Concrete Syntax Metapatterns

Miljak, Luka; Poulsen, Casper Bach; Corvino, Rosilde

DOI
10.1145/3687997.3695637

Publication date
2024

Document Version
Final published version

Published in
SLE 2024 - Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering, Co-located with

Concrete Syntax Metapatterns

Luka Miljak
Delft University of Technology
Delft, Netherlands
L.Miljak@tudelft.nl

Casper Bach Poulsen
Delft University of Technology
Delft, Netherlands
C.B.Poulsen@tudelft.nl

Rosilde Corvino
TNO-ESI
Eindhoven, Netherlands
rosilde.corvino@tno.nl

Abstract

Software engineers should be able to apply massive code refactorings to maintain large legacy code bases. A key aspect of developing restructurings is matching and transforming code snippets using abstract syntax trees (ASTs).

Matching on ASTs is typically done through AST patterns with holes. AST patterns can be extended to become *metapatterns*, which increase their expressivity. Metapattern examples include disjunctions, descendant patterns, and patches where we inline transformations into the pattern itself.

Despite their expressivity, abstract syntax (meta)patterns can become verbose and require restructuring engineers to be intimately familiar with the data types that define the AST. A better approach is to use *concrete syntax patterns*, which allows us to denote our patterns in the syntax of the object language. Previous work has shown that we can use external *black-box parsers* of the object language to compile concrete syntax patterns for arbitrary languages.

In this paper, we scale this compilation method to support concrete syntax *meta*patterns, which allows for a more declarative way of expressing restructurings. We evaluate this method through an implementation written in Kotlin.

CCS Concepts: • **Software and its engineering → Translator writing systems and compiler generators**; **Domain specific languages**.

Keywords: refactoring, restructuring, metaprogramming, concrete syntax, black-box parsers

ACM Reference Format:
Luka Miljak, Casper Bach Poulsen, and Rosilde Corvino. 2024. Concrete Syntax Metapatterns. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (SLE '24), October 20–21, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3687997.3695637

SLE '24, October 20–21, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1180-0/24/10
https://doi.org/10.1145/3687997.3695637

1 Introduction

Software restructuring is important to keep large software projects maintainable. This can range from small-scale refactorings, like renaming a variable, to large project-wide restructurings, such as migrating from one API to another (e.g., switching to a different logging framework). For large projects, performing manual project-wide restructurings can be prohibitively expensive. For this reason, modern IDEs and code transformation tools provide integrated support for *automated* refactorings. However, many software projects require restructurings beyond what IDEs support. Implementing such automated restructurings is expensive, error-prone, and requires specialist knowledge of metaprogramming.

Implementing restructurings typically involves matching, traversing, and manipulating abstract syntax trees (ASTs). With this come two usability concerns. The first concern relates to matching on ASTs using abstract syntax patterns. These patterns are represented as terms with metavariables, which can be verbose and depend on our knowledge of the data types and constructors defining the abstract syntax of the object language. The second concern is about how close our implementation resembles the specification of the restructuring, i.e., how declarative is our implementation?

This concern can best be illustrated through some example. Consider a restructuring for some object-oriented language where we wish to improve our logging messages by prepending each message with the name of the class surrounding the log. For example, consider the following C++ program:

```
class Foo {
  void bar() { logger::log(myMsg); }
};
```

The `myMsg` argument supplied to the logger should be updated to `"Foo: " + myMsg` after restructuring.

A typical implementation for this restructuring performs the following steps: (1) Match the input AST on a class declaration pattern, then (2) traverse each AST node in the body of the class. For each node, (3) if this node matches a call to `log`, then (4) rewrite the message argument so that the name of the matched class is prepended to it. This is an *operational* way of thinking and constructing a restructuring.

Concrete syntax patterns. We can mitigate the concern of verbose patterns using concrete syntax patterns [3, 15]. Rather than using abstract syntax to denote our patterns, we use the concrete syntax of the object language. Existing language-parametric tools for restructuring, such as the

```
class@|struct @name {
    @<...EXPRESSION
        <<logger::log(@msg)>> --> <<logger::log("@name: " + @msg)>>
     @+ <<logger::log(@level, @msg)>> --> <<logger::log(@level, "@name: " + @msg))>>
    ...>
};
```

Listing 1. A concrete syntax metapattern for a C++ restructuring that prepends the class name before each log message.

Rascal Metaprogramming Language [1, 7] and the Spoofax Language Workbench [5, 17] already support such concrete syntax patterns. Internally, these patterns are still compiled into abstract syntax patterns.

Metapatterns. Addressing the other concern, we make restructurings more declarative through *metapatterns*. While normal patterns can be viewed as just terms with metavariables and holes, metapatterns are inlined with more logic. For example, to handle AST traversals, a descendant pattern $/p$ matches an AST node if any of its descendants match p. A disjunctive pattern $p_1|p_2$ inlines case analyses inside the pattern. Similarly, we can inline patches $p_1 \longrightarrow p_2$, used to perform transformations, into a larger pattern. We introduce these metapatterns in Section 2.

Concrete syntax metapatterns. If we embed these metapatterns in the concrete syntax of the object language, we get *concrete syntax metapatterns* (CSMPs). For example, we can represent the logger refactoring using a single CSMP as shown in Listing 1. The operators `@|` and `@+` are used for case splitting, `@<... ...>` is for deep matching, and `-->` to patch code that we have matched on. While the exact details of the syntax and semantics of CSMPs in this example are explained in the remainder of this paper, we already see that this is a declarative way of implementing such a restructuring.

Although some existing tools do support some formalism of metapatterns, there are limitations. For instance, Coccinelle [9] lets us inline patches and disjunctions in the concrete syntax, but this tool is limited to the C language. Scala supports disjunctions (pattern alternatives) in case analyses [11], but is not a metaprogramming language and thus is not suitable for concrete syntax. Rascal supports concrete syntax patterns and the descendant pattern [14], but the descendant pattern can only be used in abstract syntax.

What we wish for is a tool that is (1) language parametric (scalable to different programming languages), and (2) supports CSMPs. To that end, in this paper we present a generalized technique for compiling CSMPs into abstract syntax metapatterns. To make this compiler scalable to different programming languages without significant effort, we extend a method by Aarssen et al. [1] to compile concrete syntax patterns into abstract syntax patterns. Their method depends on a *black-box parser* of the object language (Section 3).

We summarize our contributions in this paper as follows:

$$
\begin{array}{lcll}
b & := & \text{numbers, strings} & \\
c & := & \text{constructor name} & \\
x & := & \text{identifier} & \\
p & ::= & b & \text{base pattern} \\
 & | & c(p^*) & \text{constructor pattern} \\
 & | & x & \text{metavariable} \\
 & | & p|p & \text{disjunctive pattern} \\
 & | & /p & \text{descendant pattern} \\
 & | & p \longrightarrow p & \text{patch pattern}
\end{array}
$$

Figure 1. Syntax of ASMPs with disjunctive, descendant, and patch patterns.

- We define four different CSMPs: two variants of the disjunctive pattern (Section 4 and Section 5), the descendant pattern (Section 6), and patches (Section 7).
- We present methods for compiling these CSMPs as relational definitions using black-box parsers.
- We present an implementation of this compiler in Kotlin (Section 8).
- We evaluate and discuss the effort required to implement new CSMPs, support for new languages, performance, and expressivity of CSMPs (Section 9).

2 Abstract Syntax Metapatterns

Informally, we call a pattern a *metapattern* if it contains more logic than just being a term with metavariables and holes; logic that otherwise would be contained outside the pattern. In this section, we look at some metapatterns using *abstract syntax* and call them *abstract syntax metapatterns* (ASMPs). We informally describe the behavior of these patterns and leave out their formal semantics to focus more on their embedding in the *concrete syntax* of the object language.

We define the syntax of ASMPs in Figure 1. "Ordinary" patterns would be made solely out of constructor patterns and metavariables, whereas we consider disjunctive patterns, descendant patterns, and patches to be metapatterns.

A disjunctive pattern $p_1|p_2$ successfully matches some AST a if either p_1 or p_2 matches a. For example, the ASMP `FunCall("foo" | "bar", args)` represents a function call to either `"foo"` or `"bar"`. Ordinarily, this would have required duplication of the entire pattern, one case for both names, or replacing the disjunction with a metavariable `name` and

then having a separate constraint for the allowed values of this name. In Section 4, we explore the disjunctive pattern further in the context of concrete syntax.

The descendant pattern $/p$ successfully matches some AST a if any descendant node of a matches p [14]. For example, the ASMP `FunDef(n, params, /FunCall("f", args))` matches any function definition that contains some function call to `"f"`. Ordinarily, this would have required capturing the body inside a metavariable `b`, then performing some traversal over `b` which matches each node against the inner pattern `FunCall("f")`. We explore this further in Section 6.

A patch "pattern" $p_1 \longrightarrow p_2$ is used for more than matching and getting a substitution for the metavariables. For some AST a, the match is a success if p_1 matches a. The result of the match will then also yield a *patch*, which will replace a with p_2 when applied. Note that p_2 can only consist of constructor patterns and metavariables, and each metavariable should be captured. A patch pattern can be viewed as a *rewrite rule* that can be inlined deeper into the pattern. For example, the ASMP `FunCall("foo"` $\longrightarrow$ `"bar", args)` will match function calls to `"foo"` and replace them with function calls to `"bar"`. Ordinarily, rewrite rules can only be written on the top level of the pattern, requiring a near-duplication of the pattern. Alternatively, we can replace the patch pattern with a metavariable `name` and add a separate rewrite rule that is applied on the `name`. See Section 7 for more information on patches in concrete syntax.

In general, these metapatterns aid in making implementations of restructurings more concise. By allowing us to inline their logic into the pattern itself, the behavior of our implementations becomes less scattered.

The three metapatterns presented here (disjunctive, descendant, patch) is not an exhaustive set. However, we argue that these three make for a representative set of metapatterns when discussing their embedding into concrete syntax, as we shall demonstrate in later sections. The support of other metapatterns is discussed in Section 9.1.

3 Concrete Syntax with Black Box Parsers

Abstract syntax patterns (ASPs) require the user to know the underlying structure of the AST. Furthermore, ASPs can be quite verbose, particularly for languages with complex structures. Therefore, we often use *concrete* syntax patterns (CSPs) instead; these patterns can be written using the underlying concrete syntax of the object language. Internally, CSPs are still *compiled* into abstract syntax patterns. For example, the concrete syntax pattern `5 + @x` compiles to `Add(Int(5), x)`. The `@x` denotes a metavariable named `x`.

We define the syntax of CSPs in Figure 2, extended in later sections to support the metapatterns introduced in Section 2. Note that we require metavariables to be annotated with their corresponding syntactic type, e.g., `5 + @<x : Expr>`. In the paper by Aarssen et al. [1], they developed Concretely,

s	:=	strings	
x	:=	identifiers	
t	:=	syntactic type	
cp	::=	s	string
	|	`@<`x : t`>`	metavariable
	|	cp cp	concatenation

Figure 2. Syntax of CSPs.

a technique for compiling CSPs to ASPs. To perform this compilation, Concretely requires some *black-box parser* of the object language. The compilation process we define below is directly based on Concretely; we will highlight the minor differences at the end of this section.

The compilation is split up into three phases.

1. *Encoding.* We encode our concrete syntax pattern into a string that the parser can parse. We do this by substituting each metavariable with some parseable string. For example, `5 + @<x : Expr>` can be encoded into `5 + 0`, where metavariable `x` encodes to `0`.
2. *Parsing.* We invoke our black-box parser to parse the encoded pattern into an AST. For example, `5 + 0` becomes `Plus(Int(5), Int(0))`.
3. *Decoding.* We decode all parts of the AST that originated from an encoded metavariable, resulting in an ASP. For example, `Plus(Int(5), Int(0))` becomes `Plus(Int(5), x)`. We identify whether AST nodes originated from a metavariable using *location information*.

ASTs. We define an AST a as some base term b (strings and numbers) or nodes $c(a_1, ..., a_n)$ where c represents a constructor name and $a_1, ..., a_n$ are the child nodes. AST nodes a are always annotated with *location information*, written as $a\langle l \rangle$ where l is a location (i, n). This is the location of the substring representing this AST node within the source string; i is the offset, and n is the length of the substring. For example, parsing the string `10+9` yields the AST

`Plus(Int(10)<(0,2)>,Int(9)<(3,1)>)<(0,4)>.`

We will use location information in the compiler's decoding stage to uniquely identify encoded metavariables.

Black-Box Parser. We interface over a black-box parser B with a tuple $(T, parse, gen, typeof)$. Here T is the set of types that AST nodes can have.[1] Next is the function $parse_B : T \times String \rightharpoonup AST$ which parses the input string and produces an AST of type T (or potentially fails). The resulting AST should be annotated with location information. The function $gen_B : T \rightarrow String$ generates some parseable string from the given type such that $parse(t, gen(t))$ will not fail. This function is used in the encoding phase of the compiler. The $typeof : AST \rightarrow T$ function yields the syntactic type of an

[1] Also known as a *sort* or *syntactic category*, e.g., `Expr` or `Decl`.

AST node based on its constructor. For example, constructors `Add` and `Mul` would have type `Expr`. We assume that that the types have no hierarchy.

Encoding. We describe the encoding process formally through an inductively defined relation $\downarrow$. We write $cp \downarrow^i s, d$ to denote that CSP cp encodes into string s. Index i tracks the current offset of s. The symbol d represents a *decoder* generated by the encoding, used in the decoding phase. A decoder is a set of mappings $l \rightarrow cp$ from locations to CSPs.

$$(\downarrow\text{-string})\frac{}{s \downarrow^i s, \emptyset}$$

$$(\downarrow\text{-metavar})\frac{s = gen_B(t)}{\texttt{@<}x : t\texttt{>}\downarrow^i s, \{(i, |s|) \rightarrow \texttt{@<}x : t\texttt{>}\}}$$

$$(\downarrow\text{-conc})\frac{cp_1 \downarrow^i s_1, d_1 \qquad cp_2 \downarrow^{i+|s_1|} s_2, d_2}{cp_1\ cp_2 \downarrow^i s_1\ s_2, d_1 \cup d_2}$$

When encoding a metavariable `@<`$x : t$`>`, we use the gen_B function to generate an encoding. This creates a corresponding decoder from the current location to the metavariable. For example, encoding the CSP `5+@<x:Expr>` yields string `5+0` (if we define gen_B(`Expr`) = `0`) and decoder $\{(2, 1) \rightarrow$ `@<`$x : t$`>`$\}$

Decoding. We describe the decoding process formally through an inductively defined relation $\uparrow$. We write $a, d \uparrow p$ to denote that AST a decodes into ASP p using decoder d. Each mapping in d has to be used *once*, which ensures that every metavariable that has been encoded will be decoded.

$$(\uparrow\text{-base})\frac{}{b\texttt{<}l\texttt{>}, \emptyset \uparrow b}$$

$$(\uparrow\text{-metavar})\frac{typeof_B(a) = t}{a\texttt{<}l\texttt{>}, \{l \rightarrow \texttt{@<}x : t\texttt{>}\} \uparrow x}$$

$$(\uparrow\text{-nomatch})\frac{a_1, d_1 \uparrow p_1 \qquad ... \qquad a_n, d_n \uparrow p_n}{c(a_1, ..., a_n)\texttt{<}l\texttt{>}, d_1 \cup ... \cup d_n \uparrow c(p_1, ..., p_n)}$$

The rule $\uparrow$-metavar is applied when we have an AST node with a location found in the decoder, meaning that this location likely originated from an encoded metavariable and we have to decode it. Rules $\uparrow$-base and $\uparrow$-metavar require the decoder to be empty and a singleton set respectively, which enforces our requirement that each mapping in d has to be used once. For the inductive case $\uparrow$-nomatch, we split the decoder into n parts $d_1 \cup ... \cup d_n$ to enforce this requirement.

The $\uparrow$-metavar rule has a premise that requires the type of the AST node to match the type of the metavariable. This solves a problem where location information is not a *unique* attribute of AST nodes; multiple AST nodes can have the same location. This is the case when there are *injections* in the language. For example, in many imperative languages, declarations like `int x = 0;` are also statements. Parsing this as a statement might give the AST `DeclStmt(Init(...))` where both the `DeclStmt` and `Init` nodes have the same location. If this location is in the decoder, which node should be decoded? To resolve this ambiguity, we require the type of the AST node we are decoding to match the type of the encoded metavariable. We assume that the combination of location and type *is* unique for each node.

Example: Let `Plus(Int(5)<(0,1)>,Int(0)<(2,1)>)<(0,3)>` be an AST obtained by parsing an encoded string. Let $\{(2, 1)$ $\rightarrow$ `@<x : Expr>`$\}$ be the corresponding decoder. Decoding yields the ASP `Plus(Int(5), x)`.

Compiler. Finally, we describe the compilation of CSPs to ASPs formally through a relation $\Rightarrow$. We write $t, cp \Rightarrow ap$ if CSP cp of type t compiles to ASP p.

$$(\text{CSP2ASP})\frac{cp \downarrow^0 s, d \qquad parse(t, s), d \uparrow p}{t, cp \Rightarrow p}$$

Note that the top-level type t of the pattern is part of the input that should be provided by the user.

Difference with Concretely. The key change we made to the work by Aarssen et al. [1] is that we use location information to uniquely identify AST nodes, whereas Concretely requires the encoded string from gen_B to be unique itself. We opted to use location information as it provides a stronger guarantee for uniqueness, as also remarked upon by Aarssen et al.'s discussion of their own work. Locations will also be necessary when inferring the type of certain metapatterns, as will be shown in later sections. A downside of our approach is that we limit ourselves to external parsers that are able to provide this location information. Other than location information, the compilation process presented in this section is analogous to their implementation.

4 Context-Free Disjunctive Patterns

We will extend our compiler from Section 3 to also support *disjunctive patterns* as introduced in Section 2. Disjunctive patterns reduce duplicate code and makes it more concise. For example, if we want to match function calls to both `foo` and `bar`, without using disjunctive patterns, we could write the following (pseudo)code:

```
  match(a, `foo(@<args : ExprList>)`)
| match(a, `bar(@<args : ExprList>)`)
```

Here `match` is a function that takes an AST node `a` and returns `true` if it matches the CSP on the right (delimited using backticks). The two calls to `match` are mostly duplicate code.

Alternatively, we could have written the following:

```
match(a, `@<n : Name>(@<args : ExprList>)`)
& (match(n, `foo`) | match(n, `bar`))
```

This uses one pattern to capture the name of the function call, then performs matches on the name separately. While this removes the duplicate code, the pattern itself became less

readable. Instead, if we inline the disjunction in the pattern itself using some operator `@+`, it becomes more concise:

```
match(a, `<foo@+bar>(@<args : ExprList>)`)
```

The usage of the operator indicates we are matching on both `foo` and `bar`. We call this the *context-free disjunction*. This disjunction assumes that the disjuncts can be parsed independently, i.e., for cp_1 `@+` cp_2 both cp_1 and cp_2 must be compilable CSPs. In Section 5 we will also introduce the *context-dependent disjunction*, which uses the `@|` operator, where disjuncts need not be independently compilable.

We will show two approaches for compiling the context-free disjunction; the first using *type hints* (Section 4.1 and the second through *type inference* (Section 4.2).

4.1 Context-Free Disjunction with Type Hints

The first approach to compiling the context-free disjunction is similar to how we compile metavariables in Section 3. We let the user annotate the pattern with its syntactic type t and then call $gen_B(t)$ to encode this pattern. We extend the syntax of CSMPs to include this disjunction: $cp ::= cp$ `@+` cp `@:` t. Reusing our previous example, it would look as follows: `<foo @+ bar @: Name>(@<args : ExprList>)`.[2]

We extend our encoding $\downarrow$ as follows:

$$(\downarrow\text{-cf-disj})\ \frac{s = gen_B(t)}{cp_1 \texttt{ @+ } cp_2 \texttt{ @: } t \downarrow^i s, \{(i, |s|) \to cp_1 \texttt{ @+ } cp_2 \texttt{ @: } t\}}$$

Similarly, the decoding $\uparrow$ is extended with the following rule:

$$(\uparrow\text{-cf-disj})\ \frac{typeof_B(a) = t \qquad t, cp_1 \Rightarrow p_1 \qquad t, cp_2 \Rightarrow p_2}{a\texttt{<}l\texttt{>}, \{l \to cp_1 \texttt{ @+ } cp_2 \texttt{ @: } t\} \uparrow p_1 | p_2}$$

Like $\uparrow$-metavar, this rule is applied when the location of the AST node is in the decoder and the type of the node matches the provided type hint. When this rule is applied, the left- and right-disjuncts of the pattern get compiled recursively.

4.2 Inferred Context-Free Disjunction

Requiring the user to annotate each disjunction with type hints makes the CSMP less readable and puts effort on the user to know this type. The type hint is required to generate a valid encoding for the pattern, but we can alternatively generate some encoding by encoding either the left- or right-disjunct. This means we can drop this type entirely. The syntax for this new *inferred* context-free disjunction is $cp ::= cp_1$ `@+` cp_2, which excludes the type.

Before extending our encoder $\downarrow$, we introduce a new function to our black-box parser interface: $trimlayout_B : String \to String \times \mathbb{N}$, which trims whitespace and other layout information (like comments) from the input string depending on the object language. In the result (s', k), s' represents the trimmed string, and k is the amount of characters that have been trimmed on the left-hand side of the string. We will need this to provide accurate location information to the decoder, which we will explain in a later example.

We extend our encoding $\downarrow$ as follows:

$$(\downarrow\text{-icf-disj})\ \frac{cp_1 \downarrow^i s, \_ \qquad trimlayout_B(s) = s', k}{cp_1 \texttt{ @+ } cp_2 \downarrow^i s, \{(i + k, |s'|) \to cp_1 \texttt{ @+ } cp_2\}}$$

This rule uses the left-disjunct cp_1 to generate an encoding.[3]

The following example shows why we need *trimlayout*:

```
</*hello*/ 5 - @<x : Expr>> @+ 5
```

This generates the encoding `/*hello*/ 5 - 0`. After parsing, we get the AST `Sub(Int(5), Int(0))`, which will be annotated with location $(10, 5)$ and only covers the substring `5 - 0` of our encoding. The layout `/*hello*/` gets ignored. If our encoder would not use the trimmed encoding to compute the location information, the decoder would include location $(0, 15)$ and the compiler will fail to decode the parsed AST.

Now that we have extended the encoder, we also add the following rule to the decoder:

$$(\uparrow\text{-icf-disj})\ \frac{typeof_B(a) = t \qquad t, cp_1 \Rightarrow p_1 \qquad t, cp_2 \Rightarrow p_2}{a\texttt{<}l\texttt{>}, \{l \to cp_1 \texttt{ @+ } cp_2\} \uparrow p_1 | p_2}$$

Notice that we now infer the type t by retrieving the type of the corresponding AST node.

Location ambiguity. This relational definition of the decoding now holds an ambiguity if two AST nodes have the same location. Consider the following line of code that is both a statement and a declaration if in the context of a function body: `int x = 0;`. This gives an AST of the form `DeclStmt(Init(...))`, where the outer `DeclStmt` node has type `Stmt`, and the inner `Init` node has type `Decl`. Both nodes hold the same location. We used to disambiguate this by checking whether the type of the AST node matches the given type hint, but we no longer have a type hint.

The solution is to decode the *outermost* node, as it represents the "most-general" type. For example, in the CSMP `<int x = 0;> @+ <println(0);>`, both disjuncts have type `Stmt`, but only the left-disjunct has type `Decl`. If we infer the type of this disjunct using the inner node (`Decl`), the compilation of the right-hand disjunct fails.

5 Context-Dependent Disjunction

Using the context-free disjunction, suppose we have a CSMP for C++ `<class @+ struct> Foo {};`, matching classes and structs. Compiling this might fail for two reasons: Keywords `class` or `struct` are not represented as individual nodes in the AST, or the black-box parser might not provide locations for these nodes. For example, the Eclipse CDT parser for C/C++ [13] does not retain locations for keywords.

If we allowed disjunctions to be *context-dependent*, we would have support for such cases. A context-dependent

[2] To avoid ambiguities, we use angled brackets to delimit blocks of code.

[3] Using cp_2 would also have worked to generate an encoding.

disjunction cp_1 `@|` cp_2 wraps around the entire outer CSMP. We can compare it to the context-free disjunction as follows:

$$C[cp_1 \texttt{ @| } cp_2] = C[cp_1] \texttt{ @+ } C[cp_2],$$

where $C[\dots]$ represents the entire context.

In Section 5.1 we extend the compiler to support context-dependent disjunctions. This will show that these disjunctions incur a significant performance cost. In Section 5.2 we discuss this cost and the trade-offs between the context-free and context-dependent disjunction.

5.1 Compiling Context-Dependent Disjunction

We extend the syntax of CSMPs with the context-dependent disjunction: $cp ::= cp$ `@|` cp. When compiling this disjunction, we must consider the *entire context* surrounding the disjunction. Compiling CSMP `<class @| struct> Foo {};` means compiling `class Foo {};` and `struct Foo {};` separately, then merging the resulting ASMPs into a disjunctive pattern. If we have multiple context-dependent disjunctions like in `<a @| b><c @| d>`, we would have to consider all combinations `ac`, `ad`, `bc`, and `bd`. This causes an exponential blowup resulting in 2^n combinations, where n represents the amount of disjunctions. This assumes a worst-case scenario where all disjunctions are disjoint (not nested in one another). We will discuss the implication of this in Section 9.2.

Updating the Encoder. For some CSMP cp, our encoder will no longer yield one encoding and decoder, but a set of encoding results E, where in a result $(s, d) \in E$, the encoding is represented by s and the decoder by d. We update most previous rules (↓-string, ↓-metavar, ↓-cf-disj, and ↓-icf-disj) by yielding a singleton set. To illustrate, we give the updated ↓-string rule, the other rules would be updated analogously:

$$(\downarrow\text{-string})\ \frac{}{s \downarrow^i \{(s, \emptyset)\}}$$

When encoding context-dependent disjunctions, we recursively encode both disjuncts and take the union of the encoding results. For example, encoding `a @| b` yields two encoding results $\{(\texttt{a}, \emptyset), (\texttt{b}, \emptyset)\}$.

$$(\downarrow\text{-cd-disj})\ \frac{cp_1 \downarrow^i E_1 \qquad cp_2 \downarrow^i E_2}{cp_1 \texttt{ @| } cp_2 \downarrow^i E_1 \cup E_2}$$

The ↓-conc rule needs to consider *all* possible combinations of the left- and right-hand sides. For example, `<a @| b>c` yields two encoding results $\{(\texttt{ac}, \emptyset), (\texttt{bc}, \emptyset)\}$.

$$(\downarrow\text{-conc})\ \frac{\begin{array}{c} cp_1 \downarrow^i E_1 \\ E = \{(s_1\ s_2, d_1 \cup d_2) \mid (s_1, d_1) \in E_1, (s_2, d_2) \in E_2 \\ \text{where } cp_2 \downarrow^{i+|s_1|} E_2\} \end{array}}{cp_1\ cp_2 \downarrow^i E}$$

Note that there is not a single set of encoding results E_2 of cp_2, but that we have one for each encoding in E_1.

Finally, we update our top-level compilation rule to construct a top-level disjunctive pattern from all decodings:

$$(\text{CSP2ASP})\ \frac{\begin{array}{c} cp \downarrow^0 E \\ P = \{p \mid (s, d) \in E,\ parse(t, s), d \uparrow p\} \\ P = \{p_1, \dots, p_n\} \end{array}}{t, cp \Rightarrow p_1 | \dots | p_n}$$

5.2 Comparison with Context-Free Disjunction

There seems to be a clear trade-off between the two disjunctive variants. The context-dependent disjunction has no restrictions on where in the CSMP it can be used. This has clear benefit in that the user can write shorter and more concise patterns that internally get compiled to complex ASMPs, but also that the user requires no knowledge of the underlying AST structure. However, there are obvious performance drawbacks: This disjunction causes an exponential blowup in compilation time depending on the amount of disjunctions in the pattern, whereas the context-free disjunction remains linear. The amount of top-level disjunctions in the output pattern also rises exponentially, which in turn slows down the matching of the ASMP against an AST.

Furthermore, even when ignoring performance, there is a slight semantic difference between the two disjunctions. This is where the terms *context-free* and *context-dependent* come into play. The former will recursively compile its disjuncts, regardless of context surrounding the disjunction, whereas context-dependent disjunctions do take the context into account. This difference becomes visible when considering an example that involves *operator precedence*.

Consider the two CSMPs `<1 @+ 1+2>*3` and `<1 @| 1+2>*3`. The former will first be encoded into some string `0*3`, which gets parsed and decoded into

```
Mul(<1 @+ 1+2>, Int(3)).
```

Both disjuncts will then be recursively compiled into

```
Mul(Int(1) | Add(Int(1), Int(2)), Int(3)).
```

The latter will be encoded into two strings `1*3` and `1+2*3`, both parsed and decoded separately. This will yield the ASMP

```
Mul(Int(1),Int(3)) | Add(Int(1), Mul(Int(2),Int(3))).
```

The two ASMPs are different because the context-dependent disjunction took the context surrounding the disjunction into account, where operator precedence comes into play. Conversely, the context-free disjunction sees no such behavior; as if we had placed brackets around disjunction.

This phenomenon also causes some problems for the *inferred* context-free disjunction where the inference will fail. Suppose we have CSMP `<1+2 @+ 1>*3`. Because our encoding rules specify that we take the left-disjunct for encoding, the encoding will be `1+2*3`. Because multiplication takes priority, the substring `1+2` will not have a corresponding node in the AST after parsing. This means that we will be unable to decode the AST and the compilation will fail.

6 Descendant Pattern

The *descendant pattern* is used to inline deep AST traversals into patterns. Consider an example in pseudocode where we wish to find all recursive calls of some function `foo`:

```
match(a, `fun foo() {
            @<body : StmtList>
          }`)
for (n := traverse(body)):
    match(n, `foo()`)
    yield n
```

Instead, with a descendant pattern we would get a more concise pattern as follows:

```
match(a, `fun foo() {
            @<...Expr foo() ...>
          }`)
```

In a similar process to how we introduced the compilation of the context-free disjunction in Section 4, we will first show the compilation of the descendant pattern with type hints (Section 6.1) and then show how we can remove the type hint and infer it instead (Section 6.2).

6.1 Descendant Pattern with Type Hints

We extend the syntax of CSMPs to include the descendant pattern with $cp ::= \texttt{@<}t_1 \ldots t_2\ cp \ldots\texttt{>}$. This pattern requires the programmer to provide two types. Type t_1, the *outer* type, represents the type of the entire metapattern. In the example from the beginning of this section, this would be `StmtList` as the body of a function is represented as a list of statements. The second type t_2 is the *inner* type, which represents the type of the inner metapattern. This would be an `Expr` in the example, as function call `foo()` is an expression.

We need the outer type to encode the entire descendant pattern, and the inner type for recursively compiling the inner pattern. Extending the encoding $\downarrow$ and decoding $\uparrow$ relations follows a similar process to metavariables and the context-free disjunctions with type hints.

$$(\downarrow\text{-desc})\ \frac{s = gen_B(t_1)}{\texttt{@<}t_1 \ldots t_2\ cp_1 \ldots\texttt{>}\downarrow^i \quad \{(s, \{(i, |s|) \rightarrow \texttt{@<}t_1 \ldots t_2\ cp_1 \ldots\texttt{>}\})\}}$$

$$(\uparrow\text{-desc})\ \frac{typeof_B(a) = t_1 \qquad t_2, cp_1 \Rightarrow p}{a\texttt{<l>}, \{l \rightarrow \texttt{@<}t_1 \ldots t_2\ cp_1 \ldots\texttt{>}\} \uparrow /p}$$

The ASMP $/p$ is a descendant pattern as in Section 2 [14].

6.2 Inferred Descendant Pattern

The type hint approach requires the user to annotate each descendant pattern with *two* type hints, so we would like to infer these types if possible. Unfortunately, inference is not as feasible here as we would like.

In particular, the inner type t_2 might be *any* type, because the inner pattern cp_1 will be matched against *all* descendants of the AST node we are matching, which can take on a wide range of types. We can try to brute force the compilation of cp_1 by attempting all possible values for t_2. Apart from the obvious performance drawbacks, this approach is also highly ambiguous. For example, in `@<... x ...>`, the inner pattern `x` could have type `Expr`, but also type `Name`.

Similarly, there exists no principled method to infer the outer type t_1. However, *guessing* its type is not as infeasible as with guessing the inner type t_2. Firstly, we do not really need to guess the type itself, we only need to generate some encoding that will successfully pass through the parser. Secondly, while ambiguities with guessing the outer type are still present, they are not as prominent as with the inner type as we shall discuss below. We will therefore design the *inferred* descendant pattern such that we still need to specify the inner type, but no longer need to know the outer type.

Guessing the outer type. We extend the syntax of CSMPs with the *inferred* descendant pattern $cp ::= \texttt{@<}\ldots t_2\ cp\ldots\texttt{>}$. To "guess" possible encodings for the pattern, we extend our black-box parser interface B with a new field $guess_B$; a set of strings representing possible encodings.

In our approach, we compile some CSMP cp by substituting each inferred descendant pattern with some $s \in guess_B$. If after the encoding phase of the compiler, the parsing phase fails, we re-run the encoder using some different substitution from $guess_B$. We extend our encoder and decoder as follows:

$$(\downarrow\text{-idesc})\ \frac{s \in guess_B}{\texttt{@<}\ldots t_2\ cp_1 \ldots\texttt{>}\downarrow^i \quad \{(s, \{(i, |s|) \rightarrow \texttt{@<}\ldots t_2\ cp_1 \ldots\texttt{>}\})\}}$$

$$(\uparrow\text{-desc})\ \frac{t_2, cp_1 \Rightarrow p}{a\texttt{<l>}, \{l \rightarrow \texttt{@<}\ldots t_2\ cp_1 \ldots\texttt{>}\} \uparrow /p}$$

The encoding rule is now explicitly ambiguous through the guess of $s \in guess_B$. An implementation of this should be able to generate all potential encodings and find at least one for which the parsing phase succeeds.

The amount of encodings blows up exponentially depending on the size of the $guess_B$ set and the amount of descendant patterns in our CSMP. If $n = |guess_B|$ and m is the amount of *disjoint* descendant patterns (not nested in one another), then we get $O(n^m)$ encodings. Nested descendant patterns have a linear increase, as they are compiled independently.

Determining the guess set. A complete $guess_B$ set should cover each type $t \in T$. We can generate this as follows: $guess_B = \{gen_B(t) \mid t \in T\}$. To increase performance, we can shorten the $guess_B$ set by limiting types that are supported. We can limit it those types that are commonly used for descendant patterns in the object language. This would make our inference less complete, but improve performance.

A second approach is to find encodings that cover multiple types. For example, in many C-like languages, the encoding `int x;` covers (sequences of) statements and (sequences of)

declarations. Similarly, `x` covers both names and expressions. These types are also some of the most common in C-like languages. Therefore, the set $guess_B$ = {`int x;` , `x`} would be quite a complete set, while only having a size of two.

Location ambiguity. Similar to the inferred disjunction, we consider ambiguities in the decoding phase if two AST nodes hold the same location. Suppose we have the CSMP `class Foo { @<...Expr "hello" ...> };` where `int x;` is a successful guess for the descendant pattern. This parses to[4]

```
Class("Foo", ClassBody([Field(...)]))
```

where nodes `ClassBody` and `Field` have the same location. Decoding this yields either one of the following patterns:

```
Class("Foo", /String("hello"))
Class("Foo", ClassBody([/String("hello")]))
```

There exists a large semantic difference between these two patterns: The first pattern finds all strings `"hello"` anywhere inside the body of class `Foo`. The second pattern will match class `Foo` only if the body has exactly one declaration, and then finds all strings `"hello"` in that declaration.

To resolve this ambiguity, we always decode the outermost AST node that matches the location, as this is always the "most-general" option. If we were to choose the inner AST node, our pattern might become too "dependent" on the guessed encoding. In the example above, nothing from our CSMP indicates that we are expecting the body of class `Foo` to be a single declaration. This was completely dependent on our choice of encoding `int x;`. It would have been entirely possible to make other guesses such as `int x; int y;` which would have resulted in a body with two declaration.

Multiple successful guesses. While we only want *one* guess for which the encoding successfully parses, how likely is it for two different guesses to be correct? If we assume that $guess_B$ has minimal overlap of types between the different encodings, we estimate these ambiguities to be rare.

If the ambiguous guesses have the same type after the parsing phase, their decoding will be equivalent and thus do not pose a problem. However, we can come up with some contrived examples that do pose a problem, which we will call *context ambiguities*. Consider some fictional language that allows us to print both expressions and statements, e.g., both `print(x)` and `print(x = 0;)` are allowed. If these two variants of printing are represented in the AST with different constructors (e.g., `PrintExpr` and `PrintStmt`), then we have a serious ambiguity if these are two possible encodings. We call this a context ambiguity because the difference lies not in the encoded descendant pattern, but in the context (parent constructor) of the descendant pattern. This would result in two possible decodings `PrintExpr(/p)` and `PrintStmt(/p)` (where `/p` is our descendant pattern).

A more likely and less contrived parser of such a language, where the this problem would not occur, would have a single constructor for both variants of the print statement. The argument given to this constructor would have information on whether this is a declaration or expression (e.g., `Print(PExpr(...))` and `Print(PStmt(...))`).

7 Patches

While pattern matching on ASTs is an important aspect of software restructuring, we should also be able to make transformations on those ASTs and *unparse* the result. One approach is to construct a set of *patches P*. A patch is a pair (l, s), where l is a location representing a part of the source code and s is a replacement string. Applying a patch means replacing the substring at the location with the replacement string. In pseudocode, this would look as follows:

```
P = {}
for (match(a, `class Foo {
                 @<...Expr
                   @<s : "hello">
                 ...>
               }`)):
  P += {(s.loc, `"world"`)}
applyPatches(a.source, P)
```

This transformation replaces each occurrence of the string literal `"hello"` in class `Foo` with `"world"`. We use a descendant pattern, iterating over every occurrence of the string `hello` and capturing this in a metavariable `s`.[5] For each occurrence we add a patch to replace `s` with `"world"`. This method of patching has been used on a larger scale restructuring by Schuts et al. [12] for migrating legacy C/C++ test code.

We can make patching more declarative by introducing a *patch pattern* that lets us inline this patch into the pattern:

```
P = match(a, `class Foo {
                @<...Expr
                  "hello" --> "world"
                ...>
              }`)
applyPatches(a.source, P)
```

We extend the syntax of CSMPs to include the patch pattern with $cp ::= cp_1 \longrightarrow cp_2$.[6] The right-hand side pattern cp_2 may only contain strings, (captured) metavariables, and concatenations of those. We could compile this pattern using type hints. However, similar to the context-free disjunction, we can infer this by recursively encoding cp_1.

$$(\downarrow\text{-patch})\frac{cp_1 \downarrow^i s, \_ \qquad trimlayout_B(s) = s', k}{cp_1 \longrightarrow cp_2 \downarrow^i \{(s, \{(i+k, |s'|) \rightarrow cp_1 \longrightarrow cp_2\})\}}$$

$$(\uparrow\text{-patch})\frac{typeof_B(a\texttt{<}l\texttt{>}) = t \qquad t, cp_1 \Rightarrow p_1 \qquad t, cp_2 \Rightarrow \_}{a\texttt{<}l\texttt{>}, \{l \rightarrow cp_1 \longrightarrow cp_2\} \uparrow p_1 \longrightarrow cp_2}$$

[4] We use the notation $[a_1, ..., a_n]$ to represents lists in the AST.

[5] A pattern `@<`x `:` p`>` captures the node that p matches into metavariable x.

[6] In code listings we will denote the arrow with `-->`.

We have seen most of the ideas behind this method in Section 4.2 on the inferred disjunctive pattern. One surprise in the ↑-patch rule is that the output ASMP uses the concrete-syntax pattern cp_2 as its right-hand side, whereas we originally defined patches in Section 2 to be fully abstract syntax. The advantage that we get here is that because cp_2 uses *concrete* syntax, there is no need for an unparser of our language, which allows us to keep the black-box parser interface simple. The compilation of cp_2 pattern is unnecessary, as we can directly construct a replacement string by substituting each metavariable in cp_2 for its assigned value, then concatenating each part together. The premise $t, cp_2 \Rightarrow \_$ exists only to ensure that cp_2 has the same type as cp_1.

Similar to other metapatterns we have seen before, the decoding is ambiguous if two AST nodes have the same location. For reasons equal to the inferred descendant pattern (Section 6.2), we always decode the outermost AST node.

8 Implementation

Based on the methods presented in this paper, we implemented a compiler from CSMPs to ASMPs in Kotlin and applied it using a black-box parser for C++ based on Eclipse CDT [13]. The compiler supports the main metapatterns introduced in the previous sections. That is, both variants of the disjunction, the descendant pattern, and patches.

Listing 2 shows an example restructuring implemented using this library. We introduced this example before in the introduction (Listing 1). This example restructuring ensures that logged messages in the code get prefixed with the name of the class or struct they are contained in.

We will go through the example from Listing 2 on a step-by-step bases. On line 2, we initialize the C++ parser. This `CDTParser` uses Eclipse CDT internally and implements our black-box parser interface `BBP`. On the next line, we initialize our CSMP compiler using this black-box parser. The `CDTType` class represents the syntactic types of C++.

We create our CSMP on lines 5-12, represented as a multiline string. Using the context-dependent disjunction `@|` we match on both classes and structs. By default, these meta-operators are delimited by whitespace; we can optionally delimit blocks of code with double-angled brackets << ...>>. In the body of the class or struct, we have a descendant pattern where we search for expressions. Using the context-free disjunction `@+`, we specify that either we want to match on `logger::log(@msg)`, or `logger::log(@level, @msg)`. On a successful match, we patch these messages so that they are prefixed with the name of the class or struct. Note that the patch operator `-->` takes precedence over the `@+` operator.

On lines 13-15 we specify the syntactic type hints for each metavariable in the pattern. Inspired by Coccinelle [9], we do this outside of the pattern to leave the pattern itself cleaner, and to avoid duplicate type hints if a metavariable is used more than once in the pattern.

We are now able to compile our pattern on line 16. The compiler also needs to know the top-level syntactic type of the pattern, which in this case is a `TRANSLATIONUNIT`. The compiled pattern is an `ASMP` (*abstract* syntax metapattern).

After we obtain the AST of our source code that we wish to restructure (lines 19-21), we use the `match` function to match this input AST against our ASMP (line 23). The output of `match` is an iterable of `MatchResults`. A single `MatchResult` contains a substitution (a map from metavariables to the AST node they matched), and patches (a map from locations to replacement strings). Because we used a descendant pattern, we will get a number of matchresults depending on the amount of `logger::log` calls found.

Finally, on line 24, we apply all patches on the source code using the `patchAll` function. This function includes checks that the locations we are patching are not accidentally conflicting. As we only patch those locations in the source string that are subject to replacement, all layout in other parts of the source code get preserved. Layout preservation has been a prominent topic in research, as a traditional way of obtaining restructured code is by unparsing the AST at the cost of losing layout [2, 4, 16].

9 Evaluation and Discussion

In this section we will evaluate and discuss various aspects of CSMPs and our method of compilation. In Section 9.1 we discuss the effort it would require to support new metapatterns save the ones discussed in this paper. We analyze and discuss the exponential blowup problem in Section 9.2. In Section 9.3, we evaluate the expressivity of CSMPs. Finally, we discuss the effort required to implement the black-box parser interface in Section 9.4.

9.1 Supporting New Metapatterns

The metapatterns our implementation currently supports are not an exhaustive set of metapatterns. However, we do argue that it takes little effort to extend the compiler to support new metapatterns, as we can reuse the techniques already developed for the previous metapatterns. We have seen four different compilation techniques.

(1) *Type Hints.* If we let the user annotate a CSMP with a type hint, we can use the gen_B function to encode this CSMP, then use location information to decode this metapattern. We applied this technique on metavariables, and a variant of the context-free disjunction and descendant pattern.

(2) *Simple Inference.* Some metapatterns have at least one direct child that can be used to encode the pattern. For example, we can use either the encoding of p_1 or p_2 to encode p_1 `@+` p_2. Similarly, we use p_1 to encode patches $p_1 \longrightarrow p_2$.

(3) *Guess Inference.* If such a direct child is not present, like with the descendant pattern, we can instead try to guess the encoding through a $guess_B$ set. This however should be used sparingly as it comes at the cost of an exponential blowup.

```
fun main() {
  val blackBoxParser: BBP<CDTType> = CDTParser()
  val compiler: CSMPCompiler<CDTType> = CSMPCompiler(blackBoxParser)

  val pattern = """
    class@|struct @name {
        @<...EXPRESSION
             <<logger::log(@msg)>> --> <<logger::log("@name: " + @msg)>>
          @+ <<logger::log(@level, @msg)>> --> <<logger::log(@level, "@name: " + @msg))>>
        ...>
    };
  """
  val metaVars: Map<String, CDTType> = mapOf("name" to NAME.get(),
                                             "msg" to EXPRESSION.get(),
                                             "level" to EXPRESSION.get())
  val compiledPattern: ASMP = compiler.compile(TRANSLATIONUNIT.get(), pattern, metaVars)
    ?: throw RuntimeException("Could not compile pattern")

  val source: String = /* source code (string) we wish to refactor */
  val sourceAST: AST = blackBoxParser.parse(TRANSLATIONUNIT.get(), source)
    ?: throw RuntimeException("Could not parse source")

  val matchResults: Iterable<MatchResult> = match(compiledPattern, sourceAST, source)
  val restructuredCode: String = patchAll(source, matchResults)

  println("Found ${matchResults.toList().size} match(es).")
  println("Restructured code:\n $restructuredCode")
}
```

Listing 2. Example refactoring for C++ using the Kotlin implementation of concrete syntax metapatterns. The refactoring ensures that logged messages in the code get prefixed with the name of the class or struct they are contained in.

(4) *"Wild" Metapatterns.* A *wild* CSMP does not follow the typical encoding and decoding scheme. Instead, we use it to generate multiple metapatterns that can all be compiled individually. Then they are merged together into one large abstract (disjunctive) pattern. The context-dependent disjunction is the prime example of this. Wild metapatterns also come at the risk of an exponential blowup.

To illustrate the reusability of these four methods, we will consider two more metapatterns we have not yet discussed: list patterns and permutation patterns. List patterns, or sequence patterns, are patterns that specifically match on (sub)lists in the abstract syntax tree. For example, the ASMP `[s1*, Print(Str("s")), s2*]` will match lists containing a statement printing `"s"`, where `s1` and `s2` are sequence metavariables that match on sublists. Using concrete syntax, we could rewrite this to `@s1* print("s"); @s2*`. To encode this, we can use the type hint approach by specifying that this pattern represents a `StmtList`. However, a cleaner approach would be to use *simple inference* by encoding any of the child patterns; in this case `print("s");` would be a valid encoding for the CSMP.

Permutation patterns, or set patterns, are patterns that match on sets; this can also be interpreted as matching on lists where the order is irrelevant. One example use case is for languages that support passing keyword arguments to function calls (e.g., `foo(a = 0, b = 1)` in Python). Because the order of arguments is irrelevant, the permutation pattern `{KwArg("a", Int(0)), KwArg("b", Int(1))}` would match all desired cases. An argument can be made for designing these in concrete syntax such that we have a context-free and context-dependent variant like how we designed the disjunctive pattern. In C++ for instance, the order of modifiers added to function declarations is often irrelevant; both `virtual void foo();` and `void virtual foo();` are equivalent. If the keywords have no location information in the AST, we would need a *wild* (context-dependent) variant of the permutation pattern. E.g., `@{public virtual void} foo();` would generate six encodings; one for each permutation.

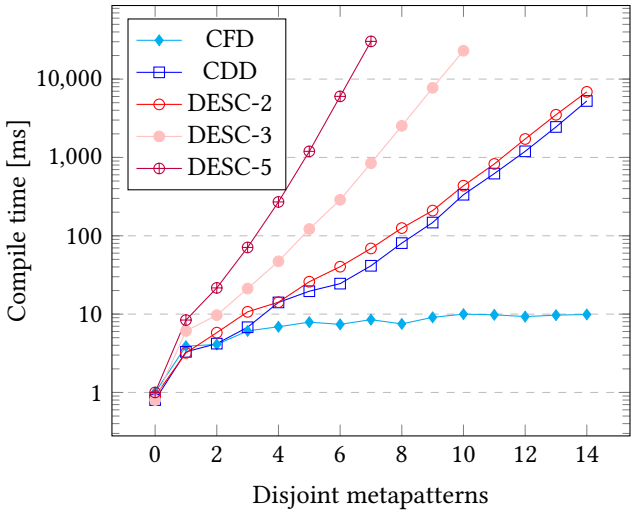

Figure 3. Runtime performance of compiling a CSMP measured against the number of disjoint metapatterns.

9.2 Performance of Exponential CSMPs

In the sections on context-dependent disjunctions (5) and the inferred descendant pattern (6.2), we predicted an exponential increase in the compile-time performance based on the number of these patterns present. To quantify at which point this blowup becomes problematic, we ran some simple experiments and visualized their results in Figure 3.

In the experiments, we ran the compiler with a black-box parser for C++ on CSMPs with varying amounts of *disjoint* context-dependent disjunctive patterns or descendant patterns. The results displayed in Figure 3 were obtained by compiling each pattern at least ten times (for a minimum of one second), and taking the average of those runs. The `CFD` and `CDD` graphs respectively plot the number of context-free and context-dependent disjunctions against the compile time. The `DESC-`*n* graphs plot number of descendant patterns, where *n* refers to the size of the $guess_B$ set. The CSMPs are artificially constructed as follows: For `CDD` we used

```
int foo() {
  return 0
  + 0 @| 1  +  0 @| 1  ... ;
}
```

with increasing number of disjunctions. `CFD` follows the same pattern, replacing the `@|` operator with the context-free version `@+`. Similarly, for `DESC-`*n* we used the pattern

```
int foo() {
  return 0
  + @<...Expr 0...> + @<...Expr 0...> ... ;
}
```

with increasing number of descendant patterns. We designed the $guess_B$ set for a worst-case scenario where we force the compiler to try every guess before finding the correct one.

Ignoring the context-free disjunction, the results clearly demonstrate an exponential blowup. Even at low amounts, we reach compile times that could be considered problematic. For example, 8 context-dependent disjunctions give a compile time of roughly 100 ms, and 12 disjunctions give a compile time of one second. As a comparison, for the context-free disjunction `@+` it took over 70 operators to reach this 100 ms average (not visible in the graph).

From these results we infer that context-dependent disjunctions and the inferred descendant patterns should be used sparingly. Luckily, in most practical cases we can actually use the context-free disjunction instead of the context-dependent disjunction. We require the context-dependent disjunction only in those cases where the disjuncts will not be represented in the AST (with location information). To further defend the context-dependent disjunction, we must also take into account that the expressivity it provides is impossible to obtain with ordinary concrete syntax patterns. There is no generalizable alternative to the context-dependent disjunction that does not involve a combinatorial explosion.

For descendant patterns, we conjecture that it is rare in practice to use many disjoint amounts in one CSMP. Furthermore, the performance blowup can be avoided with type hints or by splitting up the pattern into smaller patterns.

9.3 Expressivity of CSMPs

To evaluate the expressivity of CSMPs, we looked at an existing C++ refactoring by Schuts et al. [12] and partially rewrote their refactoring to instead use CSMPs and our Kotlin tool. The original refactoring, implemented in Rascal and using abstract syntax patterns, migrates C/C++ test code from the STX testing framework to the GoogleTest framework. We only modified the code snippets which involved C++ (the original refactoring also targets C and CMake). We also ignored C++ preprocessing directives. We have successfully rewritten the five relevant code listings from their paper, going from 116 source lines of code (SLOC), to 92.

We utilized all CSMPs introduced in our paper, which reduced often reduced the SLOC. In particular, the original refactoring manually built and kept track of a *change set* to patch the code, which we now replace with the *patch pattern*.

A limitation of our Kotlin tool is that it requires verbose solutions for any task where our CSMPs are not expressive enough, such as checking equality between two metavariables from different patterns. We have to *manually* get the substitution of a match and retrieve the metavariable from this substitution. These cases resulted in a higher SLOC count. To avoid this in future work, the CSMP language should be embedded into a proper DSL for metaprogramming (such as Rascal).

Another downside of using CSMPs involves *keywords*. The original refactoring could use holes _ in the pattern, abstracting over keywords. As mentioned in Section 5, sometimes keywords lack location information and can not be replaced

with metavariables in concrete syntax. This means that the CSMP will either be verbose or not as generic as the abstract counterpart. Using the context-dependent disjunction we mitigate some of this. For example, with `static @| /*...*/` we *optionally* match on the static keyword.

9.4 Effort in Implementing the Black-Box Parser

By utilizing external black-box parsers, our approach is generalizable to many languages. But what is the effort required to implement this black-box parser interface? We reaffirm the conclusion by Aarssen et al. [1] that the work load is most prominent in *marshalling* of the native ASTs from the external parser to our internal representation of ASTs in Kotlin, which is tedious and error-prone to write manually. In comparison, the other elements of the black-box parser interface (*gen*, *typeof*, *trimlayout*, and *guess*) need little effort.

10 Related Work

Previous work on supporting concrete syntax patterns in a language-parametric setting often relies on the existence of a syntax specification of the language in some formalism. The ASF+SDF system [6, 15] supports rewrite rules, where the syntax and parser of these rules is dynamically constructed based on the syntax definition of the object language and a user-defined syntax for metavariables, giving rise to concrete syntax patterns. Both Spoofax [5, 17] and Rascal [7] support concrete syntax patterns using a similar method. Aarssen et al. [1] remark that this approach, which requires a syntax specification for the object language in the respective system, is costly. As an alternative, they present Concretely, a technique for using external black-box parsers of the object language to support concrete syntax patterns. Our compiler for CSMPs is an extension to this technique, with the difference that we use location information to decode metavariables rather than assuming that encodings are unique.

Additionally, Aarssen et al. [1] introduced Tympanic, a DSL for mapping Java class hierarchies to Rascal's algebraic data types. This tool significantly reduces the effort required to write mapping from ASTs generated by external black-box parser (that are Java-based) to their internal definition of ASTs. We have not created such a DSL for our Kotlin library.

Abstract metapatterns supported by Rascal include regular expressions for matching strings, multi-variable patterns for sequences, and the descendant pattern [14]. However, the support for these metapatterns in concrete syntax is limited.

The C program transformation tool Coccinelle [9], applied extensively on Linux Kernel code [8], uses a semantic patch language SmPL which lets users express transformation using concrete syntax with line-based patch patterns. On a successful match, lines in the pattern prefixed with - will be removed, and lines prefixed with + are added. Our definition of a patch pattern uses an arrow --> instead. SmPL also introduces the ellipsis "..." metapattern, which matches any control-flow path from a term matching the pattern before the "..." to a term matching the pattern after the "...". Their other metapatterns include disjunctions `|`, optionals `?`, and conjunctions `&`. We consider Coccinelle to be a quintessential example of a restructuring tool with fundamental support for CSMPs. Its widespread use is an inspiration for our work, which lays a groundwork for generalizing tools such as Coccinelle to any programming language.

11 Conclusion

Concrete syntax metapatterns (CSMPs) are patterns written using the concrete syntax of some object language, and contain logic that is more expressive than regular patterns with metavariables and holes, which allows for more declarative implementations of software restructurings. In this paper, we introduced techniques to compile CSMPs to *abstract syntax patterns*. Extending the work by Aarssen et al. [1], these methods require some external black-box parser of the object language, making the compiler scalable to multiple languages. Specifically, we defined a compiler for disjunctions, descendant patterns, and patches.

To spare users from annotating each CSMP with syntactic type annotations, we employed methods to infer the type instead (through simple or guess inference). Furthermore, for the disjunction we support a context-dependent variant which can be used at any position in the code, regardless of whether it represents a singular AST node or not. Some of these methods have an exponential blowup in runtime performance, however, we conjecture that in practical cases, the amount of these "exponential CSMPs" required is not significant enough to cause real problems.

We implemented our compiler as a library in Kotlin, together with some infrastructure to implement restructurings. While our implementation comes with support for C++, supporting new languages requires an implementation of the black-box parser interface. To extend the compiler with new CSMPs, we argued that we can mostly reuse the methods presented for the existing CSMPs.

Directions for future work include the embedding of CSMPs in a metaprogramming DSL, finding optimization techniques in the compilation process, and a study on the usability of CSMPs and their applicability in practice.

Data-Availability Statement

The software artifact associated with this paper is available at DOI 10.5281/zenodo.13741142 [10]. It includes the Kotlin tool from Section 8 and data related to the evaluation.

Acknowledgments

We would like to thank the anonymous reviewers for their thorough feedback. We thank Jurgen Vinju for his involvement. This work is supported by the Programming and Validating Restructurings project (17933, NWO-TTW, MasCot).

References

[1] Rodin Aarssen, Jurgen J. Vinju, and Tijs van der Storm. 2019. Concrete Syntax with Black Box Parsers. *Art Sci. Eng. Program.* 3, 3 (2019), 15. https://doi.org/10.22152/PROGRAMMING-JOURNAL.ORG/2019/3/15

[2] Rodin T. A. Aarssen and Tijs van der Storm. 2020. High-fidelity metaprogramming with separator syntax trees. In *Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2020, New Orleans, LA, USA, January 20, 2020*, Casper Bach Poulsen and Zhenjiang Hu (Eds.). ACM, 27–37. https://doi.org/10.1145/3372884.3373162

[3] Annika Aasa, Kent Petersson, and Dan Synek. 1988. Concrete Syntax for Data Objects in Functional Languages. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming, LFP 1988, Snowbird, Utah, USA, July 25-27, 1988*, Jérôme Chailloux (Ed.). ACM, 96–105. https://doi.org/10.1145/62678.62688

[4] Maartje de Jonge and Eelco Visser. 2011. An Algorithm for Layout Preservation in Refactoring Transformations. In *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6940)*, Anthony M. Sloane and Uwe Aßmann (Eds.). Springer, 40–59. https://doi.org/10.1007/978-3-642-28830-2_3

[5] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax language workbench. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, part of SPLASH 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 237–238. https://doi.org/10.1145/1869542.1869592

[6] Paul Klint. 1993. A Meta-Environment for Generating Programming Environments. *ACM Trans. Softw. Eng. Methodol.* 2, 2 (1993), 176–201. https://doi.org/10.1145/151257.151260

[7] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*. IEEE Computer Society, 168–177. https://doi.org/10.1109/SCAM.2009.28

[8] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 Years of Automated Evolution in the Linux Kernel. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin C. Reed (Eds.). USENIX Association, 601–614. https://www.usenix.org/conference/atc18/presentation/lawall

[9] Julia Lawall and Gilles Muller. 2022. Automating Program Transformation with Coccinelle. In *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*. 71–87. https://doi.org/10.1007/978-3-031-06773-0_4

[10] Luka Miljak, Casper Bach Poulsen, and Rosilde Corvino. 2024. *Concrete Syntax Metapatterns - Artifact.* https://doi.org/10.5281/zenodo.13741142

[11] Scala Development Team. 2021. *Pattern Matching | Scala 2.13.* Retrieved June 25, 2024 from https://github.com/scala/scala/blob/2.13.x/spec/08-pattern-matching.md#pattern-alternatives

[12] Mathijs T. W. Schuts, Rodin T. A. Aarssen, Paul M. Tielemans, and Jurgen J. Vinju. 2022. Large-scale semi-automated migration of legacy C/C++ test code. *Softw. Pract. Exp.* 52, 7 (2022), 1543–1580. https://doi.org/10.1002/SPE.3082

[13] The Eclipse Foundation. 2012. *Eclipse CDT.* Retrieved June 26, 2024 from https://github.com/eclipse-cdt

[14] UseTheSource. 2009. *Descendant Pattern | Rascal Language Reference.* Retrieved June 25, 2024 from https://www.rascal-mpl.org/docs/Rascal/Patterns/

[15] Mark van den Brand, Arie van Deursen, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. 2001. The Asf+Sdf Meta-Environment: a Component-Based Language Development Environment. In *First Workshop on Language Descriptions, Tools and Applications, LDTA 2001, a Satellite Event of ETAPS 2001, Genova, Italy, April 7, 2001 (Electronic Notes in Theoretical Computer Science, Vol. 44)*, Mark van den Brand and Didier Parigot (Eds.). Elsevier, 3–8. https://doi.org/10.1016/S1571-0661(04)80917-4

[16] Mark GJ van den Brand and Jurgen J Vinju. 2000. Rewriting with layout. In *Proceedings of RULE2000*, Vol. 98.

[17] Eelco Visser. 2002. Meta-programming with Concrete Object Syntax. In *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2487)*, Don S. Batory, Charles Consel, and Walid Taha (Eds.). Springer, 299–315. https://doi.org/10.1007/3-540-45821-2_19

Received 2024-06-27; accepted 2024-08-30