Designing Language-Agnostic Code Transformation Engines

Jason Lecerf

Université de Lille

Département de formation doctorale en informatique **École doctorale SPI Lille**

Designing Language-Agnostic Code Transformation Engines

Construction de moteurs de transformation de code automatique agnostiques du langage

THÈSE

présentée et soutenue publiquement le 26/11/2019

pour l'obtention du

Doctorat de l'Université de Lille

(spécialité informatique)

par

Jason Lecerf

Composition du jury

Président :	Pierre-Etienne Moreau	(Professeur des Universités, Université de Lorraine)
Rapporteurs :	Julia Lawall	(Directrice de recherche, Inria)
	Johan Fabry	(Ingénieur de recherche, Raincode Labs)
Directeur de thèse :	Stéphane Ducasse	(Directeur de recherche, Inria Lille - Nord Europe)
Encadrant de thèse :	Thierry Goubier	(Ingénieur de Recherche, CEA-LIST)

Département Architecture, Conception et Logiciels Embarqués
CEA-LIST

Acknowledgements

First of all, I would like to thank my advisor Stéphane Ducasse for his continuous involvement in this thesis. Despite the distance and the resulting complications, he managed to stay up to date and give me critical feedback to finish my PhD. He also introduced me to members of the Smalltalk community and of his lab in Lille, many of which are now my friends.

I would also like to thank my day-to-day supervisor Thierry Goubier. Without his knowledge and guidance, I would have drowned in the parsing abyss. Our conversations, work-related or not, were always insightful.

Besides my supervisors, I would like to thank my thesis committee: Julia Lawall and Johan Fabry for their reviews on this text and their precise questions during my defense, but also Pierre-Etienne Moreau for presiding said defense.

My next thanks go to my friends at CEA for making the daily grind exciting in the best periods and bearable in the worst ones: Pierre-Guillaume, Thibault, Thibaut, Guillaume, Gabriel, Johannes, Vincent, Joël, Aurore, Emmanuel, Dina... I probably forgot some of you, but know that if you cheered me up after a disastrous review from reviewer 2, came by for a game night or had heated beer arguments with me, you deserve your spot here.

And last but not least, my special thanks go to my almost life-long friends. Thomas Poullain is always there to discuss anything and everything, and crucially is not afraid to strongly disagree with me on tons of topics. Auriane Duquesne continuously supported me even in her harshest moments, always seeing a bright side to every situation. Gautier Berthou is one of the few souls that does not relinquish talking about personal problems, and solved most of mine in the process. My room-mate Alexandre Coden trained my cat to be the cute little pest he is (among other things). Douglas Raillard, Sergueï Lallement, Romain Tching Chi Yen and Sarah Leclerc made the last eight years feel like a walk in the park.

2

Contents

Chapter 1
Introduction

Contents

1.1 Context

Software systems age away by increasing in complexity, reducing in maintainability to satisfy priorities or simply by its experts rarefying. The loss of knowledge of a system calls for diverse solutions. Rewriting the system from scratch is one of them but feels like a waste of invested time and resources. Program exploration and program understanding [KDSD+09, Vok06, GJ05] tools help bridging the knowledge gap, but require a time investment difficult to outsource. For highly complex projects with systemic problems, there is a clear need for modernization. Modernization opportunities can be discovered by bug checkers [BNE16], but in the end, modernization solutions are implemented through system transformations.

As a result, developers often face the need to perform code transformations over large bodies of code [SAE+15, AL11]. Such code transformations are tedious and error-prone to perform when done manually, meaning developers need tools [BPM04] to assist them in these tasks. Semi-automated code transformations based on pattern matching facilities are 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 (IDE). Those semi-automated code transformations are based on writing two patterns, a matching pattern and an associated transformation pattern. A rewriting engine parses the

matching pattern, finds all its occurrences in the source code and transforms each occurrence using the transformation pattern. With a simple set of two patterns, developers are able to rewrite all the targeted code fragments in very large code bases.

Amongst the common use cases for code transformations, refactorings [RBJO96, RBJ97, Fow18] are the most mature in terms of their tooling quality. Refactorings are a special subset of code transformations, where the functional behavior of the program being transformed must be preserved after the transformation. Refactorings are usually designed to improve code quality by increasing maintainability, readability or performance. These kinds of transformations are very attractive to developers because they ensure that the functional semantics of the code are kept, meaning it cannot break their code through the transformation process. However, formal verification of refactorings is a difficult problem, with only simple ones with reasonable verification strategies; verifying more complex refactorings remains a computation-hungry task. In practice, a refactoring engine uses preconditions-postconditions; properties are checked before the refactoring to ensure that the transformation can be applied and after to ensure the transformation has been successful, where maintaining functional behavior is assimilated to an absence of regression in testing. So the transformation is neither proven as correct, nor is functional behavior considered apart from the tested expected behaviors.

Another complex modernisation task is migrating code from one language to another. There is no perfect binding between two languages where every language feature in the first always has an equivalent in the second language. In addition, a problem of missing code appears if the target language does not possess a certain library present in the original language. For both these reasons, transformations for migration do not inherently preserve functional behavior and require arbitrary, even incomplete, transformations. Migration transformations need flexibility for experts to craft transformations outputing correct and readable code.

Refactorings and code migrations already require a large amount of language knowledge to be written and as such must be crafted by language experts. On the other hand, there are other use cases where transformations would ideally be crafted by standard users. Such cases include source-to-source transformations for optimization purposes [LRDJ17] where multiple transformations are applied to a program before the compilation of a High Performance Computing (HPC) program to perform optimizations that are badly handled by the compiler alone. Typically, transformations for optimization require in-depth knowledge of the target architecture and experts in both language transformation and hardware architecture are hard to come by. Therefore, they are usually handled by standard users (from the point

of view of language transformations) who need easy-to-craft transformations without deep knowledge of the language's intermediate representation (IR).

Code specialization is another example of source-to-source transformations operating just before the compilation. This use case covers simple transformations of variables to constant values, to fit multiple particular use cases with a single generic source, all the way to custom data layout transformations [NRL17] for a specific target architecture. Code specialization in itself combines the metaprogramming and aspect-oriented programming paradigms: a generic code base is translated into target-specific code bases based on aspects implemented through scripted code transformations.

We focus our efforts on the two previous use cases: code migration and specialization. The first one is performed by language expert users, while the second one is performed by users with experience in their domain but without language expertise. Both use cases emphasize arbitrary and source-to-source transformations. In the next section, we will create requirements for a source-to-source transformation engine, featuring arbitrary transformations, from these user profiles and use cases. These requirements will serve as a basis to implement a source-to-source transformation engine targeting code migration and specialization.

1.2 User requirements

Mainly from the last two use cases (migration and specialization), we derive requirements for an arbitrary code transformation engine, from the user's point of view.

Multi-language. The code transformation engine must be multi-language. Since code bases usually feature multiple languages for different purposes, it should reflect in the code transformation engine. If the user already knows said languages, it should be no harder to match one or the other and it should be easy to switch between known languages.

No IR knowledge required. The code transformation engine must not require the user to know internal representation of languages. For standard users, it is important to not rely on language IRs, meaning they must be able to craft patterns and transformations with no prior knowledge of the IR of the language they are currently transforming. This greatly increases the accessibility of the tool to target standard and expert users alike.

Hybrid approach. The code transformation engine should allow access to the intermediate representation. While the default setting for matching and transforming should be independent of the IR, not all pattern and transformations are easy to craft in such a language-agnostic way. For experts, who already know the language and its IR, the engine should also let them express their patterns and transformations according to the IR. An hybrid approach defaulting to "no IR" is desirable.

DSL-supported. A well known option to improve productivity and accessibility in niche domains are Domain-Specific Languages (DSL). DSLs provide custom abstraction for a specific domain, in our case, pattern matching and transformation of source code, hiding implementation details away from the users. To help new users and non-expert users, a DSL for source-to-source code transformations should be implemented on top of the engine.

These user requirements affect the design of the transformation engine, yielding additional requirements for the engine implementor, discussed in the following section.

1.3 Implementor requirements

Easy language back-end implementation. The code transformation engine should support efficiently the implementation of additional back-ends for additional languages. A language back-end for transformation tools encompasses a parser, pattern matching and transformation integration with the parser. These language back-ends constitute a major engineering effort when written from scratch for each new language. To easily support multiple languages for the same transformation engine, the language back-ends must either reuse existing on-the-shelf back-ends or have tooling to ease their implementation.

Seamless hybrid integration. The pattern matching engine must not rely on separate engines for IR-based patterns and non IR-based patterns. To support an hybrid approach with patterns using or not using IR information, there must exist a high level of integration between both kinds of user patterns. This essentially means being able to combine both at will without having to support two distinct transformation engines, each for a different kind of matching mechanism.

Interchangeable DSL front-end. The code transformation DSL should be ameneable to different syntax styles. The code transformation DSL must have a strong modular back-end on top of which different syntaxes could be implemented. A major point of adoption for DSLs is the familiarity of the syntax and what it changes for different user groups. It should be possible to easily craft a new front-end and link it to the transformation back-end.

1.4 Our approach in a nutshell

How do we build a source-to-source code transformation engine whose transformations require no prior knowledge of language constructs at a reduced cost for the engine implementor?

To answer this question and meet the requirements exposed in the previous section, we choose to design a pattern matching engine whose pattern use the same syntax that their matched language as a default, but can also be expressed by refering to its IR. The pattern matching engine is coupled with a Generalized LR parser generator to both generate the IR from the source code, but also to automate part of the matching process to hide the IR complexity from the user. To reunite parsing of the source code and matching of the pattern under the same algorithm, we adapted the *parsing as intersection* result [BHPS61] to pattern matching purposes. Direct IR matching can still be expressed in the patterns as more traditional approaches, but the default matching does not require it. In that way, we build a language-IR-agnostic code transformation engine.

1.5 Contributions

The main contributions of this thesis are:

- An application of parsing as intersection to a GLR parser generator to ease the implementation of language back-ends required for hybrid pattern matching,
- A GLR parsing algorithm based on the scheduling of isolated parsing alternatives to allow for classic LR side-effect disambiguation in GLR,
- A reflection on the required back-end components for a code transformation engine in the context of a specific LR/GLR parser generator.

1.6 Structure of the dissertation

Chapter 2 presents an overview of the domain, at a crossroad between parsing, pattern matching and rewriting engines.

Chapter 3 explains how to automatically obtain a source code pattern matching engine from language grammars through a modified version of parsing as intersection.

Chapter 4 describes a compromise algorithm between the two main implementations of Generalized LR parsing providing a clearer fork model and allowing side effects to occur during parsing.

Chapter 5 concludes this work, discusses challenges in the domain and future work to reach a complete multi-language source-to-source code transformation engine.

Chapter 2
State of the art

In this chapter, we will go over the vocabulary of transformation engines, in addition to exploring similar works in the domain. Most pattern-based source-to-source transformation engines parse the program to transform and create an intermediate representation. As such, the first part of this chapter is dedicated to grammars, languages and parsing to understand how IRs (in particular AST) are built from its source. Pattern matching consists in interpreting a pattern and finding its occurences in the program IR previously built from parsing the program source. The pattern matching step is the focus of the second part. The last part of this chapter details what kind of transformations can be performed on the occurences of the pattern.

Contents

2.1 Languages and grammars

In this section, we will explain how to obtain an IR, starting from a grammar in the process called parsing. Since parsing usually refers both to the complete process of deducing an IR from an input string and the parser component in itself, Figure 2.1 disambiguates the two.

Figure 2.1: Parsing workflow

Scanner. Before parsing, a scanner preprocesses the input character stream by tokenizing it according to token definitions expressed through regular expressions. Working as the token level is far easier for the parser, since working directly with single characters as tokens tend to explode the number of rules in the grammar and add a lot of ambiguities. However, there are *scanner-less* approaches to parsing [EKV09], where the base element of rules are characters and not tokens.

Parser. A *parser* creates a representation of the string if it belongs to the language through a process called derivation. The semantics of the language then uses said representation. The two most common representations created by a parser are the derivation tree and the abstract syntax tree (AST).

In the rest of this work, we assume a parser in working in tandem with a scanner unless specified otherwise. Now that the scanner is out of the way, we will explain the derivation of an input string and how to create a derivation tree or an AST from this derivation, both being the basis for parser construction.

2.1.1 Grammars

A reader already familiar with grammars may directly skip to the next section. As a foreword, although other types of grammar exists, for the rest of this work, a grammar will refer to a Context-Free Grammar (CFG) unless specified otherwise. A CFG is a subclass of grammars that parses context-free languages. CFGs are less expressive than other classes of grammars but are typically sufficient to parse programming languages and, crucially, they allow for workable implementation of parsers without prohibitive runtimes.

A CFG is a set of production rules describing a language. Let G be the tuple $G = \{T, N, R, S\}$. T is the set of terminal symbols in the language. Terminals are the most basic elements in a language, usually associated with words. N is the set of nonterminal symbols in the language. Nonterminals are more complex elements that are abstracted away from other elements through production rules. R is the set of production rules of the language. A production r is of the form $A ::= \alpha$ where α is a string of symbols in $T \cup N$ or the empty symbol ϵ. S is the starting symbol of the grammar ($S \in N$). Usually, the grammar is extended with a unique starting symbol S' such as $S' ::= S$ and $S' \notin N$.

Grammar derivation is the process of applying consecutive derivation steps to transform the start symbol S' into the string to recognize. A derivation step $\beta A\gamma \Rightarrow \beta\alpha\gamma$ derives $\beta\alpha\gamma$ from the application of the rule $A ::= \alpha$. A is transformed as α in its context $\beta A\gamma$ by recognizing that α is another form of A. If the derivation process terminates with the string s through zero or more derivation steps $S' \overset{*}{\Rightarrow} s$, the string s belongs to the language defined by the grammar.

Example. The grammar in Listing 2.1 is described in a variant of the Extended Backus-Naur Form (EBNF) format. This grammar G_1 is defined by $T_1 = \{\text{“}a\text{”}, \text{“}b\text{”}\}$, $N_1 = \{S, E\}$, the set of production rules $R_1 = \{S ::= E\text{“}a\text{”}, S ::= \text{“}a\text{”}, E ::= E\text{“}b\text{”}, E ::= \text{“}b\text{”}\}$ and a unique start symbol such as $S'_1 ::= S$.

```
S
    : E "a"
    | "a"
    ;

E
    : E "b"
    | "b"
    ;
```

Listing 2.1: Toy grammar for the language `b + a`

As an example, to derive the string `bba` from G_1, we need to go through the derivation steps described in Listing 2.2.

```
S'              -> start
S               -> S' ::= S
E "a"           -> S ::= E "a"
E "b" "a"       -> E ::= E "b"
"b" "b" "a"     -> E ::= "b"
```

Listing 2.2: Derivation of `bba` using the grammar of Listing 2.1

2.1.2 Sentence representation

Derivation tree. The most simple representation of a sentence is the derivation tree (or parse tree) of the string. The derivation tree is obtained by remembering the path the derivation took throughout the grammar and creating nodes for each rule whose children are the nodes of the subsequent rules.

```
<number> : [0-9]+ ;
<identifier> : [a-z][a-zA-Z0-9_]+ ;

Exp
    : Exp "+" Exp
    | Exp "-" Exp
    | Operand
    ;

Operand
    : <number>
    | <identifier>
    ;
```

Listing 2.3: Toy grammar for an arithmetic expression language.

Listing 2.3) represents a new grammar to handle a subset of arithmetic expressions.

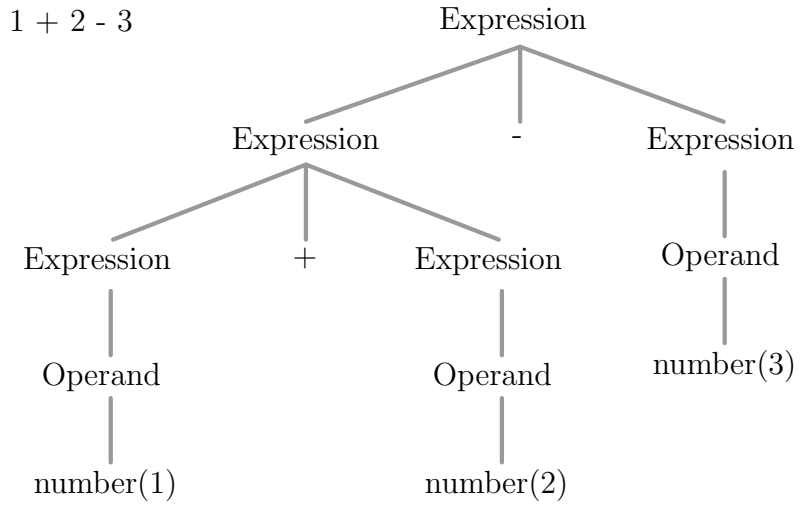

Figure 2.2: Derivation tree for the string `1 + 2 - 3` according to the grammar of Listing 2.3

The derivation tree for the input string `1 + 2 - 3` is represented in Figure 2.2. The tree follows the exact structure of the grammar rules, leading to multiple nodes of the same type with different kinds of children and superfluous intermediary nodes.

Abstract Syntax Tree. Derivation trees are basically a one-for-one mapping of production rules in a tree form. However, in most cases they yield a tree difficult to interpret, cluttered with intermediary nodes, unnecessarily increasing its depth. For this reason, parsers tend to generate an Abstract Syntax Tree (AST) instead, where intermediary recursion nodes can be converted to collections. They are many ways to create an AST, the main point being to abstract away from the concrete syntax of the language and get a clear structure that is useful to further tools down the chain.

```
1  <number> : [0-9]+ ;
2  <identifier> : [a-z][a-zA-Z0-9_]+ ;
3
4  Exp
5      : Exp "+" Exp        {{Addition}}
6      | Exp "-" Exp        {{Subtraction}}
7      | Operand
8      ;
9
10 Operand
11     : <number>           {{Number}}
12     | <identifier>       {{Variable}}
```

```
13      ;
```

Listing 2.4: Toy grammar for an arithmetic expression language with AST generation.

Expression example. The grammar in Listing 2.4 is enhanced with custom AST node creation. Nodes given between double curly braces are created when the rule is applied and when omitted, their values are pulled up from previous rules without creating intermediary nodes.

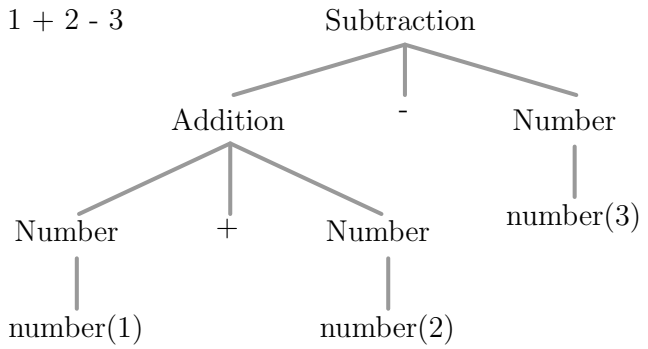

Figure 2.3: AST for the string 1+2−3 according to the grammar of Listing 2.4

An AST example for the same string `1+2-3` is given in Figure 2.3. This AST is more compact than the derivation tree and its nodes have useful names for subsequent analysis.

Collection example. Listing 2.5 adds an extra production to the previous grammar in Listing 2.4.

```
1 ExpList
2     : ExpList "," Exp
3     | Exp
4     ;
```

Listing 2.5: Additional production rule to handle lists of expressions in Listing 2.4.

The production `ExpList` is used recursively to create a comma-separated list of expressions.

In Figure 2.4, the resulting parse tree is on the left and the modified AST is on the right. The recursion introduces additional uninteresting `ExpList` nodes that are replaced in the AST by collections of `Exp` and `","`. Note that multiple other techniques could have been employed to create an AST,

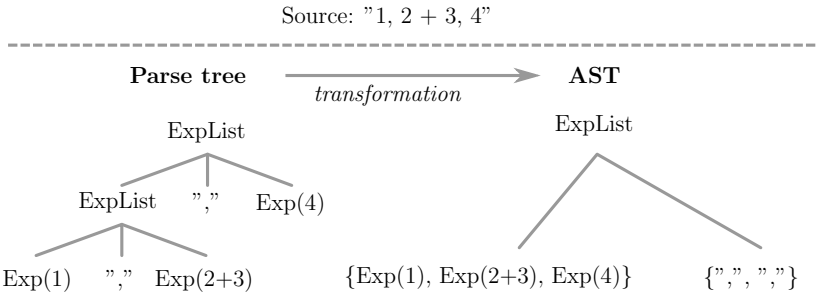

Figure 2.4: AST transformation of recursion to collections.

depending on its future use, but both techniques presented in this section are common to most engines.

2.2 Parsing in practice

Parsing is the process of deriving a structure from a string of language elements (non-terminals and terminals) according to rules defined by the grammar of the language in question. When an input string has multiple valid representations according to the grammar, the grammar (and input string) is said to be ambiguous. The root cause of ambiguities is the possibility to apply different rules at a single point during the parse. Parsers derived from such grammars have a worse runtime since they need to pursue all the possibilities. Deterministic languages are void of ambiguities and are parsed by classic parsing techniques, whereas generalized parsing techniques should be used to parse ambiguous languages.

The most common deterministic and generalized parsing techniques will be detailed in this section. For more, the author refers the readers to the *Parsing Techniques, a Practical Guide* book [GJ08b], a massive archive of parsing algorithms from their inception to 2008.

2.2.1 Deterministic parsing

Deterministic parsing techniques offer efficient parsing without backtracking for unambiguous grammars. Programming language grammars, contrary to those of natural languages, tend to be almost deterministic with a few identifiable ambiguities. Deterministic techniques have been designed to parse efficiently deterministic languages. Deterministic parsing algorithms are of three main flavors: LL parsing, LR parsing and parser combinators.

LL parsing. LL [RS70] stands for Left-to-right Leftmost derivation. LL

parsers are top-down parsers that operate the derivation top-down, meaning it start from the rules and find the terminals. In fact, it starts from the starting symbol of the grammar, predicts the next rule to apply based on the next word (called lookahead), applies the rule and so on.

LR parsing. LR [AHU74] stands for Left-to-right Rightmost derivation. LR parsers are bottom-up parsers that operates in the inverse, they start with the string and try to make sense of the string by applying the rules bottom-up until the start symbol is reached.

Parser combinators. Parser Expression Grammars (PEG) [For04] using the Packrat [Gri04] algorithm operates on different grammars than CFGs, grammars where the unordered choice of rules is replaced by an ordered choice. It means that for a given non-terminal, the second rule can only be applied if the first rule did not match. This ordered choice forces the grammar to be unambiguous by choosing the first rule (in order) in the event of an ambiguity.

2.2.2 LR parsing

LR usually requires a number of lookahead tokens for the parser to take a decision: LR(k) has k lookahead tokens, LR(1) one and LR(0) none. LR(k) is proven to be the most general non-ambiguous parsing technique for deterministic CFGs. In addition, a rewrite of a LR(k) grammar, with k tokens of lookahead, to a LR(1) grammar is always possible [Knu65]. LR(1) grammars are therefore sufficient to parse deterministic CFGs. Combined with their parse time depending solely on the length of the input and not its depth, both aspects make it an attractive option to parse programmming languages.

LR parsers [AHU74] are table-based bottom-up parsers: table-based because they rely on generated parsing tables and bottom-up because they create their derivation tree starting from the leaves (terminals) and progressively generalizing until the root non-terminal is reached. They are also called shift-reduce parsers because they execute two main actions, namely shift and reduce.

LR parse table. The standard way to create an LR(1) parser from an LR(1) grammar is to derive a pushdown automaton from the grammar. The automaton is then converted into two tables. The *action* table states which LR actions are available from each automaton state according to the lookahead token. In the case of a shift, the information in the *action* table is the

state to shift to. Otherwise, for a reduce, it contains the grammar rule to apply to perform the reduction. The *goto* table holds the reduction information: the next state to push on the stack after the reduction.

LR actions. LR parsers execute four kinds of actions: shift, reduce, accept and error. Shift actions push a new state on the state stack and accept the current token. Reduce actions remove the last m states from the state stack (where m is the size of the right hand side of the rule associated to the reduce) and pushes a new state onto the state stack. No token is consumed during a reduce. Shift and reduce are the two main actions and further references to LR actions will refer to shift and reduce. The error action occurs when there is no valid action for the lookahead token, terminating the parse with an error. In contrast, the accept action acknowledges a valid parse and terminates without error.

Figure 2.5 shows one example of stack activity for each of the four LR actions. Given a state stack and lookahead on the first line, the parser will get an action from the parse tables, resulting in the state stack and lookahead on the second line.

Shift

state stack	*lookahead*	*action*
[1 5 7]	"a"	shift 9
[1 5 7 9]	nil	

Error

state stack	*lookahead*	*action*
[1 5 7]	"a"	none
[]	nil	

Reduce

state stack	*lookahead*	*action*
[1 5 7]	"a"	reduce 2 states, goto 13
[1 13]	"a"	

Accept

state stack	*lookahead*	*action*
[1]	EOF	accept
[0]	nil	

Figure 2.5: Example of applying an LR action of each type (Shift, Reduce, Error and Accept) on the state stack and lookahead token of the parser.

LR runtime in a nutshell. The LR runtime starts with an initial state on its state stack. The parser asks the scanner for the next lookahead token, then gets the next possible LR action for this token in its parse tables. If the token is not recognized, i.e., if the parser gets an error action or no action, the parser stops. Otherwise, it executes the next actions until the next shift, where it gets a new lookahead token and begins a new loop. Once an accept action is reached, the input string is recognized.

Parser generation. Creating the automaton and deriving the parse tables from it is a cumbersome and error-prone task when done by hand. Therefore most LR parsers are in fact generated by a parser generator that compiles a grammar into a scanner, parse tables and the LR runtime. Yacc [LMB92, Joh75] is an early example of a generator producing a table-based scanner-parser combination. Yacc inspired a plethora of LR/LALR parser generators such as GNU Bison [Lev09] and SmaCC [BRPP10], but also parser generators for other parsing techinques such as LL(*) [PF11, AKB15] with ANTLR [Par13, BP08].

Derivation tree generation. To generate a derivation tree of the input string, it is common to follow a similar process to the one of the state stack. Now, the parser holds two separate stacks, the state stack to handle the path in the automaton and the node stack to build the derivation tree. The shift actions now also push the current token on the node stack. The reduce actions now also pop the last m items from the node stack and push a new node with these items as children onto the same stack.

For a derivation tree, the node type of the created node is the symbol name on the left-hand side of the reduce production rule. To build another structure (typically an Abstract Syntax Tree or AST), arbitrary execution is allowed on reduce in the form of semantic actions. The result of the execution of a semantic action is the node that will be pushed onto the node stack, resulting in more freedom in the construction of the tree for a better semantic analysis. As an example, Figure 2.6 shows one shift of token on the node stack followed two reduces into nodes `Num(2)` and `Add(1+2)`, according to the grammar in Listing 2.4 (node names have been shortened for conciseness).

Semantic actions. Semantic actions are used to alter the parse *result* (creating an AST instead of a derivation tree) but also to alter the parse *process* by directly modifying the parser state on a reduce (a side effect).

The main side effect used in semantic actions is to manipulate the lexical analysis, for example by adding symbol information into the lexical analysis. A typical case is the lexical analysis of the C language, where a distinction must be made between typenames and identifiers, based on the existence of a previous definition (famously called the "lexer hack"). Otherwise typenames and identifiers, which have the same syntactic definition, can appear in a lot of the same rules, leading to many ambiguities.

Shift

state stack	*node stack*	*lookahead*	*action*
[1 5 7]	[Num(1) "+"]	"2"	shift 9
[1 5 7 9]	[Num(1) "+" "2"]	nil	

Reduces

state stack	*node stack*	*lookahead*	*action*
[1 5 7 9]	[Num(1) "+" "2"]	"-"	reduce 1 state, goto 11
[1 5 7 11]	[Num(1) "+" Num(2)]	"-"	reduce 3 states, goto 3
[1 3]	[Add(1+2)]	"-"	

Figure 2.6: AST node creation through a node stack.

2.3 Generalized parsing

Generalized parsing is powerful since it can resolve local conflicts without having to manually craft code for each conflict. First we will come back to ambiguities and conflicts, then explain which parsing algorithms solve conflicts. The end of the section is focusing on Generalized LR parsing, the generalized variant of the previously described LR approach (see Section 2.2.2).

2.3.1 Ambiguities and LR conflicts

Conflicts. In LR parsing, when multiple actions could be executed from the same lookahead token from a single state, there is a conflict. LR conflicts exist in two forms: shift-reduce and reduce-reduce. Shift-reduce means the parser either shifts the current token or reduces using a rule, whereas reduce-reduce is strictly between two different reduces. Note that if a generated LR parser has conflicts, the grammar is not deterministic anymore. Depending on implementations, LR parsers either stop or just pick one of the alternatives and continue parsing.

Figure 2.7 is an example of a famous shift-reduce conflict in grammars of expressions (see Listing 2.6.

```
1 <number> : [0-9]+ ;
2 <identifier> : [a-z][a-zA-Z0-9_]+ ;
3
```

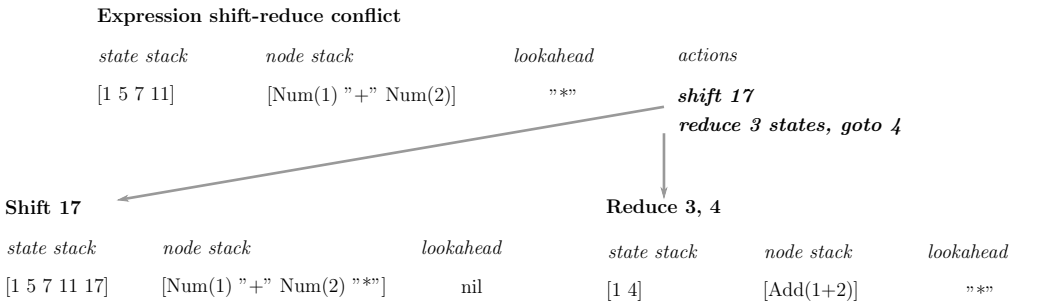

Figure 2.7: Example of a shift-reduce conflict in the LR parser of an expression grammar.

```
Exp
    : Exp "+" Exp        {{Addition}}
    | Exp "*" Exp        {{Multiply}}
    | Operand
    ;

Operand
    : <number>           {{Number}}
    | <identifier>       {{Variable}}
    ;
```

Listing 2.6: Ambiguous grammar for expressions.

When the parser reaches `1 + 2` in the input `1+2*3`, the next token is `"*"`. The parse has two choices: shifting the token to end up with the final AST `Add(1,Mult(2,3))`, or reducing the current stack to end up with the final AST `Mult(Add(1,2),3)`. In this case, even if one makes more sense than the other from a semantical point of view, both are valid according to the grammar. Since the two different ASTs result from a single string, the shift-reduce conflict here has caused an ambiguity.

Non-deterministic and unambiguous. A deterministic grammar is parsed in linear time by a deterministic parser. Deterministic parsers do not feature any conflict at any point in the grammar. In LR, this translates to never having more than one action available at any time. However, even if a grammar is *non-deterministic* (meaning it has conflicts), it can still be *unambiguous*, because a "local" conflict could be resolved before the end of the parse.

From a parser point of view, if a generalized parser answers one AST for a given string, this means the string is unambiguous according to the grammar of the language. Even if conflicts appear in the middle of the parse, if a single solution subsists, it means the other (faulty) ones originating from the

conflict lead to errors and got discarded. If the parser answers two or more ASTs however, the grammar is ambiguous. Figure 2.8 illustrates the three different possibilities here: deterministic, ambiguous or non-deterministic unambiguous.

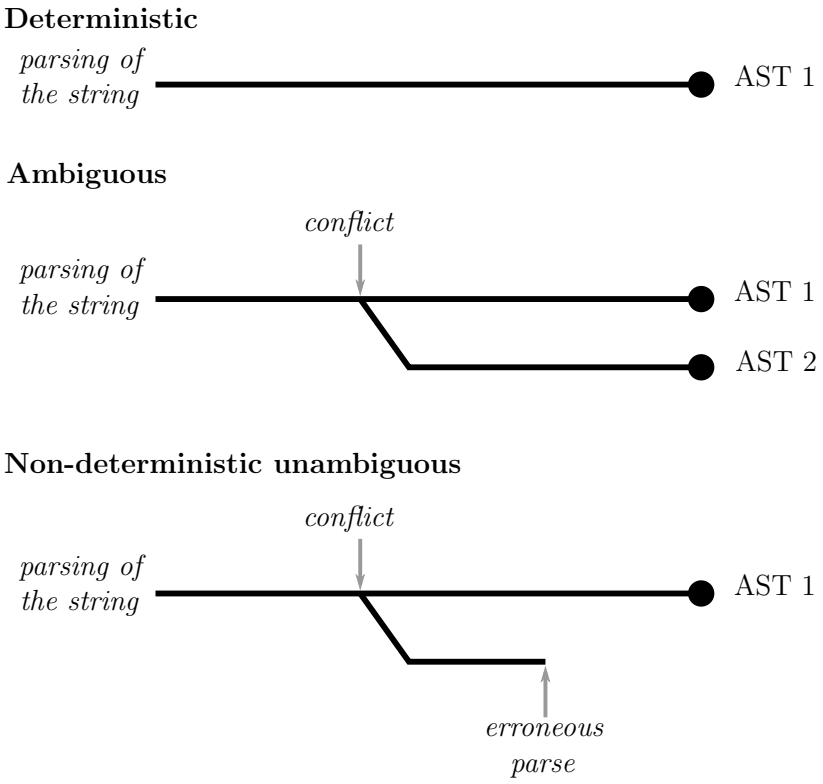

Figure 2.8: Conflicts, ambiguities and generalized LR parsers.

Most programming languages have non-deterministic unambiguous grammars, and even if some can be converted to deterministic grammars, the resulting grammars are often much more complex and hard to maintain.

2.3.2 Generalized parsing algorithms

We identify four classes of generalized parsing algorithms, most of them originally designed to handle natural languages.

Earley. Earley's algorithm [Ear70] builds a set of states using three steps: prediction, scanning and completion. The prediction phase expands the existing states of the form $A ::= \alpha \bullet B\beta$ in the set with $B ::= \bullet\gamma$. The scanning phase recognizes terminals by adding the state $A ::= \alpha b \bullet \beta$ from $A ::= \alpha \bullet b\beta$

when the lookahead is b. The completion phase performs a similar job to LR reductions by adding the state $A ::= \alpha B \bullet \beta$ from $A ::= \alpha \bullet B\beta$ and $B ::= \gamma\bullet$. If at the end of the stream, the states contain the end state, the parse succeeds otherwise if fails. Earley has a complexity of $O(n^3)$ in the general case, $O(n^2)$ on unambiguous grammars and $O(n)$ on deterministic grammars.

CYK and Valiant. CYK's algorithm [You67], contrary to the other algorithms presented here does not work with BNF grammar, but operates on a Chomsky Normal Form (CNF) grammar. CNF's production rules only have two symbols on their right hand side, meaning they must be of the form $A ::= XY$ where $X, Y \in T \cup N$. The CYK algorithm builds a table M with rows giving the recognized nonterminals in the substrings of size 1 to the size n of the string to recognize. Each $M(i, j)$ holds the set of recognized nonterminals for the substring $s[j; i + j - 1]$, so that $M(n, 1)$ holds the set of recognized nonterminals for the entire string. Valiant [Val75] improved the original $O(n^3|G|)$ complexity of CYK by using multiplications of boolean matrices to lower the input size factor. Although it is possible to transform BNF grammars to CNF, it leads to an explosion of the number of rules and since this algorithm scales with the size of the grammar, the runtime for non-worst-case grammars is usually better with other algorithms. We do not know of any grammar that justifies the use of CYK or Valiant, since the constant factor is far too important.

GLR. Originally Tomita's algorithm [Tom87], the Generalized LR (GLR) parsing algorithm generalizes the classic bottom-up parsing algorithm to non-deterministic grammars. GLR is implemented in one of two main ways: list of stacks or Graph-Structured Stack (GSS), both of which we will discuss in the following section and in Chapter 4. As implied by the name, GLR parsers use the same parse tables as a standard LR parser. In terms of runtime, GLR scales with ambiguities, meaning if ambiguities accumulate the parse time will explode whereas few ambiguities locally resolved will have a parse time close to linear. In fact, Tomita's algorithm has a $O(n^{k+1})$ where k is the size of the longest rule, so for pathological grammars on the other hand it yields disastrous parse times. For both these reasons, we will take a longer look at GLR in the rest of this work.

GLL. Generalized LL parsing (GLL) [SJ10] is a generalized variant of the popular top-down LL parsing algorithm. As its non-generalized variant, GLL using a recursive descent through the grammar, mimicking a derivation by

hand in the grammar. GLL also uses a GSS similar to the one of GLR with the addition of loops to handle left recursion. To the best of our knowledge, GLR and GLL are the two generalized parsing techniques used in practice to parse programming languages.

In the next section, we will discuss the two main implementations of GLR and explain their differences and shortcomings.

2.3.3 Generalized LR parsing

Premise of GLR

The gist of Generalized LR (GLR) parsing is that, when the parser reaches an ambiguity, it processes both parsing alternatives in parallel. On one hand, if the grammar is truly unambiguous, along the way, one of the parses should reach an error state. In that case, the faulty parse is discarded and the remaining parse alternative resumes its execution. On the other hand, if the grammar is truly ambiguous, both parses will successfully finish yielding two parse trees. For more than one ambiguity, the process is simply repeated for each ambiguity, leading to possibly many parse alternatives in parallel yielding a parse forest. GLR parsing can be viewed as a breadth-first traversal of the grammar rule alternatives, contrary to a backtracking approach which is a depth-first traversal.

Two main approaches exist to implement GLR parsing: the list-of-stacks approach and the Graph-Structured Stack approach.

List of stacks implementation

The naive way to implement a GLR algorithm is to fully copy the current state of the parser when reaching a conflict. That way, both parse alternatives are straightforward to process in parallel. The state stack from the parser is copied in another parse alternative process. If an alternative reaches an error state, it is deleted without concern for the rest of the alternatives since it is independent. However, with only this forking mechanism and without merging, this approach is very memory-hungry since for each new ambiguity, a new parsing process is spawned. A merging technique is mandatory to achieve non-prohibitive memory consumption and run time.

Example. Figure 2.9 is an example depicting what happens when a list-of-stacks GLR reaches a conflict. The parser starts with a single stack, the same as an LR parser. Once a conflict is reached, a copy (fork) of the stack is performed and each possible action of the conflict is then executed on a

separate forked stack. Each stack continues to parse as if it was alone, except that it can be merged with another one if they became identical. If a stack reaches and error, it is discarded. If multiple stacks complete, the string is ambiguous and two ASTs are produced as it is the case with this example.

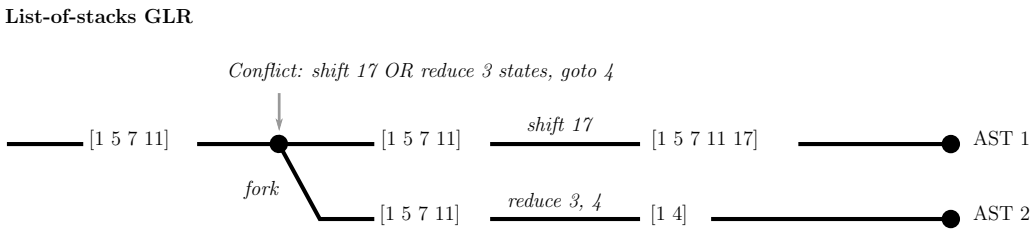

Figure 2.9: Example of parsing with a list-of-stacks GLR.

Merging in list-of-stacks GLR. Parser generators, such as Bison [Lev09] and SmaCC [BRPP10], implement a simple merging algorithm by synchronizing the parse alternatives on shift actions and merging alternatives with identical state stacks. Parsing alternatives are checked for equality once every shift, if their state stacks are equals, they are merged. However, the merge is *only* done on shifts, so it is possible that two parse alternatives performed multiple identical reductions and an identical shift before merging. It means they could have been merged earlier after a reduction, executing the following actions on a single merged parsing alternatives instead of executing the same actions on both parsing alternatives and merging later on (on a shift). This merging technique is far from perfect but achieves practical parse time when parsing programming languages with mainstream parser generators (i.e. Bison).

GSS implementation

Tomita [Tom87, TN91] designed an algorithm to share the states from the prefix of the state stacks and merge the identical top states. He called the resulting structure a Graph-Structured Stack (GSS). This representation achieves maximum state sharing, leading to a lower memory footprint than a list of stacks. A significant body of work went into implementing a complete version of this algorithm, starting with the first Tomita paper in 1984 up to RNGLR from Scott and Johnstone in 2002. We give here a short overview; we recommend reading [SJ06] for a detailed overview of its state of the art and implementation.

Intuitively, the algorithm is equivalent to sharing identical stack prefixes and suffixes in a list-of-stacks implementation, by combining identical states

into a single one. The resulting GSS is a graph with the maximal number of states shared. However, to construct the GSS in practice, the duplicate parts are never added in the first place and only non-duplicated states are added.

A GSS is constructed to never unnecessarily add an already existing state. The GSS is organized in successive *frontiers* synchronized on shifts. Frontiers are composed of a set of states, and each state in the current frontier is connected to states in the current frontier (via reduction arcs) or to states in previous frontiers (via reduction and shift arcs). The first frontier is composed of the single starting state. Algorithm 1 shows a simplified version of the GLR algorithm, enough to understand the mechanisms of GLR and its main differences with LR.

Algorithm 1 Simplified GLR frontier algorithm

```
1: function CreateFrontier(previousFrontier, reduceSet, shiftSet)
2:     U ← ∅
3:     while reduceSet ≠ ∅ do
4:         (m, N, i) ← (reduceSize, reduceSymbol, originState) ∈ reduceSet
5:         R ← reduction paths of size m from i
6:         while R ≠ ∅ do
7:             r ← state ∈ R
8:             j ← gotoState(r, N)
9:             if j ∉ U then
10:                add j to U
11:            end if
12:            if ∄ arc(j, r) then
13:                add arc(j, r)
14:            end if
15:        end while
16:        add new actions to reduceSet and shiftSet
17:    end while
18:    while shiftSet ≠ ∅ do
19:        (i, j) ← (originState, destState) ∈ shiftSet
20:        if j ∉ U then
21:            add j to U
22:        end if
23:        if ∄ arc(j, i) then
24:            add arc(j, i)
25:        end if
26:    end while
27: end function
```

Each new frontier computation follows the same pattern: perform all possible reductions and then execute the next shift.

Executing the next shift means going through all the states in the previous frontier, seeing if they accept the lookahead token as a possible shift, creating

the new state in the current frontier and adding a shift arc from the new state to the corresponding state in the previous frontier. If the new state already exists in the current frontier, no new state is added, and the shift arc is simply added from the existing state.

For reductions, the parser gathers all states in the current frontier that accept a reduce from the current lookahead. Then, it searches for reduction paths of the size of the production rule of the reduction down the GSS, starting from the current state down to the initial state in the first frontier. Then, arcs are added from the goto state of the reduction to the state at the end of the reduction path previously found. As with shifts, if the goto state does not exist in the current frontier, it is created with its arc, otherwise only the arc is added.

As for termination, if a newly built frontier is empty, the parse is considered terminated. If an accept state is present in the last frontier, the parse is a success, otherwise it is a failure.

For each shift and reduction, every new state in the frontier is kept unique to achieve the maximum state sharing.

Differences between list of stacks and GSS

While a list-of-stacks approach copies the stacks and then tries to merge them to reduce the duplication of states, a GSS approach does not, by construction, duplicate states between stacks.

A list of stacks is a local structure, each stack has only knowledge of its own stacks and only when merging does it interact with the other stacks. It means states are duplicated, but it also means we are always working with a single stack. The standard LR shift and reduce algorithm are reused, semantic actions can even be executed. The only additions are: forking when there is a conflict and trying to merge after before every shift.

On the contrary, a GSS is a global structure, at any time, it contains all the states the parse went through. Since it is global, it checks whether a new state should be added or if it already exists and only an arc (symbolizing a shift or a reduce) needs to be added to the GSS. However, since the GSS is a global structure, individual stacks cannot be modified without impacting other stacks. Stack independence is important for reduces with semantic actions, since code needs to be executed *while* reducing. Indeed, on a stack, it is obvious which last m states to remove since it is linear, but on a graph, multiple paths must be considered and new arcs could still be added in the future. For these reasons, semantic actions and AST construction are done once the GSS is complete, when the parse is finished.

Table 2.1 summarizes the differences of approach between a list of stacks and a GSS.

Table 2.1: List of stacks and GSS differences in GLR

	List of stacks	GSS
States sharing	duplication of states	maximal state sharing
Structure	local	global
Similitude to LR	high	limited
Reductions	online	at the end

2.3.4 Improvements and variants of GLR

Tomita's algorithm builds a GSS where each state is present once and only once in each layer of the GSS. Tomita then builds a Shared Packed Parse Forest (SPPF) from the GSS at the end of the parse. A SPPF is a graph of all the possible parse trees of the input string, that is also constructed with node sharing in mind. A number of improvements were made to the original GLR algorithm, mainly to fix termination issues and runtime, leading to numerous variants of the algorithm.

Farschi's algorithm. Farshi's algorithm [NF91] is a correction of the original Tomita GLR to ensure termination in the case of hidden left recursion in the grammar. The fix involved modifying the way reductions are handled by the original algorithm to search down all paths in the frontier of the GSS, bringing a significant execution overhead.

RNGLR. RNGLR [SJ06] is another approach to solve the problem of hidden left recursion by modifying the LR parse tables. Reductions of size 0 are added to the parse tables when hidden left recursion is discovered during the parser generation process. RNGLR also produces more compact SPPF [Rek92] for ϵ-reductions.

BRNGLR. BRNGLR [SJE07] improves the worst case runtime of GLR from $O(n^{k+1})$ to $O(n^3)$ by binarising the recognition process (effectively limiting k to 2). Note that BRNGLR does not binarize the parse tables, only the search through the tables at parse time. The algorithm also has linear complexity on LR(1) grammars. Johnstone et al [JSE06] compares BRNGLR,

RNGLR and Farschi's GLR to demonstrate the efficiency of the right-nulled parse tables with the binarized GLR runtime.

SGLR. Scannerless GLR [EKV09] is a version of the classic GSS-based GLR parsing algorithm adapted to remove the need for a separate scanning phase to create tokens. SGLR defines disambiguation filters to improve performance by removing the additional ambiguities introduced by the absence of a scanner.

Elkhound. Elkhound [MN04] is an LR and GLR parser generator for C++. Elkhound builds an hybrid parser that switches between a LR and a GLR mode for a better parse time. The GLR part is GSS-based, but an ordering is performed on the GSS node construction, contrary to the original algorithm.

2.4 Pattern matching

After discussing parsing of the source code to retrieve an AST, we will now focus on how to match the subtrees in an AST through patterns. The differences between explicit and syntactic patterns are covered in this section.

2.4.1 Pattern matching implementors

Pattern matching is the process of finding all the occurences of a given description in a given source. Pattern matching originates from functional language and metalanguage research through languages like ML and later Erlang, Haskell, OCaml, Scala, F#, and so on. In these languages featuring built-in pattern matching, the users can ask for language structures with respect to certain conditions. The source in built-in pattern matching is directly the syntactic structures of the language. Many paradigms later incorporated built-in pattern matching to their languages, most notably Object-Oriented (OO) languages [RSDT16, GHB10, EOW07]. TOM [BCM15, MRV03] is an external pattern matching engine to bring pattern matching extensions to languages lacking its support.

Other projects created external pattern matching engines whose goals are to separate the matched language from the implementation of the engine. External pattern matching can originate from a will to support multiple languages in a single tool, or because pattern matching for a language is costly to implement in the language. Pattern matching engines are used by bigger language tools, such as code transformation engines [YKS+14], refactoring

engines [RBJO96, KBD17], search engines [DRI14] or clone detection engine [KT14]. They implement a pattern matching engine, either specific to the language they work with or more generic when dealing with multiple languages.

2.4.2 Pattern and matches

A pattern is a description of the IR structure to match in the input. Different transformation engines operate on different types of IRs. For example, Stratego [BKVV08] matches terms while Rascal [HKV12] matches subtrees in ASTs. For the rest of this work, we will focus on matching subtrees in ASTs. As such, language IR will refer to ASTs unless specified otherwise.

Classical approaches to pattern matching offer patterns that are a textual representation of a node (mainly its name). The pattern is then converted to its corresponding subtree and the pattern matching engine checks for all of its occurences or *matches* in the source AST.

Contextual information can also be added to refine a pattern's matches. This typically includes precisions on its parent node or its children to get fewer more accurate matches. Some engines offer wildcards to match any node.

Listing 2.7 is an example of a simple pattern that matches nodes with the name `ExpressionNode` whenever they are encountered in the AST.

```
1 ExpressionNode
```

Listing 2.7: Example of pattern.

Listing 2.8 is more specific, only matching such a node when its right operand is the variable `"a"`. The rest of its children are wildcards, so they are not restricted to match a specific type of node.

```
1 ExpressionNode(left=_,operator=_,right=Variable("a"))
```

Listing 2.8: Example of pattern with children information.

2.4.3 Explicit pattern matching

Pattern matching [BBC+03, VD03, JK06] feeds a pattern and a program to an engine and gets in return all the possible matches of the pattern in the program. A pattern matching engine is a search engine returning elements of said source code IR. In the context of matching programming languages, these structures are AST node types. The pattern matching engine will

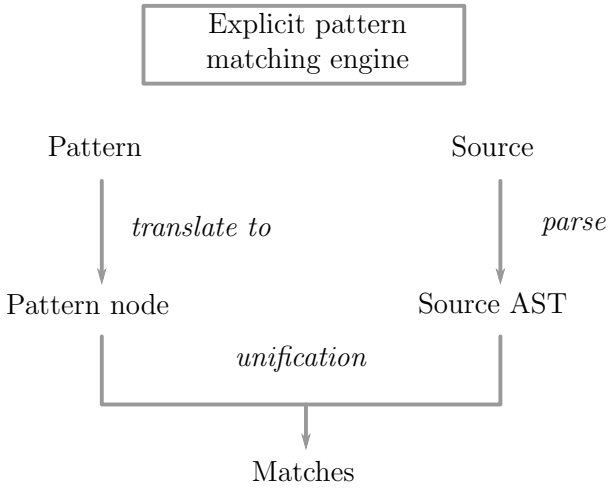

Figure 2.10: Explicit pattern matching engine.

answer all the matches of said entity by simply comparing the node type of the pattern with the ones in the input AST in a unification process (Figure 2.10).

However, for a single node matching, the set of matches can be massive for big code bases, so filtering techniques are applied before or after the match to limit the number of results to more reasonable quantities. Filtering can be done on the name of sub-entities, actual source code of the nodes, file location, and so on.

2.4.4 Syntactic pattern matching

Language ASTs are constituted of many nodes that are specific to the language. As an example, an AST for a general purpose language such as Java usually comprises of more than 200 different node types. In the context of non-expert users crafting their own patterns, this knowledge burden of the AST adds a high barrier to entry. To somewhat lift this burden, instead of *explicitly* describing the node type to match through explicit patterns, syntactic pattern matching proposes to describe the concrete syntax of what to match and letting the engine infer the node type.

Syntactic patterns are also called code templates, such as in Ekeko/X [MDR16b]. Syntactic pattern matching offers the opportunity for users to express patterns without relying on AST node types. Syntactic patterns are written using the input language, enhanced with some sort of wildcards. These wildcards hold any node type available in the language, and so does the syntactic pattern. Wildcards are also called holes or metavariables. For consistency reasons, they will be refered as metavariables in the rest of this work.

Syntactic pattern example. Listing 2.9 shows an example of a syntactic pattern designed to match a while statement from languages such as C.

```
1 while(‘aCondition‘)
2 {
3     ‘someStatements‘
4 }
```

Listing 2.9: Syntactic pattern while example.

The pattern is just like C code, with the addition of two metavariables: ‘`aCondition`‘ and ‘`someStatements`‘ acting as wildcards and matching anything inside a while and inside a block respectively. Nowhere did we have to specify the types of AST node these metavariables should have.

Retrieving IR types from syntax. Since metavariables are only variables and not explicit types, no explicit node type information is required to express syntactic patterns. Of course, since type information is absent but is still required to perform the actual matching, it needs to be inferred. Previously, the matching step only consisted in a unification process where a pattern is unified with the input AST. Now, before the unification, the pattern undergoes a "type inference" step during which a node type is inferred for the pattern and node types are inferred for each of the metavariables. Then, the unification proceeds as previously described with the output of the type inference.

Syntactic pattern matching engine are particularly interesting because they lower the barrier to entry for non-experts in language representation. We will also show in Chapter 3 that they rely on a powerful result, ensuring that the steps described here (matching and inference) are guaranteed to work for any language for which one can write a context-free grammar.

2.5 Code transformation engine

A code transformation engine automatically transforms the occurences of a pattern found in an input source according to a transformation pattern and returns the modified version.

Figure 2.11 summarises a code transformation engine architecture. The engine is comprised of two sub-engines: a pattern matching engine handling the search for the occurences of the pattern in the input and a rewriting engine handling the rewriting of these occurences according to a transformation pattern.

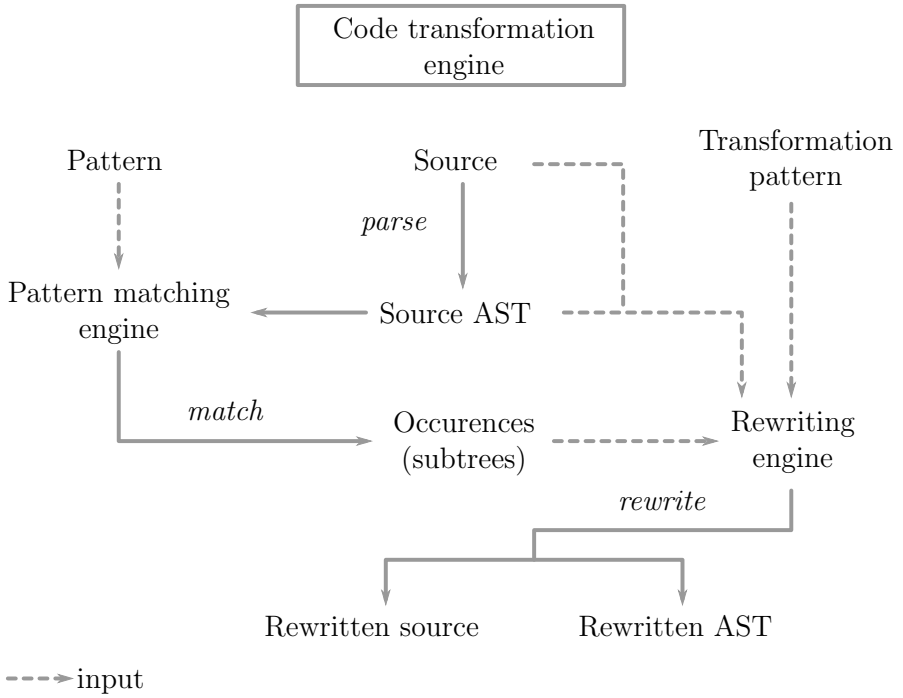

Figure 2.11: Anatomy of a code transformation engine.

2.5.1 Code transformation use cases

Typically, use cases shape code transformation engines since they implicitly derive requirements.

Refactorings. Refactorings [Rob99, RBJ97, RBJO96] are code transformations from a language to the same language while preserving the semantics after the transformation. Refactorings modify non-functional requirements such as the code size, the memory consumption, the general performance, the readability, and so on. The general goal of refactorings is improving code quality while giving the user confidence in the transformation. Refactorings are developed by language experts then integrated into IDEs for end users. Preconditions check that the code base is in a state to be refactored by a given refactoring, while postconditions validate the transformation. Refactorings are used in a variety of contexts such as code clones [KT14], software product lines [KBD17] and pattern matching [WGMH13], and all kinds of languages, even preprocessed languages such as C [OBH14, Ove13, HOBH13, Spi10, MB05, GG12] notably hard to refactor into readable code close to the original source [GJ13, GMJ06].

Custom arbitrary transformations. Arbitrary, non behaviour-preserving, transformations assume nothing on the end result of said transformations. The ability to craft custom transformations is a powerful tool in expert hands, but can be confusing and dangerous for others. Arbitrary transformations tend to be source-to-source since there is no presupposition on the output's form. Node rewriting is also a possibility, but mainly inside a language, not between languages since it would require the user to create an IR for the target language from scratch. Rascal [HKV12] is a metaprogramming language leveraging both explicit and syntactic matching through separate engines for transformation tasks. Rascal, as a full-fledged programming language and not a DSL, is aimed at language implementors and not at non-expert users. The Semantic Patch Language (SmPL) of Coccinelle [PLHM08] and its transformation back-end are designed to handle evolution of C code in Linux. The patch-like syntax, the accessibility of the patterns and the specialization to C derive from the target users: driver or kernel experts without a strong emphasis on language expertise. Ekeko/X [DRI14, MDR16a] is a transformation engine working with syntactic pattern for transformation of Java code. Ekeko/X clearly aims at simplifying pattern matching and code transformation for non-expert users.

Code migration. Code migration transforms a code base in a given language to another language. In the context of code transformation engine, it is meant to be done semi-automatically. This task usually cannot be done completely automatically since there is no 1-to-1 feature match between the old and new language. And even in the ideal case of perfect feature mapping between the two languages, due to the sheer size of code migration projects have to work with, the old code base is doomed to make use of a library present in the ecosystem of the first language, but absent in the new language. This library would need to be developed from scratch or mapped to a similar library in the new language. Code migration shares with refactoring the fact that developers will need to maintain the output code. Thus, the transformed code must be readable and close to what a human developer would have written. Such transformation finesse is hard to achieve without arbitrary transformations that let the expert not only handle the migration in itself (language mapping) but also readability of the code.

Code specialization. Code specialization is a vast umbrella under which can be put all processes of applying transformations to a generic code base to accomodate to features of a given target. The targets cover architecture specialization, OS specialization, input data specialization. Code specializa-

tion is done online through a Just-In-Time (JIT) compiler or offline before the compilation or interpretation. Code specialization is usually done from one language to the same, but it is not always true. For example, part of a C/C++ program can be modified to use a specialized piece of assembly for a specific platform for performance reasons instead of using a generic function. In our case, we are mostly interested in offline code specialization integrated in the compilation process, since this use case generally involves architecture experts writing transformations instead of experts.

Language workbenches. Language workbenches [JCB+15, WKV14, KRV10] offer a large array of features to develop languages, from parsers to refactoring engines, to code generators, and so on. They aim at providing a complete package to create language syntax and semantics from scratch with all the support of a state-of-the-art IDE. Inherently, language workbenches are designed with multiple languages in mind, meaning their underlying parsing and transformation techniques tend to be more generic than engines target to specific use cases and languages.

2.5.2 Transformation rule and transformation pattern

A transformation rule is the combination of a matching pattern and transformation pattern, the left hand side being the matching part and the right hand side the transformation part. Both sides of the transformation rule share the same context, the same binding of its metavariables. A transformation pattern is a description of the transformation to apply, and can be either imperative or declarative. Imperative transformation patterns directly describe the way to transform the matches. These snippets of code need to be written in the engine's language (or a DSL) and, given total access to the matches, perform the transformation. Declarative transformation patterns describe what the matches must be transformed into, letting the transformation engine handle the transformation in itself.

```
1 `left` + `right` => `left` `right` +
```

Listing 2.10: Declarative transformation pattern.

For example, in the rule shown in Listing 2.10, the metavariables `left` and `right` of the matching pattern are reused as context for the declarative transformation pattern yielding a reverse Polish notation addition.

2.5.3 Rewriting engine

A rewriting engine handles the transformation of subtrees in the AST according to a transformation pattern. The transformation is either a source-to-source rewrite, meaning the source of the matched nodes is modified, or a node rewrite where a subtree is replaced by another subtree.

Source-to-source rewrites. A source-to-source rewrite transforms the source code of the matched subtree into a new string. To perform such an operation, a binding between the node and its source must exist. In addition, since a pattern can have multiple (overlapping) matches for a single transformation, the transformation is usually performed by a transaction of individual transformations of each subtree. Overlapping matches and rewrites benefit from a string structure that represent the transaction and its overlapping rewrites, so they can be cancelled at any time. String origins [IVDSE14] is an example of string representation specialized for code transformation. Source-to-source transformations are fit to arbitrary transformations since the raw source of the input is modified, allowing for syntax-breaking code transformations. Code migration is a good example of arbitrary source-to-source transformations where control over the source output is crucial to produce readable code for the maintainers of the new code base.

Node rewrites. A node rewrite replaces a matched subtree in the source AST with a new subtree. Either a binding already exists between the old and the new subtree in the back-end of the engine, in which case it is done automatically without user intervention, or it does not. If the binding does not exist or the user wants different transformations than the one provided by the engine, the user must convert the subtree by himself. Most node rewriting engine give the user access to transformation primitives to replace nodes. Overbey [OJ08] shows how to generate rewritable ASTs with additional primitives to conserve the original layout.

Rewriting strategies. For a given matching pattern, there usually are more than one match and due to the tree nature of ASTs, the matches overlap each others. A pattern matching additions will match once a top level addition which can also have additions as children, which will be matched in turn. If they need to be rewritten using a transformation pattern, there are multiple ways to rewrite this set of matches. Some may need to be overwritten by a parent match transformation while other need to be transformed before their parent. Rewritting strategies control the execution of

the transformations. Stratego [BKVV08] for example is a language to define transformation rules and strategies to control the execution of those rules.

2.6 Conclusion

As it is unreasonable to demand of new users to learn the IR of the language from scratch, an engine targeting new users should use syntactic pattern matching. As it is also unreasonable to ask experts to only match using syntactic patterns and not explicit patterns for which they know the IR, an hybrid approach is desirable. The challenge now is to build a code transformation engine based on a hybrid syntactic-explicit pattern matching engine and source-to-source rewriting engine featuring rewriting syntactic patterns. The engine also needs to make it easy for the implementor to add support for new languages without having to write a language back-end from scratch.

Chapter 3
Parsing as intersection for pattern matching

In this chapter, we discuss the advantages of a hybrid explicit-syntactic pattern matching approach which defaults to syntactic. We show that the implementation of a hybrid pattern matching engine can be language-agnostic enough to only require an additional line in the grammar of a new language to be able to match this new language. This is done through an adaptation of parsing as intersection to handle the type inference in syntactic pattern matching, the typically language-dependent part of the engine.

Contents

3.1 Hybrid explicit-syntactic pattern matching

In this section, we examine deficiencies of pure explicit pattern matching and pure syntactic pattern matching. We promote an hybrid approach featuring both kinds of matching in a single pattern, and describe the additional step of type inference to add syntactic matching in an explicit pattern matching engine.

3.1.1 Deficiencies of explicit pattern matching

Traditional explicit pattern matching (described in Section 2.4.3) expects the user to *explicitly* state what should be matched. Explicit pattern matching relies on the assumption that the user knows the pattern matching engine's internal representation of the matched language.

This assumption severely limits the use of pattern matching in practice, as learning an internal representation is a significant knowledge burden. Basically, the user needs to know the exact node type of the pattern to be matched, for example it could be `IfStatement` for an if-then statement.

Table 3.1: Possible IR sizes for popular languages

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

Table 3.1 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, and yet it is still composed of over 150 node types. All these general purpose languages have sizable IRs that are tedious to learn for a user who is not a language expert. The more complex the language, the more constructs the user has to learn. The problem is much worse for languages similar to COBOL where the grammar is reverse engineered from existing code bases [VDBSV+97] (more than 300 production rules).

Of course, it is still possible to learn one language's IR for the sake of rewriting a massive code base for a migration for instance, but most of the time said code base is written in a multitude of languages handling different parts of the system. Learning each of the language IRs is time-consuming for developers. Each language has its own features, differing from the others, and even when features are similar, they usually do not use the same IR construct.

Most users do not have the time and resources to become experts in pattern matching. As a result, only those already 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 even environments that have existed for decades such as RB see very little user creation of new transformation patterns [dSS17].

3.1.2 Alternatives to explicit pattern matching

Alternatives and improvements exist to lessen the knowledge burden caused by explicit pattern matching.

Unified representation. Pushing the limits of unification of language representation is OMG's AST Metamodel (ASTM) [Obj11], which aims at unifying language programming concepts (even from different paradigms) under a common AST model. The concepts in the metamodel corresponds to core nodes available in most programming languages, such as the notion of expressions, statements, and so on, 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. The unified representation ends up being the union of all the nodes for all the languages, making the representation even more complex and more of a burden to learn. Ultimately, this approach does not solve the problem at hand.

Island parsing. Another solution to reduce the knowledge burden on the user is to use island parsing. Island parsing [Kur16] allows one to parse a small select subset of the language, that way the IR only incudes the few construct the implementer deemed interesting. As a result, it is also easier to implement at first [BNE16], as only a subset of the language is parsed and matched. However, island parsing in this context suffers three problems. First, the engine implementer decides what is interesting in the language, meaning only those parts are matched. Second, if more constructs of the language need to be matched, it is an additionnal burden on the engine implementer to add this support, since he/she is recreating a full parser a posteriori. Last, this approach works well for a single language, but as soon as the engine implementer wants to add support for a new language, all parsing and matching facilities must be rewritten from scratch.

Query-by-example. 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. While this approach works for program understanding for example, its heuristic nature makes it unsuitable to program transformation.

A better solution to alleviate the knowledge burden on the user is to use patterns written in the matched language. Since the user already knows

the syntax of the language to match, he/she should be able to write patterns that closely resemble code snippets of the language. These patterns are called syntactic patterns or code templates.

3.1.3 Syntactic pattern (code template)

Contrary to *explicit* pattern matching where the user is expected to name the type of AST node to match, *syntactic* pattern matching is based on writing snippets of code in the syntax of the matched language [PLHM08, Bar13, Cor06, DRI14, MDR16b].

```
1 IfStatementNode(_,BlockNode(_*))
```

Listing 3.1: Explicit conditional statement pattern.

As an example, matching a conditional statement in Java using an explicit pattern matching engine will look like the pattern in Listing 3.1. To create this pattern, the user would need to know that the name of the construct to look for is an `IfStatementNode`, which has two ordered children (a `ConditionNode` and a `BlockNode`). Since the user wants to match this pattern for every condition, the first child is replaced by a wildcard "_" matching anything. The second child must now be a `BlockNode` with any number of statements as its children.

```
1 if ( `condition` )
2 {
3     `statements*`
4 }
```

Listing 3.2: Syntactic conditional statement pattern.

A syntactic version of the explicit pattern of Listing 3.1 is in Listing 3.2. Contrary to its explicit counterpart, the syntactic pattern does not require any knowledge of the IR: the structure of the pattern is inferred from the pattern syntax and wildcards are replaced by metavariables. The pattern is expressed in an extension of the syntax of the *host language*, meaning the underlying language to match in which metavariables are embedded.

3.1.4 Hybrid syntactic-explicit patterns

If the user is an expert in the IR of the language to match, explicit pattern matching would be beneficial in addition to syntactic pattern matching. It should be best for a pattern matching engine to support both syntactic patterns and explicit patterns, such as Rascal [HKV12]. But in addition to supporting both kind of patterns, it is desirable to have an hybrid pattern

representation for patterns, so that explicit *and* syntactic parts coexist in the same pattern, with the same engine (instead of having a separate engine for each kind of pattern). An hybrid syntactic-explicit pattern is syntactic by default, but allows for individual typing of metavariables or patterns. That way, the strengths of both approaches are kept for language experts: they can type some metavariables explicitly and fallback on type inference for the rest.

```
1 `a:Add` + `b`
```

Listing 3.3: Hybrid explicit-syntactic pattern.

The pattern in Listing 3.3 is by default syntactic but the user specified that the metavariable `` `a` `` should be of type `Add`.

3.1.5 Metavariables

A metavariable is a variable in the logic sense, it holds no value until one is bound to it through matching. Any IR element or collection of IR elements is a possible value for our metavariables. A metavariable goes through 3 states: untyped, typed and bound. In the untyped state, the metavariable is simply an identifier, with no type and no value. For example, before matching, the metavariable `` `condition` `` from Listing 3.2 is untyped. In the typed state, the metavariable is also assigned a type and still has no value, but later, its value will only be of the type it got. Following the previous example, `` `condition` `` would be of type `ConditionNode`. In the bound state, the metavariable received a value of its type, the value being a node in the AST to match in our case. The process through which a metavariable gets from the untyped state to the typed state is a process we named type inference and the one to go from the typed state to the bound state is called unification.

3.1.6 Type inference to support syntactic pattern matching

Type inference is a mandatory step of syntactic pattern matching that explicit pattern matching does not require, since patterns are already typed by the user. The type inference phase must not only infer the type of the pattern, but also the types of each of its metavariables. For a single pattern, there exists a forest of trees where the top node of the tree is of the pattern type and each of pattern metavariables are leaves inside the tree. Before the unification phase, metavariables are leaves since no concrete node has been assigned to it, it is simply given a type. We named this kind of tree with metavariables as leaves *abstract matches*.

Let L_1 be the host language of the pattern, defined in the grammar in Listing 3.4.

```
<number> : [0-9]+ ;

Expression
    : Expression "+" Expression {{Add}}
    | Expression "-" Expression {{Sub}}
    | <number>                  {{Num}}
    ;
```

Listing 3.4: Toy expression grammar.

As an example, let us consider the simple addition pattern of Listing 3.5.

```
‘a‘ + ‘b‘
```

Listing 3.5: Addition syntactic pattern.

For this pattern, we need to find all the subtrees in the source matching an addition of two values `‘a‘` and `‘b‘` that themselves match any subtree as long as it is within an addition. The first step, the type inference consists of getting abstract matches from the textual description of the pattern. An abstract match is the abstract syntax tree of the pattern, where metavariables are special leaf nodes annotated with a given type. Each combination of pattern type and metavariable types for each metavariable yields a forest of abstract matches.

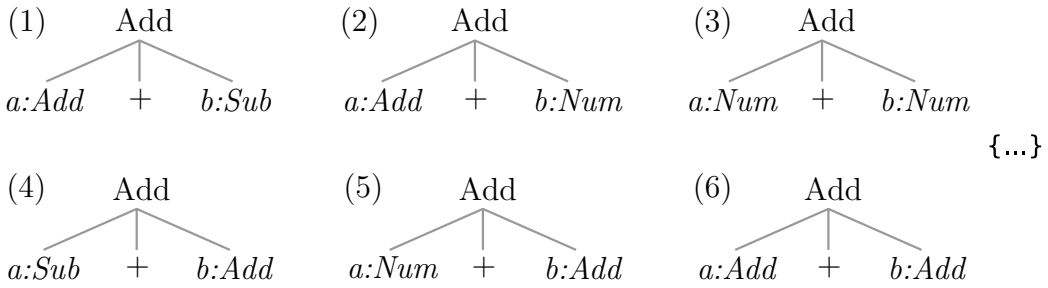

Figure 3.1: Abstract matches for the pattern `‘a‘ + ‘b‘`. The abstract matches have been numbered for referencing in Figure 3.2.

For example, Figure 3.1 represents some abstract matches of the pattern `‘a‘ + ‘b‘` according to the grammar of Listing 3.4. The pattern type of this pattern, meaning the type of its top node, must be `Add`, while `‘a‘` and `‘b‘` can be any one of `Add`, `Sub` or `Num`.

3.1.7 Unification

During the unification phase, the AST of the source tree is traversed using different strategies and nodes are compared to the abstract matches obtained from the type inference. Metavariables are assigned the values of the subtrees that match their types. The subtrees with each of their metavariables bound to a concrete subtree of the source AST are named *concrete matches*.

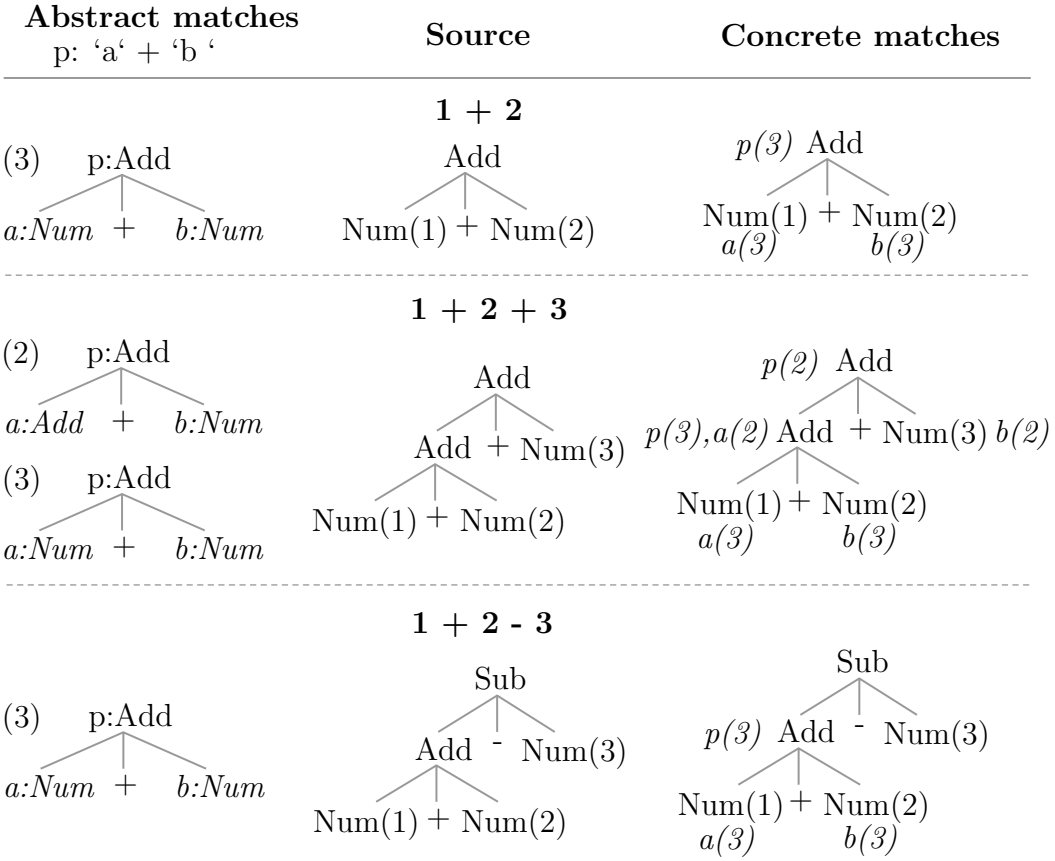

Figure 3.2: Concrete matches for the pattern `‘a‘ + ‘b‘`. The first column repeats the abstract matches from Figure 3.1, indexed by a number in parenthesis. For each abstract match "`(X)`", we bind its metavariables `a(X)` and `b(X)` and its pattern type `p(X)` to a subtree in the concrete match column.

If we go back to the previous example of pattern in Listing 3.5, and unify its abstract matches with a collection of source examples, we get the concrete matches in Figure 3.2. Metavariables `‘a‘` and `‘b‘` are now bound to a given subtree of the concrete match. The top node of the matched subtree represents the inferred pattern type for the concrete match.

3.1.8 Towards a language-agnostic pattern matching engine

To build a pattern matching engine supporting hybrid patterns, we need to translate syntactic patterns and explicit patterns into IR elements. Explicit patterns are easy in that regard since they already describe an IR element. Syntactic patterns require additional work, through a type inference phase, to transform them into a compatible structure. Type inference is heavily dependent on the matched language, since it will provide the IR. In the next sections, we will describe a technique to perform the type inference without having to write by hand a language back-end for each new language.

3.2 Implementing syntactic pattern matching

In this section, we will discuss how parsing as intersection is adapted and combined with a parser generator to solve the type inference reliance on hand-crafted language back-ends.

3.2.1 Type inference implementation issues

Unification is a typical step of every pattern matching engine and as such has been covered in all of them. As explained in Section 3.1.6, type inference resolves around solving two problems:

- finding the type of the pattern,
- and finding the type of each of its metavariables.

Both problems are intermingled since the type of a pattern depends on the type of its metavariables. In other words, the problem of type inference can be reformulated to searching all the unique combinations of metavariable types for each possible pattern type. We named a single combination of a pattern type, each of its metavariable types and the pattern tree, an abstract match, so type inference searches for every valid abstract match for a pattern.

In a syntactic pattern matching engine, this search is performed by parsing the pattern with a parser. Such a parser is a parser for the host language, modified to handle metavariables.

Metavariable type problem. First of all, to be fed to a parser, a metavariable needs to be recognized by the parser. In parsers for pattern matching, metavariables are viewed as special tokens but would still require custom code to handle it. In scannerless parsers, the metavariable could have a special rule that would need to be checked constantly and, again, would require custom code to handle. In both cases, custom code is required once a metavariable is recognized by the parser to get the metavariable to a typed state. Such glue code is language-dependent and has to be rewritten for each language the engine aims to support. Syntactic pattern matching engines deal with this problem by skipping one node, creating a metavariable node that matches anything in its stead, and try to continue parsing with this incomplete input leading to backtracking introduction into a possibly non-backtracking parser.

Pattern type problem. In addition to finding a type for every valid type combination for the metavariable, type inference needs also to provide the pattern type for each combination. Note that this is also be divided into two problems: where in the language does the pattern start and from that point what types are reachable.

```
1 `a` + `b` * `c`
```

Listing 3.6: Arithmetic expression syntactic pattern.

In the example of Listing 3.6, the pattern is obviously some kind of arithmetic expression. However, to parse this pattern and get the proper type, the parser cannot start at the top level (for example, a *CompilationUnit* for C) because it would not match this pattern which belongs to the language. It would require every pattern to be written from the top level to match anything at all, which is unreasonable and impractical. As such, the parser needs to start at some intermediary point in the language to get the valid type. Of course, it is impossible to find the correct starting points for a given pattern before parsing it, so every possible starting point in the language must be considered. Identifying those starting points of a pattern is usually done in two ways: either the engine requires the user to provide the starting point for the pattern or the engine implementer provides the list of all possible starting points manually by hard coding them in the engine. The first approach severely limits the specter of users that can write patterns (essentially experts), whereas the second approach is error prone (missing a starting point) and has to be repeated for each new language the engine supports.

Work by Aarssen et al [AVvdS19] provides facilities to ease the writing of glue code for metavariable typing and pattern typing when using external

black box parsers. While it is still required to write some glue in a format or another this helps leverage the power of well known parsers used by the community.

In this work, the opposite approach is taken: instead of using black box parsers, we integrate our pattern matching engine to a parser generator to forgoe completely the problem of glue code. Our approach leverages parser generators for their automation capabilities and the result of *parsing as intersection* to add pattern matching to said parser generators.

3.2.2 Typing in parser generators

Parser generators are used when building language independent tools for a collection of languages. A parser generator takes a grammar of the language to parse as its input and from it, generates a parser for the language. That way, when maintaining the parser, only the grammar (typically orders of magnitude smaller in lines of code) has to be maintained, the code for the parser is generated. While the generation allows one to do a lot with very little lines of code, grammars in BNF are an entirely different beast from standard code, leading to difficult debugging tasks on a grammar when not equipped with a parser debugger.

As explained in Section 2.2.2, LR parsers are a popular parsing technique and are always generated through parser generators (referencing an LR parser usually means referencing its generator). In LR parsers, the grammar is distilled into parse tables and the AST creation code is also generated. Since we want to infer AST node types, all the generated information is sufficient to perform type inference.

In a parser generation environment, two kinds of types coexist in a single typing system. The first kind is the *symbol types*, they consist of the types of the grammar itself, so terminals and non-terminals. Note that these are also the types used by the parse tree since the names of the parse tree nodes are non-terminal names in the grammar. The second kind is the *node types*, they consist of the types of the AST nodes. In most parser generators that provide AST generation, the creation of ASTs is either done by transducers that transform the parse tree into an AST or by directly building the AST during the parse, without going through an intermediate parse tree node. In this work, we focus on the second approach with intermediate parse tree nodes. This approach relies on semantic actions to build the AST from scratch in the same way they would be needed to build a parse tree. Semantic actions are tied to production rules in the grammar, from which we obtain a binding between symbol types and AST node types.

```
<number> : [0-9]+ ;

Expression
    : Expression "+" Expression     {{Addition}}
    | Expression "-" Expression     {{Subtraction}}
    | <number>                      {{Number}}
    ;
```

Listing 3.7: Example of grammar with AST generation.

In the example of Listing 3.7, the symbol type `Expression` leads to three different node types: `Addition`, `Subtraction` and `Number`.

The job of the type inference phase is to get the abstract matches of a pattern by typing its metavariables and recognizing a root node type. The types in question are AST node types, but syntactic patterns will be parsed during the type inference, so it is important to have a binding between the grammar types and the node types.

3.2.3 Parsing as intersection for type inference

Type inference is needed to solve the metavariable type problem and the pattern type problem. A grammar result called *parsing as intersection* can be adapted to perform type inference without rewriting language back-ends by hand. The rest of this section is dedicated to an explanation of the principle of parsing as intersection and its adaptation to suit type inference in a pattern matching engine.

Parsing as intersection is a property on context-free grammars described by Bar-Hillel et al. [BHPS61]. This property has been studied and explained in a chapter of *Parsing Techniques: a Practical Guide* [GJ08a], but the book questions its use cases.

Parsing as intersection. Parsing as intersection states: "the intersection of a context-free language with a regular language is again a context-free language". In other terms, $L \cap L_R = L_I$, where L is the original context-free language (defined by a Chomsky Type-2 grammar), L_R the regular language (Chomsky Type-3) and L_I is the context-free language resulting from the intersection.

While this result is daunting and quite abstract in its original form, we will discuss its implication for the pattern matching domain and how to implement it.

In the context of a pattern matching engine, the original language L is described by grammar G, and corresponds to the host language, the one being matched. The regular language L_R is assimilated to our syntactic pattern

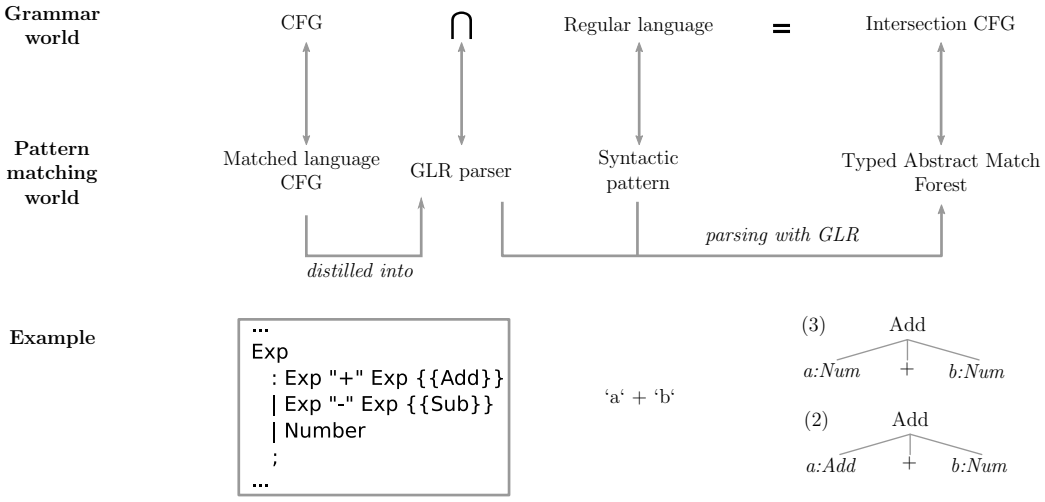

Figure 3.3: Parsing as intersection: from the grammar world to the pattern matching world

language. This language is simply using the same symbols as G, with the caveat that metavariables hold the function of regular language wildcards and that Kleene operators (star, union and concatenation) are valid on those symbols.

The intersection grammar G_I of language L_I is obtained by applying the intersection operator between G and the syntactic pattern. In this context, where the syntactic pattern language and the host language both have the same set of symbols, the intersection operator $\cap$ is assimilated to a generalized parser of the host language trying all the possible parses of the pattern. Performing this will result in a forest of abstract matches, one abstract match per pattern type - metavariable types combination. Note that instead of generating all the abstract matches (trees), we could instead generate the grammar which once we expand all of its possible branches would give us back the abstract matches. However, since the later purpose is to compare trees during the unification phase, we prefer to directly generate abstract matches.

Figure 3.4 shows a pattern matching engine updated with type inference based on parsing as intersection.

Solving the metavariable type problem. The gist of the approach to solve the problem of finding all the possible types for a metavariable is to treat metavariables in the same way as ambiguities (see Section 2.3.1). Upon reaching an ambiguity in a generalized parser, the parser will fork for each possible parsing alternative, processing them in parallel until one fails. In our

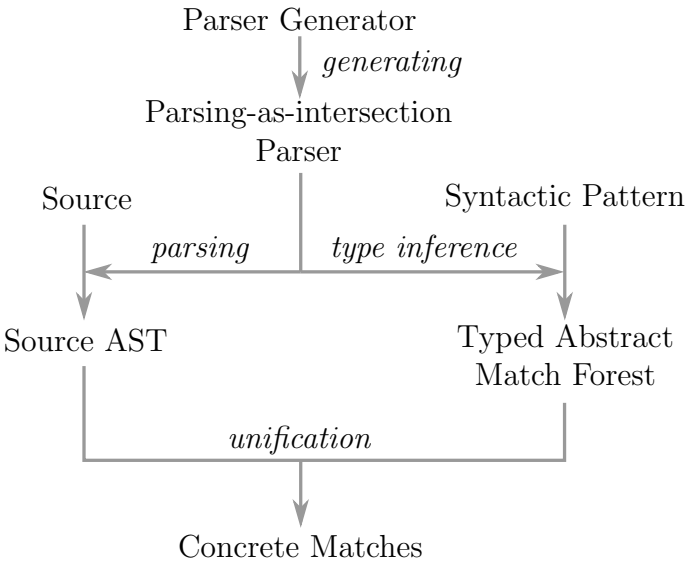

Figure 3.4: Pattern matching engine using parsing as intersection

case, upon reaching a metavariable, the generalized parser will fork for each possible type the metavariable can have at this point in the parse (not all possible symbol types) and then continue the parse as usual. This set of types is obtained from the binding between grammar types and AST node types. If the parse reaches an error state, as for ambiguities, the current parsing fork is discarded without impacting the others. All the successful parsing forks yield a valid abstract match according to the grammar. Reusing the forking mechanism of generalized parsers that are generated further lessens the burden on the engine implementer.

Solving the pattern type problem. Since it is not acceptable to define all patterns from the start of the grammar, a solution must find the possible starting states automatically, without relying on glue code from the engine implementer. Solving this problem is also done using a generalized parser. Once the possible *starting* locations for the pattern are identified, the parser simply performs an initial fork for each of those locations in the grammar. The possible starting locations correspond to the start of production rules in the grammar that accept a token, meaning all rules that only reduce to other rules must be excluded. Since the starting locations can be deduced from every CFG and incorporated in a generated parser, it removes the need for glue code.

3.3 Type inference through GLR-based parsing as intersection

3.3.1 Gist of the approach

Fitting all together, we want to use a parser generator to generate a GLR parser aware of metavariables to use a single algorithm to parse the host language and the syntactic patterns, thus improving back-end reuse between languages of the pattern matching engine.

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.

Our approach implements a version of parsing as intersection using a specific kind of generalized parser: GLR parsers. We modify the GLR parsing runtime to handle metavariables as ambiguities and automated identification of starting locations. This solves the metavariable type and pattern type problems by a clever use of the GLR parser forking mechanisms.

In addition, the pattern matching engine also supports an explicit-inferred hybrid approach that lets the user specify explicitly individual metavariable or pattern types while relying on inferrence for the rest.

Figure 3.5 presents the main components of our approach to syntactic pattern matching. The gist of the parsing as intersection approach is to use the same parsing algorithm to parse the program source and the pattern. Whereas the program parsing yields an AST, the pattern parsing corresponding to type inference yields a forest of trees corresponding to each valid abstract match.

The GLR parse starts and when it reaches a metavariable, the parser forks into multiple subparsers, one for each possible type that appears at this specific point during the parse. The parser does it by inspecting its own LR parse tables, it checks for all the available transitions from the current state, relates them to potential AST node types and forks for each one of them. If a subparser fails to parse, it means the configuration is invalid and it is discarded. The subparsers that survive at the end of the parse will each generate one tree, each tree having a specific configuration of metavariable-type pair, an abstract match. Each abstract match in the forest is then confronted against the program AST. For the matching subtrees, each metavariable is bound to the concrete AST node (meaning subtree) it represents.

Activating pattern matching for a new language is easy if a grammar is available. It consists of only a single line of code to add the metavariable

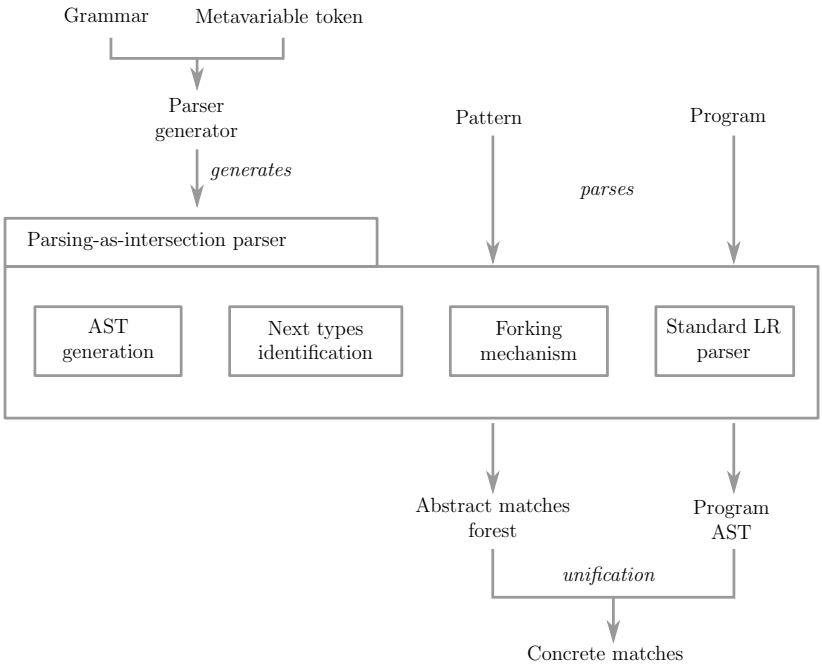

Figure 3.5: General pipeline of the pattern matching engine.

token into the grammar, so that the scanner recognizes it.

3.3.2 Parsing a syntactic pattern

Performing parsing as intersection on a pattern to get the pattern's abstract matches requires some modification of the parsing runtime of GLR that we will now present.

First, to be able to gather every type of the pattern and not describe the pattern and parse it from the root of the grammar, we need to identify candidate starting states in the parser. Without other assumptions about the pattern in question, the best starting points we can find are the production rules that accept a token. In other words, we only reject production rules that are only reductions of other rules. This simply means that the parser must be able to consume a token first, without any previous input.

In LR terms, starting states corresponds to states in the automaton that display at least one outgoing shift transition. That way, if all the possible actions are reduces, accepts or errors, the state is not selected as a starting candidate.

```
<number> : [0-9]+ ;
<identifier> : [a-z][a-zA-Z0-9_]+ ;

Expression
    : Expression "+" Term    {{Add}}
    | Expression "-" Term    {{Sub}}
    | Term
    ;

Term
    : Term "*" Factor        {{Mult}}
    | Term "/" Factor        {{Div}}
    | Factor
    ;

Factor
    : "(" Expression ")"
    | Operand
    ;

Operand
    : <number> {{Number}}
    | <identifier> {{Variable}}
    ;
```

Listing 3.8: Unambiguous arithmetic expression grammar.

```
‘a‘ + ‘b:Mult‘
```

Listing 3.9: Hybrid explicit-syntactic expression pattern example.

According to the grammar of Listing 3.8, the pattern of Listing 3.9 can start from any of the production rules because none of them only reduces.

For each starting state candidate, a subparser is created with its own state as the top of the state stack. Obviously, this will overshoot the number of starting states that will yield a correct result in the end. However, since we are dealing with a GLR parser, a subparser reaching an error state will be discarded. Uninteresting starting states will rapidly reach an error state resulting in their removal. The subparsers behave normally until they reach metavariables.

3.3.3 Reaching a metavariable

To be able to recognize metavariables, a token definition needs to be added to the grammar, so that the compiled scanner recognizes it and creates a proper metavariable token. To this end, a metavariable token must first be defined in the host grammar. Since it will coexist with the rest of the host

language, it is imperative the token definition does not conflict with other token definitions in the language.

In all examples of syntactic patterns, the metavariable token will enclose metavariables with the backquote character ‘ since it is mostly absent from programming languages. As stated in Section 3.3.1, our approach is hybrid, meaning the default matching method is syntactic but the user can still choose to explicitly type individual metavariables or patterns. To do so, a dedicated metavariable parser is called from the generated scanner to ensure the metavariable token holds the type the user specified if any.

When reaching a metavariable token, the subparser inspects the parse tables and retrieves all the possible actions for the current parsing state (every reduce and shift)[1].

If we go back to our previous example in Listing 3.9 the metavariable `‘a‘` could of types `{Number, Variable, Add, Sub, Mult, Div, "(", <number>, <identifier>}` depending on the parser's starting states, while an explicitly typed `‘b‘` is of type `Mult`.

3.3.4 Forking the parser

To support an hybrid approach, if a node type is specified by the user, the subparser selects only the actions tied to a symbol whose node types include the user node type. If no user type is provided (the default case), the subparser forks for each action.

Algorithm 2 describes the main steps leading to forking of the subparser. The algorithm forks the parser for every possible interpretation 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. If this set is empty (lines 4 to 6), it means the parser is in an error state, unable to accept any token at the time. The function returns and the parser in question will be discarded later on.

In lines 8 to 10, 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.

If the symbol type is a token (lines 14 to 17), 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 a state where the metavariable can be pushed onto the node stack

[1] It also makes sure to retrieve only reduces that do not pop more states than available on the state stack of the subparser.

Algorithm 2 Forking on metavariable algorithm.

```
 1: function ForkOnMetavariable(currentState, metavariableToken)
 2:     userType ← type(metavariableToken)
 3:     transitions ← transitionsFrom(currentState)
 4:     if transitions = ∅ then
 5:         return
 6:     end if
 7:     for all (symbol, action) ∈ transitions do
 8:         if defined(userType) & ! compatible(symbol,userType) then
 9:             continue
10:         end if
11:         if isNode(symbol) then
12:             subparser ← forkParser(symbol, action)
13:             emulateNodeShift(subparser, action, symbol, metavariableToken)
14:         else if isToken(symbol) then
15:             subparser ← forkParser(symbol, action)
16:             performReducesAndShift(subparser, action, symbol, metavariableToken)
17:         end if
18:     end for
19: end function
```

(as for any LR shift). Some of these reductions may be invalid, but their subparsers will quickly be killed when no valid action is found at the newly reduced state.

In the case of the symbol type being a node (lines 11 to 13), 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.

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 pattern in the form of one abstract match per successful subparser. The result is a forest of the valid abstract matches where each AST has a unique set of metavariable-type pairs. From this point on, unification (the second phase of pattern matching) finds the subtrees in the source AST matching the abstract

matches, and if it does, binds each typed metavariable to its concrete subtree.

3.3.5 Unification

The next phase is more standard, unification works in the same way as for an explicit pattern matching engine, it is not tied to syntactic patterns. Although not part of type inference, the unification process (the next step in pattern matching) is described here for the sake of clarity.

The unification process compares the pattern forest of abstract matches (special 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 is found, the algorithm stops there. We focus on a simple depth-first traversal of the program AST (see Algorithm 3).

Algorithm 3 Unification algorithm.

```
1: function UNIFY(programAST, patternForest)
2:     for all programNode ∈ programAST do
3:         programRoot ← programNode
4:         for all patternAST ∈ patternForest do
5:             patternRoot ← root(patternAST)
6:             if patternRoot = programRoot then
7:                 acceptMatch(patternAST)
8:             else
9:                 discard(patternAST)
10:            end if
11:        end for
12:    end for
13: end function
```

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 4, 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 5), the actual token strings are compared and the result of the equality is returned to the calling node.

Algorithm 4 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 5 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 in a dictionary to its concrete match. 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 subsequent rewriting or analysis is then based on the typed matches and their respective contexts.

Trying to match a pattern that has no metavariable may too result in multiple abstract matches, but only if the target language is inherently ambiguous. Intuitively, if our pattern (regular language) is simply a string belonging to our host language, the type inference through parsing as intersection is just a standard parse: if the language is ambiguous, the pattern could be ambiguous too. For example, the pattern `(‘a‘) * ‘b‘` in C has (at least) two abstract matches, one for the multiplication expression, and the second one for the declaration of a pointer variable of type `a`.

3.4 Matching complex types

Parsing as intersection is a result on grammars, and by extension on the derivation tree that parsing theoretically yields. However, parser and parser generator implementations transform derivation trees into ASTs. This transformation exists to simplify the tree by removing artifacts of no use in representation and analysis, artifacts that only exist for recognition purposes. Our technique is adapted to work with ASTs and works well when the mapping between AST nodes and derivation tree nodes is given. In the case of list

idioms, the mapping is more complex and requires work described in the rest of this section.

3.4.1 Matching of list idioms

When building the AST, a parser generated from a parser generator generaly uses techniques to reduce the tree from its parse tree size into a more manageable size, without insignificant intermediate nodes. As presented in Section 2.1.2, recursion (being left or right) is one of the main culprits for the potential node cluttering. In EBNF grammars, recursion is explicitly specified either via rules or by Kleene star and plus operators, leaving the generator to handle the conversion (Listing 3.10).

```
<number> : [0-9]+ ;

NumberListLeftRecursion
    : <number> NumberListLeftRecursion
    | <number>
    ;

NumberListRightRecursion
    : NumberListRightRecursion <number>
    | <number>
    ;

NumberListKleenePlus
    : <number> +
    ;

NumberCommaList
    : Number ("," Number) *
    ;

Number
    : <number>
    ;
```

Listing 3.10: Recursion in parser generators.

Instead of storing the head or tail of the recursion in a separate node and the value of the current list element in the current node, some parser generators aggregate all list elements into a collection. For the user, it also crucial to being able to match such list idioms through patterns and metavariables. While the implementation described in the previous section matches nodes and tokens in an AST through syntactic patterns, there is no clear way to describe these list idioms in a pattern. If these list idioms cannot be described

in a pattern, they cannot be matched.

```
1 ‘list: <number> *‘
2 ‘list: <number> +‘
3 ‘commaList: Number ("," Number)*‘
```

Listing 3.11: List matching examples.

Listing 3.11 shows examples of common list matching patterns using Kleene star and Kleene plus. In our case, a comma separated list in a production will yield two different child collections for the parent node: one for all the commas, and the other for the other node (here, `Number`). This section is dedicated to explaining how to implement the matching of such collections and its impacts on the syntactic patterns.

3.4.2 AND metavariable types

Since the pattern matching language is a regular language over the symbols of the host language, supporting Kleene operators is in theory possible. The aggregation of nodes into collections complicates the parsing as intersection inference algorithm to generate abstract matches and the unification algorithm to search through the source AST. The type inference must be able to handle metavariables featuring common AND types, types denoting of sequences of other types, and create abstract matches for those to be unified later on. To this end, we propose that metavariables now hold one or more types in a sequence with optional Kleene plus, Kleene star, option operator and group operator.

```
cList: Num ("," Num)*
```

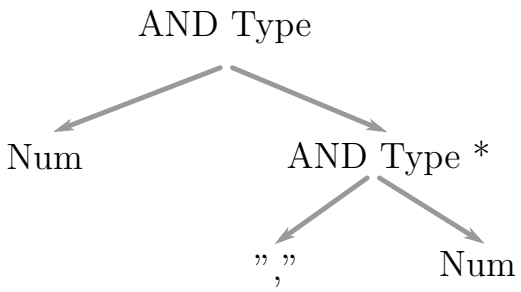

Figure 3.6: Metavariable model with AND types.

Figure 3.6 illustrates the model of the last example of Listing 3.11, ‘ `commaList: Number ("," Number)*`‘. The model of an AND type created by a user is essentially a tree of subtypes. As any other type, a AND type has a

cardinality resulting from Kleene operators. Group operators create a new AND type with elements of the group being subtypes.

3.4.3 Type inference for AND types

During type inference, when the parse reaches this metavariable, it deconstructs the AND type into primitive types that the parser can expect. Figure 3.7 summarizes the steps of parsing the metavariable in Figure 3.6 which has an AND type.

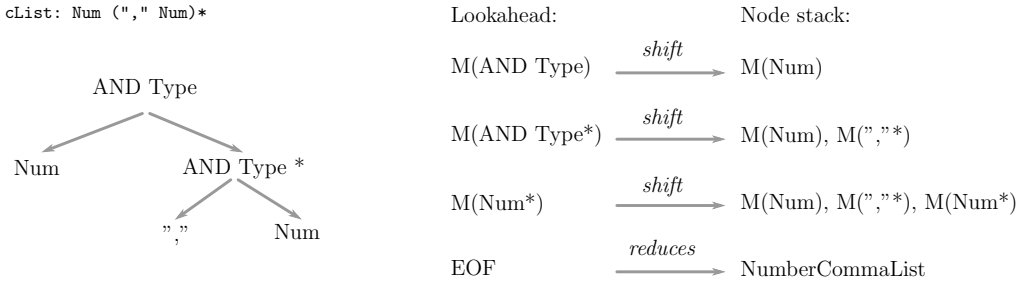

Figure 3.7: Parsing an AND type metavariable.

First, the parser will react as if a `Number` was its next symbol to parse and shift it. Then, the group `("," Number)*` will be separated into `","`, which the parser will parse immediately, and `Number`. Once the second `Number` has been shifted, the pattern is reduced to a `NumberCommaList`. The cardinality of the group must be shared by its children to remove any problem of them matching a different number of elements in the unification phase. The group will be spread into two different collections (a comma collection and a number collection), but they will crucially reference the same cardinality object. All the subtypes in an AND type metavariable are still referencing their original metavariable.

The resulting `NumberCommaList` (Figure 3.8) holds both number and comma collections, themselves containing the AND type metavariable divided into three metavariables.

3.4.4 Unification for AND types

The unification algorithm also needs to be adapted to accomodate the new abstract matches produced by the type inference of AND types. The Kleene star and plus operators force collection matches, and we choose to implement a greedy matching algorithm for those collections.

```
cList: Number ("," Number)*
```

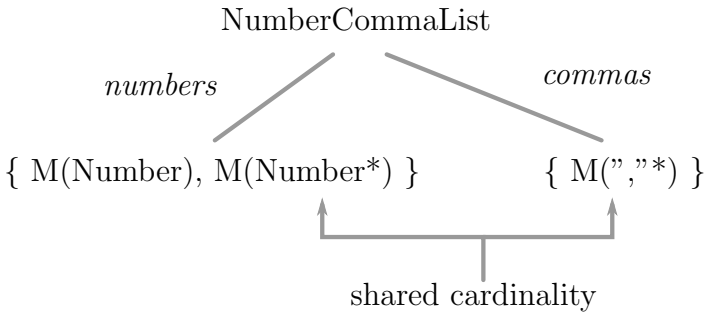

Figure 3.8: AND type abstract match.

Algorithm 6 matches *matchCollection* of the abstract match with *sourceCollection* of the source AST starting respectively at *matchIndex* and *sourceIndex*. Lines 2 to 4 initialize a temporary index for the *sourceCollection*, the current abstract match node in the collection and the collection of matching nodes in the source collection. Lines 5 to 12 greedily match the current abstract match node with as many source collection nodes. The matching nodes are added to the temporary *currentNodes* collection. Lines 13 to 15 exit the function with a failed match if there are no matched nodes and the cardinality of the abstract match node is not a Kleene star (meaning it could be empty). Lines 16 to 20 register *currentNodes* in the context as a valid match for the abstract match node unless it already existed. This occurs if multiple metavariables with the same name are used, they should all match the same kind of subtree. If the current match and the existing match differ, the function fails to match. Lines 21 to 23 ensure that if we reach the end of the *matchCollection*, we also reach the end of the *sourceCollection*, otherwise the match fails. Line 24 recursively calls the function with the next abstract match index and the next source index.

3.4.5 OR metavariable types

On the language expert side of the hybrid approach, it is useful for the user to specify multiple types for a single metavariable or pattern. The resulting abstract matches contain only metavariables of one of the given type. We call these OR types since they are a disjunction of multiple types. Listing 3.12 provides examples of ways to express metavariable OR types on an expression language.

```
1 ‘a: Add | Sub‘ + ‘b: Num | Mult‘
```

Algorithm 6 Greedy collection matching algorithm.

```
1: function GREEDYMATCHCOL(matchCollection, matchIndex, sourceCollection, sourceIndex, context)
2:     currentIndex ← sourceIndex
3:     matchNode ← matchCollection[matchIndex]
4:     currentNodes ← []
5:     while true do
6:         if currentIndex <= size(sourceCollection) & matchNode.match(sourceCollection[currentIndex]) then
7:             currentNodes.append(sourceCollection[currentIndex])
8:         else
9:             break
10:        end if
11:        currentIndex ← currentIndex +1
12:    end while
13:    if currentNodes = [] & ! matchNode.isStar() then
14:        return false
15:    end if
16:    if matchNode.isMetavariable() then
17:        if context[matchNode] = ∅ then
18:            context[matchNode] ← currentNodes
19:        else if context[matchNode] ! = currentNodes then
20:            return false
21:        end if
22:    end if
23:    if size(matchCollection) < matchIndex +1 then
24:        return currentIndex > size(sourceCollection)
25:    end if
26:    return greedyMatchCol(matchCollection, matchIndex +1, sourceCollection, currentIndex, context)
27: end function
```

Listing 3.12: OR type pattern on expressions.

In comparison to AND types, OR types are relatively easy to integrate in the type inference phase and costless for the unification phase. During the type inference phase, the parsing as intersection parser will provide all the different types that appear at this point in the parse. The valid types are retrieved from the intersection of this set and the set of types the user

specified in its OR type metavariable. The rest of the matching process works in the exact same way as with a single or no user type.

3.5 Experiments and results

This section presents the implementation technology of the pattern matching engine, gives a use case of code migration for an unformalized version of the approach and discusses the expressivenes of patterns in the formalized approach introduced in this chapter.

3.5.1 The Smalltalk Compiler-Compiler

This approach was implemented on top of the Smalltalk Compiler-Compiler (SmaCC) [BRPP10]. SmaCC is a parser generator producing LR(1), LALR(1) and GLR parsers from an EBNF grammar. SmaCC is written entirely in Smalltalk and generates Smalltalk code. In addition to a parser, SmaCC also generates AST node classes from annotated productions in the grammar, and a generic visitor for said AST that is easily extended. At parse time, an AST is created from those annotations and the generated node classes. SmaCC automatically detects recursions in the grammar and converts them into collections of nodes in the AST, reducing the need for intermediate nodes. A `BlockNode` may hold a child collection of `StatementNode`s for example.

To make the pattern matching available for a given language, the generated parser must be GLR and the metavariable token needs to be added to the grammar of the language. Of course, the metavariable token must not conflict with existing token definitions.

Currently, the engine already has open-source back-ends for the following languages: C#, Java, Javascript, Delphi, Swift and Smalltalk. Parsers for Powerbuilder, C, IDL and Ada have also been built with SmaCC, but remain closed-source.

3.5.2 Industrial Validation

The rewriting engine of SmaCC had an intuition of this technique to perform type inference, which we later formalized and adapted to better reflect parsing as intersection.

This previous version has been used on several industrial projects using different programming languages: Delphi, PowerBuilder, C#, Java and Ada [BRPP10]. These projects range from refactorings to migrations from

one language to another. A typical migration project with SmaCC is composed of two transformation passes.

First pass: migration. The first pass converts all of the source code in the original language to the new target language. The transformation pass involves applying numerous rules in bulk on the same valid input, not by applying a rule, parsing the new program and applying another one, etc. The entire source code is transformed into the target language by applying all the transformations to generate a new code from the AST.

Second pass: artifact elimination. Small transformations post-migration are important to make the code more natural to future developers handling the project. Listing 3.13 is an example of a Delphi to C# transformation rule that is a special case of the general loop conversion. The Delphi for loop used `"- 1"` in the end condition. In C#, we could eliminate the `"- 1"` from the converted code by using `"<"` instead of `"<="`.

```
1 for ‘a‘ := ‘b‘ to ‘c‘ - 1 do ‘d‘
2 >>>
3 for (‘a‘ = ‘b‘; ‘a‘ < ‘c‘; ‘a‘++) ‘d‘
```

Listing 3.13: Delphi to C# syntactic pattern in SmaCC

The second pass overwrites some transformations through specific use cases to make the code easier to maintain and more akin to what code in the target language looks like. While this transformation step was not necessary, it eliminates some conversion artifacts.

Powerbuilder to C# example. An example of such a project was the migration from 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. The code base contained over 1.1 MLOC[2] and were 153MB in size. The resulting C# code contained almost 7 600 files with 3.3 MLOC and was 161MB in size. 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. These files were converted using 578 conversion rules. Of

[2]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.

the 578 rules, 356 (62%) of them used syntactic patterns, and the other 222 rules used more traditional explicit pattern matching. The first pass (migration) accounted for 509 conversion rules, while the second pass (artifact elimination) accounted for 69 rules. 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 chapter 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.

While no project of such a scale has yet been performed with the approach formalizing parsing as intersection, the study gives us confidence in the practicality of the approach.

3.5.3 Expressiveness of hybrid patterns

Pattern language. As our approach to type inference is derived from parsing as intersection, it suffers from the same limitations in terms of expressiveness. The pattern matching language is a regular language on top of symbols from the grammar of the matched language. As such, it only supports sequences and Kleene operators. Union could also be supported but is not implemented at the moment, since its interaction with AND types would need to be carefully looked at. For convenience, metavariables act as any symbol from the alphabet unless explictly typed. While limited, the language is still powerful enough to infer all types of the AST and sequences of types in the case of AND types.

Hybrid patterns on grammars. We explore the expressiveness of hybrid patterns on the metagrammar of SmaCC. As an example, the grammar from Listing 3.14 acts as the input program of the pattern matching engine. The grammar itself and its syntactic patterns are written in the language defined by the metagrammar of SmaCC.

```
1  S
2       : A B "c"
3       | A "c"
4       ;
5
6  A
7       : "a"
8       ;
9
10 B
```

```
11      : "b"
12      ;
```

Listing 3.14: Source code of the grammar to match with patterns written in the metagrammar of SmaCC.

The first pattern (Listing 3.15) is a hybrid pattern, mostly syntactic with a metavariable featuring an AND type to match the alternates of the production rule `A`.

```
1 A : ‘alts: AlternativeNode ("|" AlternativeNode)*‘ ;
```

Listing 3.15: Hybrid pattern to match production rule `A` and get its alternates.

The result of applying this pattern to the input grammar is a single match of the `ProductionRuleNode` of `A`. The match context contains the alternates of the rule indexed at the `‘alts‘` metavariable.

To achieve the same result using only explicit pattern matching would require the user to first get all the `ProductionRuleNode` nodes through this type. Then, a custom visitor would go through all the matched production rules and gather those of production name `A` (which is a `SymbolNode` in this case). And, for those, the collection of `AlternateNode` should be found in the `ProductionRuleNode` children. The two first part of this explicit matching could be matched with an explicit pattern of the form `ProductionRuleNode( SymbolNode("A"),_)`, but it would require the user to know that this particular node has two children. The purely explicit matching requires to know 3 different node types and their hierarchy in the AST, while the hybrid one only requires one.

Using syntactic patterns for filtering. In the next example, we want to find the production rule that has a given pattern as its right hand side. The pattern in question (Listing 3.16), alone, matches an `AlternateNode` with a metavariable of type `TermNode` in its context.

```
1 ‘firstSymbol‘ "c"
```

Listing 3.16: Syntactic pattern to match all the alternates featuring a first unknown symbol followed by `"c"`.

To find its production rule, we could add code to follow the parent chain of the node and find that the parent is a `ProductionRuleNode`, but it is extra code written in the engine language. Instead, we can use the hybrid pattern in Listing 3.17 to get all the production rules on the input.

```
1 'prodName' : 'alts: AlternativeNode ("|" AlternativeNode)*' ;
```

Listing 3.17: Hybrid pattern to matches all the production rules in the grammar and get their names and alternates.

Then, we can filter those production rules by running the previous syntactic pattern (Listing 3.16), on each matched subtree and get only the production rule for which the syntactic pattern matches. Creating a long pattern to match this in a single pattern could be done, but it would be much more complex than to use a second pattern as a filter for the matches of the first.

3.6 Discussion

3.6.1 Prerequisites of the approach

Our approach assumes the following:

Generalized parser generation. The approach requires a generalized parser to handle metavariables as ambiguities. In our case, we implemented it using a GLR parser. Originally, Earley's algorithm [Ear70] was used to describe parsing as intersection [GJ08b], so we believe other generalized parsing techniques (such as GLL [SJ10]) could also be used to implement pattern matching based on parsing as intersection, as long as it fulfills the other requirements.

Grammar for the language. Since the approach is based on a parser generator, a grammar should be available for the language. Having or creating the grammar remains the biggest requirement, but since it is one for most language tools, we do not deem it unreasonnable.

AST generation. The LR parser should be able to generate an AST of the source source at parse time. Note that our approach also works at the parse tree level if it is generated from the grammar. The key point is to generate the tree at parse time, which is difficult to achieve in some parsing technologies and algorithms.

Next type identification. During the parse of the pattern, the parser should be able to find the next types from any state in the parser. In our engine, the binding between state, grammar type and node type is available in the parse tables.

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.

3.6.2 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 a linear parse time. This property does not hold for GLR-type parsing, where the worst case is $O(n^{k+1})$ in complexity, with n being the size of the input and k being the size of the longest production rule. Note that the run-time complexity is heavily dependent on the number and density of ambiguities, the more ambiguities, the more we fork, the more subparsers we need to maintain in parallel and the higher density of ambiguities; the longer they take to resolve. Even if it is easy to create a worst case grammar that reaches the worst case complexity, the average programming language grammar with few local ambiguities.

In practice, for the subparsers created for pattern matching purposes, the average complexity 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. What would need to be examined with caution however are patterns consisting of a sequence of untyped metavariables, which may lead to exceptionally bad runtime. Such patterns could match almost anything, which probably means these are uninteresting patterns in the first place.

3.6.3 Application to other parsers and parser generators

Parsing as intersection, in essence, is independent of the parsing technique used. As long as they respect the previously described requirements (Section 3.6.1), the technique could be applied as is to other GLR-based parser generators.

As for other parser classes such as GLL, the implementation of the prerequisites would change, as forking and type identification for metavariables would need to be introduced. Equivalents to these concepts would be required to work with parsing techniques other than LR.

3.7 Conclusion

A previous version of this work [LBGD18] has been published in the *International Conference on Software Maintenance and Evolution* (ICSME'18).

In this chapter, we proposed a syntactic pattern matching engine using parsing as intersection to perform type inference. This approach allows for complete automation of the type inference implementation for a given language by generating a modified GLR parser, removing the need to implement new hand-crafted back-ends for the engine. As a result, users may express their patterns combining both syntactic and explicit pattern matching, while defaulting to syntactic to remove the need to learn internal representation of pattern matching engines. The engineering cost for the implementer is lowered to finding or writing the grammar. Integrated to a parser generation framework, activating the pattern matching engine for a new language requires only a single line in the grammar.

Chapter 4
Side-effect-enabling GLR parser

Pattern matching based on parsing as intersection provides an easily generated type inference mechanism relying on a generalized LR parser treating metavariables in a similar way to ambiguities. In cases where the original grammar is non-deterministic, forking for both metavariables and ambiguities could lead to an impractical runtime. To ensure this does not happen, we would need to allow custom *online* disambiguation of individual forks. Custom disambiguation consists mainly of adding temporary data structures such as a symbol table and inspecting those data structures during the parse to choose an action to perform instead of forking and trying all valid actions. Since this new behaviour modifies the parse, we use the term "side effect" and side-effect parsing when a parser involves side effects. In standard LR parsers, side effects are expressed through semantic actions executed on reduce. In a GSS implementation of GLR, side effects are difficult to obtain since they impact the parser in its entirety since forks are implicit (and lost in the GSS) and semantic actions are delayed until the end of the parse or *a minima* to the end of the frontier. In a list-of-stacks implementation of GLR, forks are explicit and thus it is easier to write side effects to modify forks, but the merge mechanism is not as efficient as a GSS implementation.

We propose a compromise GLR algorithm named Fibered-GLR (FGLR). Each parsing fork is explicitly modeled as a separate entity, a non-preemptive task called a fiber (or coroutine). Thanks to a scheduler controlling the order of the parsing alternative execution, semantic actions are executed online (at the same time as their LR action) and can alter their own parse independently by modifying all their local fiber elements. While being less efficient than the memory-optimal GSS approach, FGLR allows parse-altering semantic actions. Compared to the list of stacks approach, FGLR is more efficient by merging earlier.

Contents

4.1 Ambiguities and conflicts in LR parsing

A grammar is ambiguous when a string belongs to the language and has multiple valid derivations. In other words, at a point during the parsing of the string two different rules could be applied, expressing two different parsing alternatives, both succeeding to parse and yielding different derivation trees. On the other hand, an unambiguous grammar should have only one single interpretation for each string in the language.

A popular example of ambiguity is a basic grammar for arithmetic expressions (see Listing 4.1).

```
<number> : [0-9]+ ;

Expression
    : Expression "+" Expression
    | Expression "*" Expression
    | "(" Expression ")"
    | <number>
    ;
```

Listing 4.1: Ambiguous arithmetic expression grammar.

Such arithmetic expressions are present in a similar form in most programming languages. Here, if the parser receives the input string `1+2*3`, it cannot determine whether the correct parse is `(1+2)*3` or `1+(2*3)`. Section 2.3.1 provided more in-depth examples.

4.1.1 LR limitations

In LR(1), ambiguities manifest through multiple actions being available for the current lookahead token, instead of a single one. When two different reductions could occur, this is a reduce-reduce conflict. If the actions are a shift and a reduce, it is called a shift-reduce conflict.

There are *unambiguous* grammars that are not *deterministic*, and thus cannot be parsed by an LR(1) parser. It means that for some grammars, even if their LR parsers have conflicts, the grammars could still be unambiguous. Programming languages often fall in this category, and as such many techniques have been developed to adapt LR parsers to slightly conflicting grammars.

4.1.2 Rewriting the grammar

The first option is simply to remove the conflict, if possible, by eliminating the local ambiguity (one that is resolved after a few more tokens). Rewriting

the grammar to remove the ambiguous production rules removes the origin of the problem.

If we rewrite our previous example of Listing 4.1, the resulting grammar contains more rules and is of greater depth (Listing 4.2).

```
<number> : [0-9]+ ;

Expression
    : Term "+" Expression
    | Term
    ;

Term
    : Factor "*" Term
    | Factor
    ;

Factor
    : "(" Expression ")"
    | <number>
    ;
```

Listing 4.2: Rewritten unambiguous arithmetic expression grammar

Grammar rewriting proves tedious since changes to remove a conflict could lead to another conflict to appear with another production rule. Grammars are difficult artifacts to handle. Rewriting the grammar is also detrimental to its maintainability, because it leads to an increased size and complexity. As such, while rewriting the grammar does not impact the parsing technology, it is not always desirable.

4.1.3 Hacking the parser

The second option consists in adding some disambiguation mechanisms that will modify the parser. Thus, precedence rules have been introduced to handle conflicts. They consist in annotating tokens with a priority and an associativity. For example, the `"+"` operator has a lower priority than the `"*"` operator for arithmetic reasons and both are left-associative. In typical Yacc/Bison/SmaCC formalism, precedence rules are represented as in Listing 4.3.

```
%left "+" ;
%left "*" ;
<number> : [0-9]+ ;

Expression
    : Expression "+" Expression
```

```
7     | Expression "*" Expression
8     | "(" Expression ")"
9     | <number>
10    ;
```

Listing 4.3: Unambiguous expression grammar using operator precedence.

Precedence rules are an enhancement technique of deterministic parsing techniques, but they solve a specific conflict for a specific target grammar and requires modifying grammar. Actually, when a precedence rule is added, the generated LR parse tables are modified to prefer an LR action over another. The change does not affect the runtime performance because it is performed at parser generation time, but it hides the conflicts in the grammar making it hard to determine what the underlying language is. A more generalized parsing technique is needed to parse ambiguous grammars without having to attack specific conflicts.

4.2 Generalized LR parsing

As explained in Section 2.2.2, LR(1) parsers can be derived from any deterministic CFG written as a LR(1) grammar. However, not all unambiguous CFGs are deterministic CFGs, meaning a LR(1) parser cannot be derived. In addition, for readability and maintainability reasons, it may not be possible or reasonable to rewrite a grammar from LR(k) to LR(1) since it greatly increases its size. For both of these reasons, a generalized LR parsing technique is desirable. Past generalized parsing algorithms such as Earley [Ear70] or CYK [Coc70] had major problems in terms of parse time. Generalized LR parsing brings efficient compiled parse tables to the generalized parsing of Earley, yielding far better results.

4.2.1 Differences with LR

The reduction and shift algorithms in GLR are vastly more complex than the classic LR ones and share almost no similarity. While the GSS is theoretically the best performing algorithm with minimal data duplication, it has evolved over a long time to solve its initial problems of non-termination in the presence of hidden left recursion [GJ08b, NF91, SJ06]. The added complexity of implementing a correct GLR and surrounding structures is a real burden, which tends to lead parser generator developers to choose a much simpler structure (list of stacks), even if it means less optimal memory management

(for example in Bison). Also, the GSS and the necessary memoization[1] to make it optimal are considered to bring a significant overhead, as shown by Elkhound [MN04] and its toggle between LR-mode and GSS-mode, the latter being used only if necessary. GSS GLR is a difficult algorithm to handle due to its remoteness to LR, and besides its obvious advantage of maximal state sharing presents the drawback of delaying semantic actions, forbidding the use of semantic actions.

4.2.2 Semantic actions in GLR

A list-of-stack approach keeps track of individual stacks separately. Each stack is viewed as a unique reduction path: applying a reduction does not impact the other stacks, at the price of state duplication. The GSS takes the opposite approach, state duplication is minimum but the unique *reduction path* information is not retained. This is why reduction paths must be found again by searching the reduction arcs down the GSS.

Online semantic actions are tightly coupled with reductions since they are executed at the same time, during the parse, hence the "online" component. Since semantic actions are allowed to modify their current parsing alternative, they should be executed as soon as their reduction is encountered so the parsing alternative can be discarded, merged or altered. If all the reduction paths are recovered only at the end of the computation of each frontier (meaning between shifts), as it is done in the GSS, semantic actions cannot be applied at the proper time: the parser could have taken a parse decision based on the "old" configuration of the parsing alternative before the semantic action is executed [MN04].

Even if GLR is powerful enough to take care of conflicts, it is not sufficient to completely disambiguate a grammar. Disambiguation is usually present in various attire, ranging from precedence rules, priority of alternatives in grammar rules (PEG parsers [For04]), filters [EKV09] or custom side-effect-based disambiguation in semantic actions. Support for side effects in semantic actions is crucial for disambiguation even in GLR.

Online semantic actions are also used to create custom ASTs for each parsing alternatives during the parse. They can then be used to debug parsing by having a rich partial structure of each parsing alternative.

In the next section, we will propose a compromise between list-of-stacks GLR and GSS GLR. This approach is less memory-efficient than the GSS approach but it allows one to use semantic actions to impact the parse (side-effect parsing). Additionally, when compared to the list of stacks approach,

[1] Memoization of reduction paths is key to not search for them multiple times.

FGLR merges stacks ealier.

4.3 A scheduling approach to GLR

We propose a scheduling approach to GLR parsing. The concurrent parsing possibilities are be viewed as execution threads (or fibers in our case, since they cannot be preempted) and a scheduler formalizes the architecture for the ordering of concurrent parsing alternatives (forks).

Individual parser forks are isolated in their respective fiber. Side effects affect the fiber's inner components but not the other parse fibers. It works the same way for other semantic actions, they only impact the AST or parser state of their fiber, providing a good encapsulation.

Each fiber holds a single valid reduction path down an LR state stack and its own node stack. Since the reduction path is unique to a fiber, there is no need to search down the stack to perform a semantic action on reduce. Semantic actions are always executed online and modify the entire fiber freely. Both LR actions and semantic actions are executed in a bottom-up order fitting of a standard LR parser.

Of course, keeping a path in the state stack instead of states decreases state sharing compared to a GSS. Fortunately, an ordering of fiber execution makes sure FGLR fibers merge as soon as possible and as such do not execute more LR actions than necessary. The scheduler's ordering ensures the number of concurrent fibers is reduced to its minimum.

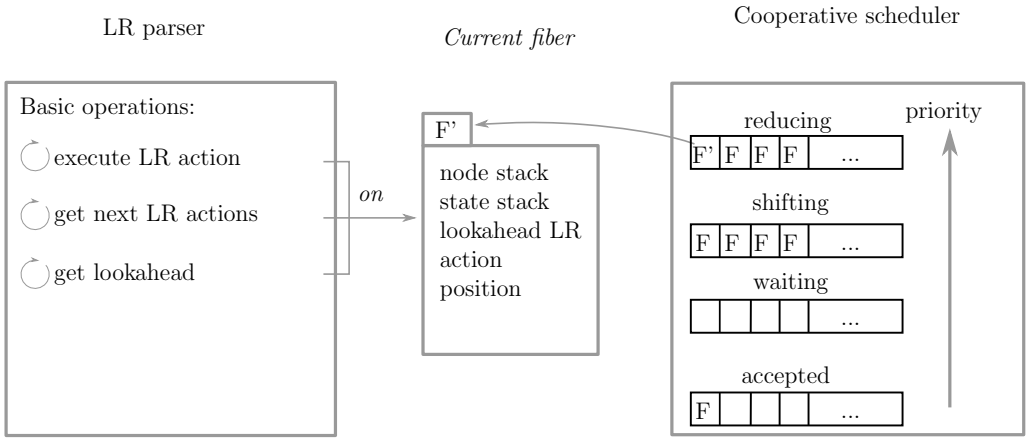

Figure 4.1: Global FGLR architecture with its three main components: the LR parser, the FGLR scheduler and the parsing fibers.

Figure 4.1 illustrates the general approach of FGLR and its scheduling activities. The LR parser exposes three main functions as an API to the

scheduler: it executes an LR action, gets the next possible actions for a given lookahead token, or gets the new lookahead token. Instead of manipulating an internal state, the LR parser manipulates the current parsing fiber as its execution context. The scheduler initiates the parse by starting a scheduling loop over its priority lists. A list of lower priority cannot have its elements scheduled unless all the other lists of higher priority are empty. The parser is given access to the currently scheduled parser fiber before executing the functions of its API. The execution of an element of the two lists of higher priority (*Reducing* and *Shifting*) leads to the execution of the LR action of the parsing fiber. The execution of *Waiting* means getting the new lookahead token and the possible LR actions for each parsing fiber, then proceeding to schedule them in the lists according to their actions. At the end of the parse, the *Accepted* list contains all the parser fibers that successfully completed their parse.

Section 4.4 goes more in depth on the fibers, the LR parser and the fiber scheduler, while Section 4.5 develops the execution order choices in the scheduler.

4.4 Structure of the Fibered-GLR Parser

The Fibered Generalized LR parsing technique relies on 3 main components: an evolving set of GLR parser fibers, a slightly-modified LR parser and a FGLR fiber scheduler.

4.4.1 FGLR parsing fiber

The FGLR scheduler works with GLR parsing fibers (we will refer to those as "fibers" in the rest of this chapter). Fibers (or coroutines) are a general concept in scheduling similar to threads. Fibers must finish their execution (cooperative scheduling) before the scheduler executes another fiber, contrary to threads that can be preempted by the scheduler at any point in their execution. In our context, fibers describe an execution context (a parser fork internal state) for the parser tasks (processing LR actions, getting a new lookahead token). A task will be executed once, then the scheduler takes over and schedules another fiber.

FGLR fibers describe the internal state of the LR parser required for their execution. A FGLR parser fiber contains:

- the state stack,
- the node stack,

- the lookahead token,
- the LR action to be executed,
- the position in the input.

In this scheduling approach, each fork gets a different lookahead according to its internal state. A different lookahead means a different position in the scanner, and this needs to be stored for merging purposes. Additional information can be added depending on what needs to be available to the semantic actions due to specific use case in a parser or due to the implementation of the parser generator. A good example is parsers requiring scopes to distinguish identifiers from typenames (and more generally for any parser which needs to inspect and modify its internal structures to parse successfully). These scopes are added to the fiber.

The fibers are the structure holding all the parser internals for an alternative, it is modified by the LR parser and scheduled by the FGLR scheduler.

4.4.2 The LR parser

FGLR requires a modified LR parser and scanner. Since the fibers contain all the specific information of a fork, every access to an attribute in the parser (or scanner) must be replaced by an access to the corresponding fiber attribute.

As for the behaviour side of the modification, the standard LR parsing loop, in Algorithm 7, must be replaced by the scheduling loop of the FGLR scheduler (Section 4.4.4).

The entire parse is directed by the scheduler. Three fiber-relient routines exist in an LR parser:

- execute the LR action of the current fiber,
- get a new lookahead token for the current fiber,
- get the next actions for the current fiber.

Every routine is called from the scheduler using the same pattern. The scheduler plugs a chosen fiber in the LR parser, then the scheduler asks the parser to run the routine and finally the scheduler unplugs the current fiber.

4.4.3 The FGLR scheduler

The FGLR scheduler is organized around 4 lists of fibers, each having a distinct priority in its execution and a distinct ordering of its fibers.

Algorithm 7 Classic LR parsing loop.

```
1: function parsingLoop
2:     while lookahead ≠ EOF do
3:         if lookahead = null then
4:             lookahead ← getLookahead(scanner)
5:         end if
6:         action ← getAction(lookahead, stateStack)
7:         if action = REDUCE then
8:             performReduce(action, lookahead, stateStack, nodeStack)
9:         else if action = SHIFT then
10:            performShift(action, lookahead, stateStack, nodeStack)
11:            lookahead ← null
12:        else if action = ACCEPT then
13:            return first(nodeStack)
14:        else
15:            return error
16:        end if
17:    end while
18: end function
```

Reducing. This list contains all the fibers which next LR action is *reduce*. This *Reducing* list is sorted using a specific metric described later in Algorithm 10. The *Reducing* list has the highest priority of execution.

Shifting. This list contains all the fibers which next LR action is *shift*. While not sorted in the current algorithm, the scheduler's structure allows for specific sorting mechanisms to be added effortlessly. The *Shifting* list has a lower priority than the *Reducing* list: its fibers will only be executed after all the fibers from *Reducing* have been processed.

Waiting. This list contains all fibers that previously shifted and thus are waiting for a new lookahead token from the scanner. Whereas the *Reducing* list is processed element by element since processing one may add new elements, the *Waiting* list is processed in a single step. Furthermore, the *Waiting* list has a lower priority than the two previous lists.

Accepted. This list contains all the fibers having successfully completed their parse. Fibers are only added to this list once we are sure they cannot be merged with fibers from other lists. Once in the accepted list, a fiber

will never be executed again (lowest priority list). At the end of the whole parsing process, all accepted fibers node stacks are returned to the caller.

4.4.4 The FGLR scheduling loop

Main loop. The main LR parsing loop is replaced by the scheduling loop described in Algorithm 8.

Algorithm 8 Scheduling loop

```
1: function SCHEDULINGLOOP
2:     while reducing ∪ shifting ∪ waiting ≠ ∅ do
3:         while reducing ≠ ∅ do
4:             executeLRaction(nextReduce())
5:         end while
6:         while shifting ≠ ∅ do
7:             executeLRaction(nextShift())
8:         end while
9:         if waiting ≠ ∅ then
10:            processWaiting()
11:        end if
12:    end while
13:    return accepted
14: end function
```

The scheduling loop is straightforward: while there are reduces in the *Reducing* list, choose the next reduce and execute it. The shifts are then processed in the same way once *Reducing* has been emptied. Lastly, *Waiting* fibers get new lookahead tokens and fork if more than one action is assigned to a fiber. New and old fibers are scheduled in *Reducing* or *Shifting* depending on their action. The next iteration of the scheduling loop then starts.

Executing an LR action. Executing an LR action with the LR parser impacts the scheduler since the fiber state changes (see Algorithm 9).

First, the scheduler gives the reference of the new fiber to the parser, then depending on the action related to the fiber, a shift or a reduce is asked of the parser. Every modification of state stack, node stack, and so on occurs on the fiber and not on the parser itself. Then, if the current fiber can be merged with other fibers in the scheduling lists, it is done, leaving the current fiber as sole fiber from this pool. The current fiber is rescheduled by getting the next action without changing the lookahead (otherwise it is rescheduled to *Waiting*).

Algorithm 9 Execute LR action

```
1: procedure executeLRaction(fiber)
2:     restoreFiber(parser, fiber)
3:     if action(fiber) = REDUCE then
4:         performReduce(parser, fiber)
5:     else if action(fiber) = SHIFT then
6:         performShift(parser, fiber)
7:     end if
8:     if canBeMerged(fiber) then
9:         merge(fiber)
10:    end if
11:    reschedule(fiber)
12: end procedure
```

4.4.5 Forking mechanism

Each time the scheduler queries the parser for the next actions on the current fiber, the fiber is forked into a fiber per new action[2]. The new fibers are then scheduled in the correct list according to their action. Forking is performed during the *reschedule* process (for actions following a reduce) and during the *processWaiting* process (after getting the new lookahead token). For now, a fork entails copying the entire fiber. We are aware that this is far from the best in terms of memory footprint and alternative implementations are discussed in Section 4.7. It is important to note that both forking and merging heavily influence the peak memory consumption of a parse.

4.4.6 Merging mechanism

GSS-based implementations try to minimize the number of states by never adding a state twice on the frontier of the GSS. However, new edges need to be added: the reduction paths through the GSS. FGLR allows duplication of states to always have a unique path for a reduce, leading to straightforward reductions (identical to an LR reduction) applied down a single path. In other words, the GSS has unique states but multiple reduction paths and FGLR has unique reduction paths but duplicate states. While Tomita's GLR minimizes the number of states in parallel, FGLR tries to minimize the number of state stacks in parallel, and, by extension, the number of fibers in parallel. To do that, after each LR action, the state stack of the current fiber must be confronted with the state stacks of the other active fibers in the

[2]The original fiber is still reused.

system. All the other active fibers (from *Reducing*, *Shifting* and *Waiting*) are candidates for merge. The fibers merge if their state stacks are identical and if their positions in the scanner are identical.

If the merge condition is respected, the top of each node stack is merged into the top node of the current fiber [3]. The merge is done by moving each top node of each node stack into a new "ambiguous" node, and replacing the top node of the current fiber by the ambiguous node [4]. This ambiguous node now keeps track of ambiguous parts in the new node stack. As a side note, when parsing is done, an AST with ambiguous nodes could be converted into a forest of unambiguous ASTs or other formats such as SPPFs.

Then, after merging the node stacks, all the "mergeable" fibers except the current one are dropped from the scheduling list and the current "merged" fiber is rescheduled. Since the parsing is done on fibers, fibers can be customized with additional merge conditions should the parser implementor need it.

4.4.7 Rescheduling

Immediately after attempting to merge the current fiber with other active fibers, the scheduler reschedules the fiber. If the executed action was a shift, the fiber is waiting for a new lookahead and as such will be transfered to the waiting list. However, if the action was a reduce, we ask the parser for the next possible actions and schedule them (if more than one, we fork the current fiber for each new action). This means adding new reducing fibers and new shifting fibers, but more importantly it ensures that all possible fibers are available for merging and execution in the scheduling lists. The ordering of fibers in the different lists and the priority of the lists will be discussed in the next section.

4.5 Execution order choices

To ensure the LR actions and their respective semantic actions are executed at the same time, we must execute semantic actions in a bottom-up order and without delaying. If we do not want to delay semantic actions (executed during reductions), we need a valid reduction path at this point.

[3] As a reminder, if two fibers are merged, the rest of the node stacks are identical, and thus does not concern us here.

[4] As is common in AST nodes, recursion is represented by a collection of nodes. It adds a step in the node stack merging: the ambiguous node must be pulled down the lists and nodes to the lowest level in the subtree.

Of course, if we keep track of the valid reduction paths, we need to duplicate more states. To prevent an explosion of the number of concurrent fibers, it is important to merge identical fibers as soon as possible. A special ordering of fiber execution is required to ensure we do not execute more action than would be necessary before merging fibers. This results in a global scheduling of all active fibers during the parse.

The global scheduling of fibers is influenced by two local orderings: the priority of the scheduling lists and the ordering in each list. The former is used to make sure LR actions execute in the right LR order and the latter is used to merge sooner.

4.5.1 Inter list ordering

The *Reducing* list is of the highest priority because a reducing fiber produces new actions (new reduces or a new shift) without affecting the lookahead token. The scanner part of the merging condition states that to merge as soon as possible, the scanner should be kept as stable as possible too. That is why, the *Reducing* list is scheduled first: its execution schedules new LR actions (*Reducing*) without modifying the lookahead token.

Then, the *Shifting* list should be scheduled since it modifies the state stack and resets the lookahead. The modifications from the shifts offer fewer merging opportunities, so they should be scheduled after every current reductions are done. When no actions are executed (the *Reducing* and *Shifting* lists are empty), all active fibers are present in the *Waiting* list.

Fibers from the *Waiting* list can be merge candidates, but they do not execute LR actions thus will never initiate merging themselves. Waiting fibers modify the scanner part (position) of the fiber by getting a new lookahead from the scanner, but do not touch the state stack. Due to those two factors, the *Waiting* list is the last list to be executed, filling the other two of higher priority. This essentialy functions in the same way as the GSS, the synchronization is done on the next lookahead, with the shifts being the previous action and reduces before that (a loop of reduces, shifts and lookahead).

The *Accepting* list is never merged and all node stack tops from accepted fibers will be returned. Each list that contains active fibers (not accepted ones) will be ordered as described in the next sections.

4.5.2 Reducing list ordering

The *Reducing* list is ordered using the sort function described in Algorithm 10.

Algorithm 10 Reducing list sort metric

```
1: function SORT(first, second)
2:     firstReduceSize = size(first.stateStack) − size(first.action)
3:     secondReduceSize = size(second.stateStack) − size(second.action)
4:     return firstReduceSize >= secondReduceSize
5: end function
```

Given two fibers f_1 and f_2, f_1 is scheduled first iff the size of its stack after the reduce would be greater than or equal to the one of f_2 after its own reduce. In other words, $|s'_1| >= |s'_2|$ where $|s'_1| = |s_1| - |r_1|$ and $|s'_2| = |s_2| - |r_2|$, $|s_k|$ and $|r_k|$ being respectively the size of the stack before the reduce and the size of the right-hand side of the rule of the reduce[5].

This metric ensures we always perform the smallest reductions with minimum impact on the stacks first. Since the merge is done on every LR action (shift and reduce on any fiber), this should allow one to merge as soon as possible. No LR action should be executed multiple times due to this merging mechanism.

While we leave a formal proof that this ordering leads to the earliest merging possible to future work, the scheduler is designed to be easily modified to introduce another sorting mechanism in case this metric proves obsolete. This metric is similar to the approach used in Elkhound [MN04], giving us confidence in our ordering.

4.5.3 Shifting list ordering

The *Shifting* list ordering is unimportant in this setting. To merge, both fibers must have the same state stack and the same scanner position. Let us consider fibers F_1 with $stack_1 = (s_0, ..., s_n)$, $action_1 = shift(k)$ (where k is its new state), position p_1 and F_2 with $stack_2 = (s_0, ..., s_n, s_m)$, $action_2 = shift(l)$ (where l is its new state), position p_2. $stack'_1 = (s_0, ..., s_n, s_k)$ is the hypothetical state after $action_1$ has been executed. Shifts add a state to the state stack, the only way they merge is if the stacks become equal (in other words, the new added state k is the top of the stack $stack_2$) and $p_1 = p_2$. $stack'_1 = stack_2$ only if $s_k = s_m$ and the rest of the stacks are equals, meaning F_1 and F_2 had an identical state stack and token (position). This is impossible, as two such fibers with identical actions cannot be scheduled, so they must occur after an action and, in this case, they would have been merged beforehand (after a reduce).

[5]The actual size of the stacks would in fact be 1 more (each reduce's goto), but it is omited since it does not affect the comparison.

4.5.4 Waiting list processing

The *Waiting* list processing algorithm is presented in Algorithm 11.

Algorithm 11 Waiting list processing

```
1: procedure PROCESSWAITING
2:     for all fiber ∈ waiting do
3:         restoreFiber(parser, fiber)
4:         getLookahead(parser, scanner, fiber)
5:         actions ← getNextActions(parser, fiber)
6:         unregister(fiber)
7:         if size(actions) = 1 then
8:             register(fiber, first(actions))
9:         else
10:            for all action ∈ actions do
11:                forkFiberForAction(fiber, action)
12:            end for
13:        end if
14:    end for
15: end procedure
```

Each fiber queries the parser for a new lookahead and then for new actions given the new lookahead token. The fiber is then removed from the scheduling lists. If there is a single possible action, the fiber is registered with its new action. If there are multiple actions (a conflict) for the fiber's lookahead, the fiber is forked for every action. Each new fiber is registered in the scheduling lists with its new action.

As a side note, the scheduler puts fibers whose next actions are *ACCEPT* inside the *Waiting* list (once) so they can be merged before joining the *Accepting* list. It was not included in Algorithm 11 due to being an implementation choice (one could have created a new list instead for example) dependent upon the node stack merging mechanism. That way identical or compatible ending nodes on the node stack can be merged into an ambiguous node.

4.6 Experiments

This section is dedicated to evaluating FGLR's scaling with regards to the two different types of conflicts: shift-reduce and reduce-reduce. We also take a look at FGLR's number of executed actions compared to Bison using a

list of stacks and SmaCC also using a list of stacks, all on the same micro grammar and input.

4.6.1 Implementation

FGLR has been implemented in SmaCC (Smalltalk Compiler-Compiler) [BLG+17], a parser generator written in Smalltalk and originally developed by John Brant and Don Roberts. The FGLR implementation is purely a runtime change to LR/GLR and does not impact the LR/LALR parse table generation. The only modifications are the addition of the scheduler, fibers and the access patterns of the LR parser. The whole FGLR runtime implemented in SmaCC also contains an implementation of syntactic pattern matching through parsing as intersection [LBGD18].

4.6.2 Experimental setup

While Bison and SmaCC GLR cannot hope to compete against a proper GSS GLR in terms of memory footprint and complexity, they are still used in practice when parsing programming languages. We want to validate that the FGLR algorithm follows the same trend and does not become prohibitive in terms of memory or computation. We also validate that FGLR merges more often than the Bison or SmaCC GLR, thus executing fewer actions. In that vein, we first evaluate FGLR against Bison and SmaCC on a small ambiguous grammar. Then we compare FGLR's scaling in the presence of reduce-reduce and shift-reduce conflicts relative to the one of SmaCC GLR.

4.6.3 Comparison with Bison GLR and SmaCC GLR

To illustrate FGLR differences with standard Bison and SmaCC GLR, we propose an example of an ambiguous grammar containing both shift-reduce and reduce-reduce conflicts (in Listing 4.4).

```
1  S
2       : S "a" S
3       | S "b" S
4       | C
5       | D
6       ;
7
8  C
9       : "n"
10      ;
11
```

```
12 D
13     : "n"
14     ;
```

Listing 4.4: Ambiguous grammar featuring reduce-reduce and shift-reduce conflicts.

The input string `"nanbn"` has 3 reduce-reduce and 1 shift-reduce conflicts. Every `"n"` results in a reduction to `C` or `D`. Every `"nan,b"`, meaning `"nan "` has been recognized and `"b"` is the lookahead, results in a shift of `"b"` or a reduction of `"nan"` to `S`. Parsing this input string with each parser derived from this same grammar leads to the number of actions presented in Table 4.1.

Table 4.1: Action comparison of GLR implementations with FGLR

	# Shifts	# Reduces	# Actions
Bison GLR	8	24	32
SmaCC GLR	10	24	34
FGLR	**7**	**20**	**27**

On this simple example, FGLR executes slightly fewer actions than Bison or SmaCC GLR because it merges faster than the other two approaches. On highly ambiguous inputs, the number of concurrent ambiguities (and thus the number of forks) shapes the general profile of the runtime performance: exponential with concurrent ambiguities. However, the number of concurrent ambiguities can be reduced if they can be resolved earlier (by merging) thus impacting the performance of the parse. So we decided to take highly ambiguous input (reduce-reduce and shift-reduce conflicts) and check their parse time to deduce information about the number of executed actions on ambiguous input.

4.6.4 Scaling of FGLR on highly ambiguous grammars

We reuse the previous conflicting grammar in Listing 4.4 on input strings of increasing size to compare the scaling of FGLR to SmaCC GLR. We generated input strings of the form of `n([ab]n){k}`, where `k` is a repetition factor, and fed these to their respective FGLR and GLR parsers in SmaCC. These experiments were realized on Pharo 6.1[6] using an OpenSmalltalk Virtual

[6]Pharo Smalltalk: https://pharo.org/

Machine[7], monothreaded on an Intel i7-7700. Figure 4.2 shows the resulting parse times. These results should be taken with a grain of salt since they ran on a monothreaded virtual machine, but they are useful for comparison.

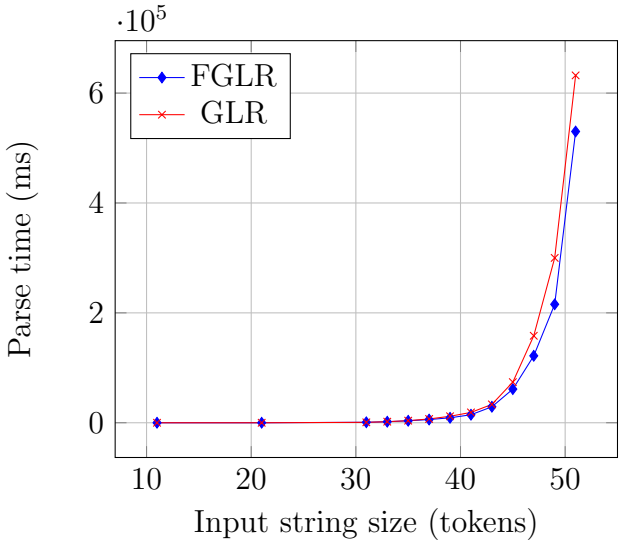

Figure 4.2: FGLR and GLR execution time for highly ambiguous inputs.

Both approaches exhibit an exponential growth, which should not come as a surprise because GLR algorithms are highly sensitive to the density of ambiguities. Here, every token leads to an ambiguity (shift-reduce or reduce-reduce) which is a very bad case for GLR. Even if both are exponential, FGLR grows slower due to merging some forks earlier and thus not duplicating unnecessary actions between forks. On the contrary, the SmaCC GLR implementation merges later and thus performs the same actions in parallel before merging the two forks.

To identify on which type of ambiguity FGLR performs best, two new grammars are introduced: the first one has been rewritten to only contain shift-reduce conflicts (Listing 4.5) and the second one to only contain reduce-reduce conflicts (Listing 4.6). Since FGLR adds the possibility to merge after a reduce, we expect it to show a better run time gain on reduce-reduce conflicts compared to shift-reduce.

4.6.5 Sensitivity to shift-reduce conflicts

Next, we evaluate the impact of shift-reduce conflicts on FGLR compared to SmaCC GLR. The grammar is modified to remove the reduce-reduce conflict,

[7]OpenSmalltalk initiative: http://opensmalltalk.org/

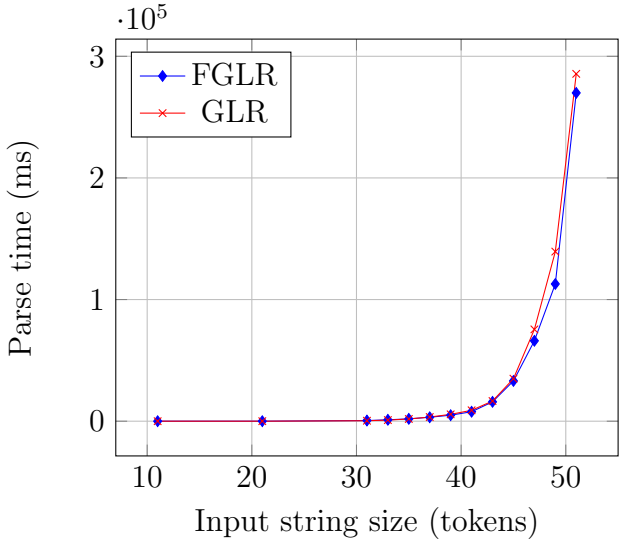

Figure 4.3: Shift-Reduce: FGLR and GLR execution time for highly ambiguous inputs.

leaving only shift-reduce conflicts as illustrated in Listing 4.5.

```
S
     : S "a" S
     | S "b" S
     | C
     ;

C
     : "n"
     ;
```

Listing 4.5: Ambiguous grammar with shift-reduce conflicts.

Figure 4.3 presents the results of parsing the same generated input strings `n([ab]n){k}`. The difference of FGLR vs list-of-stacks GLR is not as clear cut on a shift-reduce only grammar and slightly worse on lower input size. This is mainly due to the overhead of the scheduler when there is no good merge opportunity to be found, even with a good scheduling. Shift-reduce conflicts are often resolved much later, after all the token of their common "ancester" rule have been consumed, leading to an impossibility to merge. For example, the input `"nanbn"` has two possible parses `na(nbn)` and `(nan)bn` which cannot be merged before the end of the parse where they both become an `S`. FGLR works best on local conflicts that have a short delay between their appearance and their possibility to be resolved.

4.6.6 Sensitivity to reduce-reduce conflicts

Lastly, we evaluate the run time impact of reduce-reduce conflicts by creating a new grammar (Listing 4.6) where shift-reduce conflicts are removed.

```
S
     : E S
     |
     ;

E
     : C
     | D
     ;

C
     : "n"
     ;

D
     : "n"
     ;
```

Listing 4.6: Ambiguous grammar with reduce-reduce conflicts.

The results of parsing input strings generated from `n{k}` are presented in Figure 4.4. The list-of-stacks (with merging) implementations will tend to reduce `"n"` to `C` then `E` and `"n"` to `D` then `E` before merging the two forks whereas FGLR will reduce `"n"` to `C`, `"n"` to `D`, merge and reduce only once to `E`. FGLR is effective to detect merge opportunities inside a deep reduction list. In other words, when two reduces are followed by the same reductions, they will merge to keep only one unique reduction path. Most reduce-reduce conflicts tend to be localized and thus easier to resolve for FGLR compared to shift-reduce ones.

FGLR exhibits a similar general behaviour to list-of-stacks GLR on highly ambiguous input string while having slightly better performance due to merging earlier.

4.7 Discussion

FGLR on LR(1) grammars. FGLR behaves identically to an LR parser on LR(1) grammars since there will only be a single fiber, always being scheduled alone, never being merged. Reductions and semantic actions are trivially compatible. FGLR also trivially behaves akin to a LR parser on unambiguous parts of the parse or if no ambiguities are encountered.

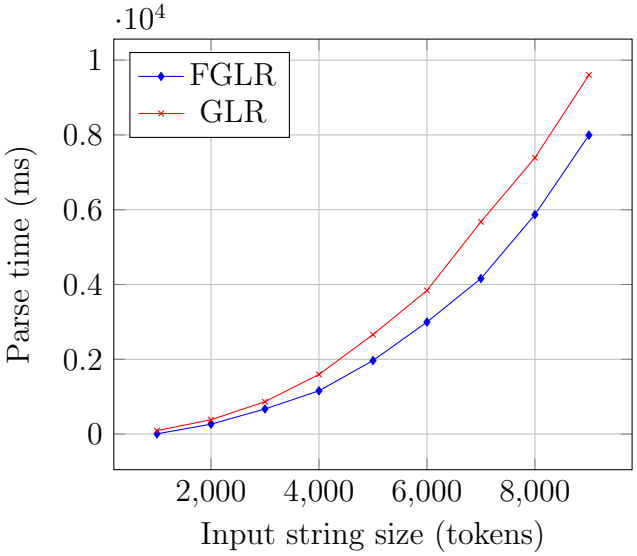

Figure 4.4: Reduce-Reduce: FGLR and GLR execution time for highly ambiguous inputs.

Targeted grammars. Due to the fiber fork, FGLR is not suited for parsing natural languages or other highly ambiguous languages. FGLR works best for grammars that are near-deterministic with a few local ambiguities. FGLR's main targets are programming languages: generic ones or Domain Specific Languages (DSL) which mostly respect this condition.

Memory footprint. Compared to a GSS with unicity of state on the frontier, FGLR is worse in terms of memory consumption. It is the price to pay to have unique paths for reductions. Compared to list of stacks where the merge only happens on shifts, FGLR merges more often (thanks to the ordering) and performs fewer LR actions, leading to less memory consumption. In certain occurences, a reduce triggers the merge of fibers even before they are executed (as long as they have been scheduled).

The fiber duplication is costly in theory if many hard-to-resolve ambiguities appear, but this limitation is true for every GLR algorithm. Optimizing FGLR by prefix sharing may seem a good idea at first glance, but semantic actions typically cost more than simple LR actions on the automaton, thus state stack operation should not be optimized at the expense of semantic actions, defying the purpose of the approach.

As for memory optimizations, these should be treated carefully to not deviate from the original purpose of the algorithm and adding complexity in the handling of semantic actions.

Execution time. LR actions in FGLR have the same cost as in standard LR parsing. FGLR adds an extra scheduling cost that is decomposed into three main parts: sorting the reducing list, checking for merge candidates and unregistering fibers from the scheduler. According to McPeak et al. [MN04], the GSS costs lie in handling state sharing to avoid duplication and its memoization of work items to avoid unnecessary computation. All of FGLR's costs (sorting, checking for merge and unregistering) are dependent on the number of parallel fibers, in the same way as other list of stacks. Many of the FGLR checks for merge could use some sort of memoization to avoid multiple identical checks. In addition, better data structures could be introduced in the scheduler to make unregistering easier to perform.

4.8 Conclusion

A previous version of this work [LG19] has been submitted to the journal *Science of Computer Programming*, and at the time of the writing is still under review.

In this chapter, we proposed the Fibered-GLR parsing algorithm, a reinterpretation of the original GLR parsing algorithm designed to enable side-effect-based parsing. To the difference of classic GLR implementations, FGLR does not delay semantic actions and ensures the same order of execution between LR actions and semantic actions, a bottom-up order. FGLR is based on a scheduler ordering fibers, a clear representation of parser internals for a given parsing alternative. This enables for side effects to be executed in an online fashion, a mandatory requirement to disambiguate during the parse. The algorithm scales better with ambiguities than a list of stacks GLR and while having a worse complexity than a GSS GLR, it allows for side-effect parsing.

Chapter 5
Conclusion

Contents

5.1 Summary of contributions

In this work, we built a source-to-source code transformation engine. This transformation engine is designed to handle arbitrary source-to-source transformations written by both language experts and non experts alike. Our main contributions are its pattern matching engine to handle hybrid patterns while requiring grammars to generate a language back-end, and the generated GLR parser for side-effect-based disambiguation.

Both contributions have been implemented on top of SmaCC, a Smalltalk parser generator. The Smalltalk source code for our engine is available on a fork of SmaCC's original repository at: https://github.com/ToshRaka/SmaCC.

5.1.1 Hybrid explicit-syntactic pattern matching engine

Pattern matching engines in transformation tools come in two forms. Explicit pattern matching requires the user to write patterns in the form of element

of the IR of the matched language. Learning an IR for a language is typically a heavy knowledge burden, making explicit patterns ill-suited for new users. On the contrary, syntactic pattern matching feature patterns written in the syntax of the matched language, with metavariables to act as wildcards. Syntactic patterns are great for new users, because they do not need to learn the IR to match the language, but some patterns are still easier to express with explicit patterns. Thus, an hybrid explicit-syntactic pattern matching engine is desirable.

Implementing the syntactic part of the engine requires an additional step compared to the explicit part: the type inference phase relying heavily on hand-written language back-ends. To ease the implementation of the type inference, we reduce language back-ends to their parsers, generated by a parser generator. With an adaptation of parsing as intersection for pattern matching, we can use the same parsing algorithm for parsing the source code of the program to match and to perform the type inference of the hybrid pattern. We modified SmaCC's GLR parser generator to generate parsers compatible with type inference from the BNF/EBNF grammar of the language. The grammar needs only a single additional line of code, defining how to recognize a metavariable, and the parser generator can generate the language back-end.

Parser generators typically perform a number of modifications to the parse tree of the language to create a compact and reasonable AST. An example of modification is the removal of intermediate recursion nodes and the addition of lists of nodes to represent recursion instead. Matching this list idiom is challenging since it removes the direct mapping between AST nodes and parse tree nodes. We also proposed a method to express and match said AST list idioms in our hybrid explicit-syntactic patterns.

5.1.2 Scheduler-based GLR

Our modified GLR parsers relies on exploring all type alternatives for each metavariable in the same way these parsers explore all parsing alternatives when they reach a conflict. Exploring the alternatives is done by forking, and forking for both conflicts and metavariables is runtime and memory intensive. Other disambiguation techniques would prove beneficial for the engine, however the main alternative disambiguation technique of exploiting side effects in the parser is not available in GLR. Side effects let implementors write disambiguation code and use additional structures to choose between alternatives during the parse, to be precise, during the semantic actions. Semantic actions need to run in an online fashion to allow for side effects. In other words, semantic actions need to be executed in bottom-up order, the

same order as reductions.

We proposed a GLR parser based on a scheduler to ensure the execution order of the parsing alternatives in a bottom-up order. Parsing alternatives are isolated as independent structures, not only containing all the parser internals for said alternative, but also any additional structure the implementor choose to implement for disambiguation. As in list-of-stacks implementations of GLR, Fibered-GLR has separated state stacks for each alternatives, merged when identical. Due to its ordering, FGLR merges faster than standard list-of-stacks GLR implementations. Keeping independent stacks is vital to implementing online semantic actions. The other technique to implement GLR uses a single graph, a GSS, instead of independent stacks to represent the state stacks. While reducing the algorithm complexity by sharing the maximum number of states, it cannot execute online semantic actions with side effects.

5.2 Validating the requirements

In the introduction (Chapter 1), we derived requirements from our two main use cases: code migration and code specialization. We divided the requirements in two categories, those from the point of view of the user and those from the point of view of the language implementor. In this section, we will discuss their realizations and the shortcomings of the said realizations.

5.2.1 User requirements

Multi-language. The transformation engine is designed to support multiple languages through multiple grammars. The language back-ends are in fact our parsers generated from those grammars with a modified parser runtime to handle hybrid explicit-syntactic patterns. These hybrid patterns validate the second part of the requirement: easily match a language that is already known to the user. A combination of syntax-based metavariables and patterns for unknown IR constructs and IR-based patterns for known ones fit this requirement.

No IR knowledge required. As for not needing to learn a language's IR to match it, we only partially validate this requirement. The patterns can be purely syntactic without IR types, and will match every simple type, just as an explicit pattern could. However, the current implementation of complex types does not yet support the matching of AND types (sequences) without adding some type information.

Listing 5.1 shows an example of a pattern containing a single metavariable with an AND type. Both `Node` and `","` are explicit type information, the first one refers to a specific AST node and the second a specific token.

```
1 ‘aList: Node ("," Node)*‘
```

Listing 5.1: AND types with IR knowledge.

To remove type information entirely from the patterns would require to get rid of these node and token types in AND types. Section 5.3.1 gives leads on how to match list idioms without type information.

Hybrid approach. Our transformation engine is indeed featuring hybrid explicit-syntactic patterns. By default, the pattern behave in a syntactic way, except individual metavariables can be typed explicitly by the user. Pattern types can also be specified, but they are only checked at the end. The current implementation does not yet use this type to reduce the set of possible starting states. This would be an immediate improvement and remove impossible parsing alternatives for the pattern from the start.

DSL-supported. Sadly, the transformation engine does not feature a DSL for code transformation. Work has been done on the back-end to fit a DSL, such as filtering aid to narrow down the matches of a pattern. Pattern refinement is a process by which a pattern can be applied on the result of another pattern (its matches) to get other more interesting matches. This iterative process makes each individual pattern easier to write than a single complex pattern would be. Pattern refinement primitives have been added to the engine. Currently, both filtering and pattern refinement need to be used in the engine's implementation language and not a dedicated DSL, which would be desirable.

5.2.2 Implementor requirements

Easy language back-end implementation. To ease the creation of language back-ends, we chose to go the way of automated tooling to generate the back-end instead of going for on-the-shelf back-ends. In that regard, having a single algorithm to parse the source code and to perform the type inference of the patterns helps. The language back-end implementation is reduced to finding a grammar for the language (the best case) or creating one. Grammars contain orders of magnitude less raw lines of code than typical language back-ends in a general purpose languages. However, it can be opposed that grammars are more complex objects and harder to create. We think that

it is better to invest time in creating tools to help with the development of grammars, rather than invest this time in implementing language back-ends for pattern matching from scratch.

Seamless hybrid integration. Our pattern matching engine is unique for both explicit and syntactic pattern matching. Patterns feature both explicit and syntactic matching inside a single pattern. We achieved a seamless integration between the two kinds of patterns: their is no need for two separate engines or back-ends, and no need to isolate the two kinds of patterns.

Interchangeable DSL front-end. Since the user requirement of developing a transformation DSL was not achieved, this one was not either. Their is no one-size-fits-all syntax across multiple domains and since we designed the engine to support multiple languages, with experts and non experts in mind, the syntax for a future transformation DSL would need to be appropriate to the domain. The back-end (the engine) should be suited to accept different syntaxes.

5.3 Perspectives

In this last section, we will explore perspectives to this work, ranging from direct improvements to the pattern matching engine, to general tools and techniques for transformation engines that would benefit an hybrid syntactic-explicit code transformation engine such as ours.

5.3.1 Improvements on matching complex types

Matching complex list idioms resulting from the parser generator's transformations of recursion into lists is done through AND types. AND types as their name suggest are a sequence of explicit types. For new users, it is always desirable to have all constructions in the AST to be matchable with syntactic patterns only. AND types, even if they solve the problem of matching collections in ASTs, go against the philosophy of matching everything syntactically. To remove this dependency on explicit types, we see multiple leads.

Submetavariables. The first method is to replace node types by submetavariables, metavariables defined inside another metavariable of AND type. In the example `‘aList: Node ("," Node)*‘`, `Node` types would be replaced by two submetavariables `‘x‘` each matching a different set of nodes,

but both of the same type. However, even with `Node` types, `","` is still type information present in the pattern. Token types are harder to get rid of since they contain syntactic information.

Subpatterns. The second method is to formalize the definition of a subpattern, a syntactic pattern embedded in a metavariable. The subpattern could even be defined outside of the metavariable and linked during the type inference. There are a number of problems to solve with syntactic subpatterns. Their abstract matches are highly dependent on the their parent pattern since the position of the subpattern in its parent determines its starting state. How should patterns with subpatterns be parsed? The regular language operators such as Kleene star are not part of the matched language. Should they be added as non-conflicting tokens in the same vein as metavariables? Should there be a preprocessor handling the regular language operators instead? Or should it be dealt with in the implementation of the parser? We hope that in time the community formalizes the concept.

5.3.2 Interactive tooling and pattern generalization

Interactive match tooling. To gain broader acceptance in other domains, syntactic pattern matching would greatly benefit from interactive tools that could relate the user to the IR and then to the grammar of the matched language. Interactive visualisation of matches (abstract and concrete) helps in explaining why a given pattern matches an unusual piece of code or on the contrary, why something is not matched even if we thought it should.

Conceptualy, a grammar is a finite representation of a potentially infinite set of sentences (trees) that derives from it. In the same way, a pattern is a finite representation of a potentially infinite set of matches. When the pattern is compared with one source AST, we get the set of matches for this AST, which contains only a small portion of what the pattern could match as a whole in the language, but that crucially we did not get to encounter in this AST. Visualization of the abstract and concrete matches, and the link between the two, would help the user understand what is being matched and why it is being matched.

Pattern generalization. In the same vein of getting the user to progressively know better how to match a program, it is desirable to have an additional tool to generalize and mutate syntactic patterns. The generalization of patterns comes in two ways. First, a syntactic pattern is created from an *example* of the source code and progressively generalized with metavariables

to match similar snippets of code. Second, a syntactic pattern is generalized from another syntactic patterns by mutating it (e.g., replacing syntactic elements with a metavariable). The idea of pattern generalization is not new, it has been for example implemented in Ekeko/X [MDR16a] through proposition of mutation operators. Proposing similar and generalized patterns automatically to the user is a great way to help understand the tool.

5.3.3 Code transformation DSL

The code transformation engine we are developing aims mainly at helping code specialization and migration, through scripted arbitrary code transformations written by experts and non experts alike. Creating code transformations and ordering code transformations is a cumbersome task in a general purpose language. However, it is specific enough to be a good target for a dedicated DSL for automated arbitrary code transformations.

Transformation DSLs have been proposed in the past, mainly focusing on the refactoring subset of transformations. RefactoringScript [YKS+14] and Wrangler [LT12] are both examples of DSLs to combine atomic refactorings into composite refactorings and combining those in turn. Refactoring implementations are inherently tightly coupled with their language. Another great example of transformation DSL is SmPL (Semantic Patch Language), the DSL of the Coccinelle [PLM07] transformation engine. Coccinelle was created to tackle collateral evolution of drivers in Linux (written in C) by providing ways to write easy-to-distribute semantic patches to automatically rewrite APIs. From the patch-like syntax of SmPL to the control-flow-based matching of Coccinelle, every part of the engine is designed with the workflow of Linux kernel and driver developers in mind. These DSL examples give an idea of what kind of DSL should we aim towards.

DSL rationale. The idea of this DSL is to provide a highly flexible language for arbitrary source code transformations, mainly with compile-time code specialization and code migration in mind. A scripting and textual format is ideal, since a transformation would be executed in a pipeline in an offline manner. As far as the syntax go, it should be quickly understood by the users and as such close to known languages in the space. Since domains and use cases for code specialization and migration are numerous, we should not restrict ourselves to a particular syntax and first consolidate a strong back-end.

The main components of the language are the syntactic patterns, metavariables, concrete matches, contexts and transformations. Each of these components should be a first-class citizen in the language. It must be possible

to assign them to variables, to modify them, to pass them as arguments and to use them as return values.

Syntactic patterns. Syntactic patterns can either represent matching patterns or rewriting patterns (see Listing 5.2). The same representation is used for the two use cases.

```
1 `a` + `a`
2 >>>
3 `a` ^ 2
```

Listing 5.2: Syntactic patterns for matching and rewriting.

Syntactic patterns expose their context and their source code. Their context can be modified in the case of input metavariables, or queried in the case of output metavariables. A syntactic pattern can be compiled into its corresponding abstract matches according to a parser. While abstract matches do not offer much use as is for the user, a debugger could exploit them for user feedback, if the pattern does not belong to the language for example.

Metavariables. Metavariables expose their name, type, cardinality, parent and children. The type results from the type inference yielding abstract matches, and the cardinality is the one specified by the user if any using Kleene operators. Child metavariables are the sub-metavariables holding the sub-types of a parent metavariable with an AND type.

Contexts. Contexts are dictionaries whose keys are metavariables and values are nodes of the source AST, or collections of nodes in case of an AND type metavariable. Metavariables in a context can be bound to a node or collection of nodes and can be accessed using their name. It is still unclear if child metavariables should be addressed through name aliasing of their parent, by index or both.

Matches. Here, matches correspond to *concrete* matches, actual source code AST subtrees matched by a syntactic pattern. Matches exposes their subtree and their context, which contains the metavariables and their matching nodes or collection of nodes. Matches can then be transformed using a rewriting pattern.

Transformations. Transformations apply a transformation pattern, a syntactic pattern, to a set of concrete matches. Each individual transformation

uses the context of its concrete match to fill in the metavariables in its pattern. The transformation should be transactional with operators to commit or abort a transformation or set of transformations.

5.3.4 Transformation control flow and strategies

Code migration is an example of sizable transformation use case where all the transformations need to be applied in one pass. Applying one transformation, parsing the result and applying another transformation is complex since it would create an intermediate result in between the original language and the new language. If we cannot iterate on intermediate transformation results, it is necessary to control the order and scope of application of our rewriting rules.

Strategies, or rewriting strategies, control the order of application of rewriting rules, both in depth, i.e. should we rewrite children before their parents, and in breadth, i.e. which transformation among those possible for a node should take precedence over the others. While transformations define actual rewriting to be performed locally, strategies provide the control flow of execution of those transformations. Some transformations need other transformations to have occured, or certain conditions to be met, before being executed.

Stratego [BKVV08] is a strategy language for term rewriting, and other strategy-based rewriting systems should be an inspiration to integrate strategies in transformation engines, particularly for a migration use case. A combination of global strategies and customizable local strategies is necessary to scale to the rewriting of hundreds of rules to a code base, without having to write the entire control flow of the rule application by hand.

Bibliography

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

[AKB15] A. M. AbdelLatif, A. Kamel, and R. Bahgat. An implementation of a fast threaded nondeterministic ll (*) parser generator. *International Journal of Computer Applications*, 130(5), 2015.

[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.

[AVvdS19] R. Aarssen, J. Vinju, and T. van der Storm. Concrete syntax with black box parsers. In *International Conference on the Art, Science, and Engineering of Programming*, 2019.

[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.

[BBC+03] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML path language (XPath). *World Wide Web Consortium (W3C)*, 2003.

[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.

[BHPS61] Y. Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. *STUF-Language Typology and Universals*, 14(1-4):143–172, 1961.

[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.

[BLG+17] J. Brant, J. Lecerf, T. Goubier, S. Ducasse, and A. Black. *SmaCC: a Compiler-Compiler*. The Pharo Booklet Collection. Pharo, 2017.

[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.

[BP08] J. Bovet and T. Parr. ANTLRWorks: an ANTLR grammar development environment. *Software: Practice and Experience*, 38(12):1305–1332, 2008.

[BPM04] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. *26th International Conference on Software Engineering*, pp. 625–634, 2004.

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

[Coc70] J. Cocke. Programming languages and their compilers: Preliminary notes. 1970.

[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.

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.

[EKV09] G. Economopoulos, P. Klint, and J. Vinju. Faster scannerless glr parsing. In *International Conference on Compiler Construction*, pp. 126–141. Springer, 2009.

[EOW07] B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *European Conference on Object-Oriented Programming*, pp. 273–298. Springer, 2007.

[For04] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 111–122, New York, NY, USA, 2004. ACM.

[Fow18] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[GG12] P. Gazzillo and R. Grimm. Superc: parsing all of c by taming the preprocessor. *ACM SIGPLAN Notices*, 47(6):323–334, 2012.

[GHB10] F. Geller, R. Hirschfeld, and G. Bracha. Pattern matching for an object-oriented and dynamically typed programming language. Master's thesis, Universitätsverlag Potsdam, 2010.

[GJ05] A. Garrido and R. Johnson. Analyzing multiple configurations of a c program. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pp. 379–388. IEEE, 2005.

[GJ08a] D. Grune and C. J. Jacobs. Parsing as intersection. In *Parsing Techniques*, pp. 425–442. Springer, 2008.

[GJ08b] D. Grune and C. J. Jacobs. *Parsing Techniques — A Practical Guide*. Springer, 2008.

[GJ13] A. Garrido and R. Johnson. Embracing the c preprocessor during refactoring. *Journal of Software: Evolution and Process*, 25(12):1285–1304, 2013.

[GMJ06] A. Garrido, J. Meseguer, and R. Johnson. Algebraic semantics of the c preprocessor and correctness of its refactorings. Technical report, University of Illinois at Urbana-Champaign, Urbana IL 61801, USA., 2006.

[Gri04] R. Grimm. Practical packrat parsing. Technical report, New York University, 2004.

[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.

[HOBH13] M. Hafiz, J. Overbey, F. Behrang, and J. Hall. Openrefactory/c: An infrastructure for building correct and complex c transformations. In *Proceedings of the 2013 ACM workshop on Workshop on refactoring tools*, pp. 1–4. ACM, 2013.

[IVDSE14] P. Inostroza, T. Van Der Storm, and S. Erdweg. Tracing program transformations with string origins. In *International Conference on Theory and Practice of Model Transformations*, pp. 154–169. Springer, 2014.

[JCB+15] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet. Mashup of metalanguages and its implementation in the Kermeta language workbench. *Software & Systems Modeling*, 14(2):905–920, 2015.

[JK06] B. Jay and D. Kesner. Pure pattern calculus. In *European Symposium on Programming*, pp. 100–114. Springer, 2006.

[Joh75] S. Johnson. Yacc: Yet another compiler compiler. Computer Science Technical Report #32, Bell Laboratories, Murray Hill, NJ, 1975.

[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.

[KDSD+09] A. Kellens, K. De Schutter, T. D'Hondt, L. Jorissen, and B. Van Passel. Cognac: A framework for documenting and verifying the design of cobol systems. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pp. 199–208. IEEE, 2009.

[Knu65] D. E. Knuth. On the translation of languages from left to right. *Information and control*, 1965.

[KRV10] H. Krahn, B. Rumpe, and S. Völkel. MontiCore: a framework for compositional development of domain specific languages. *International journal on software tools for technology transfer*, 12(5):353–372, 2010.

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

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

[LBGD18] J. Lecerf, J. Brant, T. Goubier, and S. Ducasse. A reflexive and automated approach to syntactic pattern matching in code transformations. In *IEEE International Conference on Software Maintenance and Evolution (ICSME'18)*, Madrid, Spain, Sept. 2018.

[Lev09] J. Levine. *Flex & Bison: Text Processing Tools*. " O'Reilly Media, Inc.", 2009.

[LG19] J. Lecerf and T. Goubier. A scheduling approach to generalized LR parsing. *Science of Computer Programming*, 2019.

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

[LRDJ17] Y. Lebras, A. S. C. Rubial, R. Dolbeau, and W. Jalby. ASSIST: An FDO source-to-source transformation tool for HPC applications. In *International Workshop on Parallel Tools for High Performance Computing*, pp. 39–56. Springer, 2017.

[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.

[MB05] B. McCloskey and E. Brewer. Astec: a new approach to refactoring c. *ACM SIGSOFT Software Engineering Notes*, 30:21–30, 2005.

[MDR16a] T. Molderez and C. De Roover. Automated generalization and refinement of code templates with ekeko/x. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pp. 669–672. IEEE, 2016.

[MDR16b] 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.

[MRV03] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *International Conference on Compiler Construction*, pp. 61–76. Springer, 2003.

[NF91] R. Nozohoor-Farshi. Glr parsing for ε-grammers. In *Generalized LR parsing*, pp. 61–75. Springer, 1991.

[NRL17] K. Narasimhan, C. Reichenbach, and J. Lawall. Interactive data representation migration: exploiting program dependence to aid program transformation. In *PEPM 2017 Workshop on Partial Evaluation and Program Manipulation*, 2017.

[OBH14] J. L. Overbey, F. Behrang, and M. Hafiz. A foundation for refactoring c with macros. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 75–85. ACM, 2014.

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

[OJ08] J. L. Overbey and R. E. Johnson. Generating rewritable abstract syntax trees. In *International Conference on Software Language Engineering*, pp. 114–133. Springer, 2008.

[Ove13] J. L. Overbey. Immutable source-mapped abstract syntax tree: a design pattern for refactoring engine apis. In *Proceedings of the 20th Conference on Pattern Languages of Programs*, pp. 7. The Hillside Group, 2013.

[Par13] T. Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[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.

[PLM07] Y. Padioleau, J. L. Lawall, and G. Muller. Smpl: A domain-specific language for specifying collateral evolutions in linux device drivers. *Electronic Notes in Theoretical Computer Science*, 166:47–62, 2007.

[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.

[RBJ97] D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

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

[Rek92] J. G. Rekers. *Parser generation for interactive environments*. PhD thesis, Universiteit van Amsterdam, 1992.

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

[RS70] D. J. Rosenkrantz and R. E. Stearns. Properties of deterministic top-down grammars. *Information and Control*, 1970.

[RSDT16] M. Rizun, G. Santos, S. Ducasse, and C. Teruel. Phorms: Pattern Combinator Library for Pharo. In *International Workshop on Smalltalk Technologies IWST'16*, Prague, Czech Republic, Aug. 2016.

[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.

[SJ06] E. Scott and A. Johnstone. Right nulled GLR parsers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(4):577–618, 2006.

[SJ10] E. Scott and A. Johnstone. GLL parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189, 2010.

[SJE07] E. Scott, A. Johnstone, and R. Economopoulos. BRNGLR: a cubic tomita-style glr parsing algorithm. *Acta informatica*, 44(6):427–461, 2007.

[Spi10] D. Spinellis. Cscout: A refactoring browser for c. *Science of Computer Programming*, 75(4):216–231, 2010.

[TN91] M. Tomita and S.-K. Ng. The generalized LR parsing algorithm. In *Generalized LR parsing*, pp. 1–16. Springer, 1991.

[Tom87] M. Tomita. An efficient augmented-context-free parsing algorithm. *Computational linguistics*, 13(1-2):31–46, 1987.

[Val75] L. G. Valiant. General context-free recognition in less than cubic time. *Journal of computer and system sciences*, 10(2):308–315, 1975.

[VD03] J.-Y. Vion-Dury. Xpath on left and right sides of rules: toward compact xml tree rewriting through node patterns. In *Proceedings of the 2003 ACM symposium on Document engineering*, pp. 19–25. ACM, 2003.

[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.

[Vok06] M. Vokác. An efficient tool for recovering design patterns from c++ code. *Journal of Object Technology*, 5(1):139–157, 2006.

[WGMH13] M. Wang, J. Gibbons, K. Matsuda, and Z. Hu. Refactoring pattern matching. *Science of Computer Programming*, 78(11):2216–2242, 2013.

[WKV14] G. H. Wachsmuth, G. D. Konat, and E. Visser. Language design with the Spoofax language workbench. *IEEE software*, 31(5):35–43, 2014.

[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.

[You67] D. H. Younger. Recognition and parsing of context-free languages in time n3. *Information and control*, 10(2):189–208, 1967.

Résumé

Introduction

Les systèmes informatiques vieillissent par l'augmentation de leur complexité, la réduction de leur maintenance pour satisfaire d'autres priorités ou simplement par la raréfaction de ses experts. La perte de connaissance sur un système appelle à différentes solutions suivant la gravité de la perte et les buts du mainteneur. Réécrire le système en entier en partant de zéro est certainement une possibilité, mais est considérée comme une perte nette de temps et de ressources investis. Les outils d'exploration et de compréhension de code aident à récupérer une partie de la connaissance perdue, mais ils requièrent un important investissement en temps humain. Pour des projets complexes aux problèmes systémiques, le besoin de modernisation est clair. Les opportunités de modernisation peuvent être découvertes par des vérificateurs de bogues, mais au final, les solutions de modernisation de code sont implémentées sous la forme de transformations de code.

Les transformations requises pour une modernisation sont trop complexes pour être effectuées efficacement et sans erreur à la main, les développeurs ont besoin d'outils pour les aider dans cette tâche. Les transformations semi-automatiques basées sur de la reconnaissance de motifs deviennent de plus en plus populaires pour répondre à ce problème. Elles ont prouvé leur efficacité sur des bases de code réelles, au point d'être intégrées dans des Environnements de Développpement Intégré (IDE). Ces transformations semi-automatiques se reposent sur l'écriture de deux motifs : un motif de reconnaissance et un motif de transformation. Grâce à ces deux motifs, les développeurs peuvent réécrire toutes les occurrences du premier motif en ce second motif sur des bases de code importantes.

Parmi les cas d'usage les plus communs des transformations automatiques de code, les refactorings possèdent les outils les plus matures. Les refactorings sont un sous-ensemble des transformations de code, pour lequel la fonctionnalité du programme est conservée après la transformation. Ils sont généralement construits pour améliorer la qualité du code en augmentant sa maintenabilité, sa performance, ou en le rendant plus lisible. Ces transformations sont attractives pour les développeurs car elles garantissent que la sémantique du programme est conservée : la transformation ne peut pas casser un code fonctionnel. Cependant, dans les faits, la vérification formelle des refactorings est un problème difficile. Seuls les plus simples ont des stratégies de vérification en temps raisonnable. En pratique, pour vérifier

des refactorings complexes, les moteurs de refactoring utilisent des pré et post conditions qui vérifient l'état du programme avant et après la transformation. Cette approximation permet une vérification moins coûteuse, mais peut laisser le programme dans des états instables à chaque étape de la transformation.

La migration de code d'un langage à un autre est une autre tâche de modernisation complexe. Toutes les constructions du langage d'origine ne se retrouvent pas dans le langage cible et ces dernières n'ont pas toujours leur équivalent. De plus, la migration peut se heurter à des problèmes de bibliothèques absentes dans le langage cible, mais présentes dans le langage d'origine. Une migration automatique ne peut donc pas être complète et demande le développement additionnel de cette bibliothèque. Pour ces raisons, les migrations de code requièrent des transformations arbitraires flexibles qui ne préservent pas la sémantique du code en question.

Les transformations arbitraires de code sont aussi utilisées pour implémenter la spécialisation de code avant la compilation. La spécialisation de code vise principalement à améliorer les performances d'une application pour laquelle l'adéquation à la plateforme est mal gérée par le compilateur seul. Les spécialisations couvrent de simples transformations de variables à constantes jusqu'aux transformations de l'agencement des données.

Nous concentrons nos efforts sur les deux cas d'usage précédents : la migration et la spécialisation de code. La migration est réalisée par des experts des langages, tandis que la spécialisation est réalisée par des experts des plateformes sans forcément une connaissance poussée des langages. Les deux cas d'usages mettent l'emphase sur des transformations arbitraires et source à source. De ces deux cas d'usages, avec leurs spécificités, nous dérivons diverses exigences pour l'utilisateur et pour l'implémenteur de l'outil.

Exigences

Exigences utilisateur

Multi-langage. Le moteur de transformation de code doit être multi-langage. Les bases de code sont généralement écrites dans de multiples langages, pour différents usages. Ceci doit être réfléchi dans le moteur de transformation. Si l'utilisateur connaît déjà le langage, il ne doit pas avoir besoin de plus de connaissance pour en écrire les motifs et il doit être aisé de changer de langage.

Pas de connaissance préalable de l'IR requise. Le moteur de transformation de code ne doit pas requérir de l'utilisateur l'apprentissage de la représentation intermédiaire du langage. Pour des nouveaux utilisateurs, il est important de ne pas se reposer sur l'IR des langages, ce qui implique que les utilisateurs doivent pouvoir écrire des motifs de reconnaissance et transformation sans connaissance a priori de l'IR. Cela baisse fortement la barrière à l'entrée de l'outil pour pouvoir cibler des experts comme des non experts.

Approche hybride. Le moteur de transformation de code devrait tout de même permettre d'exprimer des motifs avec l'IR si il le souhaite. Même si le moteur utilise des motifs indépendants de l'IR par défaut, toutes les transformations ne sont pas faciles à réaliser de cette manière. Un accès optionnel à l'IR est désirable, particulièrement pour les experts des langages ayant déjà une bonne connaissance de l'IR en question.

Aidé par un DSL. Un Langage Spécifique au Domaine (DSL) devrait être ajouter pour aider aux activités de transformation de code source. Les DSLs sont un moyen connu d'améliorer la productivité et l'accessibilité à des outils pour des domaines spécifiques et la transformation de code ne fait pas exception. Les DSLs proposent des abstractions personalisées pour un certain domaine, qui dans notre cas seraient particulièrement utiles pour cacher des détails d'implémentation de la reconnaissance de motifs et des transformations avec des abstractions de plus haut niveau. Pour aider nouveaux utilisateurs et experts, un DSL de transformation de code source à source est désirable.

Exigences implémenteur

Implémentation simplifiée des back-ends de langage. Le moteur de transformation de code doit proposer une manière efficace d'implémenter les back-ends de langage à moindre coût. Un back-end de langage comprend son analyseur syntaxique ainsi que ses intégrations au moteur de reconnaissance de motifs et de transformation. Ces back-ends de langage sont un effort d'ingénierie non négligeable car écrit à partir de zéro pour chaque nouveau langage.

Intégration hybride transparente. Le moteur de reconnaissance de motifs ne doit pas se reposer sur deux moteurs séparés pour gérer les motifs sans

et avec IR. Pour supporter une approche complètement hybride, il doit exister un grand degré d'intégration entre les deux types de motifs. Cela signifie que l'implémenteur ne doit pas avoir à gérer un type de motif différemment de l'autre.

Syntaxe du DSL interchangeable. Le DSL du moteur de transformation de code doit pouvoir être adapté pour différents styles de syntaxe. Le DSL doit avoir un back-end modulaire et robuste au dessus duquel pourraient être implémentée diverses syntaxes pour divers sous-domaines. A point majeur d'adoption des DSLs est dans la familiarité de sa syntaxe qui dépend elle du sous-domaine en question. Il devrait être possible de construire facilement des nouveaux front-ends et de les lier au moteur de transformation.

Notre approche en quelques mots

Comment peut-on construire un moteur de transformation de code source à source dont les transformations ne requièrent aucune connaissance préalable des constructions du language tout en ayant un coup réduit pour l'implémenteur du moteur ?

Pour répondre à cette question et répondre aux exigences que nous nous sommes fixés dans les paragraphes précédents, nous choisissons de construire un moteur de reconnaissance de motifs qui utilisent la même syntaxe que leur langage à reconnaître par défaut, mais peut aussi faire référence à l'IR au besoin. Le moteur de reconnaissance de motifs est couplé à un moteur de génération d'analyseurs syntaxiques LR généralisés pour générer à la fois l'IR du code source, mais aussi pour automatiser une partie du processus de reconnaissance, cachant ainsi la complexité de l'IR à l'utilisateur. Pour réunir l'analyse syntaxique et la reconnaissance des motifs sous l'ombrelle d'un unique algorithme, nous adaptons le résultat de *parsing as intersection* à la reconnaissance de motifs. La reconnaissance de motifs via l'IR est toujours exprimable dans les motifs comme les approches traditionnelles, mais le moteur se rabat toujours sur une reconnaissance syntaxique par défaut. De cette manière, nous construisons un moteur de réconnaissance et transformation de code agnostique vis à vis de l'IR du langage reconnu.

Contributions

Dans cette thèse, nous avons construit un moteur de transformation de code arbitraire, source à source, dont les transformations sont adaptées aux ex-

perts et non-experts. Nos contributions principales résident dans le moteur de reconnaissance de motifs hybrides explicites-syntaxiques et les analyseurs syntaxiques FGLR générés qui permettent la désambiguïsation par effets de bord. Nos contributions sont implémentées au dessus de SmaCC, le générateur d'analyseurs syntaxiques de Smalltalk.

Moteur de reconnaissance de motifs hybrides explicites-syntaxiques

Dans les outils de transformation, les moteurs de reconnaissance de motifs existent sous deux formes. La reconnaissance explicite demande que l'utilisateur écrive des motifs décrivant explicitement la forme des éléments à reconnaître de l'IR. Apprendre l'IR d'un langage est une lourde tâche, ce qui rend la reconnaissance explicite inadaptée pour les nouveaux utilisateurs. Au contraire, la reconnaissance syntaxique propose d'écrire les motifs directement dans la syntaxe du langage à reconnaître, augmentée de métavariables qui font office de jokers. Les motifs syntaxiques sont plus adaptés aux nouveaux utilisateurs, car ils ne requièrent pas l'apprentissage de l'IR pour être exprimés. Cependant, certains motifs sont plus faciles à exprimer explicitement, il est donc désirable d'avoir un moteur de reconnaissance gérant des motifs hybrides explicites-syntaxiques.

Implémenter la partie reconnaissance syntaxique d'un tel moteur a besoin d'une étape additionnelle par rapport à la partie explicite : une phase d'inférence de type qui se repose pour beaucoup sur des back-ends de langage écrits à la main. Pour faciliter l'implémentation de l'inférence de type, nous réduisons les back-ends de langage à leurs analyseurs syntaxiques, eux-même générés par un générateur d'analyseurs syntaxiques. Grâce à une adapation de parsing-as-intersection pour la reconnaissance de motifs, nous pouvons nous servir du même algorithme pour analyser le code source du program à transformer et pour effectuer l'inférence de type du motif hybride. Nous avons modifié le moteur de génération d'analyseurs syntaxiques GLR de SmaCC pour générer des analyseurs compatibles avec l'inférence de type, à partir d'une grammaire du langage au format BNF ou EBNF. La grammaire en question ne nécessite que l'addition d'une unique ligne supplémentaire qui définit comment reconnaître une métavariable, et le générateur se charge de générer le back-end du langage pour le moteur de reconnaissance.

Les générateurs d'analyseurs syntaxiques effectuent des modifications à l'arbre de syntaxe pour créer un arbre de syntaxe abstrait plus contact et raisonnable. Un exemple d'une telle modification est la suppression des noeuds intermédiaires représentant la récursion et l'ajout de listes de noeuds

comme représentation alternative plus compact. Reconnaître cet idiome de listes est complexe car le lien direct entre les noeuds de l'arbre de syntaxe et ceux de l'arbre de syntaxe abstrait est brisé. Nous proposons également une méthode pour exprimer et reconnaître ces idiomes de listes dans les arbres de syntaxe abstraits avec des motifs hybrides explicites-syntaxiques.

GLR ordonnancé

Nos analyseurs GLR modifiés reposent sur l'exploration de tous les types possibles pour une métavariable, à l'instar de la manière dont ces analyseurs explorent les alternatives d'analyse possibles lors d'un conflit. L'exploration des alternatives est rélisée par duplication de l'analyseur lors qu'il rencontre à la fois des conflits et des métavariables, cette double duplication peut être stressante en terme de temps d'exécution et de mémoire. D'autres méthodes de désambiguïsation serait bénéfiques pour le moteur pour endiguer cette potentielle explosion de duplication. Cependant, la principale technique pour désambiguïser *durant* l'analyse est l'exploitation d'effets de bord dans l'analyseur, non disponible dans les analyseurs GLR. Les effets de bord permettent l'implémentation de code de désambiguïsation pour faire un choix entre les alternatives pendant l'analyse, ou plus précisément, pendant les actions sémantiques. Les actions sémantiques doivent être exécutée en ligne et dans un ordre bottom-up pour permettre les effets de bord, c'est-à-dire dans le même ordre que leurs réductions associées.

Nous proposons un analyseur GLR basé sur un ordonnanceur pour s'assurer de l'exéction des alternatives d'analyse dans l'ordre bottom-up. Les alternatives d'analyse sont isolées dans des structures de données indépendantes qui contiennent non seulement les données internes à l'analyseur pour l'alternative en question, mais aussi toute structure de désambiguïsation additionnelle que l'implémenteur aurait choisi d'ajouter. De même que pour les implémentations list-of-stacks de GLR, Fibered-GLR (FGLR) sépare les piles d'états pour chaque alternative et les fusionne quand elles redeviennent identiques. Grâce à son ordre bottom-up, FGLR fusionne plus souvent qu'une implémentation standard de GLR list-of-stacks. Garder les alternatives indépendantes entre elles est vital à l'implémentation d'actions sémantiques en ligne. La technique la plus efficace de GLR utilise un unique graphe, appellé GSS, pour représenter toutes les alternatives d'analyse au lieu de les séparer. Cette technique permet de réduire la complexité de l'algorithm en partageant un nombre maximum d'états, mais elle ne peut plus exécuter les actions sémantiques en ligne et donc pas désambiguïser par effets de bord.

Réponse aux exigences et perspectives

Exigences utilisateur

Multi-langage. Le moteur de transformation est construit pour supporter plusieurs langages à travers leurs grammaires. Les back-ends de langage sont réduits à leur analyseur syntaxique généré depuis la grammaire du langage avec un environnement d'exécution modifié pour géré les motifs hybrides explicites-syntaxiques. Ces motifs hybrides valident la deuxième partie de l'exigence : permettre l'écriture aisée de motifs pour un langage déjà connu de l'utilisateur. Une combinaison de métavariables et motifs syntaxiques pour les constructions inconnues de l'IR et de motifs basés sur l'IR pour celles qui sont connues répond à l'exigence.

Pas de connaissance préalable de l'IR requise. Pour ce qui est de ne pas avoir besoin d'une connaissance préalable de l'IR pour écrire des motifs et faire la reconnaissance, la validation est partielle. Les motifs peuvent être purement syntaxiques, sans types de l'IR, et pourront reconnaître n'importe quel type de l'IR tout comme un motif explicite. Cependant, l'implémentation actuelle des types complexes facilitant la reconnaissance des arbres transformés repose sur des fondements explicites.

Approche hybride. Notre moteur de transformation est en effet hybride, il propose des motifs explicites-syntaxiques. Par défaut, le motif se comporte de manière syntaxique et c'est seulement lorsque les métavariables sont individuellement typées que l'on a accès à la reconnaissance explicite. L'implémentation actuelle n'utilise pas encore les types de motifs pour réduire le nombre de possibilités explorées. Une amélioration immédiate consisterait à retirer les alternatives d'analyse pour un motif typé au début de la reconnaissance.

Aidé par un DSL. Malheureusement, le moteur de transformation n'offre pas encore de DSL de transformation de code. Certaines fonctionnalités du moteur ont déjà été ajoutées pour aller dans ce sens, comme le raffinage de motifs, procédé qui permet d'appliquer des motifs successifs pour récupérer des éléments plus intéressants. Ce processus itératif permet d'écrire plusieurs motifs simples plutôt qu'un motif unique très complexe.

Exigences implémenteur

Implémentation simplifiée des back-ends de langage. Pour faciliter la création des back-ends de langage, nous avons choisi la voie de l'automatisation via la génération du back-end depuis une grammaire, plutôt que de nous appuyer sur des analyseurs sur étagère. Pouvoir utiliser le même algorithme pour l'inférence de types et l'analyse syntaxique est le principal critère qui nous a fait choisir cette route. L'implémentation du back-end du langage est réduite à trouver une grammaire du langage dans le meilleur des cas et la créer dans le pire. Les grammaires de langages de programmation généralistes sont des objets complexes, mais contiennent plusieurs ordres de magnitude moins de code. Nous pensons qu'il est plus important d'investir dans les outils pour aider au développement des grammaires plutôt que d'investir dans l'écriture de back-ends de langages.

Intégration hybride transparente. Le moteur de reconnaissance de motifs est singulier, à la fois explicite et syntaxique. Un unique motif peut contenir à la fois des parties explicites et syntaxiques. L'exigence d'une intégration hybride transparente est réalisée : il n'y a ni besoin de séparer les deux types de moteurs ou les deux types de motifs.

Syntaxe du DSL interchangeable. L'exigence du DSL n'étant pas validée, celle-ci ne l'est pas non plus. Aucune syntaxe ne convient à tous les usages, la syntaxe est très dépendante de son domaine et donc public. Le moteur est construit pour que les motifs soient aisés à écrire pour les experts et non experts, des syntaxes interchangeables doivent représenter cela aussi.

Perspectives

Reconnaissance syntaxique de types complexes. Reconnaître les arbres de syntaxe abstraits, après transformation de la récursion en liste, est actuellement fait sous la forme de types AND explicites. Ces types représentent une séquence de types explicites. Pour de nouveaux utilisateurs, il est désirable de pouvoir reconnaître n'importe quelle sous-structure de l'AST via des motifs syntaxiques. Les séquences de types explicites vont à l'encontre de cette philosophie et il serait intéressant de trouver une solution purement syntaxique pour reconnaître ses structures.

Outils interactifs et généralisation de motifs. Pour faciliter l'accès aux motifs hybrides, il serait appréciable d'avoir des outils de généralisation de motifs. Dans un premier temps, les motifs pourraient être généralisés à

partir d'exemples de code similaire à reconnaître. Le moteur de généralisation remplacerait certaines parties de syntaxe concrète de l'exemple en métavariables pour forger un motif syntaxique. Les métavariables pourraient être typées explicitement par l'utilisateur ou par le moteur. Dans un second temps, de manière similaire, le moteur pourrait muter un motif en un autre motif en remplaçant de la syntaxe concrète par une métavariable, en aggrégeant une métavariable et des éléments de syntaxes adjacent en une nouvelle métavariable, ou par tout autre opérateur de mutation. Le moteur de généralisation devrait être interactif pour que l'utilisateur assimile l'impact de sa généralisation sur les occurrences obtenues.

DSL de transformation de code. Le moteur de transformation de code bénéficierait grandement de l'ajout d'un DSL pour simplifier l'écriture des transformations. *Parsing as intersection* aide dans l'écriture des motifs via des motifs syntaxiques, mais ne propose rien pour aider à l'après-reconnaissance. Un langage avec des motifs, des transformations, des métavariables, des contextes et des occurrences en tant qu'objets de première classe permettrait la manipulation de ses concepts plus aisément que dans le langage d'écriture du moteur de transformation.

Stratégies de transformation. À ce DSL peut venir s'ajouter des stratégies de transformation. Les stratégies définissent l'ordre d'application et donc les priorités des transformations. Elles sont particulièrement utiles pour procéder à des réécritures de base de code avec de très nombreuses transformations ou des transformations qui se chevauchent. Plusieurs stratégies prédéfinies pourraient être proposées à l'utilisateur, mais il est également important de les incorporer dans le DSL pour que des stratégies personnalisées et adaptées puissent être créées par les utilisateurs.

Abstracts

Abstract (English). Code transformations are needed in various cases: refactorings, migrations, code specialization, and so on. Code transformation engines work by finding a pattern in the source code and rewriting its occurrences according to the transformation. The transformation either rewrites the occurrences, elements of the intermediate representation (IR) of the language, into new elements or directly rewrites the source code. In this work, we focused on source rewriting since it offers more flexibility through arbitrary transformations, especially for migrations and specializations.

Matching patterns come in two different flavours, explicit and syntactic. The former requires the user to know the IR of the language, a heavy knowledge burden. The latter only relies on the syntax of the matched language and not its IR, but requires significantly more work to implement the language back-ends. Language experts tend to know the IR and the syntax of a language, while other users know only the syntax.

We propose a pattern matching engine offering a hybrid pattern representation: both explicit and syntactic matching are available in the same pattern. The engine always defaults to syntactic as it is the lowest barrier to entry for patterns. To counterbalance the implementation cost of language back-ends for syntactic pattern matching, we take a generative approach. We combine the hybrid pattern matching engine with a parser generator. The parser generator generates generalized LR (GLR) parsers capable of not only parsing the source but also the hybrid pattern. The back-end implementer only needs to add one line to the grammar of the language to activate the pattern matching engine.

This approach to pattern matching requires GLR parsers capable of forking and keeping track of each individual fork. These GLR implementations suffer the more forking is done to handle ambiguities and patterns require even more forking. To prevent an explosion, our Fibered-GLR parsers merge more often and allow for classic disambiguation during the parse through side-effects.

Résumé (Français). Les transformations automatiques de code apparaissent dans diverses situations, les refactorings, les migrations inter-langages ou encore la spécialisation de code. Les moteurs supportant ces transformations cherchent dans le code source les occurrences de motifs spécifiés par l'utilisateur, puis les réécrivent grâce à une transformation. Cette transformation peut soit modifier les occurrences elles-mêmes, des éléments de la représentation intermédiaire (IR) du langage, en nouveaux éléments ou réécrire leur code source. Nous nous concentrons sur la réécriture de code source qui offre une meilleure flexibilité grâce à des transformations arbitraires particulièrement utiles à la migration et à la spécialisation de code.

Les motifs sont divisés en deux catégories : les motifs explicites et syntaxiques. Les premiers demandent que l'utilisateur connaisse l'IR du langage, un effort d'apprentissage non négligeable. Les seconds demandent seulement de connaître la syntaxe du langage et non son IR, mais requièrent un effort d'implémentation supplémentaire pour les back-ends de langage du moteur. Tandis que les experts en langage connaissent l'IR et la syntaxe du langage, les autres utilisateurs connaissent seulement la syntaxe.

Nous proposons un moteur de reconnaissance de motifs offrant une représentation hybride des motifs : les motifs peuvent être à la fois explicites et syntaxiques. Par défaut, le moteur se rabat sur un fonctionnement syntaxique, car la barrière à l'entrée est plus basse. Pour palier au coup d'implémentation des back-ends de langage pour la reconnaissance syntaxique, nous prenons une approche générative. Le moteur de reconnaissance hybride est couplé avec un moteur de génération d'analyseurs syntaxiques. Ce dernier génère des analyseurs syntaxiques LR généralisés (GLR) capables d'analyser non seulement le code source à réécrire, mais également le motif à reconnaître. L'implémenteur du back-end de langage n'a alors qu'à ajouter une ligne à la grammaire pour avoir accès au moteur de reconnaissance de motifs pour ce langage.

L'approche est basée sur des analyseurs syntaxiques GLR pouvant se dupliquer et traquant ses sous-analyseurs. Ces implémentations particulières de GLR ne passent pas à l'échelle quand trop de duplications sont nécessaires pour gérer les ambiguïtés et notre approche ajoute de la duplication. Pour éviter une explosion du temps d'exécution, nos analyseurs syntaxiques FGLR fusionnent plus régulièrement et permettent une désambiguïsation à la volée pendant l'analyse via des effets de bord.