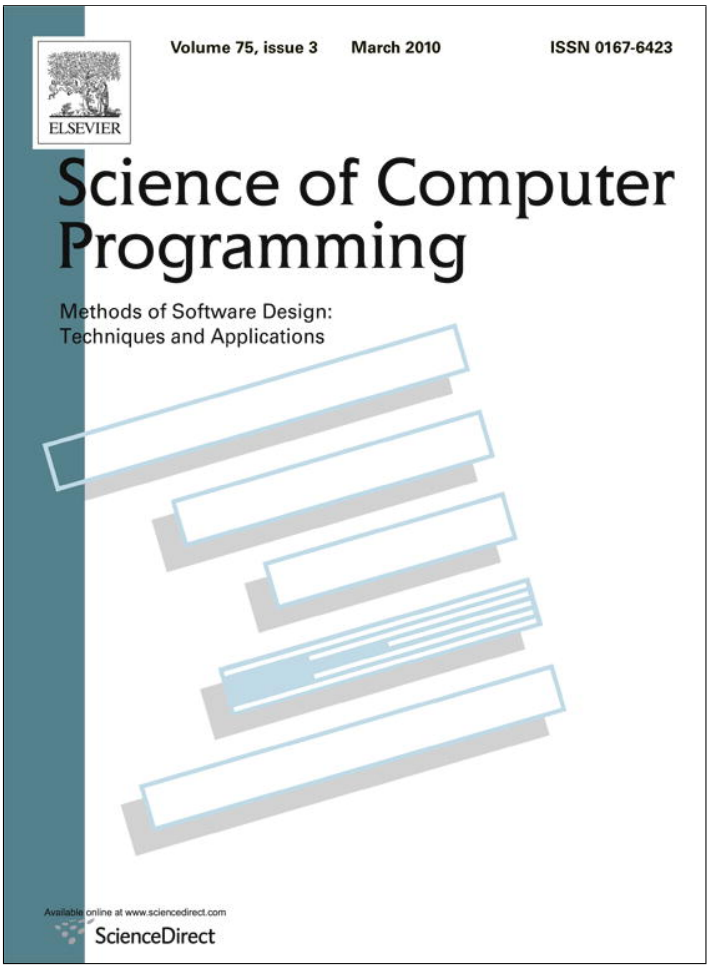

Science of Computer Programming 75 (2010) 85–105

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

# Theory and practice of unparsed patterns for metacompilation

Christian Rinderknecht [a,*], Nic Volanschi

[a] Konkuk University, 143-701 Seoul Gwanjin-gu Hwayang-dong, South Korea

ARTICLE INFO

*Article history:*
Received 6 June 2008
Received in revised form 10 June 2009
Accepted 23 September 2009
Available online 1 October 2009

*Keywords:*
Pattern matching
Tree pattern
Code checking
Metacompilation
Formal methods

ABSTRACT

Several software development tools support the matching of concrete syntax user-supplied patterns against the application source code, allowing the detection of invalid, risky, inefficient or forbidden constructs. When applied to compilers, this approach is called *metacompilation*. These patterns are traditionally parsed into tree patterns, i.e., fragments of abstract-syntax trees with metavariables, which are then matched against the abstract-syntax tree corresponding to the parsing of the source code. Parsing the patterns requires extending the grammar of the application programming language with metavariables, which can be difficult, especially in the case of legacy tools. Instead, we propose a novel matching algorithm which is independent of the programming language because the patterns are not parsed and, as such, are called *unparsed patterns*. It is as efficient as the classic pattern matching while being easier to implement. By giving up the possibility of static checks that parsed patterns usually enable, it can be integrated within any existing utility based on abstract-syntax trees at a low cost. We present an in-depth coverage of the practical and theoretical aspects of this new technique by describing a working minimal patch for the GNU C compiler, together with a small standalone prototype punned Matchbox, and by lying out a complete formalisation, including mathematical proofs of key algorithmic properties, like correctness and equivalence to the classic matching.



## 1. Introduction

Pattern matching of source code is very useful for analysing and transforming programs, as in compilers, interpreters, tools for legacy program understanding, code inspectors, refactoring tools, model checkers, code translators etc. Source code matching is especially useful for building extensible versions of these tools with user-defined behaviour [1–4]. As the problem of tree matching has been extensively studied, the problem of source code matching has usually been reduced to tree matching, following two different ways.

**Tree Patterns.** In the first approach, code patterns are written as trees, using a domain-specific notation to describe an abstract syntax tree (AST). This approach based on tree patterns has been used for a long time, either by using pattern matching support available in the implementation language, for instance, for tools written in ML, or otherwise by explicitly implementing a tree pattern matching mechanism, for instance in inspection tools such as tawk [5] or Scruple [6] or in model checking tools such as MOPS [3]. More recently, some extensible code inspectors such as PMD[1] represent ASTs in XML (which can be considered, in general, as a standardised formal notation for trees). This allows us to write tree patterns in standardised languages such as XPath (and the languages embedding it, like XQuery and XSLT), and thus reuse the existing tree pattern matchers. The main advantage of expressing patterns as trees is that the implementation of pattern matching

* Corresponding author.
*E-mail addresses:* rinderkn@konkuk.ac.kr (C. Rinderknecht), nic.volanschi@free.fr (N. Volanschi).
*URLs:* http://konkuk.ac.kr/~rinderkn (C. Rinderknecht), http://nic.volanschi.free.fr (N. Volanschi).

[1] http://pmd.sourceforge.net/.

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.09.011

```
for(%x=0; %x<%n; ++%x) %y[%x]=0;
```

**Fig. 1.** A concrete syntax pattern.

is simple, because any appropriate tree matching algorithm can be directly used on this representation. However, an important shortcoming of this approach is that programmers writing patterns should be aware of the internal AST representation of programs, and also of a specific notation for it. As a motivating example, let us consider a very simple user-defined inspection rule over C programs that searches for code fragments resetting all the elements in an array to zero, such as `for(i=0; i<100; ++i) a[i]=0;`. When such initialisation code is found, the code reviewer may suggest that the same operation would be implemented more efficiently using the `bzero()` standard library function. Alternatively, the same rule might be predefined in a compiler in order to recognise such initialisations and automatically implement them more efficiently using the `bzero()` function. Depending on the AST representation of the C program in a particular tool, the tree pattern corresponding to the example above would typically be expressed as follows:

```
for_stmt(assign_expr(X,N), less_expr(X,M), preincr_expr(X),
         expr_stmt(assign_expr(array_expr(Y,X), int_literal(0))))
```

The name of the array, its bound and its index have been abstracted as variables of the pattern, called *metavariables*. As can be seen in the above tree pattern, the programmer writing the inspection rule must be aware of both the AST structure (for instance, that an assignment in C is an expression embedded in a statement) and of a specific notation for it (for instance, that the assignment operator is called `assign_expr`, that the `for_stmt` operator takes four arguments in a given order, etc.). Writing the same pattern in a language such as XPath does not solve any of these issues, and the pattern becomes even more verbose.

**Concrete Syntax Patterns.** In the second approach to source code matching, code patterns are expressed using the native syntax (i.e., the concrete syntax) of the subject programming language, augmented with metavariables. Then patterns are parsed to trees before being matched with the program AST. In this approach, the pattern to be searched can be written much more naturally and concisely as shown in Fig. 1 (where metavariables are escaped by the special character `%`).
This second approach has been used, for instance, in several extensible model checkers [7,1,2,4], as well as extensible tools for legacy program understanding and transformation [8,9]. The main advantage of concrete syntax patterns is that they are easy to write and read back by any programmer, without knowledge of the AST representation. However, as far as the implementation is concerned, parsing the pattern requires a modified parser of the programming language defining the application, extended to

- allow metavariables, which, by definition, are variables that are never found in the source code;
- parse patterns that represent arbitrary program *fragments* (statements, expressions, declarations), whereas the grammar specifies fixed syntactical subsets.

Extending the parser of a programming language in this way is a challenging task, even if some frameworks automate the addition of the extra grammar productions [10], because it entails usually many supplementary *ad hoc* choices in the parser strategy, often called *conflict resolutions*. These conflicts may correspond to real ambiguities in the extended language (for example, in C, the pattern `f(%x);` may represent either a function call or a function declaration with an implicit return type of `int`), or reveal a limited knowledge of the lexical context in the parsing algorithm. Indeed, existing parsers commonly use limited lexical lookahead, as those analysing LALR(1) or LL(1) grammars. In LALR(1)-based parsers, the conflicts present themselves as dilemmas (reduce or reduce, shift or reduce) but choosing one option rather than the other cannot always preserve the syntactic coverage (i.e., some valid programs may be rejected), hence rewriting the underlying grammar becomes a necessity. Systematically adding pattern productions to an existing LALR(1) grammar usually creates a large number of such conflicts, partly because patterns may represent any program fragment instead of whole programs (like the `f(%x);` example above), partly because a pattern variable in a specific context may stand for different sub-constructs (for instance, in the C or Java pattern `f(%x)` variable `x` may stand either for a unique argument or for a list of arguments). This latter kind of conflicts arises for every production in the original grammar allowing a non-terminal to be derived into a single other non-terminal (in the example, an argument list non-terminal that may reduce to a single argument non-terminal). In our extensive experience with adding patterns to LALR(1) grammars for re-engineering projects, this single task takes at least several man-weeks to well-trained developers when applied on real-size grammars, especially those of legacy languages designed before grammar models have gained a wide adoption in the industry. Furthermore, rewriting the grammar to cope with all these pattern-related conflicts leads to special pattern syntaxes (such as `#stmt f(#list %x);` in the previous example) that make the patterns look less similar to native code. This is the reason why most of the existing tools following this approach allow only restricted forms of concrete syntax patterns, described by a limited pattern grammar (for example, matching only assignments and function calls [11]).

In theory, GLR (for Generalised LR) parsers [12] can deal with both kinds of conflicts, because they compute all possible parses: conflicts due to a limited lookahead are solved later during the parsing and conflicts due to the ambiguity of the grammar itself result in several possible ASTs. However, mainstream parsers such as Bison[2] do not implement the proper

2 http://www.gnu.org/software/bison/.

GLR algorithm, but rather implement under this name an algorithm that may use exponential time and space when parsing with an ambiguous grammar. Alternatively, the proper $O(n^3)$ time GLR parsing algorithm [12] was implemented in tools such as SDF [13], used by ASF+SDF [14] and Stratego/XT [15] to implement concrete syntax rewriting rules, or very recently BRNGLR [16]. Also, a hybrid GLR/LALR parser called Elkhound [17] has been released and its running time is close to that of a standard LALR(1) parser on all the portions of a grammar that are LALR(1). Other parsing algorithms covering all context-free grammars, such as Earley's [18], may perform well on ambiguous grammars.

However, in the case of legacy parsers, that are specifically addressed by our approach, there is a very important practical issue: porting an existing parser to a different technology, like GLR or Earley, may prove quite difficult and has profound impacts on the test of the tool, which is unaffordable for most projects. An aspect that may further complicate the problem is that in many such projects, it is required not only to implement a new parser, but to reproduce exactly the same behaviour as the legacy parser, including perhaps some of its idiosyncrasies. Indeed, when a significant customer base with potentially huge legacy assets relies on an existing compiler, it may be economically unacceptable to require them migrating to a new, perhaps better, but slightly incompatible parser.

**Our Proposal.** It is fair to say that there is no simple solution today for adding concrete syntax pattern matching to existing parsers without either profoundly restructuring the parser, rewriting it in another framework, severely restricting the patterns, or compromising performance. As a consequence, concrete patterns are rarely used in existing parser-based tools such as mainstream compilers. The lack of concrete patterns is particularly limiting the implementation of convenient user extensions in tools such as extensible compilers, code inspectors, model checkers, etc. To solve this problem in a pragmatic way, we propose a pattern matching technique based on *unparsed patterns*, which makes an efficient use of unrestricted concrete syntax patterns, while requiring at the same time no extension of the parser for the subject programming language. This method is applicable to legacy parsers, based on any parsing technology, without porting them to a different framework. In a previous paper [19], one of us showed some concrete applications of unparsed patterns within a checking compiler called myGCC. However, no details were published about their implementation, nor about their theoretical foundations; it introduced the idea of unparsed patterns and briefly mentioned that they work by unparsing the AST, rather than by parsing the pattern. The present article aims at a complete exposition of unparsed patterns.

The rest of this paper is organised as follows. The next section presents the formal model which we use later to define our algorithms and related proofs. The reader familiar with functional languages or logic programming can skip this section upon first reading. Then, Section 3 describes a matching algorithm for unparsed patterns by means of backtracking. It allows the reader to become familiar with the issues at hand, without delving too much on the details. The following Section 4 presents our main contribution, that is, a deterministic, linear-time matching algorithm, called *ES*(1), together with some formal proofs. After, Section 5 presents our implementation of our matching algorithm and discusses its strengths and limitations. The penultimate Section 6 presents different related works and compares them to ours.

## 2. Formal model

In order to reduce the gap between the program and its specification, we restrict ourselves to basic mathematical constructs that can be mapped to data structures and functions of the implementation programming language. Beyond the obvious integers, variables and tuples, the remaining terms are *lists* and *constructors*. We shall follow the Prolog notation for lists and write [ ] for the empty list and $[e \mid l]$ for the non-empty list whose head is (denoted by) $e$ and tail is (denoted by) $l$. Some shorthands prove useful, e.g., $[a, b, c]$ instead of $[a \mid [b \mid [c \mid [\,]]]]$, or $[a, b \mid l]$ instead of $[a \mid [b \mid l]]$. Constructors are names associated with a non-empty list of terms, e.g., $c([1, d([n])])$ is a term constructed with constructors $c$ and $d$. As a consequence, the number of terms is variable and positive, e.g., we can have $c([1, d([n])])$ and $c([e([1])])$, but not $c([\,])$.

**Parsing and Unparsing.** The converse of parsing, called *unparsing*, is defined on the parse tree, so it is dependent only on the grammar. In order to cope with all programming languages, we make no assumptions on the nature of the lexemes. We denote a lexeme by $l$ and their set is noted $\mathcal{L}$. Each non-terminal symbol of the grammar corresponds to exactly one constructor of the parse tree, with a variable number of arguments—the parse tree is an unranked tree. For example, the productions `Exp ::= Integer | - Exp | Exp + Exp` allow the constructor of `Exp` to have one, two or three arguments. Let $\mathcal{C}$ be the set of constructors of parse trees. In general, we write $c(f)$, where $f$ is the list of the arguments of constructor $c \in \mathcal{C}$. By definition, the leaves of the parse trees are lexemes, so let $\mathcal{H}_0 = \mathcal{L}$ be the set of parse trees of height 0 and let us define recursively the set of trees of height $n+1$ by the equation $\mathcal{H}_{n+1} = \mathcal{H}_n \cup \{c([h_1, h_2, \ldots, h_p]) \mid c \in \mathcal{C}, h_i \in \mathcal{H}_n, p > 0\}$, for all $n \geqslant 0$. The cumulative infinite series $\mathcal{H}_0 \subseteq \mathcal{H}_1 \subseteq \ldots$ has a smallest upper bound $\mathcal{H}$, which is the set of all the parse trees. Let us note $\mathcal{T}$ the set of all trees which are not reduced to one node, i.e., $\mathcal{T} = \mathcal{H} \setminus \mathcal{L}$. We note $l$ the lexemes ($l \in \mathcal{L}$), $t$ the trees not reduced to one node ($t \in \mathcal{T}$) and $h$ the general trees ($h \in \mathcal{H}$). The *parse forest* of a given input is the list of its parse trees. The set of all the forests is inductively defined as the smallest set $\mathcal{F}$ such that

- $[\,] \in \mathcal{F}$,
- if $h \in \mathcal{H}$ and $f \in \mathcal{F}$ then $[h \mid f] \in \mathcal{F}$.

The expression $f_1 \cdot f_2$ denotes the *catenation* of the forests $f_1$ and $f_2$.

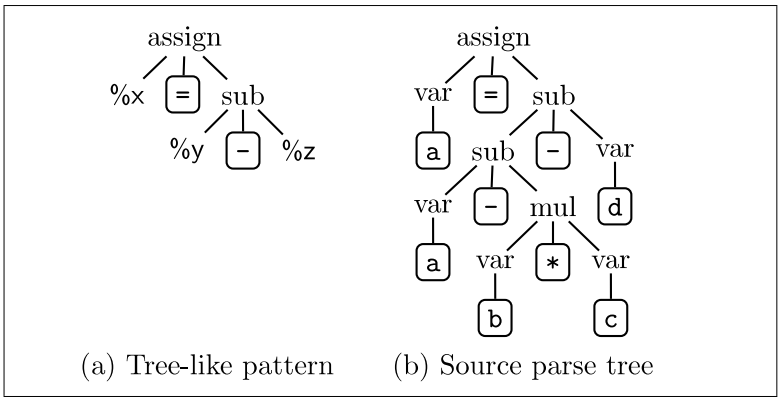


**Fig. 2.** Tree pattern matching `a = a - b*c - d`.

**Unparsed Patterns.** Unparsed patterns are series of lexemes and *metalexemes* (which cannot be found in the programming language) whose purpose is to control the matching process. All unparsed patterns can at least contain *metavariables* whose purpose is to be bound to a subtree of the parse tree, but not to a leaf. For example, consider again Fig. 1: in case of a successful match, the lexemes `for` and ++ match leaves of the parse tree, while the metavariables `%x` and `%n` are bound to some subtree of the parse tree. Let $\mathcal{V}$ be an infinite denumerable set of variables. A metavariable is formally an element of the set $\{\mathsf{meta}(x) \mid \mathsf{meta} \notin \mathcal{C}, x \in \mathcal{V}\}$, and no element of this set in included in $\mathcal{L}$. That means that a metavariable is a variable which is not a node of any parse tree. The concrete syntax of $\mathsf{meta}(x)$ is the concrete variable `x` escaped by `%`, i.e., `%x`. An unparsed pattern $\overline{p} \in \overline{\mathcal{P}}$ is a list of lexemes and metalexemes.

**Substitutions.** A *substitution* $\sigma$ is a mapping whose domain $dom(\sigma)$ is a finite subset of (meta)variables $\mathcal{V}$ and the co-domain is a finite subset of parse trees $\mathcal{H}$. We note $\sigma_\varnothing$ any substitution with an empty domain. A *binding* $x \mapsto t$ is the pair $(x, t)$, where $x \in \mathcal{V}$ and $t \in \mathcal{H}$. Conceptually, a substitution can be thought of as a table which maps variables to parse trees. The substitution $\sigma \oplus x \mapsto t$ is the *update* of the substitution $\sigma$ by the binding $x \mapsto t$:

$$(\sigma \oplus x \mapsto t)(y) \triangleq \begin{cases} t & \text{if } x = y \\ \sigma(y) & \text{otherwise} \end{cases} \tag{1}$$

Let us extend updates to substitutions as follows:

$$(\sigma_1 \oplus \sigma_2)(y) \triangleq \begin{cases} \sigma_2(y) & \text{if } y \in dom(\sigma_2) \\ \sigma_1(y) & \text{otherwise} \end{cases} \tag{2}$$

Let us define the inclusion between substitutions as

$$\sigma_1 \subseteq \sigma_2 \iff \forall x \in dom(\sigma_1).(\sigma_1(x) = \sigma_2(x)) \tag{3}$$

**Inference Systems.** An inference system is a finite set of *inference rules*, which are logical implications of the form $P_1 \wedge P_2 \wedge \cdots \wedge P_n \Rightarrow C$, or simply

$$\frac{P_1 \quad P_2 \quad \ldots \quad P_n}{C}$$

The propositions $P_i$ are called *premises* and $C$ is the *conclusion*. When premises are lacking, then $C$ is called an *axiom* and is simply written $C$. Free variables are implicitly universally quantified at the outermost level. A *proof tree*, or a *derivation*, is a finite tree whose nodes are (instances of) conclusions of an inference system and their children are the premises of the same rule. The leaves of the tree are axioms. The conclusion of the proof, is then the root of the tree, which is traditionally set at the bottom of the page. What makes this formalism interesting is that it allows two kinds of interpretation: logical and computational. The logical reading has just been sketched: if all the premises hold, then the conclusion holds. This top-down reading qualifies as *deductive*. The computational interpretation is bottom-up instead: in order to compute the conclusion, the premises must be computed first (their order is unspecified). This reading is said *inductive*. A single formalism with this double interpretation is powerful because a relationship logically defined by means of inference rules can then be considered as an algorithm as well. Conversely, the logical aspect enables proving theorems about the algorithm.

## 3. A backtracking algorithm

Before presenting a precise algorithm for unparsed-pattern matching, let us discuss informally an example. Consider the problem of matching the pattern `%x = %y - %z` against the C expression `a = a - b * c - d`. Pattern tree matching would first parse the expression into the parse tree in Fig. 2(b), then parse the pattern using an extended parser

into the parse tree in Fig. 2(a), and then match the latter against the former. As a result, all the metavariables are correctly bound with respect to the grammar in the substitution {x ↦ "a", y ↦ "a-b*c", z ↦ "d"}. The key idea of unparsed patterns is to avoid parsing the pattern by going the other way around, i.e., by unparsing the source parse tree and comparing the result with a textual pattern. However, if the parse tree is simply unparsed into a string, matching would fall back to the case of matching between two strings, which is very imprecise, because it would yield both the substitution {x ↦ "a", y ↦ "a-b*c", z ↦ "d"}, which is correct, and {x ↦ "a", y ↦ "a", z ↦ "b*c-d"}, which is incorrect, since the subtraction operator is left-associative. Moreover, metavariables are bound to strings (i.e., concrete syntax), rather than being bound to subtrees of the parse tree. This is not suitable when using pattern matching to process the matched subtrees, which is a common case within parsing-based tools.

The technical issue is that the whole parse tree is fully unparsed (i.e., destructured) at once, dropping the references to all the subtrees. In order to avoid that, the parse tree should be unparsed level by level (in a breadth-first traversal) and the unparsed pattern (which is a list of lexemes and metavariables here) should either be partially matched against the current unparsed forest or the latter should be further unparsed. These two alternatives are sometimes possible for a given parse forest and unparsed pattern. In the first option, i.e., partial matching, can be tried first and if it leads to a failure, the second option, i.e., unparsing, is tried instead. If both options lead to a failure, then the whole matching is deemed a failure. This technique is called *backtracking* and does not lead to a linear-time algorithm in the worst case (in the size of the pattern plus the size of the source tree). Also, the order in which matching or unparsing are tried is not significant as there is no way to guess which would me more likely to be successful *a priori*. More precisely, firstly, the source parse tree is pushed on an empty analysis stack. This stack is, in general, a parse forest. We shall speak of the "left of the forest" instead of the "top of the stack". Secondly, the textual pattern, here the string `%x = %y - %z`, is transformed into a list of lexemes and metavariables. Thirdly, given an initial empty substitution $\sigma$, the algorithm non-deterministically chooses one of the two following actions, and backtrack in case of failure.

(1) **Matching.** The first element $e$ of the pattern is matched against the leftmost tree $h$ of the forest. This can be achieved in two different situations:
   (a) **Elimination.** If $h$ and $e$ are the same lexeme, then the remaining pattern is matched against the remaining forest, with the same substitution $\sigma$.
   (b) **Binding.** If $h$ is not a lexeme (i.e., it is not a leaf) and $e$ is the metavariable $x$, which is either already bound to a subtree equal to $h$ (Unparsed patterns are not linear, i.e., a metavariable can occur more than once.) or unbound in $\sigma$, then the remaining pattern is matched against the remaining forest, with $\sigma$ updated with $x$ bound to $h$.
(2) **Unparsing.** If the forest starts with a tree $t$, unparsing consists in replacing $t$ by the forest of its direct subtrees (in other words, the root of $t$ is cut out) and trying again with the same pattern and the same substitution.

The algorithm always stops because either the pattern length or the forest size strictly decreases at each step. It fails if and only if the final pattern is not empty. In case of success, the final substitution is the result (it contains all the bindings of the metavariables to subtrees of the source parse tree).

### 3.1. Pattern matching

Unparsed patterns are noted $\overline{p}$ and the set of unparsed patterns is inductively defined as the smallest set $\overline{\mathcal{P}}$ such that

- $[\,] \in \overline{\mathcal{P}}$;
- if $l \in \mathcal{L}$ and $\overline{p} \in \overline{\mathcal{P}}$, then $[l \mid \overline{p}] \in \overline{\mathcal{P}}$;
- if $x \in \mathcal{V}$ and $\overline{p} \in \overline{\mathcal{P}}$, then $[\mathsf{meta}(x) \mid \overline{p}] \in \overline{\mathcal{P}}$.

Let us extend the substitutions defined in Section 2, in order to cope with unparsed patterns, not just metavariables. The effect of a substitution on a pattern will be to replace every occurrence in the pattern of the metavariables in its domain by the corresponding parse trees. The substitutions computed by any of our matching algorithms are total, i.e., they replace *all* the metavariables of the pattern. It is handy to distinguish the forests which contain no metavariables by calling them *closed forests* or *closed patterns*, and their contents *closed trees*. In order to distinguish a substitution applied to a metavariable $x$ from a substitution on an unparsed pattern $\overline{p}$, we shall note $\sigma(x)$ the former and $\sigma[\![\overline{p}]\!]$ the latter. Consider the formal definition of substitutions in Fig. 3. The first equation ($\overset{1}{=}$) means that the substitution on the empty pattern is always the empty forest. The second equation ($\overset{2}{=}$) defines the substitution of a metavariable by its associated tree: the tree is added to the left of the resulting forest and the substitution proceeds recursively over the remaining unparsed pattern. The third equation ($\overset{3}{=}$) specifies that the substitutions always leave lexemes unchanged.

Pattern matching is defined by the inference system given in Fig. 4, where the rules are unordered and in Prolog, in Fig. 5. Let us call a *configuration* the pair $\langle \overline{p}, f \rangle$. In case the forest contains only one tree $h$, let us write $\langle \overline{p}, h \rangle$ instead of $\langle \overline{p}, [h] \rangle$. The pattern matching associates a configuration to a substitution. This system of inference rules is not syntax-directed, because the conclusions of rules BIND and UNPAR overlap: a non-deterministic choice between binding and unparsing must be done. This dilemma cannot be decided solely based on the shape of the configuration and thus the implementation must rely on a backtracking mechanism, as we said before. Note that no rule has more than one premise involving the ($\twoheadrightarrow$) relation, hence the proof trees (i.e., derivations, when read deductively) are actually lists. Rule END rewrites the empty configuration to the

$$\sigma[\![[\,]]\!] \stackrel{1}{=} [\,]$$
$$\sigma[\![[\mathsf{meta}(x) \mid \overline{p}]]\!] \stackrel{2}{=} [\sigma(x) \mid \sigma[\![\overline{p}]\!]]$$
$$\sigma[\![[l \mid \overline{p}]]\!] \stackrel{3}{=} [l \mid \sigma[\![\overline{p}]\!]]$$

**Fig. 3.** Substitutions on unparsed patterns.

$$\langle [\,], [\,]\rangle \twoheadrightarrow \sigma_{\varnothing} \quad \textsc{end} \qquad \frac{\langle \overline{p}, f\rangle \twoheadrightarrow \sigma}{\langle [l \mid \overline{p}], [l \mid f]\rangle \twoheadrightarrow \sigma} \textsc{elim}$$

$$\frac{\langle \overline{p}, f\rangle \twoheadrightarrow \sigma \qquad \sigma \subseteq \sigma \oplus x \mapsto t}{\langle [\mathsf{meta}(x) \mid \overline{p}], [t \mid f]\rangle \twoheadrightarrow \sigma \oplus x \mapsto t} \textsc{bind} \qquad \frac{\langle \overline{p}, f_1 \cdot f_2\rangle \twoheadrightarrow \sigma}{\langle \overline{p}, [c(f_1) \mid f_2]\rangle \twoheadrightarrow \sigma} \textsc{unpar}$$

**Fig. 4.** A backtracking pattern matching.

```
match([],[],[]).                                        % END
match([lex(L)|P],[lex(L)|F],S) :- match(P,F,S).         % ELIM
match([meta(X)|P],[node(C,F1)|F2],S2) :-                % BIND
  match(P,F2,S1), add(S1,{X,node(C,F1)},S2).
match(P,[node(_,F1)|F2],S) :-                           % UNPAR
  append(F1,F2,F), match(P,F,S).

add([],B,[B]).
add(S=[{X,T1}|_],{X,T2},S) :- !, T1 = T2.
add([B1|S1],B,[B1|S]) :- add(S1,B,S).
```

**Fig. 5.** The backtracking algorithm of Fig. 4 in Prolog.

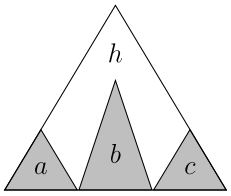


**Fig. 6.** Closed-tree inclusion $[a, b, c] \sqsubseteq h$.

empty substitution; this happens as the last rewrite step—from whence its name. Let us read the rules inductively, since this reading corresponds to an algorithm.

- Rule ELIM: if the pattern and the tree start with the same lexeme, then remove the lexemes and try to rewrite the remaining configuration.
- Rule BIND: a metavariable $x$ is bound to a tree $t$, i.e., $x \mapsto t$, if the remaining configuration rewrites to a substitution $\sigma$ which either already contains the binding or whose domain does not contain $x$ (i.e., $\sigma \subseteq \sigma \oplus x \mapsto t$); the resulting substitution is the updating of $\sigma$ with the new binding.
- Rule UNPAR: to match the same pattern against the same tree $t = c(f_1)$ whose root has been cut off (i.e., $f_1$ remains); this is *unparsing*. Note that the configuration $\langle [\mathsf{meta}(x) \mid \overline{p}], [c(f_1) \mid f_2]\rangle$ can lead both to an unparsing or a binding.

The encoding in Prolog is shown in Fig. 5. Patterns are noted `P`, variables `X`, forests `F`, substitutions `S`, trees `T`. A binding between a metavariable and a tree is a pair `{X,T}` and a substitution is a list of such bindings. $\langle \overline{p}, f\rangle \twoheadrightarrow \sigma$ is `match(P,F,S)`; $c(f)$ is `node(C,F)`; $\mathsf{meta}(x)$ is `meta(X)`; $l$ is `lex(L)` and '$\sigma \oplus x \mapsto t$ if $\sigma \subseteq \sigma \oplus x \mapsto t$' is implemented by `add(S1,{X,T},S2)`, where $\sigma$ is `S1`, $x \mapsto t$ is `{X,T}` and $\sigma \oplus x \mapsto t$ is `S2`.

### 3.2. Closed-tree inclusion

We need to defined what it means for a closed tree to be included in a parse tree. This is the *closed-tree matching*, which is a variation on the classic tree matching, as, in the latter, the pattern tree may embed metavariables and the roots are matched. This way, it becomes possible to compare the expressive power of the backtracking pattern matching with respect

$$[\,] \sqsubseteq [\,] \ \textsf{EMP} \qquad \frac{f_1 \sqsubseteq f_2}{[h \mid f_1] \sqsubseteq [h \mid f_2]}\ \textsf{EQ} \qquad \frac{f \sqsubseteq f_1 \cdot f_2}{f \sqsubseteq [c(f_1) \mid f_2]}\ \textsf{SUB} \qquad \frac{f \sqsubseteq [h]}{f \sqsubseteq h}\ \textsf{ONE}$$

**Fig. 7.** Closed-forest inclusion.

to the more familiar tree matching, used by many existing tools. Informally, let us say that a tree $h_1$ is included in a tree $h_2$ if and only if $h_1$ is included in $h_2$ and the fringe of $h_1$ is included in the fringe of $h_2$, as shown in Fig. 6. Let us note $h_1 \sqsubseteq h_2$ this relationship. Technically, we only need to define ($\sqsubseteq$) between a forest $f$ and a tree $h$ in such a manner that $f \sqsubseteq h$ implies that there exists a constructor $c$ such that $c(f) \sqsubseteq h$. But we shall not precise this further. The formal definition we propose here is based on partially ordered inference rules (see Section 2), displayed in Fig. 7. We give now a logical reading of the rules.

- Rule ONE states that if the forest $f$ is included in a forest made of a single tree $h$, then it is included in $h$. This relates a closed forest and a tree.
- Axiom EMP says that the empty forest is included in the empty forest.
- Rule EQ specifies that if a non-empty forest $f_1$ is included in another non-empty forest (possibly the same), then the forest $[h \mid f_1]$ is included in the forest $[h \mid f_2]$, where $h$ is a tree (possibly a lexeme).
- Rule SUB states that if a forest $f$ is included in the catenation of forests $f_1$ and $f_2$, such that the constructor $c$ can have $f_1$ as direct subtrees, then $f$ is included in $[c(f_1) \mid f_2]$ (i.e., grouping some trees into a new tree changes nothing). When reading the rules inductively, i.e., bottom-up, or, in other words, algorithmically, we must add the constraint that rule SUB must always be considered last, i.e., if a derivation (i.e., a proof tree) ends with SUB, its conclusion has the shape $f \sqsubseteq [c(f_1) \mid f_2]$ *and it is implied that there is no forest* $f'$ *such that* $f = [c(f_1) \mid f']$ (which would conclude EQ).

### 3.3. Soundness

The soundness of the pattern matching means that all computed substitutions, once applied to the original pattern, yield a closed forest which is included, in the sense above, in the original tree. Informally: all successful pattern matchings lead to successful closed-tree matchings. Formally:

**Theorem 1** (*Soundness*).

*If* $\langle \overline{p}, h\rangle \twoheadrightarrow \sigma$, *then* $\sigma[\![\overline{p}]\!] \sqsubseteq h$.

Proof of 1 (*Soundness*). Let $\aleph(\overline{p}, f, \sigma)$ be the proposition '*If* $\langle \overline{p}, f\rangle \twoheadrightarrow \sigma$, *then* $\sigma[\![\overline{p}]\!] \sqsubseteq f$.' Then $\aleph(p, [h], \sigma)$ is equivalent to the soundness property (by rule ONE). Firstly, let us assume that

$$\langle \overline{p}, f\rangle \twoheadrightarrow \sigma \tag{4}$$

is true (otherwise the theorem is trivially true). This means that there exists a pattern matching derivation $\Delta$ whose conclusion is $\langle \overline{p}, f\rangle \twoheadrightarrow \sigma$. This derivation is a list, which makes it possible to reckon by induction on its structure, i.e., one assumes that $\aleph$ holds for the premise of the last rule in $\Delta$ (this is the *induction hypothesis*) and then proves that $\aleph$ holds for $\langle \overline{p}, f\rangle \twoheadrightarrow \sigma$. A case by case analysis on the kind of rule that can end the derivation guides the proof.

(1) Case where $\Delta$ ends with END.

In this case, $\overline{p} = [\,]$ and $f = [\,]$. Therefore $\sigma[\![\overline{p}]\!] = \sigma[\![[\,]]\!] \overset{1}{=} [\,] \sqsubseteq [\,] = f$. We conclude that $\aleph([\,], [\,], \sigma)$ holds.

(2) Case where $\Delta$ ends with ELIM.

$$\frac{\dfrac{\vdots}{\langle \overline{p}', f'\rangle \twoheadrightarrow \sigma}}{\langle [l \mid \overline{p}'], [l \mid f']\rangle \twoheadrightarrow \sigma}\ \textsf{ELIM}$$

where, since we assumed (4),
(a) $f \triangleq [l \mid f']$,
(b) $\overline{p} \triangleq [l \mid \overline{p}']$.
Let us assume that the induction hypothesis holds for the premise of ELIM, i.e., $\aleph(\overline{p}', f', \sigma)$ holds:

$$\sigma[\![\overline{p}']\!] \sqsubseteq f'. \tag{5}$$

Besides, we have

$$\sigma[\![\overline{p}]\!] = \sigma[\![[l \mid \overline{p}']]\!] \sqsubseteq [l \mid f'] \overset{3}{=} [l \mid \sigma[\![\overline{p}']\!]] = f. \quad \text{by 2(b), (5), EQ, 2(a).} \tag{6}$$

As a conclusion, the induction hypothesis and (4) imply (6), so $\aleph([l \mid \overline{p}'], [l \mid f'], \sigma)$ holds.

(3) Case where $\Delta$ ends with BIND.

$$\dfrac{\dfrac{\vdots}{\langle \overline{p}', f' \rangle \twoheadrightarrow \sigma'}}{\langle [\mathsf{meta}(x) \,|\, \overline{p}'], [t \,|\, f'] \rangle \twoheadrightarrow \sigma' \oplus x \mapsto t}\ \textsc{bind}$$

where, because we assumed (4),
(a) $f \triangleq [t \mid f']$,
(b) $\overline{p} \triangleq [\mathsf{meta}(x) \mid \overline{p}']$,
(c) $\sigma' \subseteq \sigma' \oplus x \mapsto t$,
(d) $\sigma \triangleq \sigma' \oplus x \mapsto t$.
Let us assume that the induction hypothesis holds for the premise of BIND, i.e., $\aleph(\overline{p}', f', \sigma')$ holds:

$$\sigma' [\![\overline{p}']\!] \sqsubseteq f'. \tag{7}$$

We also have

$$\sigma(x) \quad \triangleq (\sigma' \oplus x \mapsto t)(x) = t. \qquad \text{by 3(d) and (1)} \tag{8}$$
$$\sigma' [\![\overline{p}']\!] \ = (\sigma' \oplus x \mapsto t)[\![\overline{p}']\!] = \sigma [\![\overline{p}']\!]. \ \text{by 3(c), Lemma 3 and 3(d)} \tag{9}$$

Besides, we have

$$\begin{aligned} \sigma [\![\overline{p}]\!] &\triangleq \sigma [\![[\mathsf{meta}(x) \mid \overline{p}']]\!] \overset{2}{=} [\sigma(x) \mid \sigma [\![\overline{p}']\!]] && \text{by 3(d) and Fig. 3} \\ &= [t \mid \sigma [\![\overline{p}']\!]] = [t \mid \sigma' [\![\overline{p}']\!]] && \text{by (8) and (9)} \\ &\sqsubseteq [t \mid f'] \triangleq f. && \text{by (7), EQ and 3(a)} \end{aligned} \tag{10}$$

In the end, the induction hypothesis and (4) imply (10), so $\aleph([\mathsf{meta}(x) \mid \overline{p}'], [t \mid f'], \sigma)$ holds.

(4) Case where $\Delta$ ends with UNPAR.

$$\dfrac{\dfrac{\vdots}{\langle \overline{p}, f_1 \cdot f_2 \rangle \twoheadrightarrow \sigma}}{\langle \overline{p}, [c(f_1) \mid f_2] \rangle \twoheadrightarrow \sigma}\ \textsc{unpar}$$

where, since we assumed (4),
(a) $f = [c(f_1) \mid f_2]$.
Let us assume that the induction hypothesis holds for the premise of UNPAR, i.e., $\aleph(\overline{p}, f_1 \cdot f_2, \sigma)$ holds:

$$\sigma [\![\overline{p}]\!] \ \sqsubseteq f_1 \cdot f_2 \sqsubseteq [c(f_1) \mid f_2] = f. \quad \text{by SUB (Fig. 7) and 4(a)} \tag{11}$$

As a conclusion, the induction hypothesis and (4) imply (11), so $\aleph([l \mid \overline{p}'], [c(f_1) \mid f_2], \sigma)$ holds. □

### 3.4. Completeness

The completeness of our algorithm means that every time a complete substitution on a pattern matches a tree, our algorithm computes a substitution which is included in the first one. Indeed, the computed substitution never contains useless bindings, but the other one may. Therefore, the completeness property is perhaps better stated by referring to *minimal substitutions*: all minimal substitutions that enable a closed-tree inclusion are computed by our pattern matching. Formally, this can be expressed as follows.

**Theorem 2** (*Completeness*).

*If* $\sigma [\![\overline{p}]\!] \sqsubseteq h$, *then* $\langle \overline{p}, h \rangle \twoheadrightarrow \sigma'$ *and* $\sigma' \subseteq \sigma$.

Proof Sketch 2 (*Completeness*). By structural induction on the patterns. □

**Lemma 3** (*Minimality*).

*If* $\sigma \subseteq \sigma'$ *then* $\sigma [\![\overline{p}]\!] = \sigma' [\![\overline{p}]\!]$.

In other words, for a given substitution, there exists a minimal substitution yielding the same result (if defined) for any pattern.

Proof Sketch 3 (*Minimality*). By structural induction on the patterns. □

### 3.5. Compliance

As a corollary, the algorithm is sound and complete with respect to the closed-tree inclusion:

**Corollary 4** (*Compliance*).

$\sigma[\![\overline{p}]\!] \sqsubseteq h$ *if and only if* $\langle \overline{p}, h\rangle \twoheadrightarrow \sigma'$ *and* $\sigma' \subseteq \sigma$.

In other words, the concept of closed-tree inclusion coincides exactly with the backtracking algorithm.

Proof 4 (*Compliance*). The way from left to right is the completeness. From the soundness, it comes that *If* $\langle \overline{p}, h\rangle \twoheadrightarrow \sigma'$, *then* $\sigma'[\![\overline{p}]\!] \sqsubseteq h$. The minimality Lemma 3 implies then $\sigma[\![\overline{p}]\!] \sqsubseteq h$. □

### 3.6. Further discussion

One solution to overcome the inefficiency of non-determinacy is to make explicit the tree structure in the textual pattern by adding some kind of parentheses, so each step becomes uniquely determined. For example, to match the pattern `%x = %y - %z` against the parse tree in Fig. 2(b), we would add to the pattern metalexemes called *metaparentheses* (i.e., parentheses that do not belong to the object language), which are here represented as escaped parentheses: `%(` and `%)`. For example, we may use the pattern `%(%x = %(%(%y%) - %z%)%)`. Note that in general we cannot use plain parentheses to unveil the structure because the language may already contain parentheses which may have a completely different meaning other than grouping. For instance, in the following Korn shell (ksh) pattern:

```
case %x in [yY]) echo yes;; *) echo no;; esac
```

we must use metaparentheses to explicit the tree structure, because parentheses would not pair the way we want—in fact, they would not pair at all. Fully metaparenthesized patterns enable linear-time pattern matching. However, the explicit structure comes at the price of seriously obfuscating the pattern. Clearly, if we may say, fully metaparenthesized patterns are quite difficult to read and write. The legibility of the pattern can be improved if the matching algorithm allows some of the metaparentheses to be dropped. This is what is shown in the next section, where the system is syntax-directed.

## 4. Algorithm *ES*(1)

### 4.1. Patterns and substitutions

Let mlp and mrp be metalexemes corresponding respectively to the opening and closing metaparentheses, whose concrete syntax is `%(` and `%)`. These metaparentheses are inserted by the user in order to guide matching the pattern against the parse tree by forcing the enclosed pattern to match one subtree. Metaparentheses can be present anywhere in the pattern as long as they are properly paired. Unparsed patterns are the smallest set $\overline{\mathcal{P}}$ such that

- $[\,] \in \overline{\mathcal{P}}$;
- if $l \in \mathcal{L}$ and $\overline{p} \in \overline{\mathcal{P}}$, then $[l \mid \overline{p}] \in \overline{\mathcal{P}}$;
- if $x \in \mathcal{V}$ and $\overline{p} \in \overline{\mathcal{P}}$, then $[\mathsf{meta}(x) \mid \overline{p}] \in \overline{\mathcal{P}}$;
- if $\overline{p}_1, \overline{p}_2 \in \overline{\mathcal{P}}$ and $\overline{p}_1 \neq [\,]$ then $[\mathsf{mlp}] \cdot \overline{p}_1 \cdot [\mathsf{mrp}] \cdot \overline{p}_2 \in \overline{\mathcal{P}}$.

Given $\overline{p}$, the first task is to check whether $\overline{p} \in \overline{\mathcal{P}}$. This is done by means of a metaparsing function, which either fails due to mismatched metaparentheses or returns the initial pattern where all patterns enclosed in metaparentheses (these included) have been replaced by a *pattern tree*. These trees are not parse trees because they contain patterns (thus the new patterns are still unparsed with respect to the programming language syntax). Let pat be their unique constructor. Thus we have $\mathsf{pat} \notin \mathcal{C}$. Let us define the set of *metaparsed patterns* $p$ as the smallest set $\mathcal{P}$ such that

- $[\,] \in \mathcal{P}$;
- if $l \in \mathcal{L}$ and $p \in \mathcal{P}$, then $[l \mid p] \in \mathcal{P}$;
- if $x \in \mathcal{V}$ and $p \in \mathcal{P}$, then $[\mathsf{meta}(x) \mid p] \in \mathcal{P}$;
- if $p_1, p_2 \in \mathcal{P}$ and $p_1 \neq [\,]$, then $[\mathsf{pat}(p_1) \mid p_2] \in \mathcal{P}$.

Pattern $l_1$ `%(` $l_2$ $l_3$ `%(` `%x` $l_4$ `%)` `%)` is the simplified form of unparsed pattern $[l_1, \mathsf{mlp}, l_2, l_3, \mathsf{mlp}, \mathsf{meta}(x), l_4, \mathsf{mrp}, \mathsf{mrp}]$. Its metaparsing is shown in Fig. 8. (Thereupon we write 'pattern' instead of 'metaparsed pattern'.)

Let us extend substitutions to cope with patterns, not just metavariables, as we did about the backtracking algorithm in Fig. 3. Closed patterns are defined inductively as the smallest set $\mathring{\mathcal{P}}$ such that

- $[\,] \in \mathring{\mathcal{P}}$;
- if $h \in \mathcal{H}$ and $\mathring{p} \in \mathring{\mathcal{P}}$, then $[h \mid \mathring{p}] \in \mathring{\mathcal{P}}$;
- if $\mathring{p}_1, \mathring{p}_2 \in \mathring{\mathcal{P}}$ and $\mathring{p}_1 \neq [\,]$, then $[\mathsf{pat}(\mathring{p}_1) \mid \mathring{p}_2] \in \mathring{\mathcal{P}}$.

Note the absence of metavariables in $\mathring{\mathcal{P}}$ and, instead, the presence of trees ($h \in \mathcal{H}$). The formal definition of substitutions can be found in Fig. 9. It is clear that the result of such substitutions is always a closed pattern.

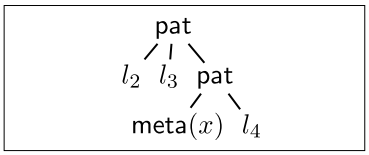


**Fig. 8.** Metaparsed pattern $[l_1, \mathsf{pat}([l_2, l_3, \mathsf{pat}([\mathsf{meta}(x), l_4])])]$.

$$\sigma[\![[\,]]\!] \stackrel{1}{=} [\,]$$
$$\sigma[\![[\mathsf{meta}(x) \mid p]]\!] \stackrel{2}{=} [\sigma(x) \mid \sigma[\![p]]\!]$$
$$\sigma[\![[l \mid p]]\!] \stackrel{3}{=} [l \mid \sigma[\![p]]\!]$$
$$\sigma[\![[\mathsf{pat}(p_1) \mid p_2]]\!] \stackrel{4}{=} [\mathsf{pat}(\sigma[\![p_1]]\!]) \mid \sigma[\![p_2]]\!]$$

**Fig. 9.** Substitutions on metaparsed patterns.

$$[\,] \sqsubseteq [\,] \ \mathsf{EMP} \qquad \frac{\mathring{p} \sqsubseteq f}{[h \mid \mathring{p}] \sqsubseteq [h \mid f]}\ \mathsf{EQ} \qquad \frac{\mathring{p} \sqsubseteq [h]}{\mathring{p} \sqsubseteq h}\ \mathsf{ONE}$$
$$\frac{\mathring{p}_1 \sqsubseteq f_1 \qquad \mathring{p}_2 \sqsubseteq f_2}{[\mathsf{pat}(\mathring{p}_1) \mid \mathring{p}_2] \sqsubseteq [c(f_1) \mid f_2]}\ \mathsf{PAT} \qquad \frac{\mathring{p} \sqsubseteq f_1 \cdot f_2}{\mathring{p} \sqsubseteq [c(f_1) \mid f_2]}\ \mathsf{SUB}$$

**Fig. 10.** Closed-pattern inclusion for $ES(1)$ (SUB is last).

### 4.2. Closed-tree inclusion

A relationship ($\sqsubseteq$) is needed to capture the concept of a closed-tree inclusion in a parse tree, just as we did about the backtracking algorithm in Section 3.3. The difference here is that closed trees may contain special nodes pat that distinguish them from parse trees. The formal definition we propose is based on ordered inference rules and given in Fig. 10.

- Rule ONE states that a pattern matches a tree if the same pattern matches the corresponding singleton forest.
- Rule EMP means: 'The empty pattern always matches the empty forest'.
- Rule EQ specifies that if the head of the pattern and the forest is the same closed tree, then the pattern is included in the forest if the remaining pattern $\mathring{p}$ is included in the forest $f$.
- Rule PAT handles the case of a pattern tree: there is inclusion if the subtrees $\mathring{p}_1$ of the pattern tree are included in the subtrees $f_1$ of the forest head and if the remaining pattern $\mathring{p}_2$ is included in the remaining forest $f_2$.
- Rule SUB states that if the two heads are different trees (this is implicit due to partial ordering and SUB being the last), there is inclusion if the pattern is included om the catenation of the subtrees $f_1$ of the tree in the forest and the remaining forest $f_2$. This rule must always be considered last, i.e., if a derivation ends with SUB, the last conclusion has the shape $\mathring{p} \sqsubseteq [c(f_1) \mid f_2]$ *and it is implied that there are no closed patterns* $\mathring{p}_1$ *and* $\mathring{p}_2$ *such that* $\mathring{p} = [c(f_1) \mid \mathring{p}_1]$ or $\mathring{p} = [\mathsf{pat}(\mathring{p}_1) \mid \mathring{p}_2]$ (which would lead to the conclusions of EQ or PAT).

### 4.3. Pattern matching

Compared to the informal description of the backtracking pattern matching given in Section 3, the binding and the unparsing steps are here specialised so to exclude each other based on the shape of the configuration at hand (this is syntax-direction). The critical case is when the current pattern starts with a metavariable: a step may be chosen that leads to a match failure whilst the other step would have not. We will come back later on this point. For now, let us describe the new binding and unparsing operations.

- **Binding.** The first element of the pattern is a metavariable $\mathsf{meta}(x)$ and the first tree $t$ of the parse forest is not a leaf. The metavariable is bound to the tree, i.e., $x \mapsto t$ in the following cases.
  - *When the parse forest is reduced to one tree.* An unparsing is possible if the tree root has only one subtree (so cutting out the root leads to another single tree), but programmers usually want to bind the biggest tree. This step is the rule $\mathsf{BIND}_3$ in Fig. 11. Notice the notation $\mathsf{meta}(x_{[c]})$ which is a shortcut for '$\mathsf{meta}(x)$ or $\mathsf{meta}(x_c)$,' where $c$ is a node constructor. $\mathsf{meta}(x_c)$ means that the metavariable $x$ must be bound to a node labeled by a constructor $c$. This gives the programmer more control on the binding process.

$$\langle [\,], [\,]\rangle \twoheadrightarrow \sigma_\varnothing \ \text{END} \qquad \frac{\langle p, f\rangle \twoheadrightarrow \sigma}{\langle [l \mid p], [l \mid f]\rangle \twoheadrightarrow \sigma}\ \text{ELIM}$$

$$\frac{\langle p, f_2\rangle \twoheadrightarrow \sigma \qquad \sigma \subseteq \sigma \oplus x \mapsto c(f_1)}{\langle [\text{meta}(x_{[c]}), l \mid p], [c(f_1), l \mid f_2]\rangle \twoheadrightarrow \sigma \oplus x \mapsto c(f_1)}\ \text{BIND}_1$$

$$\frac{\langle p, [t_2 \mid f]\rangle \twoheadrightarrow \sigma \qquad \sigma \subseteq \sigma \oplus x \mapsto c(f_1)}{\langle [\text{meta}(x_{[c]}) \mid p], [c(f_1), t_2 \mid f]\rangle \twoheadrightarrow \sigma \oplus x \mapsto c(f_1)}\ \text{BIND}_2$$

$$\langle [\text{meta}(x_{[c]})], [c(f)]\rangle \twoheadrightarrow \{x \mapsto c(f)\}\ \text{BIND}_3$$

$$\frac{\sigma_1 \subseteq \sigma_1 \oplus \sigma_2 \qquad \langle p_1, f_1\rangle \twoheadrightarrow \sigma_1 \qquad \langle p_2, f_2\rangle \twoheadrightarrow \sigma_2}{\langle [\text{pat}(p_1) \mid p_2], [c(f_1) \mid f_2]\rangle \twoheadrightarrow \sigma_1 \oplus \sigma_2}\ \text{UNPAR}_1$$

$$\frac{\langle p, f_1 \cdot f_2\rangle \twoheadrightarrow \sigma}{\langle p, [c(f_1) \mid f_2]\rangle \twoheadrightarrow \sigma}\ \text{UNPAR}_2$$

**Fig. 11.** Pattern matching $ES(1)$ ($\text{UNPAR}_2$ is last).

  - *When the second element of the pattern and the second tree of the forest are the same lexeme l.* Hence this decision is based on a lookahead of one lexeme after the first tree. This case corresponds to the rule $\text{BIND}_1$ in Fig. 11. Note that there is a slight optimisation here, since $l$ is eliminated in the same rule (otherwise rule ELIM would always have been used after this one). The metavariable $x$ can be typed so it can only bind nodes of kind $c$, which is denoted by $\text{meta}(x_{[c]})$.
  - *When the second tree of the forest is not a leaf.* This is the rule $\text{BIND}_2$ in Fig. 11. Same remark about $\text{meta}(x_{[c]})$.

  The substitution $\sigma$ resulting from the recursive call (among the premises) must be compatible with the binding $x \mapsto c(f_1)$, i.e., $\sigma \subseteq \sigma \oplus x \mapsto c(f_1)$.
- **Unparsing.** The root of the first tree in the parse forest is cut out and the tree is replaced by its direct subtrees.
  - If the first element of the pattern is a tree pattern, i.e., corresponds to a metaparenthesis in the original unparsed pattern, then this tree pattern is matched against the first tree and the remaining pattern is matched against the remaining forest. The two resulting substitutions, $\sigma_1$ and $\sigma_2$, must be compatible, i.e., no binding in one is present in the other with the same metavariable, unless the tree is the same (in short: $\sigma_1 \subseteq \sigma_1 \oplus \sigma_2$). This is rule $\text{UNPAR}_1$ in Fig. 11.
  - Otherwise, the new configuration is rewritten. This is rule $\text{UNPAR}_2$ in Fig. 11, which must be considered last, because its conclusion can overlap with the one of BIND rules or $\text{UNPAR}_1$.

Fig. 12 shows the implementation of ES(1) in Objective Caml, a functional language with static typing. The substitutions are lists of bindings. We use an exception instead of an empty list to signal a match failure. The library function `List.fold_left` is a functional iterator such that `List.fold_left f a [b1; ...; bn]` is `f (... (f (f a b1) b2) ...) bn`. The untyped metavariable $\text{meta}(x)$ is implemented as `‘Meta(x,None)`; the typed metavariable $\text{meta}(x_c)$ is implemented as `‘Meta(x,Some c)`; lexeme $l$ by `‘Lex(l)`; a metaparsed pattern $\text{pat}(p)$ by `‘Pat(p)`; the non-leaf tree $c(f)$ by `‘Node(c,f)`; a forest $f$ by a list of trees. The binary operator `@` is the list concatenation; the expression $[a \mid l]$ is implemented by `a::l` and the binder ‘`as`’ allows creating an alias in the Objective Caml pattern. The function `add` is the equivalent of the Prolog predicate `add/3` in Fig. 4.

### 4.4. Termination and worst-case complexity

The system terminates because each rule either strictly decreases the number of elements in the pattern or the number of nodes in the parse forest. A successful run happens when a configuration cannot be rewritten furthermore and it contains an empty pattern and an empty parse forest. Let us assume that the cost of applying an inference rule is constant. Then the cost of a run is upper bounded by a constant times the number of rules in the proof tree (which can be considered as a trace of the execution). The worst-case complexity is thus obtained by an initial configuration which leads to the maximum number of rules being used. The observation made about the termination of the system gives us the clue as how to proceed in finding such input. Because a metavariable cannot be bound to a lexeme (i.e., a tree leaf), the nodes of the bounded subtree are not considered by the algorithm. Thus, the worst case contains no metavariables, so no BIND rule is used. Moreover, the size of the pattern has no impact on the cost, since a metaparenthesis or a lexeme are eliminated at the same time a node or a leaf is removed from the parse forest (see rules ELIM and $\text{UNPAR}_1$). The worst-case complexity follows: it is linear in the number of nodes in the parse forest.

```
exception Failure

let rec add s b = match (s,b) with
  ([],b) -> [b]
| ((x,v)::s1 as s,(y,w)) when x = y ->
    if v = w then s else raise Failure
| (b1::s1,b) -> b1 :: add s1 b

let rec mat p f = match (p,f) with
  ([],[]) -> []                                    (* END    *)
| (‘Lex(l1)::p,‘Lex(l2)::f) when l1 = l2
    -> mat p f                                     (* ELIM   *)
| (‘Meta(x,None)::‘Lex(l1)::p,‘Node(c,f1)::‘Lex(l2)::f2)
    when l1 = l2
    -> add (mat p f2) (x,‘Node(c,f1))              (* BIND1  *)
| (‘Meta(x,Some c1)::‘Lex(l1)::p,‘Node(c2,f1)::‘Lex(l2)::f2)
    when l1 = l2 && c1 = c2
    -> add (mat p f2) (x,‘Node(c2,f1))       (* BIND1 typed *)
| (‘Meta(x,None)::p,‘Node(c,f1)::(‘Node(_,_)::_ as f2))
    -> add (mat p f2) (x,‘Node(c,f1))              (* BIND2  *)
| (‘Meta(x,Some c)::p,‘Node(c1,f1)::(‘Node(_,_)::_ as f2))
    when c = c1
    -> add (mat p f2) (x,‘Node(c1,f1))       (* BIND2 typed *)
| ([‘Meta(x,None)],[‘Node(c,f)])
    -> [(x,‘Node(c,f))]                            (* BIND3  *)
| ([‘Meta(x,Some c1)],[‘Node(c,f)]) when c1 = c
    -> [(x,‘Node(c,f))]                      (* BIND3 typed *)
| (‘Pat(p1)::p2,‘Node(c,f1)::f2)
    -> List.fold_left add (mat p1 f1) (mat p2 f2) (* UNPAR1 *)
| (p,‘Node(c,f1)::f2)
    -> mat p (f1 @ f2)                             (* UNPAR2 *)
| _ -> raise Failure
```

**Fig. 12.** Implementation of $ES(1)$ in Objective Caml.

### 4.5. Soundness

**Theorem 5** (*Soundness*).

*If* $\langle p, h\rangle \twoheadrightarrow \sigma$ *then* $\sigma[\![p]\!] \sqsubseteq h$.

Proof 5 (*Soundness*). Let $\aleph(p, f, \sigma)$ be the proposition '*If* $\langle p, f\rangle \twoheadrightarrow \sigma$ *then* $\sigma[\![p]\!] \sqsubseteq f$.' So $\aleph(p, [h], \sigma)$ is equivalent to the soundness. Let

$$\langle p, f\rangle \twoheadrightarrow \sigma \tag{12}$$

(otherwise the theorem is trivially true). This means that there exists a pattern matching derivation $\Delta$ whose conclusion is $\langle p, f\rangle \twoheadrightarrow \sigma$. This derivation is a tree; we hence can reason by structural induction on it, i.e., we assume that $\aleph$ holds for the premises of the last rule in $\Delta$ (this is the *induction hypothesis*) and then prove that $\aleph$ holds for $\langle p, f\rangle \twoheadrightarrow \sigma$. We proceed case by case on the kind of rule that can end $\Delta$.

(1) Case where $\Delta$ ends with END.

We have $p = [\,], f = [\,]$ and $\sigma = \sigma_\varnothing$. Therefore $\sigma[\![p]\!] = \sigma[\![[\,]]\!] \overset{1}{=} [\,] \sqsubseteq [\,] = f$. Thus $\aleph([\,], [\,], \sigma_\varnothing)$ holds.

(2) Case where ELIM ends $\Delta$.

$$\frac{\begin{array}{c}\vdots\\ \langle p', f'\rangle \twoheadrightarrow \sigma\end{array}}{\langle [l \mid p'], [l \mid f']\rangle \twoheadrightarrow \sigma}\ \textsc{elim}$$

where, since we assumed (12),
(a) $p \triangleq [l \mid p']$,
(b) $f \triangleq [l \mid f']$.

Let us assume that the induction hypothesis holds for the premise of ELIM, that is to say, $\aleph(p', f', \sigma)$ holds. Thus

$$\sigma[\![p']\!] \sqsubseteq f'. \tag{13}$$

Besides, we have

$$\sigma[\![p]\!] = \sigma[\![[l \mid p']]\!] \overset{3}{=} [l \mid \sigma[\![p']\!]] \sqsubseteq [l \mid f'] = f. \text{ by 2(a), (13), EQ, 2(b)} \tag{14}$$

As a conclusion, the induction hypothesis and (12) imply (14) in this case, i.e., $\aleph(p, f, \sigma)$ holds.

(3) Case where $\mathsf{BIND}_1$ ends $\Delta$.

$$\frac{\begin{array}{c}\vdots\\ \langle p'', f''\rangle \twoheadrightarrow \sigma'\end{array}}{\langle [\mathsf{meta}(x_{[c]}), l \mid p''], [c(f_1), l \mid f'']\rangle \twoheadrightarrow \sigma' \oplus x \mapsto c(f_1)}$$

where, since we assumed (12),
(a) $t \triangleq c(f_1)$,
(b) $\sigma' \subseteq \sigma' \oplus x \mapsto t$,
(c) $p \triangleq [\mathsf{meta}(x_{[c]}), l \mid p'']$,
(d) $f \triangleq [t, l \mid f'']$,
(e) $\sigma \triangleq \sigma' \oplus x \mapsto t$.

Let us assume that the induction hypothesis holds for the premise of $\mathsf{BIND}_1$, i.e., $\aleph(p'', f'', \sigma')$ holds:

$$\sigma'[\![p'']\!] \sqsubseteq f''. \tag{15}$$

From 3(b) and Lemma 3, we draw

$$\sigma'[\![p'']\!] = (\sigma' \oplus x \mapsto t)[\![p'']\!] = \sigma[\![p'']\!] \sqsubseteq f''. \text{ by 3(e) and (15)} \tag{16}$$

We have

$$\sigma(x) = (\sigma' \oplus x \mapsto t)(x) = t \text{ by 3(e) and (1).} \tag{17}$$

Besides, we have the following equalities:

$$\begin{aligned}\sigma[\![p]\!] &= \sigma[\![[\mathsf{meta}(x_{[c]}), l \mid p'']]\!] \overset{2}{=} [\sigma(x) \mid \sigma[\![[l \mid p'']]\!]] \text{ by 3(c) and Fig. 9}\\ &\overset{3}{=} [\sigma(x), l \mid \sigma[\![p'']\!]] = [t, l \mid \sigma[\![p'']\!]]. \qquad \text{Fig. 9 and (17)}\end{aligned} \tag{18}$$

Closed-tree matching (16) and the derivation

$$\frac{t \in \mathcal{H} \qquad \dfrac{l \in \mathcal{H} \qquad \sigma[\![p'']\!] \sqsubseteq f''}{[l \mid \sigma[\![p'']\!]] \sqsubseteq [l \mid f'']}\ \mathsf{EQ}}{[t, l \mid \sigma[\![p'']\!]] \sqsubseteq [t, l \mid f'']}\ \mathsf{EQ}$$

imply

$$\sigma[\![p]\!] = [t, l \mid \sigma[\![p'']\!]] \sqsubseteq [t, l \mid f''] = f. \text{ by (18) and 3(d)} \tag{19}$$

As a conclusion, the induction hypothesis and (12) imply (19) in this case, i.e., $\aleph([\mathsf{meta}(x_{[c]}) \mid p'], f, \sigma)$.

(4) Case where $\mathsf{BIND}_2$ ends $\Delta$.

$$\frac{\begin{array}{c}\vdots\\ \langle p', [t_2 \mid f']\rangle \twoheadrightarrow \sigma'\end{array}}{\langle [\mathsf{meta}(x_{[c]}) \mid p'], [c(f_1), t_2 \mid f']\rangle \twoheadrightarrow \sigma' \oplus x \mapsto c(f_1)}$$

where, since we assumed (12),
(a) $t_1 \triangleq c(f_1)$,
(b) $p \triangleq [\mathsf{meta}(x_{[c]}) \mid p']$,
(c) $f \triangleq [t_1, t_2 \mid f']$,
(d) $\sigma \triangleq \sigma' \oplus x \mapsto t_1$,
(e) $\sigma' \subseteq \sigma' \oplus x \mapsto t_1$.

Let us assume that the induction hypothesis holds for the premise of $\mathsf{BIND}_2$, i.e., $\aleph(p', [t_2 \mid f'], \sigma')$:

$$\sigma'[\![p']\!] \sqsubseteq [t_2 \mid f']. \tag{20}$$

From 4(e) and Lemma 3, we draw

$$\sigma'[\![p']\!] = (\sigma' \oplus x \mapsto t_1)[\![p']\!] = \sigma[\![p']\!] \sqsubseteq [t_2 \mid f']. \quad \text{by 4(d) and (20)} \tag{21}$$

We have

$$\sigma(x) = (\sigma' \oplus x \mapsto t_1)(x) = t_1. \quad \text{by 4(d) and (1)} \tag{22}$$

Furthermore,

$$\begin{aligned} \sigma[\![p]\!] &= \sigma[\![[\mathsf{meta}(x_{[c]}) \mid p']]\!] \overset{2}{=} [\sigma(x) \mid \sigma[\![p']\!]] \text{ by 4(b) and Fig. 9} \\ &= [t_1 \mid \sigma[\![p']\!]] \sqsubseteq [t_1, t_2 \mid f'] = f. \qquad \text{by (22), (21), EQ, 4(c)} \end{aligned} \tag{23}$$

As a conclusion, the induction hypothesis and (12) imply (23) in this case, i.e., $\aleph([\mathsf{meta}(x_{[c]}) \mid p'], f, \sigma)$.

(5) Case where $\mathsf{BIND}_3$ ends $\Delta$.

$$\langle [\mathsf{meta}(x_{[c]})], [c(f)] \rangle \twoheadrightarrow \{x \mapsto c(f)\} \ \mathsf{BIND}_3$$

where, since we assumed (12),
(a) $t \triangleq c(f)$,
(b) $p \triangleq [\mathsf{meta}(x_{[c]})]$,
(c) $f \triangleq [t]$,
(d) $\sigma \triangleq \{x \mapsto t\}$.
Because $\mathsf{BIND}_3$ is an axiom, we must prove $\aleph([\mathsf{meta}(x_{[c]})], [t], \{x \mapsto t\})$ without relying on the induction principle:

$$\begin{aligned} \sigma[\![p]\!] &= \sigma[\![[\mathsf{meta}(x_{[c]})]]\!] \overset{2}{=} [\sigma(x) \mid \sigma[\![[\,]]\!]] \overset{1}{=} [\sigma(x) \mid [\,]] \text{ cf. Fig. 9 and 5(b)} \\ &\triangleq [\sigma(x)] = [t]. \qquad \text{by 5d and (1)} \end{aligned} \tag{24}$$

Since we also have the derivation

$$\dfrac{\dfrac{}{[\,] \sqsubseteq [\,]}\ \mathsf{EMP}}{[t \mid [\,]] \sqsubseteq [t \mid [\,]]}\ \mathsf{EQ}$$

we know that $[t] \sqsubseteq [t]$, which, in conjunction with (24), implies

$$\sigma[\![p]\!] \sqsubseteq [t] = f. \quad \text{by 5(c)} \tag{25}$$

As a conclusion, the induction hypothesis and (12) imply (25), that is, $\aleph([\mathsf{meta}(x_{[c]})], [t], \{x \mapsto t\})$ holds.

(6) Case where $\Delta$ ends with $\mathsf{UNPAR}_1$.

$$\dfrac{\dfrac{(\Delta_1)}{\vdots}}{\langle p_1, f_1 \rangle \twoheadrightarrow \sigma_1} \quad \dfrac{\dfrac{(\Delta_2)}{\vdots}}{\langle p_2, f_2 \rangle \twoheadrightarrow \sigma_2}}{\langle [\mathsf{pat}(p_1) \mid p_2], [c(f_1) \mid f_2] \rangle \twoheadrightarrow \sigma_1 \oplus \sigma_2}\ \mathsf{UNPAR}_1$$

where, since we assumed (12),
(a) $p \triangleq [\mathsf{pat}(p_1) \mid p_2]$,
(b) $f \triangleq [c(f_1) \mid f_2]$,
(c) $\sigma_1 \subseteq \sigma_1 \oplus \sigma_2$.
The derivations $\Delta_1$ and $\Delta_2$ are sub-derivations of $\Delta$, therefore the induction hypothesis holds for their conclusions, i.e., $\aleph(p_1, f_1, \sigma_1)$ is true:

$$\sigma_1[\![p_1]\!] \sqsubseteq f_1 \tag{26}$$

and $\aleph(p_2, f_2, \sigma_2)$ is true as well:

$$\sigma_2[\![p_2]\!] \sqsubseteq f_2. \tag{27}$$

Directly from the definition (2), it comes

$$\sigma_2 \subseteq \sigma_1 \oplus \sigma_2. \tag{28}$$

It follows

$$\sigma_1[\![p_1]\!] = (\sigma_1 \oplus \sigma_2)[\![p_1]\!] \text{ by 6(c) and Lemma 3} \tag{29}$$

$$\sigma_2[\![p_2]\!] = (\sigma_1 \oplus \sigma_2)[\![p_2]\!] \text{ by (28) and Lemma 3.} \tag{30}$$

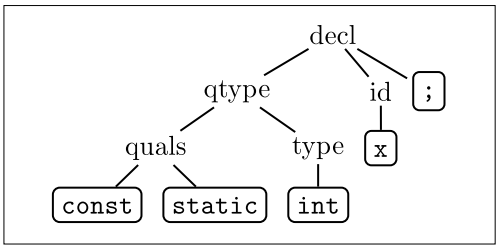


**Fig. 13.** A simplified parse tree (no empty words).

Let $\sigma \triangleq \sigma_1 \oplus \sigma_2$. Besides, we have

$$\begin{aligned}\sigma[\![p]\!] &= \sigma[\![[\mathsf{pat}(p_1) \mid p_2]]\!] \overset{4}{=} [\mathsf{pat}(\sigma[\![p_1]\!]) \mid \sigma[\![p_2]\!]] && \text{by 6(a) and Fig. 9}\\ &= [\mathsf{pat}(\sigma_1[\![p_1]\!]) \mid \sigma[\![p_2]\!]] = [\mathsf{pat}(\sigma_1[\![p_1]\!]) \mid \sigma_2[\![p_2]\!]] && \text{by (29) and (30)}\\ &\sqsubseteq [c(f_1) \mid f_2] = f. && \text{by (26), (27), PAT, 6(b)}\end{aligned} \tag{31}$$

As a conclusion, the induction hypothesis and (12) imply (31), that is, $\aleph([\mathsf{pat}(p_1) \mid p_2], f, \sigma)$ holds.

(7) Case where $\mathsf{UNPAR}_2$ ends $\Delta$.

$$\dfrac{\dfrac{\vdots}{\langle p, f_1 \cdot f_2\rangle \twoheadrightarrow \sigma}}{\langle p, [c(f_2) \mid f_2]\rangle \twoheadrightarrow \sigma}\ \mathsf{UNPAR}_2$$

where, since we assumed (12),
(a) $f \triangleq [c(f_1) \mid f_2]$.
Let us assume that the induction hypothesis holds for the premise of $\mathsf{UNPAR}_2$, i.e., $\aleph(p, f_1 \cdot f_2, \sigma)$ is true. Therefore

$$\sigma[\![p]\!] \sqsubseteq f_1 \cdot f_2 \sqsubseteq [c(f_1) \mid f_2] = f. \quad \text{by SUB (Fig. 10), 7(a)} \tag{32}$$

As a conclusion, the induction hypothesis and (12) imply (32) in this case, i.e., $\aleph(p, f, \sigma)$. (The structure of the pattern $p$ is irrelevant here.) □

### 4.6. Completeness

This algorithm is not complete in the sense the backtracking algorithm was in Theorem 2. Consider the unparsed pattern `%x = %y - %z - %t` and the same parse tree in Fig. 2(b). The execution trace is $\mathsf{UNPAR}_2$, $\mathsf{BIND}_1$, $\mathsf{UNPAR}_2$, $\mathsf{BIND}_1$ and then failure, that is, no rule apply. But there exists a successful closed-tree inclusion if the substitution $\{\mathtt{x} \mapsto \mathrm{var}(\mathtt{a}), \mathtt{y} \mapsto \mathrm{var}(\mathtt{a}), \mathtt{z} \mapsto \mathrm{mul}(\mathrm{var}(\mathtt{b}), *, \mathrm{var}(\mathtt{c})), \mathtt{t} \mapsto \mathrm{var}(\mathtt{d})\}$ is applied to the pattern first. Therefore, a match failure with $ES(1)$ can either mean that there is no matching or that one exists but was not found. In the latter case, metaparentheses must be added to the pattern in order to force an unparsing step instead of a binding. In the previous example, the successful substitution is found by $ES(1)$ if the unparsed pattern is transformed into `%x = %(%(%y - %z%) - %t%)`. Of course, as a worst case, if the pattern is fully metaparenthesized *with respect to the grammar*, $ES(1)$ becomes complete in the sense above, e.g., `%(%x = %(%(%(%y%) - %z%) - %t%)%)`. Indeed, such parenthesising leads to a metaparsed pattern tree which is isomorphic to a tree pattern, as in classic pattern matching. In other words, there is always a way to metaparenthesize an unparsed pattern to make it as expressive as the classic, tree-based, pattern matching. Moreover, in practice, only a few metaparentheses may be needed for a given pattern, as shown above. By looking closely at the rewrite rule, we can figure out necessary conditions for an unparsed pattern to lead to a loss of completeness in matching. These conditions can serve as empirical guidelines to prevent the problem from appearing. As mentioned above, the problem occurs when an instance of rule $\mathsf{BIND}_1$ or $\mathsf{BIND}_2$ is used and later leads to a failure. This failure could have been avoided, had $\mathsf{UNPAR}_2$ been selected, perhaps several times, followed by $\mathsf{BIND}_1$ or $\mathsf{BIND}_2$. Assuming that it is rule $\mathsf{BIND}_1$ that should be delayed after some $\mathsf{UNPAR}_2$ steps, it means that the lexeme $l$ is repeated in the parse forest and occurs each time after a tree (not a lexeme). Left-associative operators in programming languages usually occur in such kind of grammatical constructs. This is why the previous pattern had to be written `%x = %(%(%y - %z%) - %t%)`. Let us consider now that it is rule $\mathsf{BIND}_2$ that should have been delayed. This leads unparsed patterns as "`%q %t %x ;`" to fail to match the parse tree given in Fig. 13. An unparsing step should be taken instead of a premature binding, so that `%q` matches the type qualifiers and `%t` the type. One way out of this is to use the metaparenthesized unparsed pattern `%(%(%q %t%) %x;%)` instead. However, a simpler solution in this case is to use a typed metavariable $q$, giving the pattern '`%<quals>q %t %x ;`', which does not force the user to know anything about the nesting structure of the AST, and is also arguably more readable than the former metaparenthesized pattern.

In fact, a typed metavariable `%<c>x` forces the algorithm to perform as many unparse step as needed before obtaining a tree of the form $c(\ldots)$ at the left of the forest. Hence, the only possible ambiguity with a typed metavariable is when several trees of the form $c(\ldots)$ can be obtained by such a continuous sequence of unparse steps. It is easy to see that this

situation always corresponds to a (directly or indirectly) left-recursive grammar construct such as a left-associative operator, a left-recursive list construct, etc. The set of such constructs may be automatically produced based on the subject language grammar. For these constructs, the algorithm will bind the metavariable typed with *c* to the biggest, enclosing *c* construct. If this is not what is needed, metaparentheses must be added to force more unparse steps.

Summarising: parsing failures can always be solved by typing the conflicting metavariable, except when that variable must bind a *nested* left-recursive construct, in which case the pattern needs to be metaparenthesized. Of course, metaparentheses and typed variables can be freely combined and may complement each other.

## 5. Implementation

Avoiding the parsing of patterns dramatically simplifies the implementation of pattern matching, as can be seen from the following two implementations. Indeed, we saw that extending a programming language grammar is difficult to implement in most existing tools. In contrast, unparsing an AST is trivial to implement: it consists of just printing the AST. In most cases, parser-based tools already include functions to pretty-print the AST, for debugging reasons. Moreover, AST pretty-printers can be generated automatically based on the grammar of a language. It is straightforward to adapt an existing pretty-printer to do unparsing on demand (see below for an example).

Unparsed patterns were first implemented by the second author in the context of a lightweight checking compiler called myGCC,[3] which is an extensible version of the GCC compiler, able to perform user-defined checks on C, C ++, and Ada code. Checks are expressed by defining incorrect sequences of program operations, where each program operation is described as an unparsed pattern or a disjunction of unparsed patterns. The implementation of pattern matching within myGCC accounts for only about 600 lines of new C code, plus about 250 lines of code adapting the existing tree pretty-printer of GCC to perform unparsing on demand. The existing pretty-printer dumped the unparsed representation of a whole AST in a debug file. We added a flag `--lazy_mode` to switch between the standard dumping behaviour and the new on-demand behaviour. When in on-demand mode, the pretty-printer returns for a given AST the list of its direct children (either trees or lexemes), instead of dumping the AST entirely to a file. This modification was straightforward.

It is important to note that even though three different input languages can be checked, every single line of the patch is language independent. As a proof for that, the patched GCC compiler restricted to the C front-end was initially tested only on C code, as reported previously [19]; subsequently, by just recompiling GCC with all the front-ends enabled it became possible to check C ++ and Ada programs. Part of this extreme language independence comes from the fact that all the three front-ends generate intermediate code in a language called Gimple, and the dumper for different languages shared a common infrastructure based on Gimple, which was modified just once. However, this is not required for the pattern matching framework. The only language dependent aspects used by the matcher were already present in GCC: a parser for each language, a conversion from language-specific ASTs to Gimple ASTs, and a dumping function of Gimple ASTs for each language, sharing some common infrastructure. Our patch of the dumper (briefly sketched above) concerned only the common (or language independent) infrastructure of the dumper. The C ++ and Ada pattern matching became possible by this combination of the language independent matching algorithm with the language dependent unparsers already present in GCC. For instance, a pattern such as `%_ = operator[](%x,%y)` successfully matches any C ++ assignment of the form `%_ = %x[%y]` in which the indexing operator has been redefined, because the corresponding Gimple ASTs are dumped using the `operator[]` syntax captured by the pattern. Had the common infrastructure of the language-specific dumpers not existed, each of the dumpers should have been modified to make them unparse level by level.

The *ES*(1) pattern matching algorithm was also implemented by the second author and is distributed as a freely available, standalone prototype called Matchbox.[4] This very simple prototype, consisting of 500 lines of C code, takes a parse tree represented in a Lisp-like notation and an unparsed pattern, prints a complete trace of all the rules applied, and finally reports a successful match or a failure. The prototype may already be used to reproduce all the examples in this paper (using the *ES*(1) algorithm). The aim of Matchbox is to evolve into a standalone library for unparsed-pattern matching, which can be linked from any parser-based tool.

### 5.1. Examples

Here are some examples of pattern matching by Matchbox. First, this is the example shown in Fig. 2. Note how the parse tree is written using parentheses and also how the concrete syntax lexemes are quoted.

```
match "assign(var('a') '='
       sub(sub(var('a') '-' mul(var('b')'*'var('c'))) '-' var('d')))"
      "%x = %y - %z"
ok, sigma={x<-var('a') y<-sub(var('a')'-'mul(var('b')'*'var('c')))
           z<-var('d')}
```

[3] http://mygcc.free.fr.

[4] http://mypatterns.free.fr/unparsed.

Now the same pattern but with a simplified parse tree in which the constructors were omitted:

```
match "(('a')'='(('a')'-'(('b')'*'('c')))'-'('d')))"
      "%x = %y - %z"
ok, sigma={x<-('a') y<-(('a')'-'(('b')'*'('c'))) z<-('d')}
```

Note the metaparentheses needed to avoid failure due to the incompleteness of *ES*(1):

```
match "(('a')'='(('a')'-'(('b')'*'('c')))'-'('d')))"
      "%x = %(%(%y - %z%) - %t%)"
ok, sigma = {t<-('d') x<-('a') y<-('a') z<-(('b')'*'('c'))}
```

Consider the unparsed pattern "`%q %t %x;`" against the parse tree in Fig. 13: the first metavariable, which intends to capture the qualificators, actually binds the qualified type as a whole, leading to a later failure:

```
match "decl(qtype(quals('const' 'static') type('int')) id('x') ';')"
      "%q %t %x;"
failed
```

In order to avoid that, the variable *q* must be typed as `%<quals>q`:

```
match "decl(qtype(quals('const' 'static') type('int')) id('x') ';')"
      "%<quals>q %t %x;"
ok, sigma = {q<-quals('const' 'static') t<-type( 'int' ) x<-id( 'x' )}
```

Mixing parentheses and metaparentheses with the parse tree from "`case v in 1) exit;; esac`":

```
match "('case'('v')'in'(('1')')' ('exit')';;') 'esac')"
      "case %x in %y) %z;; esac"
ok, sigma = {x<-('v') y<-('1') z<-('exit')}
```

### 5.2. Assessment

The technique of pattern matching based on unparsed patterns is completely language independent. Patterns are simply strings, which exist as a base type in any programming language. No pattern parser is required, which means also that the implementation of the pattern matcher is language independent. The only part tied to a specific language is the unparser, but they can be automatically generated from any grammar.

Unparsed patterns are an unrestricted solution to match programs, because they allow pattern variables to stand for any program subconstruct or subexpression—in fact, for any subtree in the AST. For instance, in the case of ASTs for the C language, pattern variables may stand for function names, structure field names, or types, which are all subtrees. This makes it possible to naturally express patterns such as `%x = %f (%y)` catching function names or expressions in pattern variable `f`, `static %t a` catching the type of variable `a` in pattern variable `t`, etc. As opposed to this unrestricted use, many pattern-based tools implement parsed patterns only for a few common program constructs. By being an easy-to-implement and a general solution, unparsed patterns have the potential to enable widespread use of concrete syntax code pattern matching within many parser-based tools. There are two quite different ways to use this enabling technology.

### 5.3. Usages

First, patterns can be used in the implementation of the tool itself, to simplify various code analyses and transformations. For instance, an analysis or optimisation pass could use pattern matching to look for statements matching a given pattern, e.g., increment assignments. This can simplify the implementation especially if the tool is written in a language that does not provide any support for pattern matching, like C or Java. However, as shown in the introduction, unparsed patterns may be useful even when the tool is written in a language that does provide a form of tree pattern matching (e.g., ML): some ASTs patterns – especially verbose patterns – are more easily expressed in native syntax than in tree syntax. Thus, at least, unparsed patterns offer an alternative for the programmer to freely chose the more convenient form for each pattern.

The pattern matcher implemented in myGCC offers two programming interfaces for internal usage.

The first interface implements unparsed patterns as used above, in which pattern variables are represented by the % escaping character, followed by a variable, e.g, `%x`, `%y`, etc. Pattern variables are global variables. This interface is available using the C function:

```
bool tree_match(tree t, const char *format)
```

which takes an AST `t` and a pattern format with named pattern variables and returns true if the matching succeeded, and false otherwise.

The second interface implements unparsed patterns with unnamed pattern variables, represented by the % escaping character followed by the type of the variable: `t` for a subtree, `c` for a character, `d` for a number, e.g., `%t = %t + %t` (currently only tree variables are implemented). This interface is similar to the C library functions performing I/O such as `scanf`, and is available through the C function:

```
bool tree_scanf(tree t,const char *pat,...)
```

which takes an AST `t`, a pattern `pat` with unnamed pattern variables, and a series of variable addresses corresponding to the pattern variables occurring in format, and returns true if the matching succeeded, and false otherwise. For example:

```
tree t, x, y;
succeeded=tree_scanf(t,"%t=%t+%t",&x,&x,&y);
```

matches `t` if it represents an increment, that is, if the variable on the left-hand side is the same as the first term of the addition. In case of successful match, it instantiates variable `x` to the variable or expression being incremented, and `y` to the increment.

Patterns can also be used to make the tool extensible by associating user-defined behaviour to some patterns. For example, in myGCC users can define sequences of operations that constitute bugs. As a particular case, myGCC allows searching all the code for a given pattern, by using a new option `--tree-check`. For instance, the following command will issue a warning for all the read statements whose result is unused, but otherwise compile the file as usual:

```
gcc --tree-check="read(%x, %y, %z);" foo.c
```

Indeed, because of the final semi-colon, this pattern only matches complete statements and not simple expressions consisting of a call to the function `read`. Therefore, statements using the value of the call will not be reported (which is the intended behaviour).

## 6. Related work

When comparing unparsed patterns with traditional, parsed patterns, also expressed in native syntax, the advantages of unparsed patterns include the following.

- Implementation is much simpler, almost no investment is needed. In particular, existing tools do not need to be ported to different parsing technologies.
- Implementation is completely language independent (modulo linking it with a specific unparser).
- Metavariables may appear, within a code fragment, in any position that is represented as a subtree in the AST, including types, qualifiers, etc. This is as good as classic tree matching, but is difficult to implement efficiently with parsed patterns.
- Since they are not parsed, patterns may be constructed dynamically with no performance penalty.

The limitations of unparsed patterns are as follows.

- Extra metaparentheses or typed metavariables are needed in some patterns to resolve conflicts that are not solvable by looking ahead of one lexeme. The *ES*(1) algorithm reduces this need to a limited number of well-defined situations, and signalled by the implementation as a warning (when tracing is on). Using these warnings and a fixed list of left-recursive constructs, programmers should be able to learn to correctly parenthesize the patterns.
- Ill-formed patterns are not signalled at compile time: they simply do not match any code at runtime.
- Unparsed patterns cannot be used to *build* code in native syntax. When used in code transformation tools, they can serve only for matching code (on the left side of rewriting rules). However, trees can be built using conventional AST constructors out of the subtrees selected by the left-hand sides.

Therefore, unparsed patterns are not meant to replace traditional, parsed code patterns if they are available, but rather provide a pragmatic, lightweight alternative when parsed patterns are unavailable, and would be too expensive to implement.

From the usability point of view, unparsed patterns are rather similar to regular expressions. As for regular expressions, there is no static guarantee that a pattern will match the desired data, so patterns are incrementally prototyped on a few test cases to ensure that they have the intended meaning. In spite of this limitation, which may seem rebarbative to users of (parsed) concrete syntax patterns, regular expressions are used everywhere, including in a myriad of tools for processing semi-structured text, or even source code. Therefore, there is an important potential for uses of unparsed patterns, in places where regular expressions are not powerful enough to match (multiply nestable) language constructs, and where at the same time parsed patterns are unavailable. To further help the programmer, matching with unparsed patterns can be traced using a switch in the implementation, thus showing the sequence of rewrite/inference steps executed for a given tree and pattern. Using this trace, programmers can quickly locate the source of a mismatch and fix the pattern accordingly. Summarising, unparsed patterns can be viewed as a mid-way path between regular expressions and parsed patterns: their usability is similar to those of regular expressions, while their matching power is similar to that of concrete syntax patterns.

Unparsed pattern can also be viewed as a (broad) generalization of functions such as `scanf` in C, which match a "format string" with a stream of characters and values; like in our patterns, the format string contains a mix of plain substrings and pattern variables to be filled in with the matching values. Indeed, our representation of metavariables, escaped by a '%' sign, is a tribute to this well-known C library function. However, format strings can only be used to match *flat* sequences of values, while unparsed patterns are used to match trees, containing values arbitrarily nested within language constructs expressed in concrete syntax.

The ATerm library described by van den Brand *et al.* [20] extends format strings to match trees, such as parse trees. The trees are represented as a data structure called ATerm that is particularly space efficient. ATerms have a concrete syntax similar to first-order terms and lists thereof (e.g., `[f,g([1,2]),"abc"]` is a list of ATerms). Format strings in the ATerm library consist of ATerms written in their concrete syntax but containing (typed) placeholders such as `<int>` for an integer or `<appl>` (application) for a subterm. For instance, `and(<int>,<appl>)` is a format string that can be matched with an ATerm using the library function ATmatch. At first sight, this API may seem very similar to our match function, but the fundamental difference is that the syntax of format strings is fixed—the concrete syntax of ATerms. Thus, when the ATerms are used to represent ASTs of a given object language, the ATerm patterns represent the abstract syntax for that language. In contrast, unparsed patterns are written in the *concrete* syntax of any object language; the underlying AST is mostly invisible, except for a few metaparentheses or typed metavariables that may be needed to disambiguate its structure in some cases.

The idea of matching a tree with a pattern expressed as a string without parsing the pattern was apparently first mentioned in our previous paper [19]. Combining this idea with the use of lexeme information in the AST, metaparentheses, typed metavariables and lookahead is new, and so is the study of the algorithmic complexity. Efficient algorithms for matching a tree with a pattern also represented as a tree have been known for a long time. This pattern matching problem can be solved in linear time as a particular case of unification [21] where variables may occur only in the pattern. We saw that reducing pattern matching to tree matching is either difficult to use (patterns expressed as trees) or difficult to implement in most existing tools (pattern parser). Our approach avoids both difficulties while keeping the same linear execution time. Many variants of the tree pattern matching problem have been studied. The subtree matching problem consists of deciding if a pattern matches any subtree of a subject tree. The "dictionary" version of this problem involves multiple patterns to be searched against the subject's subtrees. Some known algorithms [22] begin with a preprocessing phase that reduces both the subject tree and the pattern to their preorder strings. However, rewriting a text pattern as a preorder string requires first representing the pattern as a tree, so it cannot be used to avoid parsing the pattern. In terms of execution time, the preprocessing phase reduces the asymptotic complexity of the search, and thus ensures a very efficient search of the patterns against all subtrees. Our approach is less efficient on the problem of dictionary subtree searches.

In the last decade, XML has increasingly been adopted as a standard for representing tree-shaped data, including program ASTs as a particular case. Therefore, XML-specific tree pattern languages such as XPath are being used for matching code. Compared to XPath, unparsed patterns provide the convenience of concrete syntax, but less matching power—for instance, matching a subtree nested at an arbitrary level is not possible. XPath has been integrated in more powerful query languages such as XQuery or in transformation languages such as XSLT or CDuce [23]. Unparsed patterns can be embedded in any programming language providing a string type, eventually as a complement to tree matching mechanisms already in the language.

Native patterns [8] are completely valid code fragments, but in which some reserved names can be used as pattern variables. Actually, the reserved names are the names of non-terminals in the grammar of the language as described in its reference manual (which is supposed to be known by programmers). These non-terminal names may be suffixed by a number to denote different occurrences of the non-terminal and by $*$ or $+$ to denote lists of such non-terminals. The extended grammar of the patterns can be generated automatically from the base grammar. However, this requires expressing or porting the base grammar in a syntax definition formalism called SDF. As opposed to this constraint, unparsed patterns can be incorporated in any existing parser-based tool with minimal effort.

Scruple [6] is a generic pattern language specialised for matching source code. Scruple patterns are more general than ours, as they allow, for instance, matching lists of subtrees in a single pattern variable, or matching in the same pattern a tree and one of its subtrees nested at an arbitrary level. Execution time is not linear, as the algorithm involves backtracking. The patterns are compiled into a recursive network of automata that consume lexemes in the subject AST that correspond to its subtrees. This is rather similar to our algorithm. However, Scruple patterns are parsed by a pattern parser which extends the native parser of the subject language. Unlike our approach, the lexemes in the AST do not include lexemes in the concrete syntax (keywords, separators, etc.).[5]

An elegant language-embedding technique for metaprogramming is described by Visser [15], which allows embedding an arbitrary context-free subject language $S$ in a host context-free programming language $H$, and further using the concrete syntax of $S$ to match and build subject code fragments within $H$ programs. This technique requires defining the grammar of both languages in the SDF framework, which combines the two in a single grammar. This grammar fusion uses SDF's support for modules and user-defined grammar injection rules (that can be automated). The advantage of their approach is that syntax errors in both the host language and its subject patterns are checked by a single parser. After this combined parsing, a generic AST transformation reduces the bilingual AST to a plain $H$ AST. Hence, a metaprogram in $H + S$ is preprocessed to a program in $H$, that can be compiled with any compiler for $H$, and then linked with appropriate tree-processing libraries for using the parsed patterns. In particular, pattern matching is supported if the language or the linked libraries provide tree matching. The main inconvenience of this approach, from a practical point of view, is the porting effort of both languages to SDF. Besides, an acknowledged limitation of that work is that it cannot express context-sensitive syntax such as type identifiers in C or off-side rules in Haskell. Unparsed patterns can serve as a lightweight alternative to this approach, when rewriting the two parsers in SDF is not feasible. It requires a minimal effort to implement and is not limited to a given class of

[5] In our theoretical sections, however, we assumed, for the sake of simplicity, some tokenisation.

languages, once they are parsed to ASTs by an existing parser. Thus, unparsed patterns are meant to be easily incorporated in most existing tools. In turn, our approach does not provide support for building ASTs in concrete syntax, nor syntax checking of the patterns.

Also related to our work is the problem of pattern disambiguation, consisting in choosing among the several possible trees corresponding to a textual pattern. As we mentioned in the introduction, disambiguation is traditionally done by introducing special pattern syntax, which tends to reduce the advantages of concrete syntax patterns. A clearly better solution, used in several metaprogramming systems, consists in exploiting type information associated to the pattern variables or to the whole pattern. This technique has been applied for instance in Meta-AspectJ [24] to allow producing AspectJ code within Java metaprograms using particularly convenient concrete patterns. Patterns are parsed using a backtracking ANTLR-based parser mixing $LL(k)$ parsing and type checking. By greatly exploiting the specifics of the Java and AspectJ languages, the tool is able to infer the type of variables containing object code and automatically introduce some type conversions from basic host types to object code types when needed. However, patterns are used only for generating code, while our patterns are used only in pattern matching. Their parsing algorithm may exhibit exponential-time behaviour.

A general and language independent solution for the type-based disambiguation of patterns is described by Bravenboer *et al.* [25] Their solution consists in three phases. First, the host language code with embedded object language patterns is parsed using a scannerless GLR algorithm according to the method mentioned above [15], which returns a parse forest for each ambiguous pattern. A second phase translates each such object AST into host language code for building that AST. The disambiguation phase is done by a slightly extended version of the host language type checker, and keeps only the valid AST builders. Since the disambiguation phase sees no object code, its implementation is independent from the object language, but not from the host language. This is much like our language independent approach. Their technique itself is portable to any statically-typed language, if different AST nodes are mapped to different types in the host language.

In our framework, disambiguation is done using metaparentheses and typed metavariables. These chiefly serve to eliminate ambiguities due to some tree structure that is absent in the unparsed patterns, but can also serve to eliminate other ambiguities that would also exist in the corresponding parsed patterns. For instance, the C or Java pattern `f(%x)` may either represent a function call with a single argument (bound to variable `x`) or a function call with any number of arguments (then `x` is bound to the list of all the arguments). This ambiguity also exists with parsed patterns, where it is eliminated by specifying the type of `x` as a list of object ASTs or as a single object AST. When matched with the (greedy) $ES(1)$ algorithm, `x` will always match the AST representing the whole list of arguments. Single argument calls can be matched by introducing extra metaparentheses, yielding the pattern `f(%(%x%))`, or by typing the variable, yielding the pattern `f(%<expr>x)`.

## 7. Conclusion

We have shown how concrete syntax pattern matching can be integrated with minimal effort in any parser-based tool, written in any host language and manipulating any subject language, including multi-language tools such as legacy systems analysers or intentional programming systems. Matching concrete syntax can make the code of such tools more concise and more readable. The purely syntactic matching can be easily complemented with any other checks in the host language, due to a natural embedding of patterns as strings in the host language. Furthermore, unparsed patterns give a very simple means to make such tools extensible with user-defined behaviour. In particular, most existing tools can be made extensible with minimal effort. Extensible compilers are a particular application in which users may add their own program checks. Other possible applications may involve model checkers, program inspectors, etc. The formal model we introduced allowed us to precisely define our matching algorithms. The first makes use of backtracking and is thus of theoretical interest. The same formalism allowed us to prove its soundness and completeness with respect to the classic pattern matching. We also proved that it defines a function. The second algorithm, $ES(1)$, is of practical interest because it runs in worst-case linear time with a lookahead of one lexeme. This lookahead is naturally an approximation of the lexical context, its consequence is to lose completeness with respect to the classic pattern matching, despite being proven correct. This incompleteness can be overcome in all cases by adding metaparentheses to the pattern or by typing the metavariables. We illustrated $ES(1)$ on realistic, albeit small, examples processed by the prototype Matchbox. We also described rather in detail how the unparsed patterns have been successfully integrated into GCC.

Unparsed patterns can be improved in several respects. For instance, it would be interesting, both from a practical and theoretical point of view, to reduce even further the amount of metaparentheses and typed metavariables required to obtain a complete matching algorithm. Also, as far as expressiveness is concerned, pattern variables could be typed and could also match tokens. Moreover, a single pattern variable could match a list of elements such as found in some other matchers, even when that list of elements is not grouped as a distinct subtree. Also, it would be useful to allow for empty trees in the definition of unparsing (yielding an empty list). In terms of execution time, it is an open question whether the subtree matching problem, when using unparsed patterns, can be solved more efficiently than in quadratic time. Finally, concrete uses of dynamic patterns remain to be investigated.

## Acknowledgements

The authors wish to express their gratitude to the anonymous reviewers, whose upstanding and expert insights have greatly improved this article.

## References


[1] D. Engler, B. Chelf, A. Chou, S. Hallem, Checking system rules using system-specific, programmer-written compiler extensions, in: 4th Symposium on Operating Syst. Design and Implementation, OSDI, San Diego, USA, October 2000.
[2] G. Back, D. Engler, MJ–A System for constructing Bug-finding Analyses for Java. Technical report, Stanford University, USA, September 2003.
[3] H. Chen, D. Wagner, MOPS: An Infrastructure for examining Security Properties of Software, in: 9th ACM Conf. Comput. Commun. Security, CCS, Washington, USA, November 2002.
[4] T. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, W. Weimer, Temporal-safety proofs for systems code, in: 14th Int. Conf. Computer-aided Verif, CAV, in: LNCS, vol. 2404, Springer-Verlag, 2002.
[5] W. Griswold, D. Atkinson, C. McCurdy, Fast, flexible syntactic pattern matching and processing, in: 4th Int. Workshop on Program Comprehension, 1996.
[6] S. Paul, SCRUPLE: A Reengineer's Tool for Source Code Search, in: J. Botsford, A. Ryman, J. Slonim, D. Taylor (Eds.), in: Conf. of the Centre For Advanced Studies on Collaborative Research, Toronto, Canada, vol. 1, IBM Centre for Advanced Studies, IBM Press, 1992.
[7] K. Olender, L. Osterweil, Cecil: A sequencing constraint language for automatic static analysis generation, IEEE Trans. Softw. Eng. 16 (3) (1990).
[8] A. Sellink, C. Verhoef, Native patterns, in: Working Conf. on Reverse Eng., WCRE, IEEE Computer Society, 1998.
[9] C. De Roover, T. D'Hondt, J. Brichau, C. Noguera, L. Duchien, Behavioral similarity matching using concrete source code templates in logic queries, in: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, PEPM, ACM, New York, NY, USA, 2007, pp. 92–101.
[10] S. Burson, G.B. Kotik, L.Z. Markosian, A program transformation approach to automating software reengineering, in: 14th ACM Conf. Comput. Softw. and Applications, Chicago, USA, 1990.
[11] D. Beyer, A.J. Chlipala, T.A. Henzinger, R. Jhala, R. Majumdar, The BLAST query language for software verification, in: 6th ACM SIGPLAN Int. Conf. Principles and Practice of Declarative Prog., PPDP, Verona, Italy, ACM Press, 2004.
[12] M. Tomita, Efficient Parsing for Natural Language: a Fast Algorithm for Practical Systems, Kluwer Academic Publishers, 1985.
[13] J. Heering, P.R. Hendriks, P. Klint, J. Rekers, The Syntax Definition Formalism SDF. SIGPLAN Notices, 1992.
[14] M.G.J. van den Brand, J. Heering, P. Klint, P.A. Olivier, Compiling language definitions: The ASF+SDF compiler, ACM Trans. Prog. Lang. Syst. 24 (1999) 334–368.
[15] E. Visser, Meta-programming with concrete object syntax, in: Don Batory, Charles Consel, Walid Taha (Eds.), Generative Programming and Component Engineering, GPCE, in: Lecture Notes in Computer Science, vol. 2487, Springer-Verlag, Pittsburgh, PA, USA, 2002, pp. 299–315.
[16] E. Scott, A. Johnstone, R. Economopoulos, BRNGLR: A cubic tomita-style GLR parsing algorithm, Acta Informat. 44 (6) (2007).
[17] S. McPeak, G.C. Necula, Elkhound: A fast, practical GLR parser generator, in: Conf. Compiler Constructors, CC, April 2004.
[18] J. Earley, An efficient context-free parsing algorithm, Commun. ACM 13 (2) (1970).
[19] N. Volanschi, Condate: A proto-language at the confluence between checking and compiling, in: 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Prog., PPDP, Venice, Italy, ACM Press, 2006.
[20] M.G.J. van den Brand, H.A. de Jong, P. Klint, P. Olivier, Efficient annotated terms, Softw.- Pract. Exp. 30 (2000) 259–291.
[21] M.S. Peterson, M.N. Wegman, Linear unification, J. Comput. Syst. Sci. 16 (2) (1978).
[22] F. Luccio, A.M. Enriquez, P.O. Rieumont, L. Pagli, Exact rooted subtree matching in sublinear time. Technical Report TR-01-14, University of Pisa, Italy, 2001.
[23] D. Beyer, G. Castagna, A. Frisch, CDuce: An XML-centric general-purpose language, in: 8th ACM SIGPLAN Int. Conf. Functional Prog., ICFP, Uppsala, Sweden, ACM Press, 2003.
[24] D. Zook, S. S. Huang, Y. Smaragdakis, Generating AspectJ programs with meta-AspectJ, in: Conf. Generative Prog. and Component Eng., GPCE, in: LNCS, vol. 3286, Springer Verlag, Vancouver, Canada, 2004.
[25] M. Bravenboer, R. Vermaas, J. Vinju, E. Visser, Generalized type-based disambiguation of meta programs with concrete object syntax, in: R. Glück, M. Lowry (Eds.), 4th Int. Conf. Generative Prog. and Component Eng., GPCE, in: LNCS, vol. 3676, Springer-Verlag, 2005.