Defining Static Analyses via Syntax Tree Patterns

Master's Thesis of

Felix Kohlgrüber

at the Department of Informatics
Institute for Automation and Applied Informatics (IAI)

Reviewer: Prof. Dr. Veit Hagenmeyer
Advisor: Dipl-Inf. Oliver Scherer

January 2019 – June 2019

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 28.06.2019

. . . . . .

(Felix Kohlgrüber)

Abstract

Static analysis tools can automatically find and report a large number of issues in source code and can therefore be an important help for programmers. The number of available static analyses and their effectiveness is currently limited by the fact that writing them is difficult as well as language- and implementation-specific. This thesis proposes a declarative Domain-Specific Language (DSL) that allows specifying static analyses that operate on (abstract) syntax trees. In addition to the proposed concept, a proof-of-concept implementation in the Rust programming language is presented. An evaluation of the concept shows that common static analyses can be expressed using the proposed system and indicates that using the proposed system is easier compared to traditional approaches.

Zusammenfassung

Statische Analyse Werkzeuge können automatisch große Mengen von Problemen in Programmcode finden und sind somit eine wichtige Hilfe für Programmierer. Die Anzahl verfügbarer statischer Analysen und deren Effektivität ist aktuell dadurch limitiert, dass sie schwierig zu schreiben sowie sprach- und implementierungsabhängig sind. Diese Abschlussarbeit stellt eine neue deklarative Domain-Specific Language (DSL) vor, die das Spezifizieren von statischen Analysen, die auf (Abstract) Syntax Trees operieren, ermöglicht. Zusätzlich zu dem vorgeschlagenen Konzept wurde im Rahmen der vorliegenden Arbeit eine Proof of Concept Implementierung in der Programmiersprache Rust entwickelt. Eine Evaluierung des Konzeptes zeigt, dass übliche statische Analysen mit dem vorgeschlagenen System ausgedrückt werden können und weist darauf hin, dass das vorgeschlagene System verglichen zu traditionellen Ansätzen einfacher ist.

Contents

1. Introduction

Writing "good" programs is hard. It requires programmers not only to write correct programs (that do what they're supposed to do), but also to think about other aspects like documentation, writing idiomatic code, code complexity, formatting and so on. Even though these other aspects don't necessarily affect the quality of the resulting executable program, they can have a significant impact on the readability and maintainability of the source code.

Theoretically, all issues in a program could be revealed during code review performed by other programmers, but in practice, the effectiveness of this technique highly depends on the reviewers' skills and dedication towards the review task. Also, because it's a manual technique, it doesn't scale very well and might be too expensive or time-consuming to perform regularly. Having Humans perform similar tasks regularly also leads to careless mistakes.

An alternative to manual code reviews are programs that analyze source code and run checks for potential issues. A large subset of these programs analyze code without actually executing it and are knows as static analysis tools or linters (named after one of the first static analysis tools called `lint` [8]).

Static analysis tools

Static analysis tools can automatically find and report issues in source code. They can be run frequently in a project or be used to analyze an existing codebase. If new issues are identified, static analysis tools can be updated and used to check all existing code for occurrences of that issue. Static analysis tools can theoretically automate a lot of the work that's currently done in manual code review and could therefore improve code quality while reducing development cost at the same time.

Static analysis tools differ in the kinds of analyses they perform, ranging from simple style checking to complex type checking or program verification analyses [1]. In the following, style checking and bug finding static analyses are presented in more detail.

Style checking static analyses detect issues that might not affect a program's runtime correctness, but rather the program's readability and maintainability. This includes issues related whitespace, naming, commenting and program structure. [1]

Many programming language communities and organizations have realized that code style has a significant impact on code readability / maintainability and have creates so-called "style guides" that describe how programs in that language should be written. Examples of popular style guides include Python's PEP8 [19], Google's C++ Style Guide [6] and many more. Style checking static analyses can be used to automate most of the rules expressed in style guides.

Automatic code formatters are tools that apply a consistent formatting (whitespace, indentation, line lengths, use of parenthesis) to source code. Automatic formatting tools are available for a lot of popular programming languages and newer languages increasingly make using them part of their "best practices". More and more projects and users use them frequently (e.g. before committing code) to ensure consistent formatting across their projects. Automatic code formatters perform style checking static analyses, but the rules they enforce are usually very conservative. Automatic code formatters can nowadays enforce low-level stylistic rules like where to use whitespace reliably.

Another category of static analyses are bug finding static analyses. Bug finding static analyses search for patterns in source code that might indicate bugs [1]. An example of such a pattern could be an if statement that has a condition that can never be true (e.g. `if(false){...}`). It is unlikely that this is intended behavior and therefore most likely a bug that should be reported to the user. Another example for a pattern could be calling a pure function (a function without side-effects) without using its return value.

In the following, this thesis focuses on static analysis tools that use the following types of static analyses:

- **"High-level" style checking**: While automatic code formatting tools handle low-level styling (whitespace, line lengths, …) reasonably well, they typically can't improve code style in more advanced ways. Checking style on a higher level could detect unidiomatic code (e.g. manually implementing a for loop using a while loop and a variable) and recommend to replace that code with a better solution. High-level code style checking could even detect larger program fragments (like manually implementing binary search) and recommend using a well-tested library instead.
- **Bug finding**: Even though compilers might already enforce certain correctness properties in source code (e.g. type-checking), there are a lot of possible bugs they can't detect (e.g. `x == x`). Bug finding static analyses can also contain rules that detect higher-level logical errors in source code that is correct from the compiler's point of view.

The main task of these kinds of static analyses is finding code snippets that have certain properties. Examples of such properties could be

- an addition that has no effect (e.g. `x + 0`),
- a function that has a parameter that isn't used in the function body or
- an if statement which has a condition that's always true (e.g. `if(true){...}`).

The difficulty of finding such code snippets depends on the program representation the search is performed on.

Program Representations

Source code is usually stored in plain text files (a flat sequence of characters). While this simple representation makes storing and editing source code straight-forward, it isn't suitable for most static analyses. Plain text source code contains formatting information (which isn't of interest for the static analyses this thesis focuses on) and a lot of structure (e.g. function scopes, operator precedence, …) is encoded implicitly. Because programs are inherently hierarchical, the semantics of a program are more naturally expressed by tree data structures. For example, a class of a programming language could be represented as an object that has an attribute `method` which contains a list of method objects. Compared to plain text source code, properties like the number of methods of a class would be easier to retrieve from a structure like that.

Compilers and interpreters typically use (often multiple of) such data structures while preparing a program for execution. These data structures are known as Intermediate Representations (IRs). Examples of IRs include parse trees and Abstract Syntax Trees (ASTs).

Because most static analyses analyze the semantics of code rather than it's encoding, they work on IRs instead of the plain source text.

1.1. Problem Statement

A problem in writing static analyses is that it requires writing nested matching code that becomes complex even for simple analyses. The example below shows a simplified version of the "collapsible_if" static analysis which is part of Rust's static analysis tool Clippy [17].

The code above matches if expressions that contain only another if expression (where both ifs don't have an else branch). Even though the purpose of this static analysis can easily be explained, its difficult to tell that purpose from the code. The actual nesting can be reduced using macros (e.g. `if_chain` [21]), this doesn't reduce the inherent complexity of the code.

Specifying static analyses this way is limiting in several ways. Because static analyses are difficult to write, a lot of programmers won't be able to write them themselves. This results in less static analyses being written and therefore a smaller scope and lower effectiveness of static analysis tools. Additionally, the factor that writing static analyses is difficult also limits the complexity of static analyses. If writing static analyses was easier, writing more

```
if let ast::ExprKind::If(check, then, None) = &expr.node {
    if then.stmts.len() == 1 {
        if let ast::StmtKind::Expr(inner)
               | ast::StmtKind::Semi(inner) = &then.stmts[0].node {
            if let ast::ExprKind::If(check2, then2, None) = &inner.node {
                ...
            }
        }
    }
}
```

Figure 1.1.: Simplified version of Clippy's [17] "collapsible_if" static analysis.

advanced static analyses would become possible. Reading static analyses written this way is difficult as well which makes them more likely to contain bugs.

Another problem in writing static analyses that way is that analysis implementations are language- and implementation-dependent. They are tightly coupled to a specific IR and changes to that IR can break a lot of static analyses. Additionally, static analyses cannot easily be ported to another IR that describes the same language.

The problems this thesis aims to solve are that

- Writing static analyses is difficult and repetitive
- Static analyses are language- and implementation-dependent

1.2. Approach

A lot of complexity in writing static analyses comes from having to manually implement the matching logic (see figure 1.1). It's an imperative style that describes *how* to match a syntax tree node instead of specifying *what* should be matched against declaratively. In other areas, it's common to use declarative patterns to describe desired information and let the implementation do the actual matching. A well-known example of this approach are Regular Expressions (REs) [5] which are used to find certain character sequences within strings. Instead of writing code that detects certain character sequences, one can describe a search pattern using a Domain-Specific Language (DSL) [20] and search for matches using that pattern. The advantage of using a declarative DSL is that its limited domain (e.g. matching character sequences in the case of regular expressions) allows expressing entities in that domain in a very natural and expressive way.

While regular expressions are very useful when searching for patterns in flat character sequences, they cannot easily be applied to hierarchical data structures like syntax trees.

The goal of this thesis is to design a DSL for specifying static analyses that operate on IRs. The solution should have the following properties:

- **Declarative**: Static analyses should be specified in terms of *what* to find instead of *how* to find it.
- **Language / Implementation independent**: It should be possible to apply the solution to different programming languages and different IRs that represent programs in these programming language.
- **Extensible**: Since the DSL itself shouldn't be able to handle all use cases itself, it should be possible to handle these cases using a general-purpose programming languages.
- **Composible**: The solution should provide mechanisms to reuse parts of specifications of static analyses in other static analyses.
- **Usable**: The solution should be as simple as possible and focus on supporting common cases instead of all possible ones. It should also be easy to learn the DSL (e.g. by re-using concepts known from other programming languages) and reading and writing patterns in it should be easy as well.

1.3. Related Work

Static analysis tools exist for a number of different programming languages. Examples include Pylint [11] for Python, Clippy [17] for Rust and Lint [8] / Cppcheck [13] for C(++) code. These tools usually specify static analyses directly against the respective AST data structure (like shown previously in section 1.1) and are therefore both language- and implementation specific.

The Rerast project [7] is a tool that allows specifying AST-based code transformation rules for Rust programs. While the motivation of this project is different than that of static analysis tools, the required task remains similar. Rerast allows specifying search patterns that are Rust code snippets that may contain additional placeholders. The implementation provides a way to determine whether a search pattern matches some part of a code base. While using and extending Rust language syntax has the advantage of being familiar to Rust programmers, it also means that the concept cannot easily be applied to other languages.

Another related project is Coccinelle [10] which is a C program transformation tool used within the Linux kernel development process. Similar to Rerast, Coccinelle is language-specific to a single programming language (C in this case).

1.4. Outline of the Thesis

The rest of this thesis is structured in the following way: chapter two describes the concept of the proposed solution. Following the conceptual description, chapter three presents an implementation of the concept. The fourth chapter shows an evaluation of the concept using the implementation presented before. The fifth and last chapter draws a conclusion and presents possible future research opportunities.

2. Concept

2.1. Overview

Figure 2.1 shows an overview of the proposed concept. The central element is the *Pattern*. A *Pattern* contains the information what to look for (e.g. “an addition that has no effect”) and is specified in a syntax (*Pattern Syntax*) inspired by RE. The *Pattern Syntax* is language-independent which means that it can represent syntax tree patterns for different programming languages.

Patterns may use so-called *Pattern Functions*. A *Pattern Function* can take *Pattern Syntax* as arguments and also expands to *Pattern Syntax*. *Pattern Functions* make composition of patterns possible. Using them, redundant parts of a pattern can be replaced with function calls that expand to the desired sub-patterns. *Pattern Functions* are defined in *Pattern Function Syntax*.

A *Pattern* is defined in terms of a *Pattern Tree*. A *Pattern Tree* is a data structure that defines the set of valid patterns for a certain language. *Pattern Trees* are language-dependent (a *Pattern Tree* for the C++ programming language looks different than one for Python) and closely related to the AST of the language they’re designed for. Because a *Pattern Tree* defines the set of possible patterns, a single *Pattern* can be seen as an instance of a certain

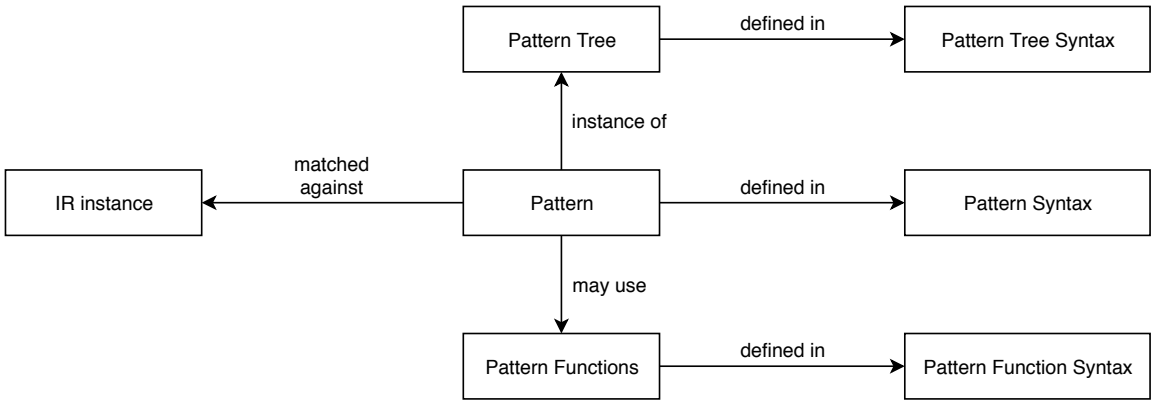

Figure 2.1.: Concept overview

Pattern Tree. *Pattern Trees* are defined in a custom syntax (*Pattern Tree Syntax*) similar to grammar definitions like (E)BNF.

Patterns can be matched against *IR instances* (e.g. ASTs, Parse Trees). How to match *IR instances* against *Patterns* created from a certain *Pattern Tree* needs to be implemented manually. The concept contains general matching implementations (like matching sequences or alternatives of elements) which implementations can build upon. The same *Pattern* can be matched against different *IR instances* if matching implementations are provided for those IRs.

This chapter is structured in the following way. The sections 2.2, 2.3 and 2.4 give an overview of the Pattern, Pattern Function and Pattern Tree concepts. Section 2.5 presents the compilation process of patterns and explains the semantics of the different concepts and their relationships in more detail. Section 2.6 describes how patterns can be matched against IR instances.

2.2. Pattern

The following code snippet shows a simple example of a pattern:

```
my_pattern: Expr =
    Lit(Bool(false))
```

Each pattern consists of three parts, a name, a type and the pattern's body. The name identifies the pattern and is used to match it against syntax tree nodes. The type specifies the top-level node within the pattern. The pattern's body specifies which syntax tree nodes the pattern matches. The syntax for these three parts is `<name>: <type> = <body>`. In the example above, the pattern's name is `my_pattern`, it's type is `Expr` and it's body is `Lit(Bool(false))`.

The name of a pattern can be any valid identifier in the implementation language. By convention, patterns use lower-case identifiers.

A pattern's type can either be a single identifier (e.g. `Expr`) or an identifier wrapped in a repetition type (e.g. `Seq<Expr>`). Valid repetition types are `Seq<<inner_type>>` for any number of occurrences of `inner_type` and `Opt<<inner_type>>` for an optional occurrence of `inner_type`.

The following sections explain the different parts of the pattern syntax that can occur in the pattern's body by example. A table summarizing the pattern syntax is shown in section 2.2.10. A formal definition of the pattern syntax is given in the appendix (section A.2.1).

2.2.1. Any

The simplest pattern is the *any* pattern. It matches a single occurrence of any node and is therefore similar to RE's `.` syntax.

The following example matches any expression:

```
my_pattern: Expr =
    _
```

While the *any* pattern doesn't make much sense on its own, it is useful in larger patterns to denote that a part isn't relevant and should be ignored.

2.2.2. Node

Nodes are used to match a specific variant of an AST node. A node has a name and a number of arguments that depends on the node type. Node names are upper-case by convention to make them distinct from function calls (see section 2.2.3).

The following pattern matches any expression that is a literal:

```
my_pattern: Expr =
    Lit(_)
```

Nodes can contain other nodes as their arguments. The following pattern matches any expression that is a boolean literal:

```
my_pattern: Expr =
    Lit(Bool(_))
```

Nodes can have multiple arguments and nodes that don't have any arguments are written without parentheses. The pattern below matches integer literals that don't have a suffix (an integer with suffix could be `1000i32` instead of `1000`).

```
my_pattern: Lit =
    Int(_, Unsuffixed)
```

2.2.3. Function call

Function calls are similar to nodes. They also have a name and a number of arguments. The syntactic difference is that function names are lower-case by convention while nodes use upper-case names.

The following pattern uses a function `any_order(<a>, <b>)`. In this case, the function could be defined to match both the sequence `<a> <b>` as well as `<b> <a>`. The definition of functions is described in section 2.3.

```
my_pattern: Seq<Lit> =
    any_order( Bool(_), Int(_, _) )
```

Function calls can also appear as arguments of nodes:

```
my_pattern: Expr =
    Array( any_order( Bool(_), _) )
```

2.2.4. Literal

A pattern can also contain literals. The exact types and syntaxes of literals can be defined by implementations as long as they're not conflicting with other syntactic elements. Common literals include floating-point (`1.234`) and literal (`123`) numbers, characters (`'c'`), strings (`"string"`) and booleans (`true` and `false`).

The following pattern matches the boolean literal `false`:

```
my_pattern: Expr =
    Lit(Bool(false))
```

The pattern below matches the character literal `x`:

```
my_pattern: Expr =
    Lit(Char('x'))
```

2.2.5. Alternative

Patterns can also contain alternatives. Similar to REs, alternatives are separated by a pipe character (`|`).

The following pattern matches boolean and integer literals:

```
my_pattern: Lit =
    Bool(_) | Int(_)
```

Alternatives can appear in node arguments as well. If there are more than two alternatives, more cases can easily be added by separating them with pipe characters. The pattern below matches character literals with a value of `x`, `y` or `z`.

```
my_pattern: Expr =
    Lit( Char('x' | 'y' | 'z') )
```

2.2.6. Empty

The empty pattern `()` can be used in places where a sequence of elements or an optional element is expected. In case of a sequence, it represents the empty sequence. In case of an optional element, it expresses that there's no element.

The pattern below matches an empty array:

```
my_pattern: Expr =
    Array( () )
```

The following pattern matches if expressions that don't have an else clause:

```
my_pattern: Expr =
    If( _, _, () )
```

2.2.7. Sequence

Sequences of elements can be expressed by simply writing them after each other.

The following example matches the array `[true, false]`:

```
my_pattern: Expr =
    Array( Lit(Bool(true)) Lit(Bool(false)) )
```

Sequence elements can optionally be delimited using semicolons (`;`) to improve readability. Trailing semicolons aren't allowed. The pattern below is equivalent to the previous example.

```
my_pattern: Expr =
    Array(
        Lit(Bool(true));
        Lit(Bool(false))
    )
```

2.2.8. Repetition

Elements my be repeated. The syntax for specifying repetitions is similar to REs. The following table shows the supported syntax:

Syntax	Meaning
x*	zero or more of x
x+	one or more of x
x?	zero or one of x
x{n,m}	at least n x and at most m x
x{n,}	at least n x
x{n}	exactly n x

The pattern below matches arrays that contain two xs as their last or second-last elements:

```
my_pattern: Expr =
    Array( _* Lit(Char('x')){2} _? )
```

For example, this pattern would match `['x', 'x']`, `['x', 'x', 'y']` and `['a', 'b', 'c', 'x', 'x', 'y']` but not `['x', 'x', 'y', 'z']`.

One important aspect is that the repetition of elements defaults to one if not specified otherwise. For example, the pattern below matches all arrays that contain **a single** element:

```
my_pattern: Expr =
    Array( _ )
```

A pattern that matches arrays of any length is written like this:

```
my_pattern: Expr =
    Array( _* )
```

Similarly, the following pattern matches if expressions that **may or may not** have an else block. Using "`_?`" is equivalent to the pattern "`_ | ()`".

```
my_pattern: Expr =
    If( _, _, _?)
```

The table below shows how optional elements are matched:

Pattern	if with else block	if witout else block
`If(_, _, _)`	match	no match
`If(_, _, _?)`	match	match
`If(_, _, ())`	no match	match

2.2.9. Named submatch

A lot of static analyses require checks that go beyond what the pattern syntax described in the previous sections can express. For example, a static analysis might want to check for variables with upper-case names or functions whose names are too long. Another example would be a static analysis that wants to match two nodes that have the same value (as needed by static analyses that check for shadowed variables).

Instead of allowing users to write these checks into the pattern directly (which would make the pattern's syntax more complex and patterns harder to read), the solution allows to assign names to parts of a pattern expression. A sub-pattern can be named by appending `#<name>` to it (where `<name>` is an identifier).

The following pattern matches character literals and gives the character literal the name "foo":

```
my_pattern: Expr =
    Lit( Char( _#foo ) )
```

Names can be given to different parts of a pattern. The following pattern gives the character literal, the literal and the expression part of the pattern the names "foo", "bar" and "baz" respectively.

```
my_pattern: Expr =
Lit( Char( _#foo )#bar )#baz
```

Named submatches can be used with repetitions. In this case the repetition is written before the named submatch. The following pattern matches arrays of length 5 and assigns the name "bar" to the array's elements.

```
my_pattern: Expr =
    Array( _{5}#bar )
```

Named submatches can be used to get references to parts of an IR instance after it has been matched (similar to IR's named capturing groups). This feature is described in more detail in section 2.5.3.

2.2.10. Summary

The following table shows a summary of the pattern syntax:

Syntax	Concept	Examples
`_`	Any	`_`
`<node-name>(<args>)`	Node	`Lit(Bool(true))`, `If(_, _, _)`
`<func-name>(<args>)`	Function	`any_order(_, Bool(_))`
`<lit>`	Literal	`'x'`, `false`, `101`
`<a> | <b>`	Alternative	`Char(_) | Bool(_)`
`()`	Empty	`Array( () )`
`<a> <b>`	Sequence	`Tuple( Lit(Bool(_)) Lit(Int(_)) Lit(_) )`
`<a>*`	Repetition	`Array( _* )`,
`<a>+`		`Block( Semi(_)+ )`,
`<a>?`		`If(_, _, Block(_)?)`,
`<a>{n}`		`Array( Lit(_){10} )`,
`<a>{n,m}`		`Lit(_){5,10}`,
`<a>{n,}`		`Lit(Bool(_)){10,}`
`<a>#<name>`	Named sub-match	`Lit(Int(_))#foo`, `Lit(Int(_#bar))`

2.3. Pattern Functions

For complex patterns, it's likely that similar subpatterns are used in different places. These repetitions can make patterns difficult to read and maintain.

In the pattern shown below, there are two `Block_(...)` subpatterns that are identical. Additionally, the subpattern `(If(_, _, _?) | IfLet(_, _?))#else_block` is repeated twice within each of these blocks.

```
pat_if_else: Expr =
    If(
        _,
        _,
        Block_(
            Block(
                Expr( (If(_, _, _?) | IfLet(_, _?))#else_block ) |
                Semi( (If(_, _, _?) | IfLet(_, _?))#else_block )
            )#block_inner
        )#block
    ) |
    IfLet(
        _,
        Block_(
            Block(
                Expr( (If(_, _, _?) | IfLet(_, _?))#else_block ) |
```

```
                Semi( (If(_, _, _?) | IfLet(_, _?))#else_block )
            )#block_inner
        )#block
    )
```

To avoid these kinds of repetitions, the concept includes so-called Pattern Functions. Pattern Functions can take any number of arguments and expand to a pattern. By defining and using two functions `expr_or_semi` and `if_or_if_let`, the pattern above can be simplified in the following way:

```
pat_if_else: Expr =
    if_or_if_let(
        _,
        Block_(
            Block(
                expr_or_semi( if_or_if_let(_, _?)#else_block )
            )#block_inner
        )#block
    )
```

The definitions of the functions used within the pattern above are shown below:

```
fn expr_or_semi($expr) {
    Expr($expr) | Semi($expr)
}

fn if_or_if_let($then, $else_block) {
    If(_, $then, $else_block) | IfLet($then, $else_block)
}
```

The syntax of Pattern Functions is similar to function syntax in c-like programming languages. An informal description of the syntax is given below:

```
fn <function_name>(<args>) {
    <body>
}
```

A pattern function consists of the function's name, a list of arguments and the function body. Function names are lower-case by convention. Function arguments are separated by comma characters and each argument is an identifier prefixed by a dollar character. The body of a Pattern Function can contain any characters.

Calls to Pattern Functions are expanded in a preprocessing step. A Pattern Function expands to the characters of it's body where all occurrences of parameters in the body are replaced by the corresponding arguments passed to the function call. This is described in more detail in section 2.5.1.

2.4. Pattern Tree

A Pattern Tree is a data structure similar to an IR that is designed to represent search patterns. The main difference between a Pattern Tree and an IR is that while an instance of an IR represents *one specific* program, an instance of a Pattern Tree may represent *a class of* programs.

For example, a pattern could match "if statements that have a boolean literal as their condition". For this pattern, it doesn't matter whether the condition is `true` or `false`, whether or not the if statement has an else branch or what statements the if statement's body contains. An IR instance would have to specify all values explicitly and couldn't express that multiple alternatives are possible.

A Pattern Tree consists of a set of node definitions. Each node definition can have multiple variants that each can have multiple arguments. For example, the following code shows an example of a node definition:

```
Lit = Bool(bool)
    | Int(u128, LitIntType)
```

The code above defines a node `Lit` representing literals that has two variants. It can either be a boolean literal (`Bool`) or an integer literal (`Int`). The `Bool` variant has a single parameter which is a boolean primitive type, the `Int` variant has two parameters (a 128-bit unsigned integer primitive type and a `LitIntType` type).

The syntax is described in more detail below. A formal specification of the syntax can be found in the appendix (section A.2.2).

2.4.1. Node definition

```
<node_name> = <variants>
```

A node definition starts with the node's name, followed by an equals sign (`=`). Node names are upper-case by convention. After the equals sign, a node definition may contain multiple variants, delimited by pipe characters (`|`).

2.4.2. Node variant

```
<variant_name> or <variant_name>(<arguments>)
```

A node variant has a name and optionally a list of arguments written in parentheses. Multiple arguments are delimited by comma characters (`,`). Variants that don't have any arguments are written without parentheses. Variant names are CamelCase by convention.

2.4.3. Node argument

A node argument is an identifier that may be followed by a repetition character and/or a custom associated type. In the simplest case, a node argument is a single identifier that specifies a type. A type can either be a primitive type provided by the implementation (e.g. `bool`, `int`, `char`, ...) or another type defined in the same pattern tree.

The following node definition uses the `char` primitive type of the implementation language in its `Char` variant:

```
Literal = Char(char)
```

The node definition below uses the `Literal` node type defined above:

```
Expr = Lit(Literal)
```

Type definitions can be recursive. The following example shows a variant `BinOp` that represents binary operations and has two arguments that use the `Expr` type recursively:

```
Expr = BinOp(OpType, Expr, Expr)
```

Repetitions

By default, a node argument expects exactly one element. To allow multiple occurrences of elements, the syntax allows appending a question mark character (`?`) or a star character (`*`) to a node argument. A question mark character expresses that the argument is optional, a star character that it's a list of elements.

For example, the following node definition defines an `Array` variant that can have any number of `Exprs` as it's argument:

```
Expr = Array(Expr*)
```

The example below shows an `If` variant that has three arguments for the if expressions's condition, body and else clause. The condition is a single `Expr` node, the body a list of `Expr` nodes and the else clause an optional `Expr` node. This `If` variant is a constructed example that assumes that the else branch may only contain a single `Expr`. In a more realistic implementation, one would use blocks instead of expressions in an if's "then"

and ”else” branch. For the purpose of demonstrating the different repetition types, the definition below is useful though and will be used in various places in this thesis.

```
Expr = If(Expr, Expr*, Expr?)
```

Custom associated types

A node argument that uses a primitive type may specify a custom associated type for it. This is done by appending angle brackets (<>) and the custom associated type's name to argument's type. Custom associated type names are upper-case by convention.

The following example shows a node definition that uses a custom associated type `Symbol` for the argument of the primitive type `str`:

```
Lit = Str(str<>Symbol)
```

Custom associated types are sometimes needed to correctly implement matching. For more details, see section 2.5.3.1.

2.5. Compilation

In order to use a pattern, it needs to be compiled. More specifically, the compilation process takes textual input of a pattern, a pattern tree and possibly multiple pattern functions as input, performs various transformations and checks and emits code that can be executed.

Figure 2.2 shows the compilation pipeline. It consists of the following four steps:

- **Parsing**: Parses the textual input of the pattern, pattern tree and pattern functions into ASTs.
- **Type checking**: Performs semantic analysis to ensure that types used within the pattern are correct.
- **Result type type inference**: Determines the types of elements that should be contained in a matching result structure.

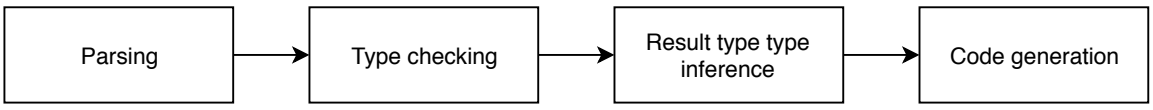

Figure 2.2.: Compilation pipeline

- **Code generation**: Uses the information acquired in the previous steps to generate executable code in a target language.

The following sections describe the compilation steps in more detail.

2.5.1. Parsing

In the first compilation step, the textual inputs of the pattern tree and the pattern functions are parsed into AST data structures separately. The exact construction of the parsers is up to implementations, but since the grammars for the three languages are relatively simple, writing parsers for them should be straight-forward. If any of the inputs don't conform to the specified grammars, the compilation should be aborted.

In a second step, the pattern itself is parsed. Pattern function calls within the pattern are expanded before the actual parsing is performed. The function expansion is described in the next section.

Function Expansion

Usages of pattern functions within the pattern are expanded by scanning the pattern's source text for occurrences of pattern function names and expanding occurrences one after the other.

A pattern function expands to the characters of it's body where all occurrences of parameters in the body are replaced by the corresponding arguments passed to the function call. If a pattern function is called with a wrong number of arguments, the compilation is aborted.

The following example shows a pattern function `any_order` and a pattern `my_pattern` that uses that pattern function:

```
fn any_order($a, $b){
    ($a $b) | ($b $a)
}

my_pattern: Expr =
    Array(
        any_order( Lit(Bool(true)), Lit(Bool(false)) )
    )
```

The preprocessing step will detect the function call in the pattern and expand to the following pattern (reformatted for better readability):

```
my_pattern: Expr =
    Array(
        ( Lit(Bool(true)) Lit(Bool(false)) ) |
        ( Lit(Bool(false)) Lit(Bool(true)) )
    )
```

After each expansion step, the pattern is checked for remaining pattern function calls. The pattern keeps expanding until it doesn't contain pattern function calls anymore. This allows writing pattern functions that expand to other pattern function calls as long as they don't form cycles.

The following example shows an example of a cycle where the pattern function `infinite` expands to itself:

```
fn infinite($a){
    infinite($a)
}
```

Implementations are expected to handle cycles by limiting the number of allowed expansion steps. If the number of allowed expansions is exceeded, the compilation should be aborted.

After all pattern function calls are expanded, the pattern is parsed into an AST. Like for the other parsers, the exact construction is up to implementations. Syntactic errors in the pattern should abort the compilation.

If parsing is successful, the pattern's and pattern tree's AST are passed to the next compilation step. Since function calls within the pattern are expanded already, the function definitions aren't needed for further analysis.

2.5.2. Semantic analysis

While the parsing step already ensured that the pattern and pattern tree definitions are syntactically correct, the purpose of this compilation step is to ensure semantic correctness.

More specifically, the following checks are performed:

- **Node / literal type checks**: A node name within the pattern needs to be a variant of the type specified in the enclosing element.
- **Repetition checks**: Depending on the repetition type of the enclosing node, only certain syntactic features are valid.

These checks are described in the following sections.

2.5.2.1. Node / literal type checks

Only certain node names are allowed in each part of a pattern. The set of valid node names always depends on the expected type. For the root level, this type is specified in the pattern definition. For nested nodes, the expected type is determined by the pattern tree definition of the enclosing node. Types need to be either names of pattern tree definitions (e.g. `Expr` and `Lit`) or names of primitive types the implementation provides. Node names in the pattern need to be pattern tree variant names (e.g. `Array` or `Char`).

For example, consider the following pattern tree:

```
Expr = Lit(Lit)
     | Array(Expr*)
     | If(Expr, Expr*, Expr?)

Lit = Char(char)
    | Bool(bool)
```

The patterns below have an expected type `Expr`, so nodes in their bodies need to be variants of the pattern tree's `Expr` node definition.

```
my_pattern: Expr =
    Lit(_) | Array(_)  // valid pattern

my_pattern_2: Expr =
    Char(_)            // error: no variant `Char` in `Expr` definition
```

If the pattern's type (`Expr` in the example above) is not one of the pattern tree definition names, the compilation is aborted.

Nodes in a pattern definition need to be used with the correct number of arguments (as defined in the pattern tree). The compilation is aborted otherwise.

```
my_pattern: Expr =
    Lit(_, _)         // error: `Lit` expects 1 argument, 2 were given
```

The expected type for nested nodes depends on the enclosing nodes' pattern tree definition. The following pattern contains a `Lit(...)` node which is defined as having a single parameter of type `Lit` (see pattern tree definition above). The argument of the `Lit(...)` node is therefore expected to contain only nodes that name a variant of the `Lit` pattern tree definition (`Char` or `Bool`).

```
my_pattern: Expr =
    Lit(Char(_))     // valid pattern
```

```
my_pattern_2: Expr =
    Lit(Array(_))    // error: no variant `Array` in `Lit` definition
```

If the expected type is a primitive type, literals of that type may be used in the pattern. Nodes cannot be used when a primitive type is expected. The exact syntax of literals is up to implementations. The example below uses single quotes to express a character literal.

```
my_pattern: Expr =
    Lit(Char('c'))    // valid pattern
```

2.5.2.2. Repetition type checks

Expected types have one of three possible repetition types. The default repetition is "single" (e.g. `Expr`). The other two repetition types are "optional" (`Opt<type>` in the pattern's type and `type?` in pattern tree definitions) and "sequence" (`Seq<type>` in the pattern's type and `type*` in pattern tree definitions).

Depending on the expected repetition type, only a subset of syntax may be used in a pattern. The following table shows which syntactic concepts can be used depending on the repetition type.

Syntax	Single	Optional	Sequence
Any (`_`)	x	x	x
Node	(x)	(x)	(x)
Literal	(x)	(x)	(x)
Alternative (`<a> | <b>`)	x	x	x
Named submatch (`<a>#<name>`)	x	x	x
Empty (`()`)	-	x	x
Sequence (`<a> <b>`)	-	-	x
Repetition (e.g. `<a>*`)	-	-	x

Any, alternatives and named submatches can always be used. Whether or not nodes and literals may be used depends on the expected type (see the previous section) and is independent of the repetition type.

The empty syntax (`()`) may only be used when the expected repetition type is "optional" or "sequence".

```
my_pattern: Expr =
    If(_, (), ())    // valid pattern

my_pattern_2: Expr =
    Lit( () )        // error: Expected single element, found "empty"
```

Sequences and repetitions may only be used when the expected repetition type is “sequence”.

```
my_pattern: Expr =
    Array( Lit(_){3,5} Array(_*) )    // valid pattern

my_pattern_2: Expr =
    Lit( Bool(_)* )    // error: Expected single element, found "repetition"
```

2.5.3. Result Type Type Inference

Matching a pattern against an IR instance produces a match result. A match result contains whether the pattern matched the IR instance and if it did, it also contains references to parts of the IR instance that were named in the pattern (see section 2.2.9).

For example, if the pattern `Array(_#var1 Lit(Bool(_#var2)))` was matched against an AST node representing a two-element array containing true and false (`[true, false]`), the match would be successful and return a structure containing two attributes (`var1` and `var2`). The `var1` attribute would be a reference to the AST expression node that represents the first array element. The `var2` attribute would be a reference to a boolean node that represents the second element.

The references in the result type are expected to be implemented as a data structure that maps identifiers to references. In statically typed languages, one would use struct- or class-like constructs for this. Because the number and type of references depends on the named submatches used in a pattern, they need to be determined during compilation. This type inference of the result type is described in the following.

2.5.3.1. Match associated types

The type of each element in the result structure is a reference to an IR node. Named submatches are used within a pattern and name pattern tree nodes. Therefore, it’s necessary to provide a mapping from pattern tree node types to IR node types. This mapping needs to be provided for each IR that should be matched against a pattern. A mapping needs to provide an IR type for each type of the pattern tree. Primitive types are assumed to be represented as themselves in the IR. For example, if a pattern tree contains a `bool` type, it is assumed that the IR type this `bool` is matched against is also a `bool`. In addition to pattern tree types, the mapping also needs to provide IR type mappings for all custom associated types (see 2.4.3). These custom associated types allow to express that a primitive type should be matched against another type. For example, this is needed when an IR uses string interning as a performance optimization.

2.5.3.2. Node type

Because the same pattern can be matched against different IRs (e.g. against a languages' parse- and abstract syntax tree), the exact types in the result structure can't be known while compiling the pattern. Instead, it is expected that any IR that is matched against the pattern provides a type mapping (see previous section). Using this assumption, it's sufficient to express the types in the result structure in terms of the pattern tree types. For each usage of the pattern, these types and the type mapping of the IR used allow to easily determine the IR node types.

The type of a named submatch is determined "top-down". Initially the type of the pattern is used. If a named submatch is used within a node, it's type is determined by the enclosing node definition.

For example, consider the following pattern tree:

```
Expr = Lit(Lit)
     | ...

Lit = Bool(bool)
    | ...
```

The following pattern uses three named submatches `#foo`, `#bar` and `#baz`:

```
my_pattern: Expr =
    Lit( Bool(_#foo)#bar ) | _#baz
```

The named submatches `#foo` and `#bar` appear within node definitions, the named submatch `#baz` on the root level. The type of `#baz` is therefore the type of the pattern (`Expr` in this case) and the types of the other named submatches are determined by the definitions of their enclosing nodes. In this example, `#foo` would have the primitive type `bool` and `#bar` the type `Lit`.

2.5.3.3. Multiple usages of the same named submatch

In some cases, it can make sense to use the same name in different parts of a pattern.

For example, the following pattern matches literals and arrays that contain a single literal:

```
my_pattern: Expr =
    Lit(_)#var | Array( Lit(_)#var )
```

Both the plain literal and the array containing the literal are named `var`. If the pattern matches, the result will contain an element named var which is a reference to a literal

IR node. Whether or not it was enclosed in an array doesn't matter. If it did, the pattern could have used different names for the two literals.

The same name may be used multiple times if the named elements have the same node type. In the example above, both named entries are literals. If two named submatches use the same name but name different types, the compilation is aborted.

The following pattern shows such an invalid pattern:

```
my_pattern: Expr =
    Lit( _#var ) | _#var  // error: multiple usages of `#var`
                          //        need to have the same type
```

The first occurrence of `#var` is of type `Lit` while the second is of type `Expr`.

2.5.3.4. Repetition type

In addition to node types, elements of a result structure may also have a repetition type. There are three repetition types: "single", "optional" and "sequence". "Optional" denotes that there may or may not be an element and "sequence" denotes that there may be any number of elements.

To obtain these repetition types, one could simply use the repetition types used in the pattern's type and the pattern tree definition's types. The problem with this would be that types in the pattern tree describe what's possible where types in the result structure should describe what's actually used.

For example, consider the following pattern that matches arrays that have only one element:

```
my_pattern: Expr =
    Array( _#var )
```

The `Array` variant is defined as `Array(Expr*)` in the pattern tree, so the approach described above would assign a "sequence" repetition type to the `var` element of the result structure. Because of the actual pattern within the array node, it can be known at compile time that there will only be a single element and the repetition type should therefore be "single" instead.

As another example, the following pattern matches arrays that have at least one element:

```
my_pattern: Expr =
    Array( _+#var )
```

Syntax	Repetition type
Any (_)	“single”
Node	“single”
Literal	“single”
Empty (())	“optional”
Sequence (<a> <b>)	“sequence”
Named submatch (<a>#<name>)	rep. type of <a>
Repetition (e.g. <a>*)	“optional” / “sequence”
Alternative (<a> | <b>)	most general child rep. type

Table 2.5.: Repetition types of pattern syntactic elements

In this case, the named submatch may indeed contain multiple elements and therefore needs to have the “sequence” repetition type.

The repetition type of a named submatch is determined in two steps. First, the children of each named submatch are analyzed to determine the number of elements that the submatch may contain. In the second step, the pattern is analyzed “bottom-up” to determine how often named submatches may occur.

Step 1

The number of elements matched by a part of a pattern depends on the pattern concept used. The repetition type of “Any”, “Nodes” and “Literals” is “single”. The repetition type of “Empty” is “optional” and the repetition type of “Sequences” is “sequence”. The repetition types of the other pattern concepts depends on their children elements and is described below. A summary is shown in table 2.5.

Named submatch (<a>#<name>)

The repetition type of a named submatch equals the repetition type of its child <a>.

Repetition (e.g. <a>)*

If <a>’s repetition type is “single” or “optional” and the repetition itself is optional (e.g. <a>?), the repetition type is “optional”. Otherwise, it’s “sequence”.

Alternative (<a> | <b>)

The repetition type of an alternative is the more general of its children’s repetition types. If one children’s repetition type is “sequence”, it’s “sequence”. Otherwise, if one children’s repetition type is “optional”, it’ “optional” and “single” otherwise.

Step 2

Repetition (e.g. <a>∗)

The repetition types of named submatches defined in subpattern <a> are adapted depending on the repetition itself. If the repetition itself is optional (e.g. <a>?), all named submatches with repetition type "single" get the repetition type "optional", all others stay the same. If the repetition itself is something else (e.g. <a>∗), all named submatches get the repetition type "sequence".

Sequence (<a> <b>)

The named submatches of <a> and <b> are joined in the following way: all named submatches that only exist in one of <a> and <b> are taken over into the result. Named submatches that exist in both children need to have the same node type (see section 2.5.3.2) and and are taken over into the result using the repetition type "sequence".

Alternative (<a> | <b>)

Similar to sequences, all named submatches that only exist in one child are taken over into the result. The only difference is that elements with repetition type "single" get the repetition type "optional". For named submatches that exist in both children, the repetition type depends on the two named submatches's types. If both types are "single", the resulting repetition type is "single" as well. If one of the types is "sequence", the result is "sequence" as well. Otherwise, the resulting repetition type is "optional".

2.5.4. Code Generation

The last step of the compilation pipeline is the code generation step. The exact code generation highly depends on the target language. It'd be possible to implement the concept as a stand-alone programming language, but this would require re-implementing a lot of functionality. Another option is to implement the concept as an internal / embedded DSL which means that the code generation emits source code in a host programming language.

2.6. Matching

Once a pattern is compiled successfully, it can be matched against IR instances. The following sections describe how matching works in more detail.

As described in section 2.5.3.1, it's necessary to provide a mapping from pattern tree node types to IR node types. For each of these pairs of types, there needs to be specified when instances of these types match. Usually, this involves checking the variant type and then matching all parameters.

For example, let's consider the following pattern tree:

```
Expr = Lit(Lit)
     | Array(Expr*)
     | If(Expr, Expr*, Expr?)

Lit = Char(char)
    | Bool(bool)
```

Matching a `Expr` pattern tree node instance against an IR node instance could first check if both instances use the same variant (e.g. that both represent a literal) and if that's the case match the variant's arguments recursively (e.g. matching the literal pattern tree node contained in the expression node against the literal IR node contained in the expression IR node). Primitive types match by equality.

Besides nodes and primitive types, a pattern may also contain sequences, alternatives and optional elements.

2.6.1. Alternative

When matching an alternative of two elements (`<a> | <b>`), `<a>` is matched first. If it matches, the match is returned. If not, the second alternative (`<b>`) is matched and the result of that match is returned.

2.6.2. Optional

Matching an optional element (`<a>?`) succeeds if the IR node matched against matches either "empty" or `<a>`.

2.6.3. Sequence

Sequences of elements are matched against a sequence of IR instances. Repetitions of elements (e.g. `<a>{1,3}`) match if the number of elements in the sequence of IR instances is within the range specified by the repetition (e.g. between one and three elements) and all elements in the sequence of IR instances individually match the pattern (e.g. `<a>`). Sequences are matched by trying all possible combinations of distributing the IR instances among the pattern's sequence elements. For example, matching the pattern sequence `Bool(true)* Bool(false)*` against a sequence of three IR instances that are boolean true values would check the following distributions:

`Bool(true)*`	`Bool(false)*`
`[]`	`[true, true, true]`
`[true]`	`[true, true]`
`[true, true]`	`[true]`
`[true, true, true]`	`[]`

In the example above, the last combination matches.

2.6.4. Matching ambiguities

Using the concepts described above, it's possible to describe patterns that are ambiguous. For example, when matching the pattern `Array( Lit(_)?#var _* )` against an IR that represents `[1, a, b]`, it's ambiguous whether the literal `1` should be matched as part of the `Lit(_)?` or the `_*` pattern. This difference is significant because it changes which elements will be part of the named submatch `#var`.

In ambiguous patterns, implementations may choose which alternative to use. This choice has no impact on whether a pattern matches or not, but it may change the contents of named submatches.

3. Implementation

As part of this thesis, a proof-of concept implementation was written in and for the Rust programming language [9]. This chapter presents the implementation in detail.

The implementation described in this chapter is available online:

`https://github.com/fkohlgrueber/pattern-matching/tree/thesis`

This chapter is structured as follows: Section 3.1 briefly describes the Rust programming language and why it was chosen for the implementation. Afterwards, section 3.2 gives an architectural overview of the implementation. Sections 3.3 - 3.8 describe important aspects of the implementation in more detail.

3.1. Rust

The concept described in chapter 2 could have been implemented in most general-purpose programming languages. The Rust programming language [9] was chosen for the implementation because some of its features and properties simplify the implementation. These properties will briefly be mentioned below.

Rust's most important feature for implementing the concept are so-called procedural macros [12] which allow writing Rust code that runs at compile-time and can transform any input (e.g. pattern syntax) into any valid Rust code. Because of procedural macros, the concept can entirely be implemented using built-in features of Rust. The advantage of this is that it allows re-using a lot of Rust's functionality. For example, one doesn't have to implement a custom build system and can use tools like Rust's package distribution system `cargo` [16] to share patterns or the Rust compiler itself to detect program errors.

While Rust's procedural macros allow writing compilers within the language, doing so by hand isn't practical. Rust has an ecosystem of packages (called "crates") that make certain parts of the development easier. There's the `syn` [4] and `quote` [3] crates which assist in writing parsers and generating code respectively. Another useful tool is `cargo expand` [2] which can be used to see the code generated by macro invocations. This information is especially useful for debugging procedural macros.

Another useful feature of Rust are so-called algebraic data types. They allow representing syntax trees naturally and Rust's pattern matching makes working with them easy and

expressive. In Rust, these types are called “enums“ but should not be confused with enums known from languages like `C` or `Java`. The main difference is that each variant in a Rust enum can have a number of attributes. [9]

The Rust ecosystem also provides a static analysis tool called `Clippy` [17] that can analyze Rust code and report potential issues. Existing static analysis implementations in Clippy can therefore be used as a baseline to evaluate the advantages the concept provides.

3.1.1. Procedural Macro Hygiene

Due to current limitations in the Rust programming language, procedural macros need to be defined in a “proc-macro” crate. This special kind of crate only allows exporting procedural macros and no other elements like functions or modules.

This becomes a problem when a procedural macro expands to code that uses other elements (e.g. calls a function). Because the procedural macro crate isn’t allowed to export functions, users of the macro are responsible to not only import the procedural macro, but also all other elements the expanded macro might use. This is confusing for users and leads to code that seems to import elements that aren’t used.

A solution to this is to use a second crate. This crate contains all elements the expanded procedural macro might need and depends on the crate that defines the procedural macro. The wrapping crate can then export all it’s elements and re-export the procedural macro. This way, a user only needs to import the wrapping crate without having to know about elements the expanded macro might use.

The implementation described below uses this pattern for all procedural macros it contains. Crates whose names end with “-macro” contain the procedural macros and should not be used directly. For each of these crates, there’s another crate with the same name (but without the “-macro” suffix) that contains additional elements and re-exports the procedural macro. These crates are meant to be used by other programs.

3.2. Project structure

Figure 3.1 shows the crates used by the implementation and the dependencies between them. Pairs of crates that implement the procedural macro pattern described above are marked in the same color. For simplicity, test crates (`test`, `test-clippy` and `pattern-func-lib`) and the `commons` crate are not shown in the figure. The full graph can be found in the appendix (section A.1).

Each crate contains different elements that are needed for the implementation to work properly. The following list gives a short overview over the contents of each crate:

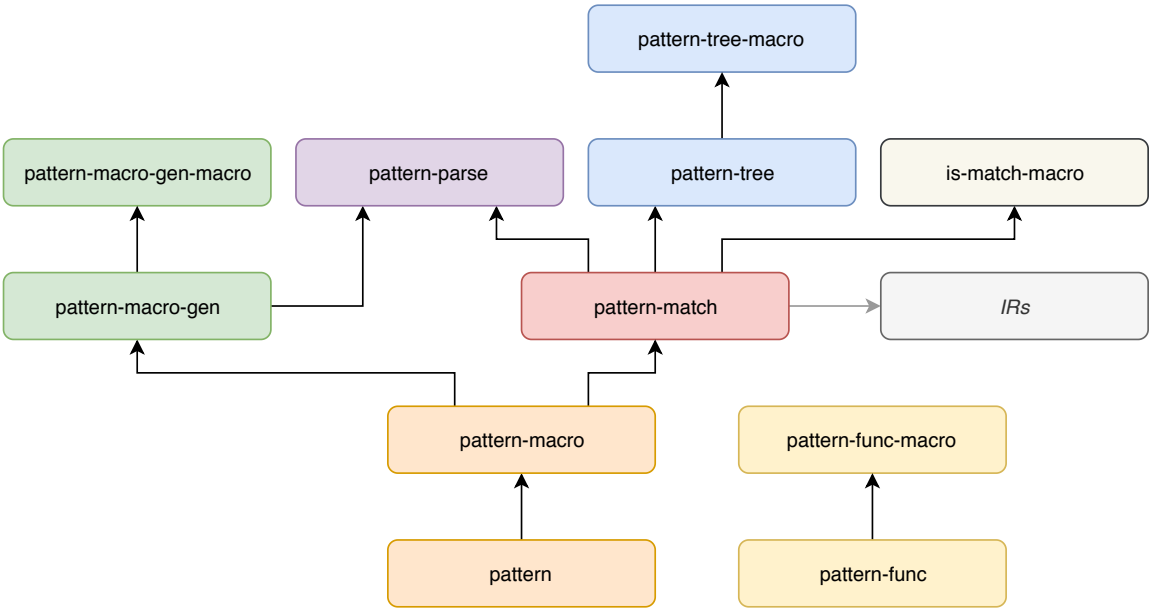

Figure 3.1.: Project Structure

- **pattern-tree-macro**: Defines the `gen_pattern_tree!{...}` macro
- **pattern-tree**: Hygienic wrapper around the `pattern-tree-macro` crate that also contains the definitions of the `Alt`, `Seq`, `Opt` and `RepeatRange` structs
- **pattern-parse**: Contains the parser for the Pattern Syntax
- **pattern-match**: Contains the `IsMatch`, `Reduce` and `ReduceSelf` traits, generic matching implementations for `Alt`, `Seq` and `Opt`, Pattern Trees for Rust and PatternSyntax and concrete matching implementations for these PatternTrees.
- **is-match-macro**: Defines the `derive_is_match_impl!` macro that can be used as a shorthand for a pattern tree's IsMatch implementation
- **pattern-macro-gen-macro**: Defines the `gen_pattern_macro!{...}` macro that's used to define pattern macros for specific pattern trees
- **pattern-macro-gen**: Hygienic wrapper around the `pattern-macro-gen-macro` crate
- **pattern-macro**: Uses the `gen_pattern_macro!{...}` to define pattern macros for specific pattern trees (e.g. `pattern_rust!{...}`)
- **pattern**: Hygienic wrapper around the `pattern-macro` crate.
- **pattern-func-macro**: Defines the `pattern_func!{...}` macro that can be used to define pattern functions

- **pattern-func**: Hygienic wrapper around the `pattern-func-macro` crate

The next sections describe these elements in more detail.

3.3. Implementation aspects

The following sections describe specific relevant details of the implementation. Because a full description of all implementation details is out of scope for this thesis, it focuses on communicating the core ideas and implementation details. The rest of this chapter contains the following sections:

- **Pattern Tree**: Describes the pattern tree implementation
- **Pattern Matching**: Describes how patterns are matched against IR instances
- **Pattern Macro Gen**: Describes how pattern macros are generated
- **Patterns**: Describes the implementation of patterns
- **Pattern Functions**: Describes how pattern functions are defined and expanded

3.4. Pattern Tree

The `pattern-tree` and `pattern-tree-macro` crates contain the `gen_pattern_tree!` macro and various structs / enums used in pattern trees.

3.4.1. Repetition Types

Three types `Alt`, `Seq` and `Opt` are used within pattern trees to express how many elements are expected at some place in a pattern tree. They are defined as Rust `enums` and their definitions are given below:

The `Alt` enum can represent alternatives of elements, named elements and the special `Any` variant:

```
pub enum Alt<'cx, 'o, T, Cx, O> {
    Any,
    Elmt(Box<T>),
    Alt(Box<Self>, Box<Self>),
```

```
    Named(Box<Self>, Setter<'cx, 'o, Cx, O>)
}
```

The `Opt` enum additionally contains a `None` variant which expresses that no element is expected:

```
pub enum Opt<'cx, 'o, T, Cx, O> {
    Any,  // anything, but not None
    Elmt(Box<T>),
    None,
    Alt(Box<Self>, Box<Self>),
    Named(Box<Self>, Setter<'cx, 'o, Cx, O>)
}
```

The `Seq` enum can also represent sequences and repetitions:

```
pub enum Seq<'cx, 'o, T, Cx, O> {
    Any,
    Empty,
    Elmt(Box<T>),
    Repeat(Box<Self>, RepeatRange),
    Seq(Box<Self>, Box<Self>),
    Alt(Box<Self>, Box<Self>),
    Named(Box<Self>, Setter<'cx, 'o, Cx, O>)
}
```

Repetitions are represented by the `RepeatRange` type that is given below:

```
pub struct RepeatRange {
    pub start: usize,
    pub end: Option<usize>  // exclusive
}
```

The `end` attribute of the `RepeatRange` is an optional unsigned integer. If the value of `end` is `None`, the range is open. If a value is given for `end`, it marks the exclusive maximum repetition count. For example, the `end` value of 10 means that the element may be repeated at most 9 times.

The `Named` variants of the `Alt`, `Seq` and `Opt` enums contain a `Setter` type which is given below:

```
type Setter<'cx, 'o, Cx, O> = fn(&'cx mut Cx, &'o O) -> &'cx mut Cx;
```

This setter type is a function that takes a context object `Cx` and an object `O` and returns a context object. Both objects are passed by reference and use respective lifetimes `'cx` for `Cx` and `'o` for `O`.

These setter functions in the pattern tree are used to populate the result structure which is described in more detail in section 3.7.1.

3.4.2. `gen_pattern_tree!` macro

The `gen_pattern_tree!` macro is used to create a pattern tree for a language. The macro expects pattern tree syntax (see section 2.4) as input and expands to the following elements:

- An enum for each node definition in the input
- A `TYPES` hashmap that maps from variant names to the types of the variant's parameters
- A `variants` module that is a namespace containing all variants of the pattern tree's enums
- A `MatchAssociations` trait that contains associated types for each pattern tree node

These elements are described in more detail below. As an example, consider the following pattern tree definition:

```
use pattern_tree::gen_pattern_tree;

gen_pattern_tree!{
    Expr = Lit(Lit)
         | If(Expr, Expr*, Expr?)

    Lit = Char(char)
        | Bool(bool)
}
```

Pattern Tree enums

The macro call above expands to the following enums:

```
pub enum Expr<'cx, 'o, Cx, A>
where A: MatchAssociations<'o> {
    Lit(Alt<'cx, 'o, Lit<'cx, 'o, Cx>, Cx, A::Lit>),
    If(Alt<'cx, 'o, Expr<'cx, 'o, Cx, A>, Cx, A::Expr>,
    Seq<'cx, 'o, Expr<'cx, 'o, Cx, A>, Cx, A::Expr>,
    Opt<'cx, 'o, Expr<'cx, 'o, Cx, A>, Cx, A::Expr>),
}
```

```
pub enum Lit<'cx, 'o, Cx> {
    Char(Alt<'cx, 'o, char, Cx, char>),
    Bool(Alt<'cx, 'o, bool, Cx, bool>),
}
```

For each pattern tree node definition in the input, an enum is created. Each of these enums has a number of variants which correspond to the variants of the respective pattern tree node definition. In the example above, the `Expr` enum contains two variants `Lit` and `If`. Each enum variant can have a number of arguments. For each argument in the pattern tree definition, an argument in the enum is created.

The type of an argument depends on the argument's definition in the pattern tree. The outer type is one of `Alt<...>`, `Seq<...>` and `Opt<...>` and depends on the repetition type specified in the pattern tree definition. The `If` variant of the `Expr` pattern tree node shows the three repetition types. A "single" repetition type (e.g. `Expr`) leads to an `Alt<...>` type, a "sequence repetition type (e.g. `Expr*`) leads to an `Seq<...>` type and an"optional" repetition type (e.g. `Expr?`) leads to an `Opt<...>` type.

The three types `Alt<...>`, `Seq<...>` and `Opt<...>` each expect two lifetime parameters and three type parameters. All usages of these types in pattern trees use `'cx` and `'o` as lifetime parameters and `Cx` as the second type parameter. The first and third type parameter depend on the argument's definition in the pattern tree. The first type parameter describes the inner type of the alternative / sequence / optional. For example, an inner type of `Lit<...>` used within an `Seq<...>` could be read as "a sequence of literals". These inner types form the tree structure of the pattern tree. The third type parameter describes the type of elements that are expected to be matched against the inner type. These types are either IR nodes or primitive types.

If an argument in the pattern tree definition is a pattern tree node (e.g. `Lit` or `Expr` in the `Expr` node definition), the first type parameter (inner type) is the corresponding pattern tree enum (e.g. `Lit<...>` or `Expr<...>`). Lifetime and type parameters are inserted appropriately. The third parameter is an associated type of the generic parameter `A`. The name of the associated parameter is identical to the inner type (e.g. an inner type of `Expr<...>` leads to the associated type `A::Expr`). Using a generic parameter `A` allows matching the same pattern tree against different IRs.

If an argument in the pattern tree definition is a primitive type (e.g. the `char` and `bool` arguments in the `Lit` node definition), both the inner and the associated type are identical to the primitive type. As described in section 2.5.3.1, primitive types are matched against themselves. The only exception to this are custom associated types (see section 2.4.3 in the concept). When using a custom associated type (e.g. `bool<>MyBool`) in an argument in the pattern tree definition, the associated type of that argument is an associated type of the generic type `A` (e.g. `A::MyBool`).

`TYPES` struct

An invocation of the `gen_pattern_tree!{...}` macro also generates a `TYPES` hashmap. In the example above, the following hashmap is generated.

```
lazy_static! {
    pub static ref TYPES: rustc_data_structures::fx::FxHashMap<
        &'static str, Vec<(&'static str, Ty)>
    > = {
        let mut p = rustc_data_structures::fx::FxHashMap::default();
        p.insert("Lit", vec![("Lit", Ty::Alt)]);
        p.insert("If", vec![("Expr", Ty::Alt),
                            ("Expr", Ty::Seq),
                            ("Expr", Ty::Opt)]);
        p.insert("Char", vec![("char", Ty::Alt)]);
        p.insert("Bool", vec![("bool", Ty::Alt)]);
    };
}
```

The implementation uses the `lazy_static!{...}` macro [18] that allows specifying static types that support allocation. Using the `lazy_static!{...}` macro is currently necessary due to a limitation of the Rust compiler's const evaluator but won't be required in future language versions. The static type is a hashmap that maps from variant names to lists of argument types. For example, the `Lit` variant of the `Expr` node definition has a single argument which is a single node of type `Lit`. In the `TYPES` hashmap this is represented as a mapping from the string `"Lit"` to a vector containing a single element `("Lit", Ty::Alt)`. Types are represented as two-tuples where the first element is the inner type and the second element is the repetition type.

The `TYPES` hashmap is needed during the compilation of patterns. During type checking, the pattern compilation needs to know the types of elements expected as children of certain pattern tree nodes. This information is contained in the `TYPES` hashmap.

`variants` namespace

In addition to previous elements, the `gen_pattern_tree!{...}` macro also generates a namespace that contains all variants of the pattern tree's nodes. This namespace is implemented as a public module `variants` that publicly uses all variants of all defined pattern tree enums. In the example above, the following module is created:

```
pub mod variants {
    pub use super::Expr::*;
    pub use super::Lit::*;
}
```

The `variants` namespace is used in the compilation of a pattern. During code generation, the node names used in the pattern are prepended by the `variants` namespce. This works because node names in a pattern refer to a variant in a pattern tree, not a pattern tree type. Using `Lit(...)` in a pattern refers to the `Lit` variant of the `Expr` pattern tree node, not the `Lit` pattern tree node.

`MatchAssociations` trait

An invocation of the `gen_pattern_tree!{...}` macro also generates a trait named `MatchAssociations`. For each node in the pattern tree (and each custom associated type), this trait contains an associated type. In the example above, the following trait is generated:

```
pub trait MatchAssociations<'o>
where
Self: Sized + Clone,
{
    type Expr: 'o + std::fmt::Debug + Clone;
    type Lit: 'o + std::fmt::Debug + Clone;
}
```

The purpose of this trait is to allow IRs to specify which of their types should be matched against which pattern tree nodes.

3.5. Pattern Matching

The `pattern-match` crate contains

- the `IsMatch`, `IsMatchEquality`, `Reduce` and `ReduceSelf` traits
- generic matching implementations for the `Seq`, `Alt` and `Opt` types
- the pattern tree definitions for (partial) `Rust` and the `PatternSyntax`
- matching implementations for these two pattern trees, consisting of:
  - `MatchAssociations` implementation
  - matching implementations for
    - `syntax::ast`,
    - `dummyAst` and
    - pattern parse tree

These elements are presented in the following sections.

3.5.1. IsMatch trait

The `IsMatch` trait is used to specify how a pattern tree node is matched against an IR node. The definition of the trait is given below:

```
pub trait IsMatch<'cx, 'o, Cx: Clone, O: ?Sized> {
    fn is_match(&self, cx: &'cx mut Cx, other: &'o O) -> (bool, &'cx mut Cx);
}
```

The trait is expected to be implemented on pattern tree nodes. It contains a single method `is_match` that takes a mutable reference to a context object `Cx` and a reference to an IR node `O` as parameters. The method returns a two-tuple that contains whether or not the match was successful as its first element and contains a mutable reference to the context object as second element. Returning a reference to the context element solves a certain lifetime problem which is explained in more detail in the appendix (section A.6).

The context object is an object that can contain references to IR nodes. When a pattern tree node contains a named submatch, the `is_match` implementation can store a reference to the corresponding IR node in the context object. Context objects are described in more detail in section 3.7.1.

For primitive types, the implementations of the `IsMatch` trait are trivial. Primitive types are matched against themselves and match if they are equal. As a shortcut for these cases, the `pattern-match` crate contains the `IsMatchEquality` marker trait that depends on the `PartialEq` trait:

```
pub trait IsMatchEquality: PartialEq {}
```

The crate also provides a generic `IsMatch` trait implementation for all types that implement `IsMatchEquality`:

```
impl<'cx, 'o, Cx: Clone, T> IsMatch<'cx, 'o, Cx, T> for T
where T: IsMatchEquality {
    fn is_match(&self, cx: &'cx mut Cx, other: &Self) -> (bool, &'cx mut Cx) {
        (self == other, cx)
    }
}
```

The implementation above simply returns whether or not the pattern tree element (`self`) and the IR element (`other`) are equal. Additionally, it also returns the context object passed to it without modifying it.

The crate contains implementations of `IsMatchEquality` for the following primitive types:

```
impl IsMatchEquality for u128 {}
impl IsMatchEquality for char {}
```

```
impl IsMatchEquality for bool {}
impl IsMatchEquality for &'static str {}
```

3.5.2. Reduce trait

IRs often contain indirections between nodes (e.g. references, boxes, ...). To make resolving these indirections easier, the `pattern-match` crate contains the `Reduce` trait:

```
pub trait Reduce {
    type Target;

    fn reduce(&self) -> &Self::Target;
}
```

The trait is supposed to be implemented on indirection types used in IRs. The trait contains a single function `reduce` that takes a reference to an indirection type as its argument and returns a reference to the trait's `Target` associated type.

The following example shows an implementation of the `Reduce` trait for `syntax::ptr::P` types:

```
impl<T> Reduce for syntax::ptr::P<T> {
    type Target = T;

    fn reduce(&self) -> &Self::Target {
        &*self
    }
}
```

This implementation converts from `syntax::ptr::P<T>` to `&T`.

The `Reduce` trait is used in the generic matching implementation for sequences and optionals (see section 3.5.3) and allows them to be generic over any kind of indirection.

For types that don't contain indirections, the crate provides a `ReduceSelf` marker trait. Types that implement this trait get a `reduce` function that simply returns the instance without performing any conversions:

```
pub trait ReduceSelf {}

impl<T> Reduce for T
where T: ReduceSelf {
    type Target = Self;
```

```
    fn reduce(&self) -> &Self::Target {
        self
    }
}
```

3.5.3. Generic Matching Implementations

The `pattern-match` crate also includes generic matching implementations for the `Alt<...>`, `Seq<...>` and `Opt<...>` types (see section 3.4.1). These implementations are described in the following sections.

Alt

The generic `is_match` implementation of `Alt<...>` types is given below:

```
impl<'cx, 'o, T, U, Cx: Clone> IsMatch<'cx, 'o, Cx, U>
for Alt<'cx, 'o, T, Cx, U>
where T: IsMatch<'cx, 'o, Cx, U> {
    fn is_match(&self, cx: &'cx mut Cx, other: &'o U)
    -> (bool, &'cx mut Cx) {
        match self {
            Alt::Any => (true, cx),
            Alt::Elmt(e) => e.is_match(cx, other),
            Alt::Named(e, f) => {
                let (r, mut cx) = e.is_match(cx, other);
                if r {
                    cx = f(cx, other);
                }
                (r, cx)
            },
            Alt::Alt(i, j) => {
                let (r_i, cx) = i.is_match(cx, other);
                // early return if first alternative matched
                if r_i {
                    return (r_i, cx);
                }
                j.is_match(cx, other)
            }
        }
    }
}
```

The implementation above is generic over three types `T`, `U` and `Cx`. The type parameter `T` is the inner type of the alternative (e.g. a pattern tree node). The type parameter `U` represents the type that `T` should be compared against (e.g. an IR node). The type parameter `Cx` represents the context object. The implementation contains a trait bound for `T` that enforces that `T` needs to be able to be matched against `U` (which means it needs to implement `IsMatch<..., U>`).

The exact matching logic depends on the variant of the `Alt` enum. If the variant is `Any`, the match returns `true`. In case of the `Elmt` variant, the alternative matches if the inner element `e` matches. The `Named` variant works similar. The only difference is that if the inner element `e` matches, the function `f` that sets a reference in the context object is executed. In case of the `Alt` variant, the first alternative `i` is matched first. If it matches, the result is returned. If it doesn't match, the second alternative `j` is matched and the result is returned.

Opt

The generic `is_match` implementation of `Opt<...>` types is similar to the implementation of `Alt<...>` types. The following code snippet shows the signature of the implementation:

```
impl<'cx, 'o, T, U, V, Cx: Clone> IsMatch<'cx, 'o, Cx, Option<V>>
for Opt<'cx, 'o, T, Cx, U>
where
    T: IsMatch<'cx, 'o, Cx, U>,
    V: Reduce<Target=U>
{
    fn is_match(&self, cx: &'cx mut Cx, other: &'o Option<V>)
    -> (bool, &'cx mut Cx) {
        ...
    }
}
```

In addition to the generic type parameters that `Alt<...>` has, the implementation of `Opt<...>` also contains another parameter `V`. This type needs to be reducible to to the type `U` (which means that it implements `Reduce<Target=U>`). Also, the implementation of `Opt<...>` uses `Option<V>` instead of `U` as the type of `other`. This means that `Opt<...>` is matched against an optional type whose inner type can be reduced to an IR node.

The implementation of the actual matching code is similar to `Alt<...>` and can be found in the appendix (see section A.3.2).

Seq

The generic `is_match` implementation of `Seq<...>` types is also similar to the implementations described above. The following code snippet shows the signature of the implementation:

```
impl<'cx, 'o, T, U, V, Cx: Clone> IsMatch<'cx, 'o, Cx, [V]>
for Seq<'cx, 'o, T, Cx, U>
where
    T: IsMatch<'cx, 'o, Cx, U>,
    V: Reduce<Target=U>
{
    fn is_match(&self, cx: &'cx mut Cx, other: &'o [V])
    -> (bool, &'cx mut Cx) {
        ...
    }
}
```

The generic type parameters are identical to those of the `Opt<...>` implementation. The `Seq<...>` implementation uses a slice of `Vs` (`[V]`) as its type for `other` because a sequence of pattern tree nodes should be matched against a sequence of IR nodes.

The actual matching implementation of sequences is similar to `Alt<...>`. Additionally, it also contains matching of repetitions and sequences. The actual implementation can be found in the appendix (see section A.3.1).

3.5.4. Pattern Tree Definitions

The `pattern-match` crate also contains pattern tree definitions for two languages.

The first pattern tree definition supports a subset of the Rust programming language and is implemented in the module `pattern_tree_rust`. The pattern tree definition can be found in the appendix (section A.4.1) as well.

The second pattern tree definition is a pattern tree that represents pattern syntax. It is implemented in the module `pattern_tree_meta` and can also be found in the appendix (section A.4.2). This pattern tree allows writing static analyses that analyze the source code of patterns. This is described in more detail in the evaluation (chapter 4).

3.5.5. Matching Implementations for Pattern Trees

Each pattern tree can potentially be matched against multiple different IRs. For each IR, the following has to be implemented:

- A new type that implements the `MatchAssotiations` trait of the pattern tree
- `IsMatch` implementations for each node of the pattern tree and the corresponding node in the IR

The implementation currently contains two matching implementations for the Rust pattern tree and a single matching implementation for the "meta" pattern tree. For the Rust pattern tree, the implementation provides matching implementations for the Rust compiler's `syntax::ast` IR [14] and a synthetic IR called DummyAST (which is defined in the module `dummy_ast_match`). For the "meta" pattern tree, the implementation provides matching implementations for the pattern's parse tree data structure (which is defined in the crate `pattern-parse`).

In the following, the Rust pattern tree matching implementation for `syntax::ast` is described in more detail. Other matching implementations work similarly.

Type that implements MatchAssociations

The matching implementation defines a new type `Ast` and implements the pattern tree's `MatchAssociations` trait (see section 3.4.2) for that type:

```
pub struct Ast {}

impl<'o> MatchAssociations<'o> for Ast {
    type Expr = ast::Expr;
    type Lit = ast::Lit;
    type Stmt = ast::Stmt;
    type BlockType = ast::Block;
    type LitIntType = ast::LitIntType;
    type IntTy = ast::IntTy;
    type UintTy = ast::UintTy;
    type Symbol = syntax::source_map::symbol::Symbol;
}
```

Implementing the `MatchAssociations` trait requires specifying which IR nodes are matched against which pattern tree nodes.

IsMatch implementations

For each pair of pattern tree- and IR types specified in the `MatchAssociations` implementation, an implementation of `IsMatch` needs to be provided. The `IsMatch` trait needs to be implemented for the pattern tree type (e.g. `Expr`) and needs to use the IR type (e.g. `ast::Expr`) for the trait's type parameter `T`.

Because pattern tree nodes are enums and most IR nodes are enums as well, most `IsMatch` implementations will be matches over the variants of those two types. The following code snippet shows a simplified version of the `IsMatch` implementation for `Expr` and `ast::Expr`. The full implementation can be found in the appendix (see section A.5).

```
impl<'cx, 'o, Cx: Clone> IsMatch<'cx, 'o, Cx, ast::ExprKind>
for Expr<'cx, 'o, Cx, Ast> {
    fn is_match(&self, cx: &'cx mut Cx, other: &'o ast::ExprKind)
    -> (bool, &'cx mut Cx) {
        match (self, other) {
            // Expr::Lit
            (Expr::Lit(l_a), ast::ExprKind::Lit(l_b))
                => l_a.is_match(cx, l_b),
            (Expr::Lit(l_a), _) => (false, cx),
            // Expr::If
            (Expr::If(check_a, then_a, else__a),
             ast::ExprKind::If(check_b, then_b, else__b)) => {
                let cx_orig = cx.clone();
                let (r, cx) = check_a.is_match(cx, check_b);
                if !r {
                    *cx = cx_orig;
                    return (false, cx);
                }
                let (r, cx) = then_a.is_match(cx, then_b);
                if !r {
                    *cx = cx_orig;
                    return (false, cx);
                }
                let (r, cx) = else__a.is_match(cx, else__b);
                if !r {
                    *cx = cx_orig;
                    return (false, cx);
                }
                (true, cx)
            },
            (Expr::If(check_a, then_a, else__a), _) => (false, cx),
        }
```

```
    }
}
```

For each variant in the pattern tree node, the `match` statement contains two variants. The first variant matches against the correct variant of the IR node and then matches the nodes' children recursively. The second variant matches all other IR node variants and returns a negative result.

Instead of using a second variant for each pattern tree variant, the implementation could have also used a default variant (`_ => (false, cx)`) at the end of the match statement. The advantage of the chosen approach is that match variants need to be defined for each pattern tree node variant. When adding a new variant to a pattern tree node, the `IsMatch` implementation described above won't compile until additional match variants are implemented. Using a default variant wouldn't trigger a compilation error and would incorrectly always return `false` for the newly added pattern tree node variant.

For variants that have multiple parameters (see `If` in the implementation above), the implementation will usually require that all pairs of parameters match. In this case, the implementation needs to ensure that the context object is processed correctly. Calling `is_match` on children may mutate the context object (e.g. when a child is a named submatch, a reference to an IR node will be added to the context object). It is important that the context object is only mutated by nodes that actually match. In the case of multiple parameters, if the first pair of parameters matched but the second pair doesn't (e.g. when an if's condition matches but it's body doesn't), the implementation needs to return a context object that hasn't been mutated by the first successful matching call (e.g. the if's condition). The implementation accounts for this by cloning the initial context (see `cx_orig`) and restores it if any of the parameters don't match.

Another type of `IsMatch` implementations needs to be implemented when an IR node is a struct that contains an enum as one of it's attributes. An example of this is the `ast::Lit` struct [15] that has a `ast::LitKind` enum as one of its attributes. In this case, an `IsMatch` implementation can simply forward the `is_match` call to the inner enum. An example of this is shown below:

```
impl<'cx, 'o, Cx: Clone> IsMatch<'cx, 'o, Cx, ast::Expr>
for Expr<'cx, 'o, Cx, Ast> {
    fn is_match(&self, cx: &'cx mut Cx, other: &'o ast::Expr)
    -> (bool, &'cx mut Cx) {
        self.is_match(cx, &other.node)
    }
}
```

Because matching implementations of pattern tree nodes against IR enums are tedious to write by hand, the `is-match-macro` crate contains the `derive_is_match_impl!{...}` macro which simplifies the specification of these matching implementations.

The following code snippet shows the complete specification of the matching implementation for `Expr` and `syntax::ExprKind`:

```
derive_is_match_impl!{
    Expr <> ast::ExprKind => Ast => {
        Lit(l) <> Lit(l)
        Block_(b) <> Block(b, _label)
        Array(a) <> Array(a)
        If(check, then, else_) <> If(check, then, else_)
        IfLet(then, else_) <> IfLet(_pattern, _check, then, else_)
    }
}
```

The `derive_is_match_impl!{...}` macro expands to the trait implementation shown in the appendix A.5.

The first line of the specification expresses which types are compared against each other (`Expr` and `ast::ExprKind` in the example above). These two types are separated by angle brackets (`<>`) and followed by an arrow (`=>`). For pattern tree types that depend on a type that implements `MatchAssociations`, this type needs to be specified afterwards and is followed by an arrow (`=>`) too. Simple enums like `IntTy` and `UintTy` in the Rust pattern tree (see section A.4.1) don't depend on such a type and can omit it. The arrow is followed by curly brackets `{...}` that contain the specification's body.

Each line of the specification's body compares a variant of a pattern tree node against a variant of an IR node. The specification needs to provide the correct number of arguments for each variant. Using the same identifier in both variant argument lists means that these parameters need to be matched recursively (e.g. the `l` parameter in the `Lit` line). Identifiers that appear on one side only (e.g. `_label` in the `Block` line) are ignored.

3.6. Pattern Macro Gen

The `pattern-macro-gen` and `pattern-macro-gen-macro` crates define and export the `gen_pattern_macro!` macro. The following example shows a usage of the macro:

```
use pattern_match;

gen_pattern_macro!{
    pattern => pattern_match::pattern_tree_rust
}
```

The macro expects an identifier (`pattern` in the example above) and a module (`pattern_tree_rust` in the example above) separated by an arrow (`=>`) as input.

The gen_pattern_macro! expands to an actual pattern macro. In the example above, this creates a new macro pattern! that uses the pattern tree defined in the pattern_tree_rust module. The macros the gen_pattern_macro! expands to are described in more detail in the following section.

3.7. Patterns

The pattern-macro and pattern crates use the gen_pattern_macro! macro to generate pattern macros. In the current implementation, they contain the two macros pattern! and meta_pattern!. The pattern! macro creates patterns for the Rust pattern tree. The meta_pattern! macro creates patterns for the parse tree of the pattern syntax.

The pattern! and meta_pattern! macros expect a pattern as their input (see section 2.2). The following code shows a simple usage of the pattern! macro:

```
pattern!{
    my_pattern: Expr =
        _
}
```

Pattern macros expand to the following elements:

- a temporary and a final result struct
- an initialization function for the temporary struct
- a conversion function from temporary to final structs
- a function that can be used to match IR nodes

These elements are described in more detail below.

3.7.1. Result structs

Two structs are created during the expansion of a pattern macro. The pattern above expands to the following result structs:

```
#[derive(Debug, Clone)]
struct my_patternTmpStruct<'o, A>
where A: pattern_tree_rust::MatchAssociations<'o> {
    root: Option<&'o A::Expr>
}

#[derive(Debug, Clone)]
pub struct my_patternStruct<'o, A>
```

```
where A: pattern_tree_rust::MatchAssociations<'o> {
    pub root: &'o A::Expr
}
```

Result structs always contain an attribute `root` which is a reference to the IR node matched against the whole pattern. Additionally, the result structs also contain attributes for all named submatches that are used within the pattern.

The difference between the temporary and final result struct is that IR node references are wrapped in an `Option<...>` type in the temporary struct. This allows the temporary struct to be initialized without providing values for its attributes. The generated pattern function (see section 3.7.2) uses this to create an "empty" result struct first and then add references to it gradually.

The initialization function that's generated for the temporary struct above is given below:

```
impl<'o, A> my_patternTmpStruct<'o, A>
where A: pattern_tree_rust::MatchAssociations<'o> {
    fn new() -> my_patternTmpStruct<'o, A> {
        my_patternTmpStruct { root: None }
    }
}
```

The macro also generates a function that converts a temporary struct to a final one. This is done by unwrapping `Option<...>` types. The implementation generated for the example pattern above is given below:

```
impl<'o, A> From<my_patternTmpStruct<'o, A>> for my_patternStruct<'o, A>
where A: pattern_tree_rust::MatchAssociations<'o> {
    fn from(cx: my_patternTmpStruct<'o, A>) -> Self {
        my_patternStruct {
            root: cx.root.unwrap()
        }
    }
}
```

3.7.2. Pattern Function

A pattern macro also expands to a function that takes a reference to an IR node as its argument and returns an optional result struct. The function definition generated for the pattern given above is shown below:

```
fn my_pattern<'o, A, P>(node: &'o P) -> Option<my_patternStruct<'o, A>>
where
```

```
    A: pattern_tree_rust::MatchAssociations<'o, Expr = P>,
    P: std::fmt::Debug + std::clone::Clone,
    for<'cx> Expr<'cx, 'o, my_patternTmpStruct<'o, A>, A>:
        IsMatch<'cx, 'o, my_patternTmpStruct<'o, A>, P>
{
```

Within the function, an instance of a pattern tree that represents the pattern is created first:

```
    // initialize the pattern
    let pattern: pattern::matchers::Alt<
        '_,
        '_,
        Expr<'_, '_, my_patternTmpStruct<A>, A>,
        my_patternTmpStruct<A>,
        A::Expr
    > = pattern::matchers::Alt::Named(
        Box::new(pattern::matchers::Alt::Any),
        |cx, elmt| {
            cx.root = Some(elmt);
            cx
        }
    );
```

Next, an empty instance of the temporary result struct is created (`cx`):

```
    // initialize the (temporary) result struct
    let mut cx = my_patternTmpStruct::new();
```

The actual matching is done by calling the `is_match` function on the `pattern` instance. The `is_match` call receives the temporary result struct instance as its context object and the IR instance (`node`) as its arguments. The call returns whether or not the match was successful (`r`) and the mutated temporary result struct instance (`cx_out`):

```
    // match input node against pattern
    let (r, cx_out) = pattern.is_match(&mut cx, node);
```

If the match was successful, the temporary result struct is converted into a final result struct and returned. Otherwise, `None` is returned:

```
    if r {
        // convert cx to final result struct and return
        Some(cx_out.into())
    } else {
        None
```

```
    }
}
```

3.8. Pattern Functions

The `pattern-func` and `pattern-func-lib` crates contain the `pattern_func!` macro that can be used to declare pattern functions. The following code snippet shows how the `pattern_func!` macro can be used:

```
pattern_func!{
    fn expr_or_semi($expr) {
        Expr($expr) | Semi($expr)
    }
}
```

The `pattern_func!` macro expects a pattern function definition (see section 2.3) as its input and expands to a pattern function macro. The example above expands to a macro `expr_or_semi!` whose signature is shown below:

```
#[proc_macro]
pub fn expr_or_semi(input: TokenStream) -> TokenStream {
    ...
}
```

Pattern function macros like `expr_or_semi!` above expect a single identifier that names a pattern macro and a pattern definition as input. A pattern macro expands to a macro call to the pattern macro given in its input. The input of this macro call is the pattern definition where occurrences of the pattern function are resolved.

Pattern function macros aren't supposed to be called directly. Instead, they should be used within patterns. If a pattern macro detects that it contains a function call, it expands to a macro call of that pattern function. The pattern function then textually replaces instances of that function call with the result of the function call. Afterwards, it again expands to a call to the original pattern macro. The following example shows the macro call chain:

This is the original pattern:

```
pattern!{
    my_pattern: Stmt =
        expr_or_semi(Lit(_))
}
```

`pattern!` macro detects that it contains a function call, so it expands to the function call `expr_or_semi!` and appends it's own name (`pattern`) to the front of the original input:

```
expr_or_semi!{
    pattern
    my_pattern: Stmt =
        expr_or_semi(Lit(_))
}
```

The `expr_or_semi!` macro resolves the function call (thereby duplicating the `Lit(_)` expression) and expands to a call to the original `pattern!` macro:

```
pattern!{
    my_pattern: Stmt =
        Expr(Lit(_)) | Semi(Lit(_)))
}
```

The pattern doesn't contain function calls anymore and can be processed as described in previous sections.

4. Evaluation

The concept was evaluated in three parts. First, an existing static analysis of Rust's static analysis tool Clippy [17] was rewritten using the concept. The concept was also used to write "meta" static analyses that analyze the source code of patterns themselves. Finally, the concept was presented to domain experts for review. These evaluations are presented in more detail in the following sections.

4.1. Collapsible-if static analysis

To evaluate the concept, Clippy's [17] `collapsible-if` static analysis was rewritten using with patterns. The `collapsible-if` static analysis detects nested `if` or `if let` statements that can be collapsed.

For example, the following code snippet contains two nested if statements:

```
if x == "Hello" {
    if y == "World" {
        println!("Hello world!");
    }
}
```

The code snippet above could be rewritten as a single `if` statement:

```
if x == "Hello" && y == "World" {
    println!("Hello world!");
}
```

4.1.1. Baseline implementation

The implementation of `collapsible-if` as of February 28, 2019 was used as the baseline of the evaluation. It is available online using the following link:

`https://github.com/fkohlgrueber/rust-clippy-pattern/blob/f69ec96906d300132ffd33151bf6641b950db96d/clippy_lints/src/collapsible_if.rs`

The relevant parts of the implementation are shown below:

```
impl EarlyLintPass for CollapsibleIf {
    fn check_expr(&mut self, cx: &EarlyContext<'_>, expr: &ast::Expr) {
        if !in_macro(expr.span) {
            check_if(cx, expr)
        }
    }
}

fn check_if(cx: &EarlyContext<'_>, expr: &ast::Expr) {
    match expr.node {
        ast::ExprKind::If(ref check, ref then, ref else_) => {
            if let Some(ref else_) = *else_ {
                check_collapsible_maybe_if_let(cx, else_);
            } else {
                check_collapsible_no_if_let(cx, expr, check, then);
            }
        },
        ast::ExprKind::IfLet(_, _, _, Some(ref else_)) => {
            check_collapsible_maybe_if_let(cx, else_);
        },
        _ => (),
    }
}

fn check_collapsible_maybe_if_let(
    cx: &EarlyContext<'_>,
    else_: &ast::Expr
) {
    if_chain! {
        if let ast::ExprKind::Block(ref block, _) = else_.node;
        if !block_starts_with_comment(cx, block);
        if let Some(else_) = expr_block(block);
        if !in_macro(else_.span);
        then {
            match else_.node {
                ast::ExprKind::If(..) | ast::ExprKind::IfLet(..) => {
                    // Report finding...
                }
                _ => (),
            }
        }
    }
}
```

```
fn check_collapsible_no_if_let(
    cx: &EarlyContext<'_>,
    expr: &ast::Expr,
    check: &ast::Expr,
    then: &ast::Block
) {
    if_chain! {
        if !block_starts_with_comment(cx, then);
        if let Some(inner) = expr_block(then);
        if let ast::ExprKind::If(
            ref check_inner,
            ref content,
            None
        ) = inner.node;
        then {
            if expr.span.ctxt() != inner.span.ctxt() {
                return;
            }
            // Report finding...
        }
    }
}

/// If the block contains only one expression, return it.
fn expr_block(block: &ast::Block) -> Option<&ast::Expr> {
    let mut it = block.stmts.iter();

    if let (Some(stmt), None) = (it.next(), it.next()) {
        match stmt.node {
            ast::StmtKind::Expr(ref expr)
            | ast::StmtKind::Semi(ref expr) => Some(expr),
            _ => None,
        }
    } else {
        None
    }
}
```

4.1.2. Pattern-based implementation

Using a pattern tree for a subset of Rust (see section 3.4), patterns and the two pattern functions described in section 2.3, the same static analysis was re-implemented in the following way:

```
pattern!{
    pat_if_without_else: Expr =
        If(
            _#check,
            Block(
                expr_or_semi( If(_#check_inner, _#content, ())#inner )
            )#then,
            ()
        )
}

pattern!{
    pat_if_else: Expr =
        if_or_if_let(
            _,
            Block_(
                Block(
                    expr_or_semi( if_or_if_let(_, _?)#else_ )
                )#block_inner
            )#block
        )
}

impl EarlyLintPass for CollapsibleIf {
    fn check_expr(&mut self, cx: &EarlyContext<'_>, expr: &ast::Expr) {
        if in_macro(expr.span) {
            return;
        }
        match pat_if_without_else(expr) {
            Some(res) => {
                if !block_starts_with_comment(cx, res.then) &&
                expr.span.ctxt() == res.inner.span.ctxt()
                {
                    // Report finding...
                }
            },
            _ => ()
        }
```

```
        match pat_if_else(expr) {
            Some(res) => {
                if !block_starts_with_comment(cx, res.block_inner) &&
                !in_macro(res.else_.span)
                {
                    // Report finding...
                }
            },
            _ => ()
        };
    }
}
```

The differences between the pattern-based and baseline implementation are discussed in section 4.4.

4.2. Meta patterns

In addition to the `collapsible-if` static analysis for Rust code, static analyses for another language were implemented as well. The language patterns are specified in was chosen as the target language. Using this language allows writing static analyses that analyze pattern specifications themselves, which makes them meta-patterns.

For example, the following pattern detects repetitions (e.g. `_{0,1}`) that can be replaced with the `?` operator (e.g. `_?`):

```
meta_pattern!{
    meta_pat_complicated_range: ParseTree =
        Repetition(_, Range(0, 1))
}
```

As another example, the following pattern detects alternatives where one branch matches any node (e.g. `_ | Lit(_)`) which makes the other branch useless:

```
meta_pattern!{
    meta_pat_any_or: ParseTree =
        Alt(Any, _) | Alt(_, Any)
}
```

The pattern tree created for the pattern syntax is shown in the appendix (section A.4.2).

4.3. Request for Comments (RFC)

The basic concept was published as a Request for Comments (RFC) to the developers of the Clippy project. The RFC can be found here:

`https://github.com/rust-lang/rust-clippy/pull/3875`

The RFC was reviewed by a number of developers and yielded no major issues. Suggestions expressed by reviewers are presented in the the discussion and marked as external contributions.

4.4. Discussion

The following sections present a discussion of the proposed concept and implementation. The structure of the discussion follows the requirements identified in section 1.1 and 1.2. The following properties are discussed:

- Expressiveness
- Language- / Implementation-independence
- Extensibility
- Composibility
- Usability

4.4.1. Expressiveness

The evaluation shows that existing non-trivial real-world static analyses can be implemented using the proposed concept. The `collapsible-if` re-implementation using patterns passes all tests written for the baseline implementation while being more compact than the baseline implementation. The pattern-based implementation uses 52 lines of code while the baseline implementation uses 68. Because lines of code aren't well suited for comparing the amount of code necessary, the two implementations were also compared in their number of characters when comments and non-necessary whitespace were removed. This comparison shows that the pattern-based implementation (using 620 characters) was only 46% the size of the original implementation (1349 characters). This shows that the pattern-based implementation can express this static analysis more succinctly and indicates that it might be more expressive in that domain in general. A major reason for this is that patterns are specified declaratively and using a domain-specific language. On the other hand, the domain-specific pattern language also limits its expressiveness compared to a general-purpose programming language. To account for this, the concept provides extension mechanisms which are discussed in section 4.4.3.

The evaluation also shows that the concept is general enough to be applied to different languages. The patterns presented in section 4.2 analyze pattern syntax itself instead of Rust syntax.

4.4.2. Language- / Implementation-independence

Implementation details are not part of pattern definitions. This means that changing the implementation doesn't necessarily require changes in patterns. For example, adding another indirection (e.g. `Box<...>`) in rust's AST data structure would only require changing the `IsMatch` implementation. Existing patterns wouldn't have to be changed. This decoupling allows IRs to evolve without requiring large changes in static analysis tools.

The second aspect is language-independence. The concept is designed to be applicable to different languages. The syntax patterns are specified in is language-agnostic and pattern trees for different languages can be specified easily. Language-independence allows using the same concepts for different languages, which reduces required implementation work. Additionally, this also means that users only need to learn the concepts once and can apply them in different contexts afterwards.

4.4.3. Extensibility

Named submatches in patterns allow patterns to return references to IR nodes. These references can then be used within a general-purpose programming language to perform additional checks. For example, the implementation of the `collapsible-if` static analysis (see section 4.1) uses named submatches for additional checks (e.g. in the `block_starts_with_comment(...)` function call.

Extensibility allows optimizing for common cases while having full flexibility when needed. It allows writing lints in two stages. In the first stage, a coarse matching is performed using patterns. In the second stage, the results of the first stage are filtered using the flexibility of a general-purpose programming language. The `collapsible-if` static analysis is an example of this.

4.4.4. Composibility

Pattern functions (see section 2.3) allow reusing parts of patterns. This includes using similar sub-patterns multiple times within a pattern and also sharing common sub-patterns between multiple patterns.

An alternative to pattern functions would have been to introduce variables that allowed using the same sub-pattern in multiple places. The concept uses pattern functions instead

of variables because they are more expressive. Variables can be seen as functions without any arguments. Allowing functions that can take arguments is therefore more expressive and more powerful as a concept of composition.

Two pattern functions were used in the `collapsible-if` static analysis (see section 4.1) which eliminated repetition within the patterns and simplified their implementation. The pattern functions used can be defined in other crates which allows creating libraries of common pattern functions.

4.4.5. Usability

The pattern syntax itself is kept as simple as possible. It reuses common function syntax (`<name>(<arg1>, <arg2>, ...)`) and syntactic elements known from REs (e.g. syntax for repetitions and alternatives). These factors should make learning the syntax relatively simple for the majority of users.

In addition to the pattern syntax, users also need to know which nodes are valid in which positions in a pattern. The corresponding pattern tree definition of a pattern provides this information in a form that should be comprehensible to most users.

A remaining usability problem is that users usually don't think about their code in a IR-like structure. Instead, they think about it in its textual representation. Mapping the textual representation users are used to to a pattern that matches it is a non-trivial task and requires users to know and understand their language's IR-like structure. This problem could be solved by a tool that takes a code sample as input and generates a pattern that matches exactly this code. This idea is described in section 5.1.5 in more detail.

The evaluation of the `collapsible-if` static analysis shows that pattern-based static analyses require less code. It also shows that the code required in the host programming language is much simpler in the pattern-based implementation.

While the observations described above suggest that the usability of pattern-based static analyses might be better than the usability of traditional static analyses, this thesis doesn't provide a definitive answer to this question.

5. Conclusion

This thesis has proposed, implemented and evaluated a DSL that allows specifying static analyses. It succeeded in providing a language- and implementation-independent solution and several factors indicate that specifying static analyses using the proposed system is easier than using traditional approaches.

In the future, systems like the one presented in this thesis could be used to write a larger number of more powerful static analyses. This could have a huge positive impact on the efficiency of software development. The general idea of automatically detecting imperfect code and replacing it with better solutions could both educate users and also reduce the accidental complexity of software systems. For example, an advanced static analysis could even detect implementations for which a maintained and mature library exists and could suggest to use that. This would reduce the amount of code a developer would have to maintain and would also help the adoption of well-maintained projects.

5.1. Outlook

While the concept presented in this thesis is able to fulfill its requirements, a number of possible extensions and opportunities were discovered during its development. These ideas are presented in the following sections.

5.1.1. Early filtering

In the current concept, static analyses are written in two stages (see section 4.4.3). This can lead to unnecessary work in some cases. For example, a static analysis that detects functions with long names would have to detect any functions in the first stage and filter their names in the second stage. Instead of applying additional filters after the whole pattern was matched in the first stage, filters could be checked while their relevant part of the pattern was matched.

The following pattern shows how this could look like:

```
pattern!{
    pat_if_without_else: Expr =
```

```
        If(
            _,
            Block(
                expr_or_semi( If(_, _, ())#inner )
            )#then,
            ()
        )
    where
        !in_macro(#then.span);
}
```

In the pattern above, the condition (`!in_macro(#then.span)`) would be evaluated as soon as the `Block(...)#then` sub-pattern was matched.

5.1.2. User study

As described in section 4.4.5, the usability of the proposed concept compared to traditional approaches remains an open research question. Future research could execute user studies to assess the concept's usability.

5.1.3. Match descendant

A common case in static analyses is that a node needs to have child nodes that conform to some criteria without the nodes having to be direct children. An example of this would be a static analysis that matches "a function that contains at least two return statements". The current concept does not support this case natively, so these static analyses need to be implemented using extensions in the host programming language. Supporting this use case in patterns themselves could simplify the implementation of a lot of static analyses.

5.1.4. Named parameters

As part of his feedback on the RFC (see section 4.3), Manish Goregaokar pointed out that named parameters could be used to improve the readability of patterns. For pattern tree nodes that have multiple parameters, names could be useful to clarify the meaning of each parameter. For example, using `If(cond=_, then=_, else=_)` in a pattern would be more descriptive than `If(_, _, _)`.

5.1.5. Clippy Pattern Author

As described in section 4.4.5, users of patterns need to know how certain code is represented in the pattern tree. A tool that'd take code as input and produces a pattern that matches this code could help users to understand the relationship between program code and patterns that match it. Such a tool would also be helpful in the development of new static analyses, where users could provide different example inputs and then create a pattern that matches all inputs from the different outputs of the tool.

A. Appendix

A.1. Project structure

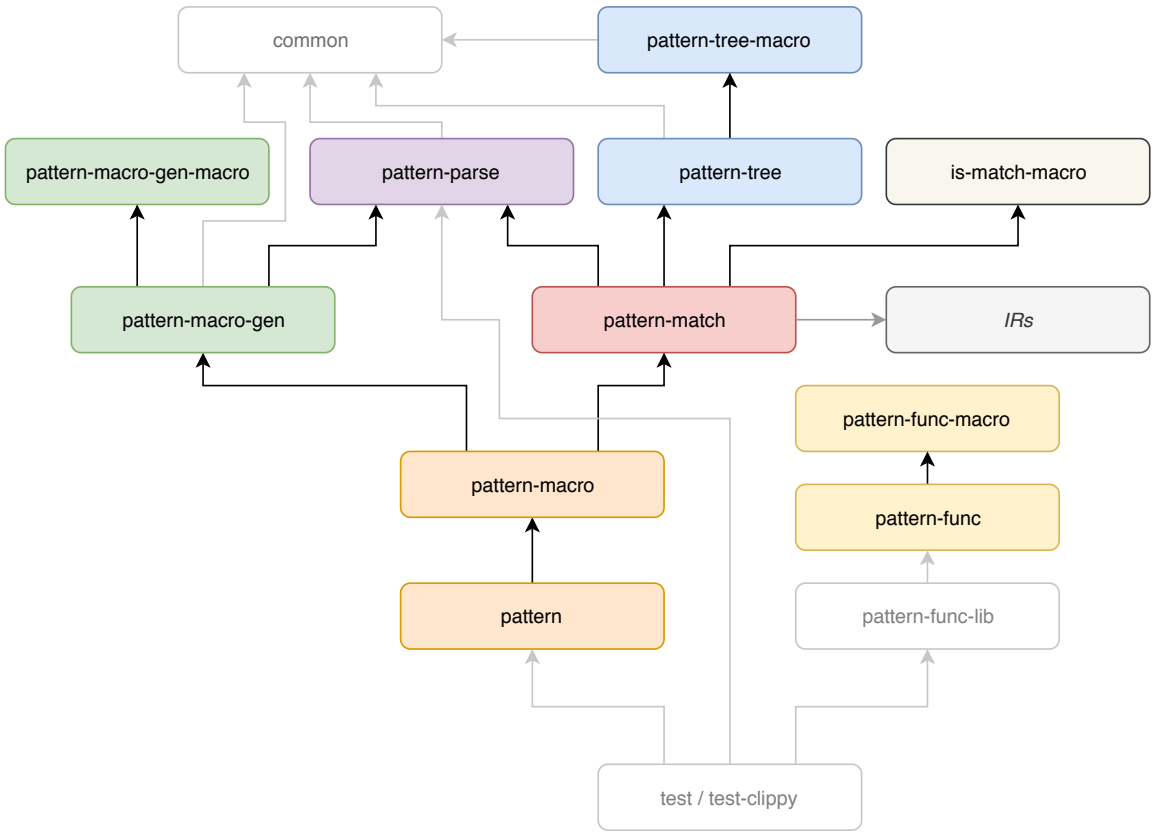

Figure A.1.: Full project structure including helper and test crates.

A.2. Syntax definitions

A.2.1. Pattern syntax

```
patternDef = ident, ":", patternType, "=", pattern;

patternType = ident | repeatType;
repeatType = repType, "<", ident, ">";
repType = "Alt" | "Seq" | "Opt";

pattern = singlePattern | sequence | alternative;
sequence = pattern, [ ";" ], pattern;
alternative = pattern, "|", pattern;

singlePattern = atomExpr, [ rep ], [ name ];
rep = "+" | "*" | "?" | repBraced;
repBraced = "{", repRange, "}";
repRange = repeat | range;
repeat = litInt;
range = litInt, ",", [ litInt ];
name = "#", ident;

atomExpr = "_" | nodeExpr | emptyExpr | "(", pattern, ")" | lit;
emptyExpr = "(", ")";
nodeExpr = ident, [ nodeExprParens ];
nodeExprParens = "(", nodeExprArgs, ")";
nodeExprArgs = pattern, { ",", pattern };
```

A.2.2. Pattern Tree syntax

```
patternTree = { ptDef };
ptDef = ident, "=", ptVariant, { "|", ptVariant };
ptVariant = ident, [ ptArgs ];
ptArgs = "(", ptArg, { ",", ptArg }, ")";
ptArg = ident, [ "*" | "?" ], ["<>", ident];
```

A.3. Generic IsMatch Implementations

A.3.1. Seq

```
impl<'cx, 'o, T, U, V, Cx: Clone> IsMatch<'cx, 'o, Cx, [V]>
for Seq<'cx, 'o, T, Cx, U>
where
    T: IsMatch<'cx, 'o, Cx, U>,
    V: Reduce<Target=U>
{
    fn is_match(&self, cx: &'cx mut Cx, other: &'o [V])
    -> (bool, &'cx mut Cx) {
        let mut cx = cx;
        match self {
            Seq::Any => (other.len() == 1, cx),
            Seq::Elmt(e) => {
                if other.len() != 1 { return (false, cx); }
                e.is_match(cx, &other[0].reduce())
            },
            Seq::Named(e, f) => {
                let (r, mut cx) = e.is_match(cx, other);
                if r {
                    for o in other {
                        cx = f(cx, o.reduce());
                    }
                }
                (r, cx)
            },
            Seq::Alt(i, j) => {
                let (r_i, cx) = i.is_match(cx, other);
                // early return if first alternative matched
                if r_i {
                    return (r_i, cx);
                }
                j.is_match(cx, other)
            },
            Seq::Empty => (other.is_empty(), cx),
            Seq::Repeat(e, r) => {
                let cx_orig = cx.clone();
                let e_range = e.num_elmts_range();
                let e_range = e_range.start..
                    e_range.end.unwrap_or(other.len()+1);
```

```
        if r.start == 0 && other.is_empty() {
            return (true, cx);
        }

        for i in r.start..r.end.unwrap_or(other.len()+1) {

            let iterators = repeat_n(e_range.clone(), i)
                .multi_cartesian_product()
                .filter(|x| x.iter().sum::<usize>() == other.len());

            'outer: for vals in iterators {
                *cx = cx_orig.clone();
                let mut skip = 0;
                for v in &vals {
                    let (r_e, cx_tmp) = e.is_match(
                        cx, &other[skip..skip+v]
                    );
                    cx = cx_tmp;
                    if !r_e {
                        continue 'outer;
                    }
                    skip += v;
                }
                return (true, cx);
            }
        }
        *cx = cx_orig;
        (false, cx)
    },
    Seq::Seq(a, b) => {
        let cx_orig = cx.clone();
        let range = a.num_elmts_range();
        for i in range.start..range.end.unwrap_or(other.len()+1) {
            *cx = cx_orig.clone();
            if i > other.len() {
                break;
            }
            let (l, r) = other.split_at(i);
            let (r_a, cx_tmp) = a.is_match(cx, l);
            cx = cx_tmp;
            if r_a {
                let (r_b, cx_tmp) = b.is_match(cx, r);
                cx = cx_tmp;
                if r_b {
                    return (true, cx);
```

```
                        }
                    }
                }
                *cx = cx_orig;
                (false, cx)
            },
        }
    }
}
```

A.3.2. Opt

```
impl<'cx, 'o, T, U, V, Cx: Clone> IsMatch<'cx, 'o, Cx, Option<V>>
for Opt<'cx, 'o, T, Cx, U>
where
    T: IsMatch<'cx, 'o, Cx, U>,
    V: Reduce<Target=U>
{
    fn is_match(&self, cx: &'cx mut Cx, other: &'o Option<V>)
    -> (bool, &'cx mut Cx) {
        match self {
            Opt::Any => (other.is_some(), cx),
            Opt::Elmt(e) => match other {
                Some(other) => e.is_match(cx, other.reduce()),
                None => (false, cx)
            },
            Opt::Named(e, f) => {
                let (r, mut cx) = e.is_match(cx, other);
                if r {
                    if let Some(o) = other {
                        cx = f(cx, o.reduce())
                    }
                }
                (r, cx)
            },
            Opt::Alt(i, j) => {
                let (r_i, cx) = i.is_match(cx, other);
                // early return if first alternative matched
                if r_i {
                    return (r_i, cx);
                }
                j.is_match(cx, other)
            },
```

```
            Opt::None => (other.is_none(), cx),
        }
    }
}
```

A.4. Pattern Tree definitions

A.4.1. Rust Pattern Tree

```
Expr = Lit(Lit)
     | Array(Expr*)
     | Block_(BlockType)
     | If(Expr, BlockType, Expr?)
     | IfLet(BlockType, Expr?)

Lit = Char(char)
    | Bool(bool)
    | Int(u128, LitIntType)
    | Str(str<>Symbol)

BlockType = Block(Stmt*)

Stmt = Expr(Expr)
     | Semi(Expr)

LitIntType = Signed(IntTy)
           | Unsigned(UintTy)
           | Unsuffixed

IntTy = Isize
      | I8
      | I16
      | I32
      | I64
      | I128

UintTy = Usize
       | U8
       | U16
       | U32
       | U64
       | U128
```

A.4.2. Pattern Syntax Pattern Tree

```
ParseTree = Node(str<>Ident, ParseTree*)
          | Alt(ParseTree, ParseTree)
          | Seq(ParseTree, ParseTree)
          | Repetition(ParseTree, RepeatKind)
          | Named(ParseTree, str<>Ident)
          | Lit(Lit)
          | Any
          | Empty

RepeatKind = Any
           | Plus
           | Optional
           | Range(u128<>LitInt, u128?<>LitInt)
           | Repeat(u128<>LitInt)

Lit = Bool(bool<>LitBool)
    | Int(u128<>LitInt)
    | Char(char<>LitChar)
```

A.5. IsMatch implementation of Expr

```
impl<'cx, 'o, Cx: Clone> IsMatch<'cx, 'o, Cx, ast::ExprKind>
for Expr<'cx, 'o, Cx, Ast> {
    fn is_match(&self, cx: &'cx mut Cx, other: &'o ast::ExprKind)
    -> (bool, &'cx mut Cx) {
        match (self, other) {
            (Expr::Lit(l_a), ast::ExprKind::Lit(l_b))
                => l_a.is_match(cx, l_b),
            (Expr::Lit(l_a), _) => (false, cx),
            (Expr::Block_(b_a), ast::ExprKind::Block(b_b, _label_b))
                    => b_a.is_match(cx, b_b),
            (Expr::Block_(b_a), _) => (false, cx),
            (Expr::Array(a_a), ast::ExprKind::Array(a_b))
                => a_a.is_match(cx, a_b),
            (Expr::Array(a_a), _) => (false, cx),
            (Expr::If(check_a, then_a, else__a),
             ast::ExprKind::If(check_b, then_b, else__b)) => {
                let cx_orig = cx.clone();
                let (r, cx) = check_a.is_match(cx, check_b);
                if !r {
```

```
                    *cx = cx_orig;
                    return (false, cx);
                }
                let (r, cx) = then_a.is_match(cx, then_b);
                if !r {
                    *cx = cx_orig;
                    return (false, cx);
                }
                let (r, cx) = else__a.is_match(cx, else__b);
                if !r{
                    *cx = cx_orig;
                    return (false, cx);
                }
                (true, cx)
            },
            (Expr::If(check_a, then_a, else__a), _) => (false, cx),
            (Expr::IfLet(then_a, else__a),
             ast::ExprKind::IfLet(
                _pattern_b, _check_b, then_b, else__b
             )) => {
                let cx_orig = cx.clone();
                let (r, cx) = then_a.is_match(cx, then_b);
                if !r {
                    *cx = cx_orig;
                    return (false, cx);
                }
                let (r, cx) = else__a.is_match(cx, else__b);
                if !r {
                    *cx = cx_orig;
                    return (false, cx);
                }
                (true, cx)
            },
            (Expr::IfLet(then_a, else__a), _) => (false, cx)
        }
    }
}
```

A.6. IsMatch lifetime problem

The `is_match(...)` function of the `IsMatch` trait returns a reference to a context object `Cx`. This section describes the problem this solves.

Consider the following simplified example:

```
pub trait IsMatch<'cx, Cx> {
    fn is_match(&self, cx: &'cx mut Cx);
}

struct Alt<T>(T, T);

impl<'cx, T, Cx> IsMatch<'cx, Cx> for Alt<T>
where T: IsMatch<'cx, Cx> {
    fn is_match(&self, cx: &'cx mut Cx) {
        self.0.is_match(cx);
        self.1.is_match(cx);
    }
}
```

The code above doesn't compile and compiler shows the following error:

```
error[E0499]: cannot borrow `*cx` as mutable more than once at a time
  --> src/lib.rs:11:25
   |
7  | impl<'cx, T, Cx> IsMatch<'cx, Cx> for Alt<T>
   |      --- lifetime `'cx` defined here
...
10 |         self.0.is_match(cx);
   |         -------------------
   |         |               |
   |         |               first mutable borrow occurs here
   |         argument requires that `*cx` is borrowed for `'cx`
11 |         self.1.is_match(cx);
   |                         ^^ second mutable borrow occurs here
```

The problem here is that both calls to the `is_match` function need to mutably borrow the variable `cx` for the same lifetime `'cx`. Since Rust only allows a single mutable borrow, the compilation fails.

To address this issue, it's possible to let the `is_match` function return the mutable reference it got as its argument and pass that to subsequent calls where the context is needed. The code below shows this:

```
pub trait IsMatch<'cx, Cx> {
    fn is_match(&self, cx: &'cx mut Cx) -> &'cx mut Cx;
}

struct Alt<T>(T, T);
```

```
impl<'cx, T, Cx> IsMatch<'cx, Cx> for Alt<T>
where T: IsMatch<'cx, Cx> {
    fn is_match(&self, cx: &'cx mut Cx) -> &'cx mut Cx {
        let cx2 = self.0.is_match(cx);
        self.1.is_match(cx2)
    }
}
```

The code above satisfies the conditions enforced by the borrow checker and compiles successfully. The reason this example works is that it uses two distinct variables `cx` and `cx2` that point to the same object. Each of these variables are only borrowed once.

Glossary

(E)BNF (Extended) Backus-Naur form. 8

AST Abstract Syntax Tree. 3, 5, 7–9, 18–20, 23, 61

DSL Domain-Specific Language. i, iii, 4, 5, 27, 63

IR Intermediate Representation. 3–5, 8, 13, 16, 23–25, 27–29, 34, 37, 39–41, 43–51, 61, 62

RE Regular Expression. 4, 7, 9–11, 62

RFC Request for Comments. vi, 60, 64

Bibliography

[1] Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Pearson Education, 2007.

[2] David Tolnay. *Cargo-Expand*. URL: https://github.com/dtolnay/cargo-expand.

[3] David Tolnay. *Quote: Rust Quasi-Quoting*. URL: https://github.com/dtolnay/quote.

[4] David Tolnay. *Syn: Parser for Rust Source Code*. URL: https://github.com/dtolnay/syn.

[5] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. 3rd ed. OCLC: ocm76945355. Sebastapol, CA: O'Reilly, 2006. 515 pp. ISBN: 978-0-596-52812-6.

[6] Google. *Google C++ Style Guide*. URL: https://google.github.io/styleguide/cppguide.html.

[7] Google. *Rerast*. URL: https://github.com/google/rerast.

[8] Stephen C Johnson. *Lint, a C Program Checker*. Citeseer, 1977.

[9] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. San Francisco: No Starch Press, 2018. 519 pp. ISBN: 978-1-59327-828-1.

[10] Julia Lawall. *Coccinelle*. URL: http://coccinelle.lip6.fr/.

[11] Python Code Quality Authority. *Pylint*. URL: http://pylint.pycqa.org/en/latest/.

[12] Rust Team. *Procedural Macros - The Rust Reference*. URL: https://doc.rust-lang.org/reference/procedural-macros.html.

[13] Cppcheck Team. *Cppcheck*. URL: http://cppcheck.sourceforge.net/.

[14] The Rust Language Developers. *Rustc - Module Syntax::Ast*. URL: https://doc.rust-lang.org/nightly/nightly-rustc/syntax/ast/index.html.

[15] The Rust Language Developers. *Rustc - Struct Syntax::Ast::Lit*. URL: https://doc.rust-lang.org/nightly/nightly-rustc/syntax/ast/struct.Lit.html.

[16] The Rust Language Developers. *The Cargo Book*. URL: https://doc.rust-lang.org/stable/cargo/.

[17] The Rust Project Developers. *Clippy*. URL: https://github.com/rust-lang/rust-clippy.

[18] The Rust Project Developers. *Lazy-Static.Rs*. URL: https://github.com/rust-lang-nursery/lazy-static.rs.

[19] Guido van Rossum, Barry Warsaw, and Nick Coghlan. *PEP 8 - Style Guide for Python Code*. URL: https://www.python.org/dev/peps/pep-0008/.

[20] Markus Völter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. In collab. with Sebastian Benz et al. OCLC: 844038960. Lexington, KY: CreateSpace Independent Publishing Platform, 2010. 558 pp. ISBN: 978-1-4812-1858-0.

[21] Chris Wong. *If_chain*. URL: https://github.com/lfairy/if_chain.