching for Functional Languages. *Science of Computer Programming* 11, 2 (1988), 133–159. https://doi.org/10.1016/0167-6423(88)90002-0

[29] Kevin Scott and Norman Ramsey. 1999. *When Do Match-Compilation Heuristics Matter?* Technical Report. University of Virginia Dept. of Computer Science. https://doi.org/10.18130/V3GB4M

[30] R. C. Sekar, Ramashubramani Ramesh, and I. V. Ramakrishnan. 1995. Adaptive Pattern Matching. *SIAM J. Comput.* 24, 6 (1995), 1207–1234. https://doi.org/10.1137/S0097539793246252

[31] Peter Sestoft. 1996. ML Pattern Match Compilation and Partial Evaluation. In *Partial Evaluation (Lecture Notes in Computer Science, Vol. 1110)*, Olivier Danvy, Robert Glück, and Peter Thiemann (Eds.). Springer, Dagstuhl Castle, Germany, 446–464. https://doi.org/10.1007/3-540-61580-6_22

[32] Jeff Smits, Gabriël Konat, and Eelco Visser. 2020. Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System. *Programming Journal* 4, 3 (2020), 16. https://doi.org/10.22152/programming-journal.org/2020/4/16

[33] Jeff Smits and Eelco Visser. 2020. Gradually typing strategies. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, Ralf Lämmel, Laurence Tratt, and Juan de Lara (Eds.). ACM, 1–15. https://doi.org/10.1145/3426425.3426928

[34] Mark Tullsen. 2000. First Class Patterns. In *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000, Boston, MA, USA, January 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1753)*, Enrico Pontelli and Vítor Santos Costa (Eds.). Springer, 1–15. https://doi.org/10.1007/3-540-46584-7_1

[35] David Turner. 1983. *SASL Language Manual*.

[36] Mark G. J. van den Brand, H. A. de Jong, Paul Klint, and Pieter A. Olivier. 2000. Efficient annotated terms. *Software: Practice and Experience* 30, 3 (2000), 259–291. https://doi.org/10.1002/(SICI)1097-024X(200003)30:3%3C259::AID-SPE298%3E3.0.CO;2-Y

[37] Eelco Visser. 1999. Strategic Pattern Matching. In *Rewriting Techniques and Applications, 10th International Conference, RTA-99, Trento, Italy, July 2-4, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1631)*, Paliath Narendran and Michaël Rusinowitch (Eds.). Springer, 30–44. https://doi.org/10.1007/3-540-48685-2_3

[38] Eelco Visser. 2001. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The*

Netherlands, May 22-24, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2051), Aart Middeldorp (Ed.). Springer, 357–362. https://doi.org/10.1007/3-540-45127-7_27

[39] Eelco Visser and Zine-El-Abidine Benaissa. 1998. A core language for rewriting. *Electronic Notes in Theoretical Computer Science* 15 (1998), 422–441. https://doi.org/10.1016/S1571-0661(05)80027-1

[40] Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, Baltimore, Maryland, United States, 13–26. https://doi.org/10.1145/289423.289425