A Reflexive and Automated Approach to Syntactic Pattern Matching in Code Transformations

Jason Lecerf*, John Brant†, Thierry Goubier* and Stéphane Ducasse‡
*CEA-LIST
Gif-sur-Yvette, France
Email: jason.lecerf@cea.fr
†RefactoryWorkers
USA
Email: brant@refactoryworkers.com
‡RMoD, INRIA Lille Nord Europe
Villeneuve d'Ascq, France
Email: stephane.ducasse@inria.fr

***Abstract*—Empowering software engineers often requires to let them write code transformations. However existing automated or tool-supported approaches force developers to have a detailed knowledge of the *internal* representation of the underlying tool. While this knowledge is time consuming to master, the syntax of the language, on the other hand, is already well known to developers and can serve as a strong foundation for pattern matching. Pattern languages with *metavariables* (that is variables holding abstract syntax subtrees once the pattern has been matched) have been used to help programmers define program transformations at the language syntax level. The question raised is then the engineering cost of metavariable support. Our contribution is to show that, with a GLR parser, such patterns with metavariables can be supported by using a form of runtime reflexivity on the parser internal structures. This approach allows one to directly implement such patterns on any parser generated by a parser generation framework, without asking the pattern writer to learn the AST structure and node types. As a use case for that approach we describe the implementation built on top of the SmaCC (Smalltalk Compiler Compiler) GLR parser generator framework. This approach has been used in production for source code transformations on a large scale. We will express perspectives to adapt this approach to other types of parsing technologies.**

***Index Terms*—syntactic patterns; pattern matching; code templates; GLR parsing; parser generation**

I. Introduction

Developers often face the need to perform code transformations over a large body of code [SAE+15], [AL11], activity often termed refactoring a code base. Such code transformations are tedious and error-prone to perform when done manually. Semi-automated code transformations based on pattern matching facilities is becoming more and more mainstream as an answer to that issue, and have proven their effectiveness in real-life code bases [PTS+11], to the point of being integrated inside integrated development environments. Those code transformations are based on writing patterns of code and their associated transformations. A transformation engine parses the pattern, finds all its occurrences in the source code and transforms each occurrence using the transformation rule. With a simple pattern, this feature enables developers to rewrite and refactor large portions of their source code. That being said, writing these patterns requires a large amount of language knowledge: the internal structures (for example, see Listing 1) of the transformation engine for this language, that we will call an intermediate representation (IR). Most developers do not have the time and resources to become experts in parsing and in intermediate representations, making it really hard for them to apprehend pattern matching. As a result, only experts write patterns, and package them as extensions inside integrated development environments for developers to use, in the Refactoring Browser (RB) [RBJO96], [Rob99], Eclipse, IntelliJ, Photran, and many more. And those environments, even with decades of existence such as RB, see very little user creation of new transformation patterns [dSS17].

The vast majority of the pattern matching engines exposes their intermediate representations to the pattern writer [BCM15], [HKV12], [LT12], [PF11], [**?**] He needs to know what kind of IR construct he wants to match beforehand (ex. an $Expression$, a $QualifiedFunctionDeclaration$, $\ldots$).

As an example, to write the pattern in Listing 1 the user needs to know that $IFSTATEMENTNODE$ is the name of the IR element to match. Some more knowledge about the attributes of this node are also required: $CONDITION$ and $BLOCK$. Both are a burden to learn for the user.

```
1 ifPattern : ^(IFSTATEMENTNODE CONDITION=.
      BLOCK=.
2         {Host code distinguishing single
3         statement vs multiple statements
4         conditionals}) ;
```

Listing 1. Explicit conditional statement pattern.

Additionnaly, the same example should perform the matching of a one-statement conditional as opposed to a multiple-statements conditional. However, explicit pattern matching is too generic to handle this *syntactic* difference. Handling similar syntactic differences is non trivial in IR-based patterns,

cannot be automated and requires non-standard intervention using host code.

On the other hand, syntactic patterns such as the one in Listing 2 let the user write patterns as he writes his program.

```
if ( `condition` )
    `anyStatement` ;
```

Listing 2. Syntactic conditional statement pattern.

Here $condition$ and $anyStatement$ are metavariables that will match any valid IR construct (not all IR constructs of the language) and the rest of the pattern is the standard syntax of the language to match (here Java). To forgo the IR requirement, syntactic patterns have been introduced (sometimes named "code templates") to build upon the syntax of languages instead of their IRs [PLHM08], [Bar13], [DRI14].

Since syntactic patterns rely on syntax, no IR information is directly given by the user (unlike the $IFSTATEMENTNODE$ in Listing 1) to the pattern matching engine, the IR element of the pattern needs to be extracted from the syntactic pattern thanks to an inference engine. In conjunction with an inference engine, the matched IR element associated to the pattern can be retrieved and used for further analysis, rewriting, refactoring...

The challenge tackled in this paper is then how to produce a pattern matching engine independent of the matched language but that accounts for the syntactic differences in said language.

Contribution: Our solution, Reflexive Parsing, offers a pattern matching engine working on syntactic patterns in which metavariables can appear at *any* position. Reflexive Parsing makes use of GLR parsing to build a node inference engine which validates the completeness of the pattern matches, to get all the possible matches of syntactic patterns.

This approach only requires:

- LR parser generation,
- IR[1] generation by the parser,
- a token to identify metavariables, and
- a forking mechanism for the parser.

The approach is integrated in a parser generation framework, making adaptation to a new language close to effortless. Indeed, activating the pattern matching engine for a new language only requires to add a single line to the language grammar (compared to a standard LR grammar).

The main contributions of this paper are the following:

- A description of node typing in intermediate representations and its use in pattern matching,
- A description of the main mechanisms of Reflexive Parsing,
- A discussion on how Reflexive Parsing can be generalized outside of its current implementation and parsing technology.

Section II describes the problem with the IR-based pattern matching techniques. Section III presents the core mechanisms of reflexive parsing through an example. Section IV details these core mechanisms and the algorithms to implement them. Section V introduces their open-source implementation and industrial use. Section VI discusses the limitations of the reflexive parsing approach. Section VII explains the position of our work in the state of the art. Section VIII concludes this paper and opens perspectives.

[1]In our case, the IR is an Abstract Syntax Tree (AST) thus we refer it as such

II. Intermediate representations and pattern matching

A. Pattern matching

Pattern matching is a key component on which are built rewrite engines [YKS+14], refactoring engines [RBJO96], [KBD17] (with support for pre- and post-conditions), powerful search engines [DRI14], clone detection engine [KT14]...

Pattern matching is all about feeding a pattern and a program to an engine and getting in return all the possible occurrences of the pattern in the program in the form of an element of the IR per occurrence. A pattern matching engine is simply a search engine returning elements of said source code IR.

There are many words used to describe the elements of internal representation of programs. The two most prominent ones, AST nodes and terms, are named based on the rationale behind the tool that uses them. Since our approach focuses on parsing, we decide to use the term "node" to further describe elements of the IR. In this context, an AST can represent either the representation of a program or pattern occurrence. In ASTs, nodes are associated to their type, $FunctionDeclaration$ or $Statement$ are examples of such types. Since occurrences of a pattern result in ASTs, their nodes are typed in the same way.

B. Explicit typing

Node types in patterns can exist in two ways, explicit or inferred. Most current pattern matching techniques only provide patterns with explicit typing. Explicit typing requires the user to write the name of the IR element (its type) he wants to match in the pattern.

a) Example: As previous shown by Listing 1, dealing with IR in patterns is cumbersome. In the next version of ANTLR (ANTLRv4), XPath [**?**] replaced the custom tree matching as the prominent pattern matching tool. XPath has the same exact problem of exposing internal IR. The patterns in Listing 3 show different contexts in which `ID` can be encountered. [2].

```
//ID
//expr/primary/ID
//body//ID
```

Listing 3. Example of patterns in XPath

If the developer only wants variable names in his expressions, he needs to know to match `ID`, but it also means that the

[2]This example was taken from ANTLRv4's githubhttps://github.com/antlr/antlr4/blob/4.6/doc/tree-matching.md

developer should know that an ID is a subnode of `primary` which is a subnode of `expr` node. The user should not need to learn a complicated model to match source code.

b) Limitations: Patterns are inherently hard to write because they require knowledge of the language IR. For example, the user may want to search for all the $ForLoop$s, or all the $QualifiedName$s, ... Basically he already needs to know what construct he is searching for. The typing information needs to be clearly expressed in the pattern. The more complex the language, the more constructs the user has to learn.

TABLE I
IR SIZE FOR POPULAR LANGUAGES

Language	Java	C++	C#	Delphi	JS
# Nodes	114	157	149	141	78

Table I shows IR sizes for some mainstream languages taken from Eclipse's JDT plugin (Java), SmaCC (Java, C#, Delphi, JS) and IPR (C++). IPR [DRS11] is a C++ representation designed specifically to be compact with as little node duplication as possible. Yet it is still composed of over 150 node types. All these languages have sizable IRs that can be tedious to learn for a non expert.

Furthermore, this is dependent on the compiler internal representation. The problem can be much worse for languages similar to COBOL where the grammar is reverse engineered from existing code bases [VDBSV$^+$97] (more than 300 production rules).

c) Unified representation: Solutions for unified representations such as OMG's AST Metamodel (ASTM) [Obj11] could ease the burden by having a common representation for (almost) all languages. The concepts in the metamodel corresponds to core nodes available in most programming languages, such as the notion of expressions, statements, etc... arranged in three groups: Generic ASTM, Language-specific ASTM, and Proprietary ASTM, to be able to cover all languages. The Generic ASTM is composed of 189 unique types, present in most languages. This part of the ASTM is not exhaustive and must be completed with node types from the Language-specific and Proprietary ASTMs. These unified representations end up being the union of all the nodes for all the languages, making the representation even more complex.

d) Island parsing: Island parsing can be a solution to reduce the size of the IR, by allowing one to write simpler parsers restricted to the subset of the language one wants to work on, instead of parsing the entire language [Kur16]. This approach decreases the cost of writing tools [BNE16] as a tiny part of the language is parsed but at the cost of automation, because it cannot be generalized to other languages. This is significant for legacy and proprietary languages such as Mantis, NSDK, 4D where the grammar is often not public or does not even exist [KLR13], [Kur16]. However, island parsing may result in multiple different representations for each parser that is written, and thus have a hard time with genericity.

C. Inferred typing

Inferred typing refers to approaches where patterns describe the syntax of what should be matched instead of its node type. The system, in return, must infer the type of the node from the syntax.

To express inferred typing, two main approaches have been taken: query-by-example [Bal15] and syntactic patterns [Cor06], [DRI14].

Query-by-example [Bal15] consists in providing an example that will be parsed and the engine will retrieve every occurrence that is significantly similar to the example. The major drawback of this approach is its heuristic nature: it cannot grasp every single occurrence. As an example, feeding `a + 5` to a query-by-example engine will possibly yield all additions, but also all other binary expressions or even only additions between an identifier and a literal depending on the AST and the similarity heuristic of the engine. However, this approach is based on a heuristic and as such, cannot be exact, false positives and negatives are a real threat.

Syntactic patterns [Cor06], [DRI14], [MDR16], on the other hand, are written in the language to match. They are snippets of code, but with a major difference, metavariables can be inserted inside. Metavariables will match *any* element of the IR that could appear at that position (effectively being an enhanced wildcard). So, syntactic patterns look like a normal pieces of code which do not require knowledge of the IR. To illustrate, the syntactic pattern `` `i` + `i` `` should match any addition of a term with itself. In most ASTs, this means `` `i` `` will be an $Expression$, a $Number$, an $Identifier$, ... Syntactic patterns can be used in the same way explicit patterns would: in source code search engines, transformation tools, ...

Syntactic patterns are fit to be used in the same contexts as explicit patterns, but as of now, some key issues make it still hard to automate and extend from a single language tool to a multiple languages one. We will describe these problems next and provide solutions in Section IV.

D. Syntactic pattern problems

a) Starting point problem: A pattern will in the end be associated with one or more AST node types. To get a specific type in a parser, you need to start at a specific point in the grammar. The usual starting point of the grammar will yield the root node of the AST, but other starting points will yield different node types. The first challenge is how to *automatically* infer *all* of the valid starting states to get all the possible node types of a given pattern. The solution to this problem must be language-independent and not require manual specification.

b) Complete match problem: The problem is similar for metavariables: instead of all the possible types for the pattern, we only want the possible types for the metavariable in the pattern. Since the node types can appear only at specific positions, the engine must be capable of only selecting the ones that are valid in regard to the grammar of the language.

```
{ `x` }
```

Listing 4. Example of patterns in XPath

As an example, in Listing 4, the `x` metavariable inside Java curly braces can refer to a $Statement$, an $AttributeDeclaration$, a $MethodDeclaration$ depending on the type of block it is in. The set of node types, a metavariable can have, must be complete and exact. The mechanism that makes it possible should not rely on any language-specific design.

c) Hybrid approach: While not a mandatory requirement, the engine should support an explicit-syntactic hybrid. Syntactic patterns are written in the matched language but this is not always desirable (mostly for experts). For example, in C, to rewrite all K&R C function declarations into ANSI C function declarations, syntactic patterns are really useful since they distinguish between the two syntaxes. However if the user wants to get all function declarations in an old program (that may contain both K&R C and ANSI C declarations), he is better off using explicit typing to match the node $FunctionDeclaration$i (this is valid only for experts). An hybrid approach should be developed to explicitly type individual patterns and metavariables.

III. The Reflexive Parsing Approach: an example

A. Pattern matching using Reflexive Parsing in parser generators

A key component of our approach is the use of a parser generator to help us solve the three issues above. Parser generators give us access to type information from the grammar (through semantic actions) and to the parser generation process. Having both enables us to generate parsers fully compatible with our pattern matching engine with almost no overhead. And our approach, *Reflexive Parsing*, takes advantage of both properties to build a node inferring system on patterns from a GLR [MN04] parser generator. This system solves the starting state problem and the complete match problem by a clever use of the GLR parser forking mechanisms. The pattern matching engine also supports an explicit-inferred hybrid approach that lets the user specify explicitely individual metavariable or pattern types.

In the following sections, we describe (first through an example) how a GLR parser aware of its own parse tables help build a powerful pattern matcher that infer node types from syntactic patterns.

To introduce our approach, we first present an example of syntactic pattern to hint at what it can provide concretely and what it looks like. Then we give a general explanation of how the pattern matching engine work. For details, please refer to Section IV.

B. An example

```
`any` ^ 2
>>>
`any` * `any`
```

Listing 5. Power of 2 syntactic pattern in R

In this example (Listing 5), the syntactic pattern is used to match all the possible calls to the power-of-2 operator in the R programming language to transform it in a multiplication. Such a transformation is a common trick to improve performance of programs at low cost.

However it is not as trivial as it seems to be. In R, literally anything can be elevated to the power of 2 and still be valid in the internal representation (expressions, function calls, and even function definitions or for loops) [3]. Listing all the items that can appear as a power-of-2 operand is best handed to the system to infer.

This syntactic pattern can be expressed this way because the parser has acccess to both the metavariable token (in backquotes) and its own parse tables.

C. General approach

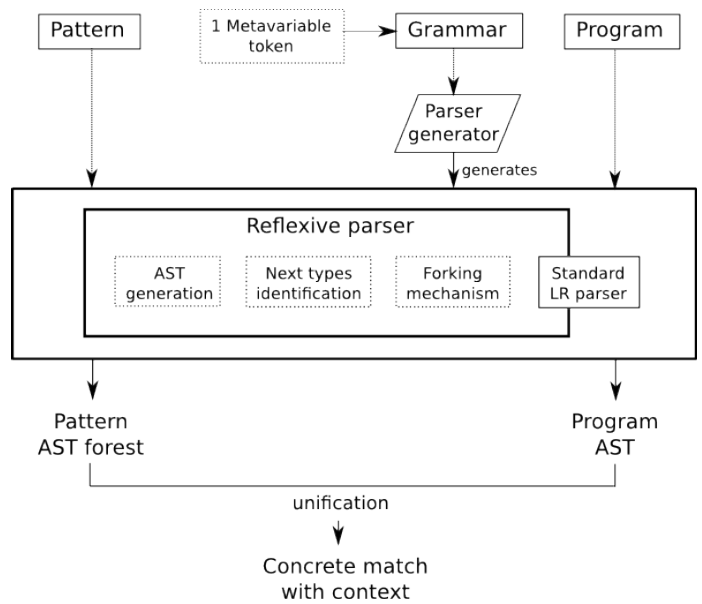

Fig. 1. General pipeline of the pattern matching engine

Figure 1 presents the main components making syntactic patterns possible. The gist of the reflexive parsing approach is to parse the pattern. Whereas the program parsing yields an AST, the pattern parsing is more complex and yields a forest of ASTs corresponding to each valid match.

The GLR parse starts and when it encounters a metavariable, the parser forks into multiple subparsers, one for each possible type that can appear at this specific point during the parse. The parser does it by *reflexively* inspecting its own LR parse tables. In the automata, it looks at all the available transitions from the current state. If a subparser fails to parse, it means the configuration is invalid and it will be discarded. The subparsers that survive at the end of the parse will each generate one AST,

[3] As visible in the R parser in the R source code at: https://svn.r-project.org/R/trunk/src/main/gram.y

each AST having a specific configuration of metavariable-type pair. Each pattern AST in the forest is confronted against the program AST. For the matching subtrees, each metavariable is bound to the concrete AST node (meaning subtree) it represents.

That way, using a syntactic pattern with metavariable, the system can grasp all the possible matches and the users need no prior knowledge of the intermediate representation of the engine.

Activating pattern matching for a new language is easy if a LR grammar is available. It consists of only a single line of code to add the metavariable token into the grammar, so that the parser recognizes it. Then syntactic patterns can be used on the new language.

Now we will describe in details each important component of the reflexive parsing approach to pattern matching.

IV. The Reflexive Parsing Approach: the mechanics

Taking the parser generation route offers us unique opportunities. First, all the syntax information about the language is present in the generated parser. Second, the type information is also retained from both syntactic and semantic actions. Third, we can modify the generation process to introduce our type inference system for any parser with a valid grammar.

A. Table-driven parsing

LR parsers are table-driven parsers, which are in fact automata. The parsing tables represent the states in which the program can be in. Each transition is fired upon receiving a specific token. If a token is not recognized by a state, the parse fails. Upon receiving a valid token, the parser can shift to a new state or reduce the current state. On shift, the current token is consumed and the automata pushes a new state on the state stack. On reduce, the n-th last states are popped from the state stack, and a new state is pushed on the state stack. Once the end of the file is reached and the first state has been reduced, the parse is successful.

Such simplicity makes LR parsers great candidates for generation. The other factor is that the tables are derived from the grammar. Lex & Yacc [LMB92] is an early example of a generator producing a table-based scanner-parser combination. In a parser generation framework, AST can be generated as well from semantic actions in the grammar. In table-based parsers, it is translated by a node stack and its manipulation. On shift, the current token is pushed on the node stack. On reduce, the n-th last stack elements are popped from the stack and a new AST node is pushed with these elements as subnodes.

Since all the grammar information is distilled into the parse tables, it is then possible to use said information in the parse tables from the parser at parse time. Accessing the parse tables and the transitions, at any point in the source, the parser knows what symbol types can be next: an $Identifier$, an $Expression$, a $Condition$, etc.... By modifying the parser generation process, we get access to this information from the parse tables.

B. Typing in parser generators

In a parser generation environment, two kinds of types coexist in a single typing system. The first kind is the symbol types: tokens and non-terminals from the grammar. They already hold type information and they are extracted from the syntactic part of the grammar. We name tokens "elementary types" and non-terminals "composite types". The composite types are the union of other types, either elementary or composite themselves. The second kind is the user types (defined by the user through semantic actions) change the types at key points in the AST. User types override composite type definitions in a parser generation setting.

```
Exp -> number
Exp -> Exp + Exp {{BinaryExp}}
```

Listing 6. Example of typing in grammar

In the example of listing 6, the token $number$ will yield an elementary type T_{number}. So, Exp should normally be of composite type $T_{Exp} = \{T_{number} \bigcup \{T_{Exp}, T_{+}, T_{Exp}\}\}$. However, the user overwrote the type of the second production with the user type $T_{BinaryExp} = \{T_{Exp}, T_{+}, T_{Exp}\}$, leading to Exp being of type $T_{Exp} = \{T_{number} \bigcup T_{BinaryExp}\}$. This type definition is intuitive, a formal type definition is outside the scope of this paper.

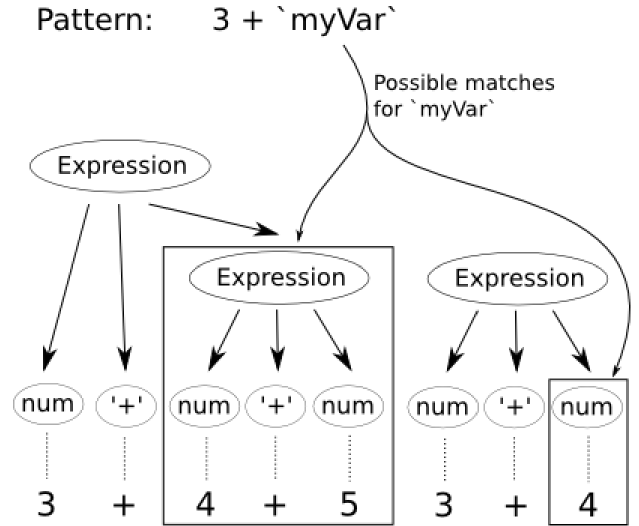

Fig. 2. Composite type example

When trying to match the pattern `3 + 'anything'`, as in Figure 2, the $anything$ part of the pattern can be the symbol $number$ or the symbol $BinaryExp$, which respectively correspond to elementary type T_{number} and user type $T_{BinaryExp}$. This composite type (T_{Exp}) is the union of both subtypes and is important to grasp all the possible types of a pattern part. To construct the valid pattern ASTs from the pattern declaration, all the subtypes of the union must be considered as valid candidates. The mechanisms to take typing into account will be described in Section IV-G.

C. Parsing a syntactic pattern

Parsers can easily take advantage of the syntax since it is encoded in their behavior, so a parser can be used to parse the program *and* the pattern.

```
for(`init`;`test`;`update`)
{
    `statements`
}
```

Listing 7. Syntactic pattern in SmaCC

Listing 7 is an example of pattern the user must be able to write. The metavariables are represented here by backquoted names, but for genericity purposes this is in fact a customizable token added to the grammar. The metavariable token is mandatory to parse syntactic patterns: otherwise, the parse will fail to recognize it as a valid part of the language. Since it is *added* to the grammar, this token must not conflict with pre-existing token definition. In this paper, every backquoted token represents a metavariable because backquotes are rarely used in programming languages.

Once the grammar is upgraded with a metavariable token and the parser generated, the parse can truly start.

D. The starting point problem

The first question that comes to mind to parse a pattern is: "Where does the parse start?". A parser needs a starting symbol to begin the parse, and a program usually starts at the state for the left symbol of the first production rule. However, it is unreasonable for a pattern since it would mean describing every pattern from the beginning of the program, a very tedious and unrealistic task.

Since we have our LR parse tables, we have an advantage. We know which states are good start candidates: states that have at least one outgoing shift. To start the parse, the parser needs to consume a token first, otherwise it cannot reduce the pattern to a valid node. So the only candidates are the states that at least consume one token, the states with at least one shift transition. States that *only* reduce are not valid.

To parse the pattern, the engine creates a number of parsers equal to the number of states that shift. That way we will get all the possible types for the pattern root node. Each parser is independent and starts at a different state, however they all work on the same input stream: the pattern. For example, parsers can start at the beginning of a function declaration, a statement or an expression. Then, for each parser, the standard LR parse begins. A failing parser will be pruned, so only matching ones subsist. That way we embrace all the *valid* starting points of the parser. The parsers behave normally until they reach metavariables.

E. Wildcards and metavariables

Wildcards are usually used to match anything and are available in most pattern matching engines. In our approach, metavariables are wildcards in the sense that they can match any node or list of nodes valid as per the grammar. The metavariables are mainly only variables with unbound value (until they match some node and their value become the node in question) in the declarative sense of variable. However, more information can be provided to the metavariable via a small DSL.

When the parser's lookahead is a metavariable token, it does not execute the next standard LR actions, instead it does two things:

- First, the parser forwards of the metavariable parsing to a small dedicated parser (for the DSL).
- Second, the parser executes special behavior for the metavariable.

F. The metavariable dedicated parser

The job of this dedicated metavariable parser is minimalistic: it simply parses the metavariable token. What is important to remember is that it is the place to add behavior for the match. For example, the user can change how the metavariable should match, such as ignoring some node types (such as parentheses in expressions), providing additional checking blocks, putting restrictions on the types (only nodes, only tokens, only lists of nodes).

Type information can be provided in the metavariable token for explicit typing. Doing that return the matching to a simple type comparison instead of the syntactic comparison. This is what allows an hybrid approach between inferred and explicit typing. The default behaviour for a metavariable matching is inferred and it is only overridden when providing the explicit type information.

G. Forking the parser

Once the metavariable token is parsed, the execution returns to the main parser. Instead of executing a standard LR action, it triggers a special method which is described in Algorithm 1.

Algorithm 1 Reflexive parsing

1: **function** FORKONMETAVARIABLE($stateStack, metavariableToken$)
2: $\quad state = top(stateStack)$
3: $\quad userType = explicitType(metavariableToken)$
4: $\quad$**for all** $(symbol, action)\ in\ transitions(state)$ **do**
5: $\quad\quad$**if** $isDefined\ (userType)\ and\ !\ compatible\ (symbol, userType)$ **then**
6: $\quad\quad\quad continue$
7: $\quad\quad$**end if**
8: $\quad\quad$**if** $isComposite\ (symbol)$ **then**
9: $\quad\quad\quad forkParser()$
10: $\quad\quad\quad emulateNodeShift\ (\ action,\ symbol,\ metavariableToken\ )$
11: $\quad\quad$**else if** $isElementary\ (symbol)$ **then**
12: $\quad\quad\quad forkParser()$
13: $\quad\quad\quad performReducesAndShift\ (\ action,\ symbol,\ metavariableToken\ )$
14: $\quad\quad$**end if**
15: $\quad$**end for**
16: **end function**

The algorithm forks the parser for all possible interpretations of the metavariable token and shifts it on the node stack. To do this, it goes over all symbols defined by the parser and possible parser actions for the symbol in the current parser state.

The symbol type is checked for compatibility with the metavariable user type if one is specified by the user in the pattern. The symbol type is compatible to the metavariable user type if it is the same type or one of the metavariable subtypes. For a composite type such as $Expression$, any kind of expressions can be compatible: $BinaryExpression$, $UnaryExpression$, ...

If the symbol type is an elementary type (i.e., a token), two LR actions are possible: shift and reduce, or in fact one shift and possibly one or more reduces. The shift is always performed but reduces may be executed earlier to put the parser in the state where the metavariable can be pushed onto the node stack (as for any LR shift). Some of these reductions may be invalid, but their subparsers will quickly be killed when no valid action can be found at the newly reduced state. In the case of the symbol type being composite, the only possible action is a shift, but instead of pushing a token on the stack, the metavariable is pushed onto the node stack. The parser forks itself into a new subparser instance before executing the LR actions in question. This new subparser now continues its parse with the type of the metavariable being identical to this specific symbol type.

Additional verifications are performed to check if the symbol types matches the restrictions on the metavariable (for example, only match tokens).

Each subparser continues its parse and invokes $ForkOnMetavariable$ each time it encounters a metavariable token. If a parser reaches an impossible configuration, it is invalidated and discarded.

At the end of the parse, we collect all the possible types for a metavariable in the form of one AST per successful subparser. The result is a forest of the valid pattern ASTs where each AST has a unique set of metavariable-type pairs.

H. Unification

The next phase is more standard: a unification process compares the pattern forest of ASTs with the subtrees of the program. This step consists in confronting each possible pattern solution to the program AST. If part of the program AST matches, the subtree should be returned and if no match can be found, the algorithm stops there. This can be done in a number of ways, but we focus on a simple depth-first traversal of the program AST (see Algorithm 2).

For each node of the program AST, we consider it as a root node and compare it to the top node of each pattern AST.

For the node equality (see Algorithm 3, we first check for type equality and then if they have the same type, each child (being a node or a token) is compared for equality with its counterpart. If all the subnodes of the pattern root node and the current program node match, we consider that this pattern tree is a valid concrete match. If any of the subnodes fails to match, the pattern tree is discarded.

For tokens (see Algorithm 4, the actual token strings are compared and returns the result of the equality to the calling node.

Algorithm 2 Unification

1: **function** UNIFY($programAST, patternForest$)
2:     **for all** programNode in programAST **do**
3:         $programRoot \leftarrow programNode$
4:         **for all** patternAST in patternForest **do**
5:             $patternRoot \leftarrow root(patternAST)$
6:             **if** $patternRoot = programRoot$ **then**
7:                 Register $patternAST$ as a valid match
8:             **else**
9:                 Discard $patternAST$
10:             **end if**
11:         **end for**
12:     **end for**
13: **end function**

Algorithm 3 AST node equality

1: **function** EQUALS($firstNode, secondNode$)
2:     **if** $type(firstNode) = type(secondNode)$ **then**
3:         **return** $subnodes(firstNode) = subnodes(secondNode)$
4:     **else**
5:         **return** $false$
6:     **end if**
7: **end function**

Algorithm 4 AST token equality

1: **function** EQUALS($firstToken, secondToken$)
2:     **return** $source(firstToken) = source(secondToken)$
3: **end function**

If a now typed metavariable node matches in the program AST, it is bound to its concrete match in a dictionary. Note that if the same metavariable is used multiple times in a pattern, their respective match in the program AST should be identical. For example, `` `i` + `i` `` cannot successfully match `3 + 4`. The dictionary is reused, at the end of the comparison, as context for the match. The rewriting or analysis can then be based on the typed matches and their respective contexts.

Trying to match a pattern that has no metavariable may too result in multiple pattern ASTs (if the target language is inherently ambiguous). For example, the pattern `` (`a`) * `b` `` in the C programming language has (at least) two pattern trees, one for the multiplication expression, and the second one for the declaration of a pointer variable of type `a`. Comparing such a pattern AST with the program AST is a more advanced version of a text search function, more advanced because the AST comparison can ignore some nodes (such as parentheses in expressions) [4].

[4]This is a setting that can be set when writing the grammar in SmaCC.

V. IMPLEMENTATION: THE SMACC REWRITE ENGINE

A. The Smalltalk Compiler-Compiler

This approach is implemented on top of the Smalltalk Compiler-Compiler (SmaCC). [5] This parser generator produces LR(1), LALR(1) and GLR parsers from a grammar in Backus-Naur Form (BNF). Compared to a LALR BNF grammar from a different tool, the only difference is the addition of the previously defined "metavariable token" IV-E as a non-conflicting token (meaning it should not interfere with other token definitions). Our pattern matcher is based on this grammar "extension" and the GLR abstract parser class of the tool.

Since the pattern matching engine has been built as part of a transformation engine used for migration projects such as Delphi to C# or PowerBuilder to C#, we will take examples from code transformations.

B. Industrial Validation

Reflexive Parsing has been used on several industrial projects using different programming languages such as Delphi, PowerBuilder, C#, Java and Ada [BRPP10]. These projects range from analysis and refactoring to migration from one language to another. As of writing this paper, the SmaCC Rewrite Engine supports the following languages: C#, Java, Javascript, Delphi, PowerBuilder, IDL, Swift, Ada and C[6].

One recent project involved converting a PowerBuilder application to C#. Code migration is a good example to illustrate the pattern matching capabilities of the engine. The PowerBuilder code contained almost 3,200 DataWindow components and over 700 code components. These files contained over 1.1 MLOC[7] and were 153MB in size. The resulting C# code contained almost 7,600 files with 3.3 MLOC and was 161MB in size[8]. These files were converted using 578 conversion rules. Of the 578 rules, 356 (62%) of them used patterns as described in this paper, and the other 222 rules used more traditional explicit type based matching.

Converting the code was done in two passes. The first pass converted all of the PowerBuilder code to C#, and used 509 conversion rules. Note that each transformation pass involves applying numerous rules in bulk on the same valid input (the first pass on PowerBuilder here), not by applying a rule, parsing the new program and applying another one, etc. The entire PowerBuilder code will be transformed to C# by applying all the transforms to generate a new code from the AST.

[5]SmaCC is freely available at https://www.refactoryworkers.com/SmaCC.html, on smalltalkhub and github

[6]The C parser is closed source for now. A sample parser can be found instead.

[7]DataWindows are generally automatically generated, declarative components where many properties are assigned on a single line of code. Lines of code are not necessarily a good measure for these components, but they are included here.

[8]The number of files increased since some files were split into their code and designer components. Also, most of the increase in overall size is due to formatting. For example, code in methods were indented with two tab characters in C#, and not indented in PowerBuilder.

```
for `a` := `b` to `c` - 1 do `d`
>>>
for (`a` = `b`; `a` < `c`; `a`++) `d`
```

Listing 8. Delphi to C# syntactic pattern in SmaCC

Listing 8 is an example of Delphi to C# rule that is a special case of the general for loop conversion. The Delphi for loop used `"- 1"` in the end condition, we could eliminate the `"- 1"` from the converted code by using `"<"` instead of `"<="`. Such small transformations are important to make the code more natural to future developers.

The second pass refactored the converted C# code using 69 rules. While the refactoring step was not necessary, it did eliminate some conversion artifacts and also simplified some code to use some C# framework methods that were unavailable in PowerBuilder. Running these two steps using the Pharo environment [BDN+09] on a six core Intel E5-1650 on the 153MB of PowerBuilder source took 2 minutes 55 seconds. Only running the first pass took 1 minute 30 seconds.

While the pattern parsing described in this paper may cause much forking during the parsing of the pattern, it does appear to be acceptable in practice. For example, parsing the 356 patterns used by the conversion rules takes 300 milliseconds.

A more detailed evaluation of the tradeoff in performance for simplicity and accessibility of syntactic pattern matching compared to explicit pattern matching is outside the scope of this paper and is left for future work. However, this evaluation shows the practicality of the method in complex industrial use cases of large code base migration.

VI. DISCUSSION

A. Prerequisites of the approach

Our approach assumes the following from the parser generation framework:

a) LR parser generation: The parser should be an LR parser to enable reflexivity on the parse tables. The behaviour of the LR parsers must be modified during the generation process to enable the parsing of the metavariable token and the actions associated to this token. This method enables greater reusability since every generated parser will be reflexive. The cost of adding this pattern matching technique to a new language is very much reduced in this setting. Having or creating the grammar remains the biggest requirement, but since it is one for most parsing tools, it does not seem unreasonnable here.

b) AST generation: The LR parser should be able to generate an AST of the parsed program. A sub-requirement is that a coupling between the syntactic symbols and their type must be present in the parser. This coupling is achieved with the semantic actions associated with the shift/reduce parser actions. As most parsers generate ASTs, we do not find this requirement particularly hard to validate.

c) Next type identification: During the parse of the pattern, the parser should be able at any state during the parse to find out what are the next types that can appear. In a LR

parser, this information is extracted from the symbol table of the parser and its transitions from the current state. And from the symbol, it is possible to deduce its symbol type (or user type if it has been overridden).

d) Forking mechanism: The parser should have a forking mechanism. If a GLR-style parser is already generated, it can be used directly by forking on metavariables in addition to ambiguities.

B. Parsing as intersection

We believe this approach to syntactic is a restricted version "parsing as intersection" adapted to a pattern matching language. This result from [**?**] simply states that the intersection of a Context-Free Grammar (CFG) and a regular language is a CFG. In our context, the initial CFG is the CFG of the target language and the regular language is our syntactic pattern language. The syntactic pattern grammar has the same alphabet (the same symbols) as the one from the CFG but only supports union, concatenation, Kleene star and Kleene plus because it is a regular language. Thus, parsing as intersection answers the question of the expressiveness of the syntactic pattern language. Our intersection yields a forest of trees containing the possible typed matches of the pattern, but this forest can also be seen as all the generated trees from a grammar if we explored all of its possibilities. This grammar is in fact the CFG resulting from the intersection of the target language CFG and the syntactic pattern (regular) language. As of the discussions in [**?**], use cases of parsing as intersection were unclear: we at least provide one, pattern matching. However, further study is necessary to produce a full fledge grammar from a GLR parser.

C. Scalability

LR and LALR parsers have the nice property of their parse time being only dependent on the input length and not the input depth [AHU74]. This leads to much appreciated linear parse time. It is not true for GLR-type parsing ($O(n^3)$ in complexity) and in our case, since we are potentially creating all possible ASTs, we can be exponential [JSE06]. Those parsers create subparsers every time an ambiguity is encountered. So, the parse time now depends on the "ambiguity depth", meaning the number of nested ambiguities we can get. In practice it does not seem to be a problem. Most general purpose languages are decently well formed and do not present much ambiguities.

As for the subparsers created only for pattern matching purposes, it depends mostly on the quantity of metavariables and their position in the pattern. We tend to think that the quantity of metavariables, and thus of "fork points", is not an issue. Every time a metavariable is added, it adds specificity to the pattern. Since it is parsed, it means fewer configurations will be accessible with each specificity. Fewer and fewer subparsers will be created as the parse progresses on the pattern and some will even get discarded.

D. Application to other parsers and parser generators

We believe the reflexive parser approach to be implementation-agnostic enough to be reimplemented in other pattern matching tools. As long as they respect the previously described requirements (Section VI-A), it can be applied as is to other *LR*-based parser and parser generator.

As for other parser classes such as GLL, the implementation of the prerequisites would change. However, forking is already available for GLL parsers and implementing a new token for metavariable identification should not be a hard task. Also, LL parser generators such as ANTLR already propose AST generation.

The main concern lies in identifying the next possible types at any point during the parse. A equivalent would be required to work with parsing techniques other than LR.

a) Adaptation to unified representations: Using type inference on the nodes of the unified representation allows us to determine the set of possible results at a certain point. This set, because of the unified representation, can be larger than the possible target language subtrees, but this is probably not an issue since the target language can't generate some of those candidate pattern trees. So even if the parser will fork for those nodes (for example, the C programming language does not have *Qualified Names*), they will then never be matched in the unification process.

VII. Related Works

Our reflexive parsing approach characterizes itself by: providing a pattern matching engine based on syntactic patterns, enhancing a language with said engine using only its grammar (LR or GLR), limiting the need for explicitly typed patterns and solving the starting point problem of the metavariable parsing.

The transformation engine Coccinelle [PLHM08] uses syntactic patterns in the form of SmPL semantic patches that describe a code transformation with patch-like syntax. Coccinelle has been used to find bugs and perform collateral envolution in the Linux kernel. However, contrary to SmaCC, the patterns are explicitly typed.

XPath [**?**] is a query language to find nodes in an XML document with pattern similar to file system paths. While model-querying (or AST-querying) is XPath's bread and butter, it is unfit to query source code since it manipulates an abstract representation.

ANTLR [PF11] is a top-down parser generator which produces LL(*) parsers with arbitrary lookahead from a BNF grammar. In ANTLRv3, the user can craft explicit patterns using the types of the language parse tree and specify custom tree traversal strategies for unification. The current version of ANTLR also relies on XPath for its parse tree transformations (and thus suffers from the same problems). However, neither approaches solve the starting point problem and both require explicit typing.

srcQL [Bar13], [**?**] is another query language based on both srcML (an XML representation of source code) and previously described XPath. srcML helps cope with abstraction problem

by providing the XML with concrete syntax information thanks to ANTLR. While srcQL supports syntactic pattern matching, it relies on heuristics to infer the pattern possible types which can lead to false positives *and* false negatives. Combined with unification being perform with partial string comparison instead of proper node comparison, this can lead to an equivalent of `'foo' * 'foo'` matching `foo * foo ()` which we do not want for exact matches.

Ekeko/X [DRI14] is a search and transformation engine on Java programs based on logic programming. It supports code templates (syntactic patterns) which instead of being parsed, are transformed into a set of Clojure logical goals. However, Ekeko/X's approach is solely restricted to Java programs and does not extend beyond this language.

Stratego [BKVV08] is a language to define rules for code transformations. Stratego uses programmable rewrite strategies to apply rewrite rules on ASTs in the form of terms. TOM [BCM15] is a language extension and an engine that bring pattern matching capabilities to a host language such as C++, Java, ... Both Stratego's and TOM's pattern matching originates from term rewriting theory, close to what exists in functional languages. These approaches can explicit typing on the pattern and can contain wildcard in the form of named generic terms.

The POET [Yi12] language is a code transformation language with custom optimizations as its main goal. Its parsing and matching features are based on code templates from island parsing: only the matched part of the language is verified, the rest of the code is not parsed. As such, POET code templates are explicitly typed contrary to a Reflexive Parsing approach based on LR parsing.

VIII. Conclusion

We proposed a new take on pattern matching based on a GLR parser capable of inspecting its own parse tables at any point in the parse to infer the node types of patterns. As a result, patterns can be expressed syntactically by the user without learning the internal representation of the engine. Integrated to a parser generation framework, activating the pattern matching engine for a new language is as cheap as it can be: a single line in the grammar. Our reflexive parsing approach has been implemented in the SmaCC parser generator and rewrite engine written in Smalltalk, and extensively used for large scale code migrations of industrial applications.

As perspective, we consider of interest the study of the extension of that technique on other transformation tools. First candidates would be other classes of parsers such as LL with GLL or parser combinators. A second perspective is the integration of our pattern matching technique with type inference on unified representations.

References

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, Reading, Mass., 1974.

[AL11] N. Anquetil and J. Laval. Legacy software restructuring: Analyzing a concrete case. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11)*, pp. 279–286, Oldenburg, Germany, 2011.

[Bal15] V. Balachandran. Query by example in large-scale code repositories. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pp. 467–476. IEEE, 2015.

[Bar13] B. M. Bartman. *srcQL: A Syntax-aware Query Language for Exploring Source Code*. PhD thesis, Kent State University, 2013.

[BCM15] E. Balland, H. Cirstea, and P.-E. Moreau. Bringing strategic rewriting into the mainstream. 2015.

[BDN+09] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[BDW03] A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of Joint Modular Languages Conference (JMLC'03)*, volume 2789 of *LNCS*, pp. 122–131. Springer-Verlag, 2003. Best Paper Award.

[BKVV08] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. a language and toolset for program transformation. *Science of computer programming*, 72(1):52–70, 2008.

[BNE16] F. Brown, A. Nötzli, and D. Engler. How to build static checking systems using orders of magnitude less code. In *ACM SIGOPS Operating Systems Review*, volume 50, pp. 143–157. ACM, 2016.

[BRPP10] J. Brant, D. Roberts, B. Plendl, and J. Prince. Extreme maintenance: Transforming Delphi into C#. In *ICSM'10*, 2010.

[Cor06] J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[DRI14] C. De Roover and K. Inoue. The Ekeko/X program transformation tool. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pp. 53–58. IEEE, 2014.

[DRS11] G. Dos Reis and B. Stroustrup. A principled, complete, and efficient representation of C++. *Mathematics in Computer Science*, 5(3):335–356, 2011.

[dSS17] G. J. de Souza Santos. *Assessing and Improving Code Transformations to Support Software Evolution*. PhD thesis, University Lille 1 - Sciences et Technologies - France, feb 2017.

[HKV12] M. Hills, P. Klint, and J. J. Vinju. Scripting a refactoring with rascal and eclipse. In *5th Workshop on Refactoring Tools*, pp. 40–49, 2012.

[JSE06] A. Johnstone, E. Scott, and G. Economopoulos. Evaluating GLR parsing algorithms. *Science of Computer Programming*, 61(3):228–244, 2006.

[KBD17] J. Kim, D. Batory, and D. Dig. X15: A tool for refactoring java software product lines. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume B*, pp. 28–31. ACM, 2017.

[KLR13] J. Kurš, G. Larcheveque, and L. Renggli. PetitParser: Building modular parsers. In *Deep Into Pharo*, pp. 36. Square Bracket Associates, Sept. 2013.

[KT14] G. P. Krishnan and N. Tsantalis. Unification and refactoring of clones. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pp. 104–113. IEEE, 2014.

[Kur16] J. Kurš. *Parsing For Agile Modeling*. PhD thesis, University of Bern, Oct. 2016.

[LMB92] J. R. Levine, T. Mason, and D. Brown. *Lex & yacc*. O'Reilly Media, Inc., 1992.

[LT12] H. Li and S. Thompson. A domain-specific language for scripting refactorings in Erlang. In *International Conference on Fundamental Approaches to Software Engineering*, pp. 501–515. Springer, 2012.

[MDR16] T. Molderez and C. De Roover. Search-based generalization and refinement of code templates. In *International Symposium on Search Based Software Engineering*, pp. 192–208. Springer, 2016.

[MN04] S. McPeak and G. C. Necula. Elkhound: A fast, practical GLR parser generator. In *International Conference on Compiler Construction*, pp. 73–88. Springer, 2004.

[Obj11] Object Management Group. Abstract syntax tree metamodel (ASTM) version 1.0. Technical report, Object Management Group, 2011.

[PF11] T. Parr and K. Fisher. LL (*): the foundation of the antlr parser generator. *ACM Sigplan Notices*, 46(6):425–436, 2011.

[PLHM08] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *ACM SIGOPS Operating Systems Review*, volume 42, pp. 247–260. ACM, 2008.

[PTS+11] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in linux: Ten years later. In *ACM SIGPLAN Notices*, volume 46, pp. 305–318. ACM, 2011.

[RBJO96] D. Roberts, J. Brant, R. E. Johnson, and B. Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, Apr. 1996.

[Rob99] D. B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.

[SAE+15] G. Santos, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente. System specific, source code transformations. In *31st IEEE International Conference on Software Maintenance and Evolution*, pp. 221–230, 2015.

[VDBSV+97] M. Van Den Brand, M. Sellink, C. Verhoef, et al. Obtaining a COBOL grammar from legacy code for reengineering purposes. In *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, electronic Workshops in Computing. Springer verlag*, 1997.

[Yi12] Q. Yi. POET: a scripting language for applying parameterized source-to-source program transformations. *Software: Practice and Experience*, 42(6):675–706, 2012.

[YKS+14] L. Yang, T. Kamiya, K. Sakamoto, H. Washizaki, and Y. Fukazawa. Refactoringscript: A script and its processor for composite refactoring. In *SEKE*, pp. 711–716, 2014.