Elsevier Editorial System(tm) for Science of Computer Programming
Manuscript Draft

Manuscript Number:

Title: Theory and Practice of Unparsed Patterns for Metacompilation

Article Type: Research Paper

Keywords: pattern matching; tree pattern; code checking; metacompilation; formal methods

Corresponding Author: Dr Christian Rinderknecht,

Corresponding Author's Institution: Konkuk University

First Author: Christian Rinderknecht

Order of Authors: Christian Rinderknecht; Nic Volanschi, Ph.D.

Manuscript Region of Origin:

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. 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.

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*

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. 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.

Key words: pattern matching, tree pattern, code checking, metacompilation, formal methods

Email addresses: rinderkn@konkuk.ac.kr (Christian Rinderknecht), nic.volanschi@free.fr (Nic Volanschi).
URLs: http://konkuk.ac.kr/~rinderkn (Christian Rinderknecht), http://mygcc.free.fr (Nic Volanschi).

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 [1,2,3,4]. 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.

1.1 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 to write tree patterns in standardised languages such as XPath (and the languages embedding it, like XQuery and XSLT), and thus reuse existing tree pattern matchers.

The main advantage of expressing patterns as trees is that the implementation of pattern matching 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.

1.2 Example

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 the loop

```
for(i=0; i<100; ++i) a[i]=0;
```

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

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 *meta-variables*. 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.

1.3 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 meta-variables. 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 figure 1 (where meta-variables are escaped by the special character `%`).

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

Figure 1. A concrete-syntax pattern

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 meta-variables, 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. Sometimes, this emendation itself leads to special pattern syntaxes (such as `#stmt f(%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 GLR parsers such as Bison [2] use an extremely inefficient, exponential-time GLR algorithm, creating a new operating system process at each local ambiguity. This scheme may quickly prove infeasible when parsing even small fragments in a pattern grammar where conflicts are scattered everywhere.

Alternatively, the polynomial-time GLR parsing algorithm [12] was used to implement 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]. Even so, there is still a performance issue, because GLR parsers are typically much less efficient than LALR(1) parsers (a factor of 10 is not uncommon [17]). Also recently, 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). But as discussed above, an

[2] `http://www.gnu.org/software/bison/`

extended pattern grammar is highly ambiguous (unless we severely restrict the patterns), so then GLR parsing time would fall back to usual GLR-class performance. Therefore, GLR parsing may not be in general an efficient, production-quality, solution to the problem of parsing source code patterns. Other parsing algorithms covering all context-free grammars, such as Earley's [18], may perform better on such highly ambiguous grammars. Anyway, besides this performance issue, porting an existing parser to a different technology, like GLR, may prove quite difficult and has profound impacts on the test of the tool, which is unaffordable in the case of legacy parsers.

1.4 Our Proposal

It is safe 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

A formal model is a framework based on logic that allows one to define precisely the concepts involved in a problem and its solution. 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. Functions here are elementary functions over sets of objects, called *terms*, so the implementor can translate them into functions, procedures, methods, processes etc., depending on the target. Let us review the different kinds of terms in our formal model.

- *Integers* are the familiar numbers.
- *Variables* are names, e.g., x, y or z. We assume that there exists a denumerable infinity of them.
- *Tuples* are finite and ordered sets of terms. Each term in a tuple is called a *component*. Tuples can be nested, as in $(a, (b, c), d)$. The empty tuple is $()$.
- *Lists* are possibly empty series of terms, called *items*. The first item is called the *head* and the remaining sub-list the *tail*. Here, 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, like writing $[a, b, c]$ instead of $[a \mid [b \mid [c \mid [\,]]]]$. It is also convenient sometimes to distinguish several items past the head, and write $[a, b \mid l]$ instead of $[a \mid [b \mid l]]$. Notice also that, from an algorithmic point of view, the lists of our model are *stacks* (the head of the list corresponds to the top of the stack).
- *Constructors* are names which can be associated with a tuple of terms. They act like a tag for a tuple and, as such, are always written with the tuple they distinguish (the constructor precedes the tuple). For example $c(1, d(n))$ is a term constructed with constructors c and d. It is possible for the tuple to be empty. For example, $c(1, d())$ contains the constructor d applied to (that is, tagging) the empty tuple. Note that, in order to distinguish constructors from variables, we always retain the parentheses, like $d()$ (whereas d alone is a variable). The number of components of a constructor c is called *arity* and noted $A(c)$.

It is useful to graphically represent terms as *trees*. Constructors then denote *nodes* and their components denote subtrees. Empty tuples, variables and integers denote *leaves*. For example $d((), e([1]))$ can be interpreted as a tree of *root* d and whose first subtree is the representation of the empty tuple () and the second corresponds to $e([1])$. Figure 2 represents the tree corresponding to $a(b(c(1), ([\,], 2)), (), d(d(x)))$, where the root is a and the leaves are 1, [ ], 2, () and x. When a tuple has no constructor, the corresponding inner node is a centred dot. Arity is constant for each kind of node, so we cannot have in the model both $c(a())$ and $c(a(), b())$. In other words, our trees are *ranked*. Never the less, for technical reasons, it is useful to have unranked trees. The way to circumvent this limitation is to encode a variable arity by an arity of 1 and a list as subtree. Then we can have both $c([1])$ and $c([1, (2)])$ and pretend that the arity of c is 1 in the former case and 2 in the latter.

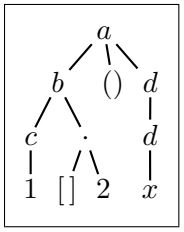


Figure 2:

Let us now use these generic terms to build the terms that model the problem at hand.

2.1 Lexemes

The first stage of a compiler consists in the lexical analysis (sometimes called scanning or lexing) of the source code: the input string of characters is analysed and segmented into the longest substrings that can be classified according to the lexicon of the input language. This is similar to transforming a line of text into words that have been checked against a dictionary. The output consists of a series of *lexemes*. Since the ambition of this study is to cope with all programming languages, we shall make no assumptions on the nature of the lexemes. We shall often designate a lexeme by l and their (unspecified) set will be noted $\mathcal{L}$.

2.2 Concrete-Syntax versus Abstract-Syntax Trees

The second stage of a compiler is the syntax analysis, also known as *parsing*, of the lexeme stream. This is akin to checking whether a series of words makes up a correct sentence, with respect to a given grammar, like the English grammar. The result of such a process is a tree called the *concrete-syntax tree*, or *parse tree*, capturing the structure of the analysed sentence in terms of the language grammar. In practice, this tree is usually not built entirely and lives in the analysis stack of the parser for the time of the processing. Instead, the programmer often chooses to output a tree, called *abstract-syntax tree* (AST),

which also captures the structure of the input program, with or without references to the grammatical rules. The inner nodes of the parse tree denote non-terminal symbols of the grammar, as usually specified in Backus-Naur Form or an extended version of it, and the leaves are the lexemes of the input. In contrast, the AST usually does not include the lexemes (see our example in the introduction) and the nodes denote kinds of constructs with or without reference to the grammar. Therefore, the converse of parsing, called *unparsing*, is naturally defined on the parse tree. On the one hand, the parse tree is not available as a data structure and, on the other hand, the AST can be arbitrary. Fortunately, the AST is often quasi isomorphic to the parse tree or is a compact version of it, which makes unparsing from the AST workable in practice. Therefore, without loss of generality, we define the unparsing inductively on the structure of the parse tree, which renders the unparsing dependent only on the grammar.

Let $\mathcal{C}$ be the set of non-terminal symbols in the grammar of the object language. By definition of parsing, these symbols are the constructors of the parse tree and each constructor has a strictly positive arity. In general, we write $c(f)$, where f is the list of the components of constructor $c \in \mathcal{C}$. By definition, the leaves of the parse trees are lexemes [3], 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 $\mathcal{H}_{n+1} = \mathcal{H}_n \cup \{c([h_1, h_2, \ldots, h_{A(c)}]) \mid c \in \mathcal{C}, h_i \in \mathcal{H}_n\}$, 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 ($h \in \mathcal{H}$). A *parse forest* is a list (algorithmically, a stack) of 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, formally defined here as

$$[\,] \cdot f = f \tag{1}$$
$$[h \mid f_1] \cdot f_2 = [h \mid f_1 \cdot f_2] \qquad \text{where } h \in \mathcal{H}. \tag{2}$$

For instance, let $a, b, c, d, e \in \mathcal{H}$ (that is to say, a, b, c, d and e are variables denoting unknown trees). Then

$$\begin{aligned}[a, b, c] \cdot [d, e] &= [a \mid [b, c]] \cdot [d, e] = [a \mid [b, c] \cdot [d, e]] \\ &= [a \mid [b \mid [c]] \cdot [d, e]] = [a \mid [b \mid [c] \cdot [d, e]]] \\ &= [a \mid [b \mid [c \mid [\,]] \cdot [d, e]]]\end{aligned}$$

[3] We leave out the empty word, ε.

$$= [a \mid [b \mid [c \mid [\,] \cdot [d, e]]]]$$
$$= [a \mid [b \mid [c \mid [d, e]]]] = [a, b, c, d, e]$$

2.3 Unparsed Patterns

Unparsed patterns are series of lexemes (which are expected to match the leaves of a given parse tree) and *metalexemes* (which cannot be found in the programming language) whose purpose is to control the matching process. Depending on the matching, we shall have different kinds of metalexemes, but 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 (contrast this with the lexemes in the patterns which only match leaves of the parse tree). For example, consider again figure 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 not 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 (i.e., the character string) of $\mathsf{meta}(x)$ is the concrete variable `x` escaped by `%`, i.e., `%x`. Sometimes, we shall call x a metavariable (instead of a variable), when it should be $\mathsf{meta}(x)$ instead. Formally, an unparsed pattern, noted $\overline{p}$, is a list of lexemes and metalexemes. The set of unparsed patterns is noted $\overline{\mathcal{P}}$. (The line over p and $\mathcal{P}$ evokes the flatness of the structure.)

2.4 Substitutions

A *substitution* σ 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 $\mathcal{H}$. 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 σ by the binding $x \mapsto t$, as defined by

$$(\sigma \oplus x \mapsto t)(y) \triangleq \begin{cases} t & \text{if } x = y \\ \sigma(y) & \text{otherwise} \end{cases} \tag{3}$$

Let us extend updates to substitutions as follows:

$$(\sigma_1 \oplus \sigma_2)(y) \triangleq \begin{cases} \sigma_2(y) & \text{if } y \in dom(\sigma_2) \\ \sigma_1(y) & \text{otherwise} \end{cases} \tag{4}$$

Let us define the inclusion between substitutions as

$$\sigma_1 \subseteq \sigma_2 \iff \forall x \in dom(\sigma_1).(\sigma_1(x) = \sigma_2(x)) \tag{5}$$

Interpreting substitutions as sets of bindings, $\sigma_1 \subseteq \sigma_2$ simply means the inclusion of the sets associated with the respective substitutions. We shall note $\sigma_\varnothing$ the empty substitution (empty domain). Notice also that the inclusion is transitive:

$$(\sigma_1 \subseteq \sigma_2) \wedge (\sigma_2 \subseteq \sigma_3) \Rightarrow \sigma_1 \subseteq \sigma_3 \tag{6}$$

2.5 Rewrite Systems

We shall use different frameworks for defining the matching function, avoiding unnecessary complexity. Perhaps the simplest modelling technique consists in defining a *rewrite system* [4] [20], i.e., a finite set of rewrite rules. A rewrite rule is a relation between two terms (made up from the formal model) and is always printed as a kind of arrow symbol. The intended interpretation of the relation is that the left-hand side can be replaced by the right-hand side in any context, defining a computation. The factorial function fact can be defined as the ordered system [5]

$$\begin{aligned} \mathsf{fact}(0) &\to 1 \\ \mathsf{fact}(n) &\to n \times \mathsf{fact}(n-1) \end{aligned}$$

Consider now a more sophisticated (unordered) system

$$\begin{aligned} \mathsf{append}(\mathsf{nil}(), s) &\to s \\ \mathsf{append}(\mathsf{push}(h, t), s) &\to \mathsf{push}(h, \mathsf{append}(t, s)) \end{aligned}$$

defining the function append which appends a stack to another; nil and push are constructors: $\mathsf{nil}()$ denotes the empty stack and $\mathsf{push}(h, t)$ is the stack denoted by t on top of which the item h has been pushed. Note that (1) the recursion on the second rule allows this system to handle stacks of all sizes; (2) the nature of the items in the stacks does not matter, enabling a generic (sometimes called polymorphic) function definition (the second stack, s, does not even need to be a stack!); (3) the function append is equivalent to the catenation $(\cdot)$ defined by equations (1) and (2): set $[\,] \triangleq \mathsf{nil}()$ and $[h \mid t] \triangleq \mathsf{push}(h, t)$, then $\mathsf{append}(s_1, s_2) = s_1 \cdot s_2$.

[4] More precisely, an orthogonal rewrite system.

[5] The rules are tried on the current term to be rewritten starting from the first and the first matching left-hand side selects the rule to be applied.

There is another interesting concept related to the rewrite systems. When there is no overlapping of the left-hand sides of the rewrite rules, the system is said to be *syntax-directed*, because choosing one rule rather than another is a decision taken only by considering the left-hand sides, not the future success or failure of the alternatives. A syntax-directed system implies that the rewrite relation is actually a function, thus so is the transitive closure. (Note that a system which is not syntax-directed can define a function.) From an implementor's point of view, this means that there is no need to order the rules or to program a backtracking mechanism.

2.6 Inference Systems

An inference system is a finite set of *inference rules*. Inference rules constitute a framework for defining relations which is more expressive than rewrite systems. For example, they allow a rewrite step to be taken only if some conditions hold, e.g., if some other rewrite step is valid (recursively). Another improvement is that, whilst rewrite systems define a binary relation, inference rules can define any kind of statement or proposition, called, in this context, *judgement*. An inference rule is a logical implication whose general form is as follows:

$$P_1 \wedge P_2 \wedge \ldots \wedge P_n \Rightarrow C$$

where $\wedge$ stands for the boolean conjunction ('and'). The more intuitive presentation of an inference rule is

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

The propositions P_i are called *premises* and C is the *conclusion* (it is the judgement defined by the rule). When premises are lacking, then C is called an *axiom* and is simply written C (without the horizontal line). An axiom is unconditionally true, that is, it is true by definition, but only in the context of the system it belongs to (as such, it is often called *non-logical axiom*).

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.

Rules and axioms may contain variables that are not explicitly quantified (universally or existentially, i.e., $\forall$ or $\exists$). In this case, they are implicitly considered as universally quantified at the rule level. For example,

$$\mathsf{even}(0) \qquad \frac{\mathsf{odd}(n)}{\mathsf{even}(n+1)} \qquad \frac{\mathsf{even}(n)}{\mathsf{odd}(n+1)}$$

is equivalent to the following propositions:

$$\begin{gathered}\mathsf{even}(0)\\ \forall n.(\mathsf{odd}(n) \Rightarrow \mathsf{even}(n+1))\\ \forall n.(\mathsf{even}(n) \Rightarrow \mathsf{odd}(n+1))\end{gathered}$$

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 (a shorthand for 'proof tree' in some contexts), is then the root of the tree, which is traditionally set at the bottom of the page.

For instance, figure 3 shows the proof that 3 is odd in the previous system. The root (the theorem) is $\mathsf{odd}(3)$ and the leaf (the axiom) is $\mathsf{even}(0)$. Note that, since the rules here only have one premise, the proof tree is actually a proof list (i.e., a degenerate tree). This was the deductive view of the inference system. The inductive view provides an algorithm to determine whether an integer is even or odd, e.g.,n is odd if the term $\mathsf{even}(n-1)$ can be derived.

$$\cfrac{\cfrac{\cfrac{\mathsf{even}(0)}{\mathsf{odd}(1)}}{\mathsf{even}(2)}}{\mathsf{odd}(3)}$$

Figure 3:

The concept of syntax-directed system, which was introduced for the rewrite systems, also applies to inference systems: an inference system is syntax-directed if the conclusions have non-overlapping shapes.

In the next section, we introduce a pattern matching based on unparsed patterns, which has pleasant properties but requires backtracking, thus too costly in practice. It is nevertheless simple and, as such, serves as an introduction to the formal specifications and proof techniques we use later on.

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 figure 4b,

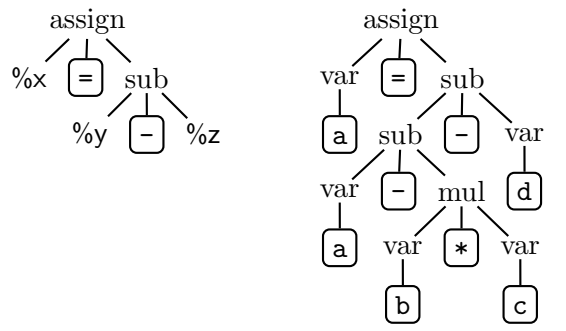
then parse the pattern using an extended parser into the parse tree in figure 4a, and then match the latter against the former. As a result, all the metavariables are correctly bound with respect to the grammar in the substitution $\{\texttt{x} \mapsto \texttt{"a"}, \texttt{y} \mapsto \texttt{"a-b*c"}, \texttt{z} \mapsto \texttt{"d"}\}$.

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 yields both the substitution $\{\texttt{x} \mapsto \texttt{"a"}, \texttt{y} \mapsto \texttt{"a-b*c"}, \texttt{z} \mapsto \texttt{"d"}\}$, which is correct, and $\{\texttt{x} \mapsto \texttt{"a"}, \texttt{y} \mapsto \texttt{"a"}, \texttt{z} \mapsto \texttt{"b*c-d"}\}$, 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 manipulate the matched subtrees, which is a quite 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 be partially matched against the current unparsed forest or the latter should be further unparsed.

These two alternatives are sometimes possible on the same configuration. The first one, i.e., partial matching, can be tried first and if it leads to a failure, the second one, 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



(a) Tree-like pattern (b) Source parse tree

Figure 4. Tree pattern matching `a = a - b*c - d`

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.[6] Thirdly, given an initial empty substitution σ, the algorithm nondeterministically 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 σ.
   (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[7] or unbound in σ, then the remaining pattern is matched against the remaining forest, with σ 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. If the final pattern is empty, then it is a success; otherwise it is a failure. 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 Definitions

3.1.1 Parse Trees and Unparsed Patterns

Trees have been formally defined in section 2. Unparsed patterns will be noted $\overline{p}$, and the set of unparsed patterns is inductively defined as the smallest set

[6] In practice, this is not necessary because the leaves of the source parse tree contain all the lexical information needed to realise a match step (read further). But, as much as the theory of unparsed patterns is concerned, it is simpler to assume some tokenisation of the pattern, hence avoiding the use of regular expressions in the formal model or the algorithm itself.

[7] Unparsed patterns are not linear, i.e., a metavariable can occur more than once.

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

Figure 5. Substitutions on unparsed patterns

$\overline{\mathcal{P}}$ 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 $[\mathsf{meta}(x) \mid \overline{p}] \in \overline{\mathcal{P}}$.

3.1.2 Substitutions and Closed Patterns.

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 figure 5. The first equation ($\stackrel{1}{=}$) means that the substitution on the empty pattern is always the empty forest. The second equation ($\stackrel{2}{=}$) 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 ($\stackrel{3}{=}$) specifies that the substitutions always leave lexemes unchanged.

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

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

Figure 6. A backtracking pattern matching

3.1.3 Pattern Matching

Pattern matching is defined by the inference system in figure 6, where the rules are unordered. 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 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 σ 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 σ 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.

3.1.4 Example

Consider the source tree in figure 4b and the unparsed pattern `%x = %y - %z`. The tree of all possible choices for the rules of the backtracking system in figure 6 is given in figure 7. The unique successful trace is in bold.

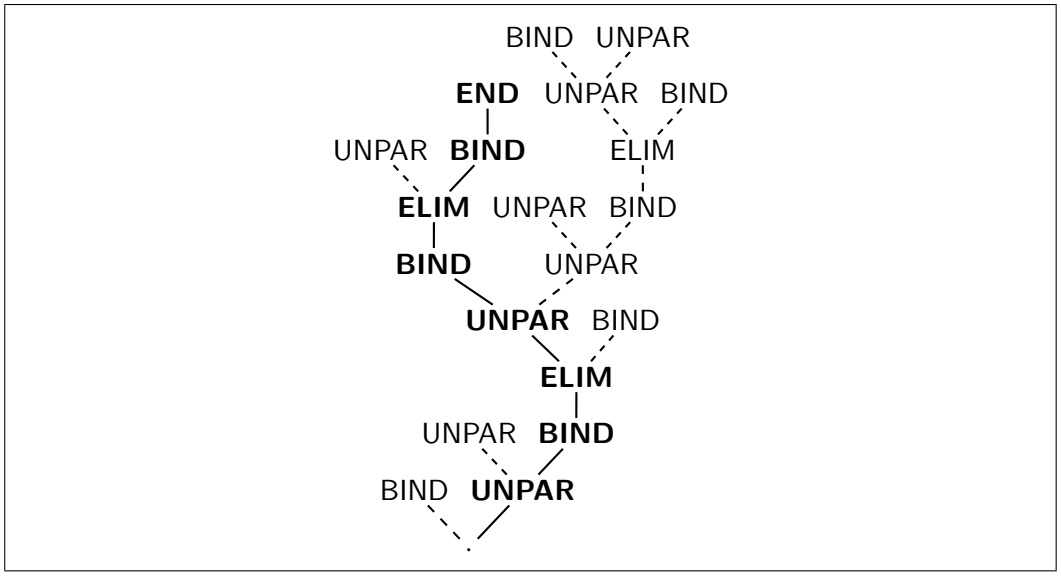


Figure 7. Matching `%x = %y - %z` (in bold)

3.1.5 Properties

Some important questions raised by this formal definition are: Is it an algorithm? (i.e., does it terminate?) Is it deterministic? With respect to what is it sound and/or complete?

3.1.6 Termination

As said before, termination is entailed by the observation that either the pattern length (i.e., the number of lexemes and metalexemes), or the forest size (i.e., the number of nodes) strictly decreases after any rule is applied to any given configuration.

3.1.7 Determinacy

It is worth wondering whether the inference system used for defining the pattern matching is a function, i.e., if it computes, for a given configuration, a unique substitution (in case of success) or none (in case of failure). This property is called *determinacy* and the concern arises whenever the system if not syntax-directed. Here, it is possible to prove that our system indeed defines a function. See appendix A.3.

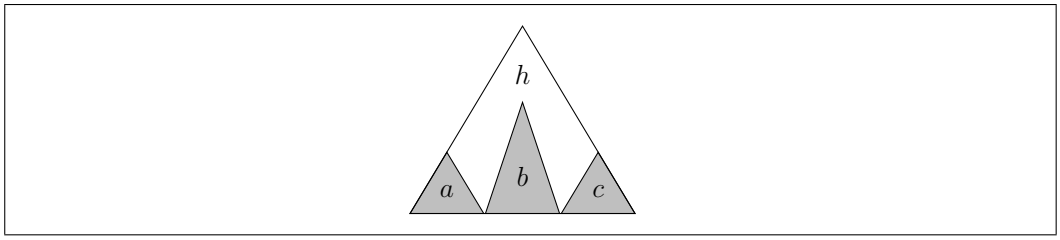


Figure 8. Closed-tree matching $[a, b, c] \sqsubseteq h$

3.1.8 Closed-Tree Matching.

Soundness and completeness are properties which suppose another level of specification, usually more abstract, or at the same conceptual level. In any case, this other description should provide a concept of matching which is more 'natural' or obvious than the one under consideration. Thus, by proving that the latter is sound with respect to the former, it is proved that it is included in it. In other words, when an algorithm is proved sound with respect to a specification, it means that all outputs satisfy the specification — thus are correct.

The intuitive notion we need here is what it means for a closed tree to match a parse tree, i.e., when a parse tree (thinking of a pattern in which all metavariables have been substituted) is contained in another parse tree (the source program). This is the *closed-tree matching*, which is a special case of the classic tree matching (in the latter, the pattern tree may embed metavariables and is called a pattern). This way, it becomes possible to compare the expressive power of the backtracking pattern matching with respect to the more familiar tree matching, used by many existing tools.

Informally, let us say that a tree h_1 matches a tree h_2 if and only if h_1 is included somewhere at the bottom of h_2, as shown in figure 8. Let us note $h_1 \sqsubseteq h_2$ the relation 'The tree h_1 matches the tree h_2.' [8] 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 figure 9. We give now a logical (or deductive) reading of the rules.

[8] The symbol $\sqsubseteq$ hints at the fact that this relation is a special case of tree inclusion, where the leaves of the trees must coincide.

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

Figure 9. Closed-forest matching

Rule ONE states that if the forest f matches a forest made of a single tree h, then it matches h. (This allows to relate a closed forest and a single tree.)

Axiom EMP says that the empty forest always matches the empty forest.

Rule EQ specifies that if a non-empty forest f_1 matches another non-empty forest (possibly the same), then the forest $[h \mid f_1]$ matches the forest $[h \mid f_2]$, where h is a tree (possibly a lexeme).

Rule SUB states that if a forest f matches the catenation of forests f_1 and f_2, such that the constructor c can have f_1 as direct subtrees, then f matches $[c(f_1) \mid f_2]$ (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 be the conclusion of EQ).

Remark 1 *There are other ways to define closed-tree matching, such as asserting the existence of a certain morphism between two trees, but our approach is more direct. It does not matter that it is very inefficient (for instance, consider that, in rule EQ, the equality of two trees must be checked), since it does not serve the purpose of an implementation but of a reference for the backtracking algorithm, in order to allow the definition of its soundness and completeness.*

3.1.9 Soundness

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

Theorem 2 (Soundness)

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

PROOF 2 (Soundness).
Let $\aleph(\overline{p}, f, \sigma)$ be the proposition

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

Then $\aleph(p, [h], \sigma)$ is equivalent to the soundness property (by rule ONE). Firstly, let us assume that

$$\langle \overline{p}, f\rangle \twoheadrightarrow \sigma \tag{7}$$

is true (otherwise the theorem is trivially true). This means that there exists a pattern-matching derivation Δ 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 Δ (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 Δ 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 Δ ends with ELIM.

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

where, since we assumed (7),
(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:

$$\sigma[\![\overline{p}']\!] \sqsubseteq f'. \tag{8}$$

Besides, we have

$$\begin{array}{lr}
\sigma[\![\overline{p}]\!] = [\![[l \mid \overline{p}']]\!] & \text{by 2b}\\
\quad \overset{3}{=} [l \mid \sigma[\![\overline{p}']\!]] & \text{cf. Fig. 5}\\
\quad \sqsubseteq [l \mid f'] & \text{by (8) and EQ (Fig. 9)}\\
\quad = f & \text{by 2a}\\
\sigma[\![\overline{p}]\!] \sqsubseteq f. & (9)
\end{array}$$

As a conclusion, the induction hypothesis and (7) imply (9), so $\aleph([l \mid \overline{p}'], [l \mid f'], \sigma)$ holds.

(3) Case where Δ ends with BIND.

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

where, because we assumed (7),
(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:

$$\sigma' [\![\overline{p}']\!] \sqsubseteq f'. \tag{10}$$

We also have

$$\begin{aligned}
\sigma(x) &\triangleq (\sigma' \oplus x \mapsto t)(x) && \text{by 3d} \\
&= t && \text{by (3)} \\
\sigma(x) &= t && (11) \\
\sigma' [\![\overline{p}']\!] &= (\sigma' \oplus x \mapsto t)[\![\overline{p}']\!] && \text{by 3c and 8} \\
\sigma' [\![\overline{p}']\!] &= \sigma [\![\overline{p}']\!] && \text{by 3d} \quad (12)
\end{aligned}$$

Remark 3 *The lemma 8, about minimal substitutions, can be applied because the meta-parsed patterns $\overline{p}$ it is defined upon are a strict superset of the unparsed patterns $\overline{p}$ of this section.*
Besides, we have

$$\begin{aligned}
\sigma [\![\overline{p}]\!] &\triangleq \sigma [\![[\mathsf{meta}(x) \mid \overline{p}']]\!] && \text{by 3b} \\
&\overset{2}{=} [\sigma(x) \mid \sigma [\![\overline{p}']\!]] && \text{cf. Fig. 5} \\
&= [t \mid \sigma [\![\overline{p}']\!]] && \text{by (11)} \\
&= [t \mid \sigma' [\![\overline{p}']\!]] && \text{by (12)} \\
&\sqsubseteq [t \mid f'] && \text{by (10) and } \mathsf{EQ} \text{ (Fig. 9)} \\
&\triangleq f && \text{by 3a} \\
\sigma [\![\overline{p}]\!] &\sqsubseteq f. && (13)
\end{aligned}$$

In the end, the induction hypothesis and (7) imply (13), so $\aleph([\mathsf{meta}(x) \mid \overline{p}'], [t \mid f'], \sigma)$ holds.

(4) Case where Δ ends with UNPAR.

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

where, since we assumed (7),
(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:

$$\begin{aligned} \sigma[\![\overline{p}]\!] &\sqsubseteq f_1 \cdot f_2. \\ &\sqsubseteq [c(f_1) \mid f_2] && \text{by SUB (Fig. 9)} \\ &= f && \text{by 4a} \\ \sigma[\![\overline{p}]\!] &\sqsubseteq f. && (14) \end{aligned}$$

As a conclusion, the induction hypothesis and (7) imply (14), so $\aleph([l \mid \overline{p}'], [c(f_1) \mid f_2], \sigma)$ holds. □

3.1.10 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 matching are computed by our pattern matching. Formally, this can be expressed as follows.

Theorem 4 (Completeness)

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

Proof 4 (Completeness). By structural induction. See appendix A.2.

3.1.11 Compliance

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

Corollary 5 (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 matching coincides exactly with the backtracking algorithm.

PROOF 5 (Compliance). The way from left to right is the completeness. From the soundness, it comes that

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

The minimality lemma 8 implies then $\sigma [\![\overline{p}]\!] \sqsubseteq h$. □

3.1.12 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 figure 4b, 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 metaparenthesised 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 metaparenthesised 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 furthermore syntax-directed.

4 Algorithm $ES(1)$

4.1 Definitions

4.1.1 Parse Trees and Unparsed Patterns

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

4.1.2 Pattern Trees and Metaparsed Patterns

Given $\overline{p}$, the first task is to check whether $\overline{p} \in \overline{\mathcal{P}}$. This is done by means of the metaparsing function metaparse, 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 $[\mathsf{pat}(p_1) \mid p_2] \in \mathcal{P}$.

The metaparsing function metaparse is formally defined by means of an ordered rewrite system defining jointly three other functions, shift, reduce and check. See figure 10, where $\overline{p}$ is an unparsed pattern, p is a metaparsed pattern, $\tilde{p}$ is, in general, a temporary mix of both, l is a lexeme and e is a lexeme or a

metalexeme. [9]

Rule ($\xrightarrow{a}$) handles the empty pattern. Rules ($\xrightarrow{b}$) and ($\xrightarrow{c}$) express that metavariables and lexemes are directly copied to the result in the same order as in the input.

Rule ($\xrightarrow{d}$) handles an opening metaparenthesis: it is pushed into an auxiliary analysis stack passed as second argument to the call of function shift. The purpose of this function is to metaparse the input with the additional knowledge that an opening metaparenthesis was found when it was first called — in other words, the analysis is performed inside metaparentheses. Thus the goal is reached when this supplementary stack contains only one pattern tree, since it must correspond to a pair of metaparentheses at the top-level in the input pattern. In this case, i.e., rule ($\xrightarrow{e}$), the pattern tree is copied to the output and the metaparsing is resumed at the top-level with metaparse.

Rule ($\xrightarrow{g}$) copies anything from the input pattern to the auxiliary stack, except closing metaparentheses because this case triggers a reduction in rule ($\xrightarrow{f}$). Indeed, finding mrp means that the shortest prefix of the auxiliary stack ending with an opening metaparenthesis must be replaced by the corresponding pattern tree and then shifting is resumed (i.e., metaparsing but not at the top-level). Rule ($\xrightarrow{i}$) pops any item, except mlp, from the auxiliary stack and pushes it to a temporary stack which contains the first-level subtrees of the pattern tree under construction. If an opening metaparenthesis is reached, in rule ($\xrightarrow{h}$), it means that the shortest prefix to be reduced has been found, therefore a pattern tree (always of root pat) is produced on top of the remaining auxiliary stack. Since the sequence `%(%)` is a metasyntax error, we must check that rule ($\xrightarrow{h}$) does not build a pattern tree without subtrees: this is the purpose of function check and the unique rule ($\xrightarrow{j}$).

As usual with rewrite systems, errors are not specified: a metaparsing error occurs if a term is reached and it cannot be rewritten furthermore. A tiny example of metaparsing is given next.

[9] The reader puzzled by the apparent complexity of the definition must remember that it uses stacks as the only data structure. It is recommended to skip it upon first reading. Note also that the given rewrite system is straightforward to implement in a functional language.

$$
\begin{aligned}
\mathsf{metaparse}([\,]) &\xrightarrow{a} [\,] \\
\mathsf{metaparse}([\mathsf{meta}(x) \mid \overline{p}]) &\xrightarrow{b} [\mathsf{meta}(x) \mid \mathsf{metaparse}(\overline{p})] \\
\mathsf{metaparse}([l \mid \overline{p}]) &\xrightarrow{c} [l \mid \mathsf{metaparse}(\overline{p})] \\
\mathsf{metaparse}([\mathsf{mlp} \mid \overline{p}]) &\xrightarrow{d} \mathsf{shift}(\overline{p}, [\mathsf{mlp}]) \\
\mathsf{shift}(\overline{p}, [\mathsf{pat}(p)]) &\xrightarrow{e} [\mathsf{pat}(p) \mid \mathsf{metaparse}(\overline{p})] \\
\mathsf{shift}([\mathsf{mrp} \mid \overline{p}], \tilde{p}) &\xrightarrow{f} \mathsf{shift}(\overline{p}, \mathsf{reduce}(\tilde{p}, [\,])) \\
\mathsf{shift}([e \mid \overline{p}], \tilde{p}) &\xrightarrow{g} \mathsf{shift}(\overline{p}, [e \mid \tilde{p}]) \\
\mathsf{reduce}([\mathsf{mlp} \mid \tilde{p}], p) &\xrightarrow{h} \mathsf{check}([\mathsf{pat}(p) \mid \tilde{p}]) \\
\mathsf{reduce}([e \mid \tilde{p}], p) &\xrightarrow{i} \mathsf{reduce}(\tilde{p}, [e \mid p]) \\
\mathsf{check}([\mathsf{pat}([e \mid p]) \mid \tilde{p}]) &\xrightarrow{j} [\mathsf{pat}([e \mid p]) \mid \tilde{p}]
\end{aligned}
$$

Figure 10. Metaparsing of patterns in $ES(1)$ (ordered rules)

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 by metaparse is $[l_1,\mathsf{pat}([l_2,l_3,\mathsf{pat}([\mathsf{meta}(x), l_4])])]$, which contains the pattern tree of figure 11.

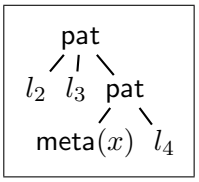


Figure 11: A pattern tree

Remark 6 *Thereupon we write 'pattern' instead of 'metaparsed pattern'.*

4.1.3 Substitutions and Closed Patterns

Let us extend substitutions to cope with patterns, not just metavariables, as we did about the backtracking algorithm in subsection 3.1. Closed patterns are defined inductively as the smallest set $\mathring{\mathcal{P}}$ such that [10]

- $[\,] \in \mathring{\mathcal{P}}$;
- 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 figure 12. It is clear that the result of such substitutions is always a closed pattern.

[10] The circles above are meant to suggest self-containment.

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

Figure 12. Substitutions on metaparsed patterns

4.1.4 Closed-Tree Matching

A relationship ($\sqsubseteq$) is needed to capture the concept of a closed-tree matching a parse tree, just as we did about the backtracking algorithm in subsection 3.1. 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 figure 13.

Rule ONE states that a pattern matches a tree if the same pattern matches the corresponding singleton forest. This allows to relate a pattern and a tree.

Rule EMP expresses that 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 matches the forest if the remaining pattern $\mathring{p}$ and forest f match.

Rule PAT handles the case of a pattern tree: there is a match if the subtrees $\mathring{p}_1$ of the pattern tree match the subtrees f_1 of the forest head and if the remaining pattern $\mathring{p}_2$ matches 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 a match if the pattern matches the catenation of the subtrees f_1 of the tree in the forest and the remaining forest f_1. 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).

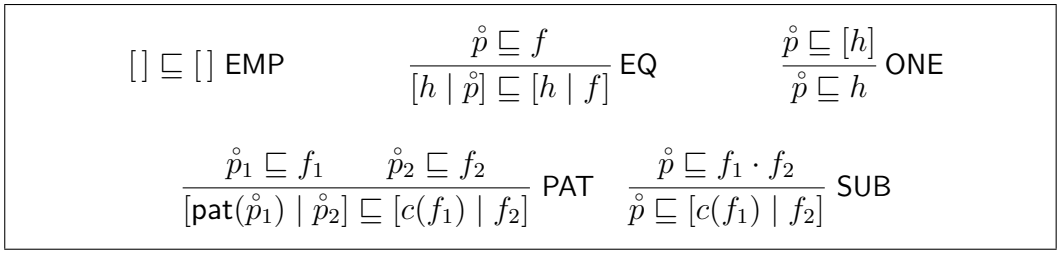


Figure 13. Closed-pattern matching for $ES(1)$(SUB is last)

4.1.5 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 the programmers usually want to bind the biggest tree. This step is the rule BIND_3 in figure 14.
  - · 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 BIND_1 in figure 14. 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).
  - · When the second tree of the forest is not a leaf. This is the rule BIND_2 in figure 14.

  The substitution σ resulting from the recursive call (among the premises) must be compatible with the binding $x \mapsto t$, i.e., $\sigma \subseteq \sigma \oplus x \mapsto t$.
- **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, σ_1 and σ_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 UNPAR_1 in figure 14.

· Otherwise, the new configuration is rewritten. This is rule UNPAR_2 in figure 14, which must be considered last, because its conclusion can overlap with the one of BIND rules or UNPAR_1.

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

$$\frac{\langle p, f \rangle \twoheadrightarrow \sigma \qquad \sigma \subseteq \sigma \oplus x \mapsto t}{\langle [\mathsf{meta}(x), l \mid p], [t, l \mid f] \rangle \twoheadrightarrow \sigma \oplus x \mapsto t} \ \mathsf{BIND}_1$$

$$\frac{\langle p, [t_2 \mid f] \rangle \twoheadrightarrow \sigma \qquad \sigma \subseteq \sigma \oplus x \mapsto t_1}{\langle [\mathsf{meta}(x) \mid p], [t_1, t_2 \mid f] \rangle \twoheadrightarrow \sigma \oplus x \mapsto t_1} \ \mathsf{BIND}_2$$

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

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

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

Figure 14. Pattern matching $ES(1)$ (UNPAR_2 is last)

4.2 Example

Consider the source tree in figure 4b and the unparsed pattern `%x = %y - %z`, as well as the tree of all possible choices with the backtracking algorithm in figure 7. Here, this tree is reduced to a list because there is no choice between rules (they are ordered). So we have the trace UNPAR_2, BIND_1, UNPAR_2, BIND_1 and BIND_3.

4.3 Properties

4.3.1 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 UNPAR_1). The worst-case complexity follows: it is linear in the number of nodes in the parse forest.

4.3.2 Soundness

Theorem 7 (Soundness)

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

Proof 7 (Soundness).

Let $\aleph(p, f, \sigma)$ be the proposition

$$\textit{If } \langle p, f\rangle \twoheadrightarrow \sigma \textit{ then } \sigma[\![p]\!] \sqsubseteq f.$$

Then $\aleph(p, [h], \sigma)$ is equivalent to the soundness. Let

$$\langle p, f\rangle \twoheadrightarrow \sigma \tag{15}$$

(otherwise the theorem is trivially true). This means that there exists a pattern-matching derivation Δ 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 Δ (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 Δ.

(1) Case where Δ 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 Δ.

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

where, since we assumed (15),
(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

$$\sigma[\![p']\!] \sqsubseteq f'. \tag{16}$$

Besides, we have

$$\begin{aligned}
\sigma[\![p]\!] &= [\![[l \mid p']]\!] && \text{by 2a} \\
&\overset{3}{=} [l \mid \sigma[\![p']\!]] && \text{cf. Fig. 12} \\
&\sqsubseteq [l \mid f'] && \text{by (16) and } \mathsf{EQ} \text{ (Fig. 13)} \\
&= f && \text{by 2b} \\
\sigma[\![p]\!] &\sqsubseteq f. && (17)
\end{aligned}$$

As a conclusion, the induction hypothesis and (15) imply (17) in this case, i.e., $\aleph(p, f, \sigma)$ holds.

(3) Case where BIND_1 ends Δ.

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

where, since we assumed (15),
(a) $\sigma' \subseteq \sigma' \oplus x \mapsto t$,
(b) $p \triangleq [\mathsf{meta}(x), l \mid p'']$,
(c) $f \triangleq [t, l \mid f'']$,
(d) $\sigma \triangleq \sigma' \oplus x \mapsto t$.
Let us assume that the induction hypothesis holds for the premise of BIND_1, i.e., $\aleph(p'', f'', \sigma')$ holds:

$$\sigma'[\![p'']\!] \sqsubseteq f''. \tag{18}$$

From 3a and lemma 8, we draw

$$\begin{aligned}
\sigma'[\![p'']\!] &= (\sigma' \oplus x \mapsto t)[\![p'']\!] \\
&= \sigma[\![p'']\!] && \text{by 3d} \\
\sigma[\![p'']\!] &= \sigma'[\![p'']\!] \\
&\sqsubseteq f'' && \text{by (18)} \\
\sigma[\![p'']\!] &\sqsubseteq f''. && (19)
\end{aligned}$$

We have

$$\sigma(x) = (\sigma' \oplus x \mapsto t)(x) \qquad \text{by 3d}$$

$$= t \qquad \text{by (3)}$$
$$\sigma(x) = t. \tag{20}$$

Besides, we have the following equalities:

$$\begin{aligned}
\sigma[\![p]\!] &= \sigma[\![[\mathsf{meta}(x), l \mid p'']]\!] && \text{by 3b}\\
&\overset{2}{=} [\sigma(x) \mid \sigma[\![[l \mid p'']]\!]] && \text{cf. Fig. 12}\\
&\overset{3}{=} [\sigma(x), l \mid \sigma[\![p'']\!]] && \text{cf. Fig. 12}\\
&= [t, l \mid \sigma[\![p'']\!]] && \text{by (20)}\\
\sigma[\![p]\!] &= [t, l \mid \sigma[\![p'']\!]]. && (21)
\end{aligned}$$

Closed-tree matching (19) and the derivation

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

imply

$$\begin{aligned}
[t, l \mid \sigma[\![p'']\!]] &\sqsubseteq [t, l \mid f'']\\
\sigma[\![p]\!] &\sqsubseteq [t, l \mid f''] && \text{by (21)}\\
&= f && \text{by 3c}\\
\sigma[\![p]\!] &\sqsubseteq f. && (22)
\end{aligned}$$

As a conclusion, the induction hypothesis and (15) imply (22) in this case, i.e., $\aleph([\mathsf{meta}(x) \mid p'], f, \sigma)$.

(4) Case where BIND_2 ends Δ.

$$\dfrac{\dfrac{\vdots}{\langle p', [t_2 \mid f']\rangle \twoheadrightarrow \sigma'}}{\langle [\mathsf{meta}(x) \mid p'], [t_1, t_2 \mid f']\rangle \twoheadrightarrow \sigma' \oplus x \mapsto t_1}$$

where, since we assumed (15),
(a) $p \triangleq [\mathsf{meta}(x) \mid p']$,
(b) $f \triangleq [t_1, t_2 \mid f']$,
(c) $\sigma \triangleq \sigma' \oplus x \mapsto t_1$,
(d) $\sigma' \subseteq \sigma' \oplus x \mapsto t_1$.
Let us assume that the induction hypothesis holds for the premise of BIND_2, i.e., $\aleph(p', [t_2 \mid f'], \sigma')$:

$$\sigma'[\![p']\!] \sqsubseteq [t_2 \mid f']. \tag{23}$$

From 4d and lemma 8, we draw

$$\sigma'[\![p']\!] = (\sigma' \oplus x \mapsto t_1)[\![p']\!]$$

$$
\begin{aligned}
&= \sigma[\![p']\!] && \text{by 4c}\\
\sigma[\![p']\!] &= \sigma'[\![p']\!]\\
&\sqsubseteq [t_2 \mid f'] && \text{by (23)}\\
\sigma[\![p']\!] &\sqsubseteq [t_2 \mid f']. && (24)
\end{aligned}
$$

We have

$$
\begin{aligned}
\sigma(x) &= (\sigma' \oplus x \mapsto t_1)(x) && \text{by 4c}\\
&= t_1 && \text{by (3)}\\
\sigma(x) &= t_1. && (25)
\end{aligned}
$$

Furthermore,

$$
\begin{aligned}
\sigma[\![p]\!] &= \sigma[\![[\mathsf{meta}(x) \mid p']]\!] && \text{by 4a}\\
&\overset{2}{=} [\sigma(x) \mid \sigma[\![p']\!]] && \text{cf. Fig. 12}\\
&= [t_1 \mid \sigma[\![p']\!]] && \text{by (25)}\\
&\sqsubseteq [t_1, t_2 \mid f'] && \text{by (24), } \mathsf{EQ} \text{ (Fig. 13)}\\
&= f && \text{by 4b}\\
\sigma[\![p]\!] &\sqsubseteq f. && (26)
\end{aligned}
$$

As a conclusion, the induction hypothesis and (15) imply (26) in this case, i.e., $\aleph([\mathsf{meta}(x) \mid p'], f, \sigma)$.

(5) Case where BIND_3 ends Δ.

$$\langle [\mathsf{meta}(x)], [t] \rangle \twoheadrightarrow \{x \mapsto t\} \;\; \mathsf{BIND}_3$$

where, since we assumed (15),
(a) $p \triangleq [\mathsf{meta}(x)]$,
(b) $f \triangleq [t]$,
(c) $\sigma \triangleq \{x \mapsto t\}$.
Because BIND_3 is an axiom, we must prove $\aleph([\mathsf{meta}(x)], [t], \{x \mapsto t\})$ without relying on the induction principle:

$$
\begin{aligned}
\sigma[\![p]\!] &= \sigma[\![[\mathsf{meta}(x)]]\!] && \text{by 5a}\\
&\overset{2}{=} [\sigma(x) \mid \sigma[\![[\,]]\!]] && \text{cf. Fig. 12}\\
&\overset{1}{=} [\sigma(x) \mid [\,]] && \text{cf. Fig. 12}\\
&\triangleq [\sigma(x)]\\
&= [t] && \text{by 5}c \text{ and (3)}\\
\sigma[\![p]\!] &= [t]. && (27)
\end{aligned}
$$

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 (27), implies

$$\begin{aligned} \sigma[\![p]\!] &\sqsubseteq [t] \\ &= f && \text{by 5b} \\ \sigma[\![p]\!] &\sqsubseteq f. && (28) \end{aligned}$$

As a conclusion, the induction hypothesis and (15) imply (28), that is, $\aleph([\mathsf{meta}(x)], [t], \{x \mapsto t\})$ holds.

(6) Case where Δ ends with UNPAR_1.

$$\dfrac{\begin{array}{cc} (\Delta_1) & (\Delta_2) \\ \vdots & \vdots \\ \overline{\langle p_1, f_1\rangle \twoheadrightarrow \sigma_1} & \overline{\langle p_2, f_2\rangle \twoheadrightarrow \sigma_2} \end{array}}{\langle [\mathsf{pat}(p_1) \,|\, p_2], [c(f_1) \,|\, f_2]\rangle \twoheadrightarrow \sigma_1 \oplus \sigma_2}\ \mathsf{UNPAR}_1$$

where, since we assumed (15),
(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 Δ_1 and Δ_2 are sub-derivations of Δ, therefore the induction hypothesis holds for their conclusions, i.e., $\aleph(p_1, f_1, \sigma_1)$ is true:

$$\sigma_1[\![p_1]\!] \sqsubseteq f_1 \tag{29}$$

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

$$\sigma_2[\![p_2]\!] \sqsubseteq f_2. \tag{30}$$

Directly from the definition (4), it comes

$$\sigma_2 \subseteq \sigma_1 \oplus \sigma_2. \tag{31}$$

It follows

$$\sigma_1[\![p_1]\!] = (\sigma_1 \oplus \sigma_2)[\![p_1]\!] \qquad \text{by 6c and 8} \tag{32}$$
$$\sigma_2[\![p_2]\!] = (\sigma_1 \oplus \sigma_2)[\![p_2]\!] \qquad \text{by (31) and 8.} \tag{33}$$

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

$$\begin{aligned} \sigma[\![p]\!] &= \sigma[\![[\mathsf{pat}(p_1) \mid p_2]]\!] && \text{by 6a} \\ &\overset{4}{=} [\mathsf{pat}(\sigma[\![p_1]\!]) \mid \sigma[\![p_2]\!]] && \text{cf. Fig. 12} \\ &= [\mathsf{pat}(\sigma_1[\![p_1]\!]) \mid \sigma[\![p_2]\!]] && \text{by (32)} \\ &= [\mathsf{pat}(\sigma_1[\![p_1]\!]) \mid \sigma_2[\![p_2]\!]] && \text{by (33)} \\ &\sqsubseteq [c(f_1) \mid f_2] && \text{by (29), (30), PAT} \\ &= f && \text{by 6b} \end{aligned}$$

$$\sigma[\![p]\!] \sqsubseteq f. \tag{34}$$

As a conclusion, the induction hypothesis and (15) imply (34), that is, $\aleph([\mathsf{pat}(p_1) \mid p_2], f, \sigma)$ holds.

(7) Case where UNPAR_2 ends Δ.

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

where, since we assumed (15),
(a) $f \triangleq [c(f_1) \mid f_2]$.
Let us assume that the induction hypothesis holds for the premise of UNPAR_2, i.e., $\aleph(p, f_1 \cdot f_2, \sigma)$ is true. Therefore

$$\begin{aligned}
\sigma[\![p]\!] &\sqsubseteq f_1 \cdot f_2 \\
&\sqsubseteq [c(f_1) \mid f_2] && \text{by } \mathsf{SUB}\,(\text{Fig. 13}) \\
&= f && \text{by 7a} \\
\sigma[\![p]\!] &\sqsubseteq f. && (35)
\end{aligned}$$

As a conclusion, the induction hypothesis and (15) imply (35) in this case, i.e., $\aleph(p, f, \sigma)$. (Note that the structure of the pattern p is irrelevant here.) □

4.3.3 Completeness

This algorithm is not complete in the sense the backtracking algorithm was in section 4. Consider the unparsed pattern `%x = %y - %z - %t` and the same parse tree in figure 4b. The execution trace is UNPAR_2, BIND_1, UNPAR_2, BIND_1 and then failure, that is, no rule apply. But there exists a successful closed-tree matching 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%)`. It is

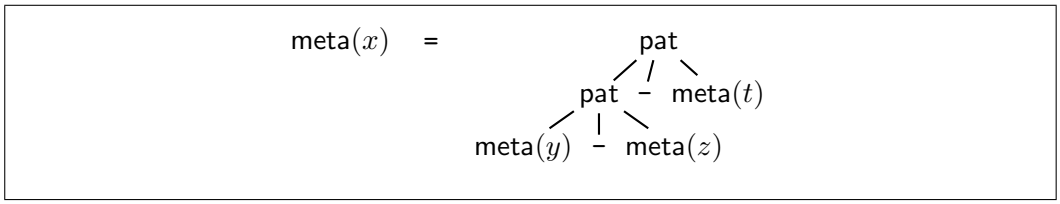


Figure 15. `%x = %(%(%y - %z%) - %t%)`

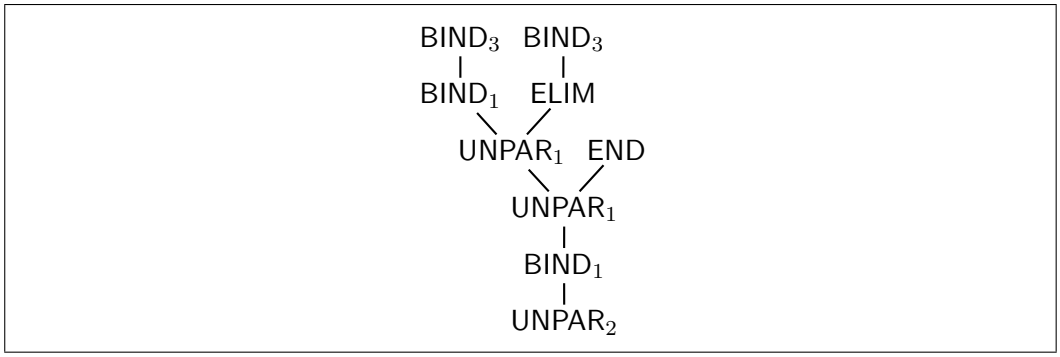


Figure 16. A simplified proof tree in $ES(1)$

metaparsed into the pattern shown in figure 15. The proof tree of the matching of this metaparsed pattern against the parse tree is given in figure 16. In figure 7, it is compulsory that (exactly) one branch from the root leads to an axiom (each branch is a logical disjunction, that is, algorithmically, a backtracking point), whereas here all branches must lead to an axiom (each branch is a logical conjunction of premises).

Of course, as a worst-case, if the pattern is fully metaparenthesised *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 metaparenthesise 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 mentionned above, the problem occurs when an instance of rule BIND_1 or BIND_2 is used and later leads to a failure. This failure could have been avoided, had UNPAR_2 been selected, perhaps several times, followed by BIND_1 or BIND_2. Assuming that it is rule BIND_1 that should be delayed after some 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%)`.

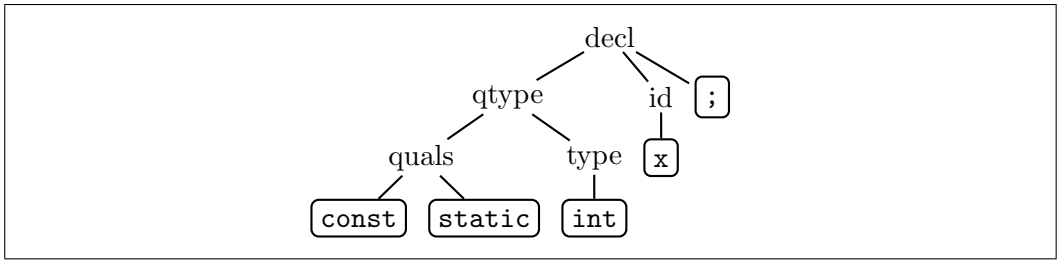


Figure 17. A simplified parse tree (no empty words)

Let us consider now that it is rule BIND_2 that should have been delayed. This leads unparsed patterns as "`%q %t %x ;`" to fail to match the parse tree given in figure 17. An unparsing step should be taken instead of a premature binding, so that `%q` matches the qualifications and `%t` the type. The way out of this is to use the unparsed pattern `%(%(%q %t%) %x;%)` instead. This situation, where two subtrees are not separated by a lexeme, is rare in practice.

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).

5.1 Implementation in myGCC

Unparsed patterns were first implemented by the second author in the context of a lightweight checking compiler called myGCC. [11] myGCC 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

[11] `http://mygcc.free.fr`

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.

5.2 Standalone Implementation

The *ES*(1) pattern matching algorithm was also implemented by the second author and is distributed as a freely available, standalone prototype called Matchbox. [12] 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

[12] `http://mypatterns.free.fr`

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.

5.3 Examples

Here are some examples of pattern matching by Matchbox. First, this is the example shown in figure 4. 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')}
```

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')}
```

The same simplified parse tree but with the unparsed pattern shown in figure 15. 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 figure 17: 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 unparsed pattern must be metaparenthesised as `%(%(%q %t%) %x;%)`.

```
~$match
  "decl(qtype(quals('const' 'static') type('int'))
   id('x') ';')"
  "%(%(%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') }
```

5.4 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.

The technique is both applicable to parsers using a separate lexical analyser and to scannerless parsers [21], as both rely on the notion of lexeme. The only difference is that scannerless parsers define lexemes using a context-free grammar while classical lexical analysers use either regular expressions or handcrafted code. From the point of view of $ES(1)$, it is irrelevant how lexemes have been recognised, once they can be found in the AST by the unparse function.

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 pat-

tern-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.

5.5 Internal 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.

5.6 External Usages

Second, patterns can 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).

5.7 Comparison with Parsed Patterns

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 stand for anything 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 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). By knowing both theoretically and practically which constructs in the language need metaparentheses, programmers should be able to correctly parenthesise 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.

6 Related Work

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, and lookahead is new, and so is the analysis of precise cases of bind/unparse conflicts and 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 [22] 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 sub-

tree of a subject tree. The "dictionary" version of this problem involves multiple patterns to be searched against the subject's subtrees. Some known algorithms [23] 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 [24]. 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 nonterminals in the grammar of the language as described in its reference manual (which is supposed to be known by programmers). These nonterminal names may be suffixed by a number to denote different occurrences of the nonterminal and by * or + to denote lists of such nonterminals. 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.). [13]

An elegant language-embedding technique for metaprogramming is described by Visser [15], which allows the 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 offside 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 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 [25] 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 generat-

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

ing 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.* [26] 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. 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%))`.

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.

We illustrated $ES(1)$ on realistic, albeit small, examples procesed 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 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.

A Appendix

A.1 Minimal substitution

A.1.1 Substitutions

A substitution on a pattern is a function which replaces all the metavariables in the pattern by abstract syntax tree. It is formally defined as

$$\sigma[\![[\,]]\!] \stackrel{1}{=} [\,]$$

$$\sigma[\![[\mathsf{meta}(x) \mid \overline{p}]]\!] \stackrel{2}{=} [\sigma(x) \mid \sigma[\![\overline{p}]\!]]$$
$$\sigma[\![[l \mid \overline{p}]]\!] \stackrel{3}{=} [l \mid \sigma[\![\overline{p}]\!]]$$
$$\sigma[\![[\mathsf{pat}(\overline{p}_1) \mid \overline{p}_2]]\!] \stackrel{4}{=} [\mathsf{pat}(\sigma[\![\overline{p}_1]\!]) \mid \sigma[\![\overline{p}_2]\!]]$$

Note that we distinguish the substitution on a metavariable x from the substitution on a pattern $\overline{p}$ by noting the former $\sigma(x)$ and the latter $\sigma[\![\overline{p}]\!]$.

Lemma 8 (Minimality)

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

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

A.1.2 Proof

Let $\beth(\sigma, \sigma', \overline{p})$ be the proposition

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

Let us prove it by induction on the structure of the pattern $\overline{p}$, that is to say, we shall assume that $\beth(\sigma, \sigma', \overline{p}')$ holds for all the immediate subpatterns $\overline{p}'$ of $\overline{p}$ (that is the *induction hypothesis*) and then prove that $\beth(\sigma, \sigma', \overline{p})$ holds too. The immediate subpatterns are the subpatterns used to build the pattern in one step, by means of the definition. More precisely, we can define the relation $(\prec)$ such that $\overline{p}' \prec \overline{p}$ holds if $\overline{p}'$ is an immediate subpattern of $\overline{p}$ as:

$$\overline{p}' \prec [e \mid \overline{p}'] \qquad\qquad \overline{p}' \prec [\mathsf{pat}(\overline{p}') \mid \overline{p}_1]$$

We shall prove, for all σ, σ' and $\overline{p}$,

$$(\forall \overline{p}'.(\overline{p}' \prec \overline{p} \Rightarrow \beth(\sigma, \sigma', \overline{p}'))) \Rightarrow \beth(\sigma, \sigma', \overline{p})$$

which, by the structural-induction principle, implies $\forall \sigma, \sigma', \overline{p}.\beth(\sigma, \sigma', \overline{p})$. We shall proceed by distinguishing firstly the case when $\overline{p}$ is the minimal pattern, i.e., $\overline{p} = [\,]$, and secondly the other cases (for which $\overline{p}'$ exists).

(1) Case $\overline{p} = [\,]$.
We have $\sigma[\![[\,]]\!] \stackrel{1}{=} [\,] \stackrel{1}{=} \sigma'[\![[\,]]\!]$, i.e., $\beth(\sigma, \sigma', [\,])$ holds.

(2) Case $\overline{p} = [e \mid \overline{p}']$.
Let us assume the induction hypothesis

$$\forall \overline{p}'.(\overline{p}' \prec \overline{p} \Rightarrow \beth(\sigma, \sigma', \overline{p}')).$$

Since in this case we have a $\overline{p}'$ such that $\overline{p}' \prec \overline{p}$, then we deduce $\beth(\sigma, \sigma', \overline{p}')$.
Let us suppose now $\sigma \subseteq \sigma'$ (otherwise the theorem is trivially true).

Therefore, we have

$$\sigma \subseteq \sigma' \tag{A.1}$$
$$\sigma[\![\overline{p}']\!] = \sigma'[\![\overline{p}']\!]. \tag{A.2}$$

Let us now consider the different kinds of e.

(a) Case $e = \mathsf{meta}(x)$.

$$\begin{aligned}\sigma[\![[e \mid \overline{p}']]\!] &\triangleq \sigma[\![[\mathsf{meta}(x) \mid \overline{p}']]\!] \\ &\overset{1}{=} [\sigma(x) \mid \sigma[\![\overline{p}']\!]]\end{aligned}$$

The definition of $\subseteq$ and (A.1) imply that $\sigma(x) = \sigma'(x)$, therefore

$$\begin{aligned}&= [\sigma'(x) \mid \sigma[\![\overline{p}']\!]] \\ &= [\sigma'(x) \mid \sigma'[\![\overline{p}']\!]] \qquad \text{by (A.2)} \\ &\overset{1}{=} \sigma'[\![[\mathsf{meta}(x) \mid \overline{p}']]\!] \\ &\triangleq \sigma'[\![[e \mid \overline{p}']]\!] \\ \sigma[\![\overline{p}]\!] &= \sigma'[\![\overline{p}]\!].\end{aligned} \tag{A.3}$$

Finally, (A.3) and (A.1) imply $\beth(\sigma, \sigma', \overline{p})$.

(b) Case $e = l \in \mathcal{L}$.

$$\begin{aligned}\sigma[\![[e \mid \overline{p}']]\!] &\triangleq \sigma[\![[l \mid \overline{p}']]\!] \\ &\overset{2}{=} [l \mid \sigma[\![\overline{p}']\!]] \\ &= [l \mid \sigma'[\![\overline{p}']\!]] \qquad \text{by (A.2)} \\ &\overset{2}{=} \sigma'[\![[l \mid \overline{p}']]\!] \\ &\triangleq \sigma'[\![[e \mid \overline{p}']]\!] \\ \sigma[\![\overline{p}]\!] &= \sigma'[\![\overline{p}]\!].\end{aligned} \tag{A.4}$$

Finally, (A.4) and (A.1) imply $\beth(\sigma, \sigma', \overline{p})$.

(c) Case $e = \mathsf{pat}(\overline{p}_1)$.
Since $\overline{p}_1 \prec \overline{p}$, the induction hypothesis also implies here $\beth(\sigma, \sigma', \overline{p}_1)$, that is

$$\sigma[\![\overline{p}_1]\!] = \sigma'[\![\overline{p}_1]\!] \tag{A.5}$$

since we already assumed (A.1).

$$\begin{aligned}\sigma[\![[e \mid \overline{p}']]\!] &\triangleq \sigma[\![[\mathsf{pat}(\overline{p}_1) \mid \overline{p}']]\!] \\ &\overset{4}{=} [\mathsf{pat}(\sigma[\![\overline{p}_1]\!]) \mid \sigma[\![\overline{p}']\!]] \\ &= [\mathsf{pat}(\sigma[\![\overline{p}_1]\!]) \mid \sigma'[\![\overline{p}']\!]] \qquad \text{by (A.2)} \\ &= [\mathsf{pat}(\sigma'[\![\overline{p}_1]\!]) \mid \sigma'[\![\overline{p}']\!]] \qquad \text{by (A.5)} \\ &\overset{4}{=} \sigma'[\![[\mathsf{pat}(\overline{p}_1) \mid \overline{p}']]\!]\end{aligned}$$

$$\triangleq \sigma'[\![[e \mid \overline{p}']]\!]$$
$$\sigma[\![\overline{p}]\!] = \sigma'[\![\overline{p}]\!]. \tag{A.6}$$

Finally, (A.6) and (A.1) imply $\beth(\sigma, \sigma', \overline{p})$. □

A.2 Completeness of the backtracking algorithm

Theorem 9 (Completeness)

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

Proof A.2 Let $\aleph(\overline{p}, f, \sigma, \sigma')$ be the proposition:

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

The completeness is equivalent to $\aleph(\overline{p}, [h], \sigma, \sigma')$. First, let us assume that

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

(otherwise the theorem would be trivially true). This means that there exists a derivation Δ in the inference system (Figure 9) defining $(\sqsubseteq)$ whose conclusion is $\sigma[\![\overline{p}]\!] \sqsubseteq f$. This derivation is a list, which makes it possible to reckon by induction on its structure, i.e., we assume that $\aleph$ holds for the premise of the last rule in Δ (the *induction hypothesis*) and then prove that $\aleph$ holds for $\sigma[\![\overline{p}]\!] \sqsubseteq f$. We proceed case by case on the kind of rule that can end the derivation.

(1) Case where Δ ends with EMP.
We have $f = [\,]$ and $\sigma[\![\overline{p}]\!] = [\,] \overset{1}{=} \sigma[\![[\,]]\!]$. Therefore $\overline{p} = [\,]$ (the substitution is injective by construction), which leads to $\langle \overline{p}, f \rangle = \langle [\,], [\,] \rangle \twoheadrightarrow \sigma_\varnothing$ by means of the axiom END (Figure 6). And we trivially have $\sigma_\varnothing \subseteq \sigma$. We conclude that $\aleph([\,], [\,], \sigma, \sigma_\varnothing)$ holds.

(2) Case where Δ ends by EQ.
Then, for all parse tree h (perhaps reduced to a single lexeme), there exist two forests f_1 and f_2 such that

$$\sigma[\![\overline{p}]\!] \triangleq [h \mid f_1] \tag{A.8}$$
$$f \triangleq [h \mid f_2] \tag{A.9}$$

and Δ has the shape

$$\dfrac{\dfrac{\vdots}{f_1 \sqsubseteq f_2}}{[h \mid f_1] \sqsubseteq [h \mid f_2]}\ \mathsf{EQ}$$

By examining the definition of substitutions in Figure 5, we deduce that we have two exclusive options for h:

(a) Case where $h = l \in \mathcal{L}$.

Definition (A.8) and definition ($\stackrel{3}{=}$) of the substitutions imply that there exists an unparsed pattern $\overline{p}'$ such that

$$\begin{aligned} \overline{p} &= [l \mid \overline{p}'] \\ f_1 &= \sigma[\![\overline{p}']\!]. \end{aligned} \tag{A.10}$$

So Δ has the refined shape

$$\frac{\begin{array}{c}\vdots\\ \sigma[\![\overline{p}']\!] \sqsubseteq f_2\end{array}}{\sigma[\![\overline{p}]\!] \sqsubseteq f}\ \mathsf{EQ}$$

Let us assume that the induction hypothesis holds for the premise of EQ, i.e.,

$$\langle \overline{p}', f_2 \rangle \twoheadrightarrow \sigma' \tag{A.11}$$

$$\sigma' \subseteq \sigma. \tag{A.12}$$

Configuration (A.11) can be the premise of pattern-matching rule ELIM (Figure 6), whose conclusion is then

$$\langle [l \mid \overline{p}'], [l \mid f_2] \rangle \twoheadrightarrow \sigma'. \tag{A.13}$$

By (A.9) and (A.10), (A.13) is equivalent to

$$\langle \overline{p}, f \rangle \twoheadrightarrow \sigma'. \tag{A.14}$$

As a conclusion, the induction hypothesis and (A.7) imply (A.14) and (A.12), i.e., $\aleph([l \mid \overline{p}'], [l \mid f'], \sigma, \sigma')$ holds.

(b) Case where $h = t \in \mathcal{T}$.

Definition (A.8) and definition ($\stackrel{2}{=}$) of the substitutions (Figure 5) imply that there exists an unparsed pattern $\overline{p}'$ and a metavariable x such that

$$\overline{p} = [\mathsf{meta}(x) \mid \overline{p}'] \tag{A.15}$$

$$\begin{aligned} f_1 &= \sigma[\![\overline{p}']\!] \\ t &= \sigma(x). \end{aligned} \tag{A.16}$$

So Δ has the refined shape

$$\frac{\begin{array}{c}\vdots\\ \sigma[\![\overline{p}']\!] \sqsubseteq f'\end{array}}{\sigma[\![\overline{p}]\!] \sqsubseteq f}\ \mathsf{EQ}$$

Let us assume that the induction hypothesis holds for the premise of EQ, i.e., $\sigma[\![\overline{p}']\!] \sqsubseteq f'$ of EQ, i.e.,

$$\langle \overline{p}', f' \rangle \twoheadrightarrow \sigma' \tag{A.17}$$
$$\sigma' \subseteq \sigma. \tag{A.18}$$

Let us consider two cases now:

(i) $x \notin dom(\rho')$.
Then, by definition of inclusions of substitutions (5) implies

$$\sigma' \subseteq \sigma' \oplus x \mapsto t.$$

(ii) $x \in dom(\rho')$.
The definition of inclusions (5) applied to (A.18) implies that $\sigma'(x) = \sigma(x)$, thus, by (A.16), we have $\sigma'(x) = t$. So $\sigma' \subseteq \sigma' \oplus x \mapsto t$, since, by definition (3), $(\sigma' \oplus x \mapsto t)(x) \triangleq t = \sigma'(x)$.

In both cases, we proved that

$$\sigma' \subseteq \sigma' \oplus x \mapsto t. \tag{A.19}$$

Configuration (A.17) and property (A.19) can be the premises to the pattern-matching rule BIND, which leads to the conclusion

$$\langle [\mathsf{meta}(x) \mid \overline{p}'], [t \mid f'] \rangle \twoheadrightarrow \sigma' \oplus x \mapsto t. \tag{A.20}$$

By (A.15) and (A.9), (A.20) is equivalent to

$$\langle \overline{p}, f \rangle \twoheadrightarrow \sigma' \oplus x \mapsto t. \tag{A.21}$$

As a conclusion, the induction hypothesis and (A.7) imply (A.21) and (A.19), i.e., $\aleph([\mathsf{meta}(x) \mid \overline{p}'], [t \mid f'], \sigma' \oplus x \mapsto t, \sigma')$.

(3) Case where Δ ends by SUB.
Then there exist two forests f_1 and f_2 and a constructor $c \in \mathcal{C}$ such that

$$f \triangleq [c(f_1) \mid f_2]. \tag{A.22}$$

The derivation Δ has thus the shape

$$\dfrac{\dfrac{\vdots}{\sigma[\![\overline{p}]\!] \sqsubseteq f_1 \cdot f_2}}{\sigma[\![\overline{p}]\!] \sqsubseteq [c(f_1) \mid f_2]}\ \mathsf{SUB}$$

Let us assume that the induction hypothesis holds for the premise of SUB, i.e.,

$$\langle \overline{p}, f_1 \cdot f_2 \rangle \twoheadrightarrow \sigma' \tag{A.23}$$
$$\sigma' \subseteq \sigma. \tag{A.24}$$

The configuration (A.23) can be the premise of the pattern-matching rule UNPAR, which leads to the conclusion

$$\langle \overline{p}, [c(f_1) \mid f_2] \rangle \twoheadrightarrow \sigma'. \tag{A.25}$$

As a conclusion, the induction hypothesis and (A.7) imply (A.25) and (A.24), i.e., $\aleph(\overline{p}, [c(f_1) \mid f_2], \sigma, \sigma')$. □

A.3 Determinacy of the backtracking algorithm

Theorem 10 (Determinacy)

$$\textit{If } \langle \overline{p}, h \rangle \twoheadrightarrow \sigma \textit{ and } \langle \overline{p}, h \rangle \twoheadrightarrow \sigma' \textit{ then } \sigma = \sigma'.$$

In other words, the fact that the system is not syntax-directed, hence, that the implementation needs a backtracking strategy, does not impede that there is none the less only one result to the pattern matching of a forest.

PROOF A.3 Let $\aleph(\overline{p}, f, \sigma, \sigma')$ be the proposition

$$\textit{If } \langle \overline{p}, f \rangle \twoheadrightarrow \sigma \textit{ and } \langle \overline{p}, f \rangle \twoheadrightarrow \sigma' \textit{ then } \sigma = \sigma'.$$

In particular, $\aleph(\overline{p}, [h], \sigma, \sigma')$ is equivalent to the determinacy. Firstly, let us assume that

$$\langle \overline{p}, f \rangle \twoheadrightarrow \sigma \tag{A.26}$$
$$\langle \overline{p}, f \rangle \twoheadrightarrow \sigma' \tag{A.27}$$

are true (otherwise the theorem is trivially true). This means that there exists a pattern-matching derivation Δ whose conclusion is $\langle \overline{p}, f \rangle \twoheadrightarrow \sigma$ and another derivation Δ' whose conclusion is $\langle \overline{p}, f \rangle \twoheadrightarrow \sigma'$. These derivations are trees (actually, lists), so we can reason by general induction on them, i.e., we assume that $\aleph$ holds for (the conclusions of) any pair of sub-derivations, one from Δ and the other from Δ', and then prove that $\aleph$ holds for the conclusions of Δ and Δ' as well — that is to say, $\aleph(\overline{p}, f, \sigma, \sigma')$ is true. (This induction schema implies that the configurations must be the same in the judgements of the sub-derivations considered as an hypothesis.) We proceed case by case on the kinds of rule that may end the derivations. Since $\overline{p}$ and f must be the same in the conclusions of Δ and Δ', some pairs of ending rules for the derivations are impossible. For example, if ELIM ends one derivation, then BIND cannot end the other because the former mandates that $\overline{p} = [l \mid \overline{p}']$, with $l \in \mathcal{L}$, whilst the latter requires that $\overline{p} = [\mathsf{meta}(x) \mid \overline{p}']$ and $l \neq \mathsf{meta}(x)$. In the following, we only give the possible combinations. Also, if Δ can end with a rule X and Δ' with another rule Y, we do not give the case where Δ ends with rule Y and Δ' with rule X (symmetric case).

(1) Case where Δ and Δ' end by END.
Then $\overline{p} = [\,]$, $f = [\,]$, $\sigma = \sigma_\varnothing$ and $\sigma' = \sigma_\varnothing$. As a consequence, $\sigma = \sigma'$, that is, $\aleph([\,], [\,], \sigma_\varnothing, \sigma_\varnothing)$ holds.
(2) Case where Δ and Δ' end by ELIM.

$$\dfrac{\dfrac{\begin{array}{c}(\Delta_1)\\ \vdots\end{array}}{\langle \overline{p}', f' \rangle \twoheadrightarrow \sigma}}{\langle [l \mid \overline{p}'], [l \mid f'] \rangle \twoheadrightarrow \sigma}\ \mathsf{ELIM} \qquad \dfrac{\dfrac{\begin{array}{c}(\Delta'_1)\\ \vdots\end{array}}{\langle \overline{p}', f' \rangle \twoheadrightarrow \sigma'}}{\langle [l \mid \overline{p}'], [l \mid f'] \rangle \twoheadrightarrow \sigma'}\ \mathsf{ELIM}$$

where, since we assumed (A.26) and (A.27),
(a) $l \in \mathcal{L}$,
(b) $\overline{p} \triangleq [l \mid \overline{p}']$,
(c) $f \triangleq [l \mid f']$.
Since Δ_1 is a sub-derivation of Δ and Δ'_1 is a sub-derivation of Δ', let us assume that the induction hypothesis holds for their conclusions, i.e., $\aleph(\overline{p}', f', \sigma, \sigma')$ holds. Therefore

$$\sigma = \sigma'. \tag{A.28}$$

Finally, the induction hypothesis, (A.26) and (A.27) imply (A.28), that is to say, $\aleph([l \mid \overline{p}'], [l \mid f'], \sigma, \sigma')$.
(3) Case where Δ and Δ' both end by BIND.

$$\dfrac{\mathsf{BIND}\ \dfrac{\begin{array}{c}(\Delta_1)\\ \vdots\end{array}}{\langle \overline{p}', f' \rangle \twoheadrightarrow \sigma_1} \qquad \sigma_1 \subseteq \sigma_1 \oplus x \mapsto t}{\langle [\mathsf{meta}(x) \mid \overline{p}'], [t \mid f'] \rangle \twoheadrightarrow \sigma_1 \oplus x \mapsto t}$$

$$\dfrac{\mathsf{BIND}\ \dfrac{\begin{array}{c}(\Delta'_1)\\ \vdots\end{array}}{\langle \overline{p}', f' \rangle \twoheadrightarrow \sigma_2} \qquad \sigma_2 \subseteq \sigma_2 \oplus x \mapsto t}{\langle [\mathsf{meta}(x) \mid \overline{p}'], [t \mid f'] \rangle \twoheadrightarrow \sigma_2 \oplus x \mapsto t}$$

where, since we assumed (A.26) and (A.27),
(a) $\overline{p} \triangleq [\mathsf{meta}(x) \mid \overline{p}']$,
(b) $f \triangleq [t \mid f']$, with $t \in \mathcal{T}$,
(c) $\sigma \triangleq \sigma_1 \oplus x \mapsto t$,
(d) $\sigma' \triangleq \sigma_2 \oplus x \mapsto t$.
Since Δ_1 is a sub-derivation of Δ and Δ'_1 is a sub-derivation of Δ', let us assume that the induction hypothesis holds for their conclusions, i.e.,

$\aleph(\overline{p}', f', \sigma_1, \sigma_2)$. Therefore

$$\begin{array}{c} \sigma_1 = \sigma_2 \\ \sigma_1 \oplus x \mapsto t = \sigma_2 \oplus x \mapsto t \\ \sigma = \sigma' \qquad\qquad \text{by 3c and 3d.} \end{array} \tag{A.29}$$

Finally, the induction hypothesis, (A.26) and (A.27) imply (A.29), that is to say, $\aleph([\mathsf{meta}(x) \mid \overline{p}'], f, \sigma, \sigma')$.

(4) Case where both Δ and Δ' end by UNPAR.

$$\dfrac{\dfrac{\begin{array}{c}(\Delta_1)\\ \vdots \end{array}}{\langle \overline{p}, f_1 \cdot f_2 \rangle \twoheadrightarrow \sigma}}{\langle \overline{p}, [c(f_1) \,|\, f_2] \rangle \twoheadrightarrow \sigma}\ \mathsf{UNPAR} \qquad \dfrac{\dfrac{\begin{array}{c}(\Delta_1')\\ \vdots \end{array}}{\langle \overline{p}, f_1 \cdot f_2 \rangle \twoheadrightarrow \sigma'}}{\langle \overline{p}, [c(f_1) \,|\, f_2] \rangle \twoheadrightarrow \sigma'}\ \mathsf{UNPAR}$$

where, since we assumed (A.26) and (A.27),
(a) $f \triangleq [c(f_1) \mid f_2]$.
Since Δ_1 is a sub-derivation of Δ and Δ_1' is a sub-derivation of Δ', let us assume that the induction hypothesis holds for their conclusions, that is, $\aleph(\overline{p}, f_1 \cdot f_2, \sigma, \sigma')$ holds:

$$\sigma = \sigma'. \tag{A.30}$$

Finally, the induction hypothesis, (A.26) and (A.27) imply (A.30), that is to say, $\aleph(\overline{p}, [c(f_1) \mid f_2], \sigma, \sigma')$.

(5) Case where Δ ends by UNPAR and Δ' ends by BIND.

$$\dfrac{\mathsf{UNPAR}\ \dfrac{\begin{array}{c}(\Delta_1)\\ \vdots \end{array}}{\langle [\mathsf{meta}(x) \mid \overline{p}'], f_1 \cdot f_2 \rangle \twoheadrightarrow \sigma}}{\langle [\mathsf{meta}(x) \mid \overline{p}'], [c(f_1) \mid f_2] \rangle \twoheadrightarrow \sigma} \qquad \dfrac{\mathsf{BIND}\ \dfrac{\begin{array}{c}(\Delta_1')\\ \vdots \end{array}}{\langle \overline{p}', f' \rangle \twoheadrightarrow \sigma''} \quad \sigma'' \subseteq \sigma'' \oplus x \mapsto t}{\langle [\mathsf{meta}(x) \mid \overline{p}'], [t \mid f'] \rangle \twoheadrightarrow \sigma'' \oplus x \mapsto t}$$

where, since we assumed (A.26) and (A.27),
(a) $\overline{p} \triangleq [\mathsf{meta}(x) \mid \overline{p}']$,
(b) $t \triangleq c(f_1)$,
(c) $\sigma' \triangleq \sigma'' \oplus x \mapsto t$.
The premise of UNPAR can only be itself the conclusion of the inference rule UNPAR or BIND. If UNPAR again, the same is true about its premise. So, the two cases can be merged into one, i.e., the shape of Δ is (by writing

only the pattern-matching judgements)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\Delta_2)}{\vdots}}{\dfrac{\langle \overline{p}', g \cdot f' \rangle \twoheadrightarrow \sigma_1}{\langle [\mathsf{meta}(x) \mid \overline{p}'], [t \mid g \cdot f'] \rangle \twoheadrightarrow \sigma_1 \oplus x \mapsto t}\ \mathsf{BIND}}}{\vdots}}{\dfrac{\langle [\mathsf{meta}(x) \mid \overline{p}'], f_1 \cdot f_2 \rangle \twoheadrightarrow \sigma_1 \oplus x \mapsto t}{\langle [\mathsf{meta}(x) \mid \overline{p}'], [c(f_1) \mid f_2] \rangle \twoheadrightarrow \sigma_1 \oplus x \mapsto t}\ \mathsf{UNPAR}}\ \mathsf{UNPAR}}{\vdots}}{}\ \mathsf{UNPAR}$$

where all the rules, if any, below BIND are UNPAR rules and

$$\sigma = \sigma_1 \oplus x \mapsto t. \tag{A.31}$$

(Because the derivation is finite, the rule BIND must appear at one point up.) The number of unparsing steps can be 0, in which case UNPAR is followed inductively by BIND. Note that the list g may be empty if $A(c) = 1$ for each tree $c(\dots)$ unparsed. Let us consider the two possible cases for g.

(a) Case $g = [\,]$.
In this case, $\langle \overline{p}', g \cdot f' \rangle \twoheadrightarrow \sigma_1$ is actually $\langle \overline{p}', f' \rangle \twoheadrightarrow \sigma_1$. Since Δ_2 is a sub-derivation of Δ_1, which is, in turn, a sub-derivation of Δ, then Δ_2 is a sub-derivation of Δ. Since Δ_1' is a sub-derivation of Δ', the induction hypothesis holds for both the conclusions of Δ_2 and Δ_1', so

$$\begin{aligned} \sigma_1 &= \sigma'' \\ \sigma_1 \oplus x \mapsto t &= \sigma'' \oplus x \mapsto t \\ \sigma &= \sigma'' \oplus \mapsto t && \text{by (A.31)} \\ \sigma &= \sigma' && \text{by 5c} \\ \sigma &= \sigma' && \text{(A.32)} \end{aligned}$$

Finally, the induction hypothesis, (A.26) and (A.27) imply (A.32) in this case, i.e., $\aleph([\mathsf{meta}(x) \mid \overline{p}'], f, \sigma, \sigma')$ holds.

(b) Case $g \neq [\,]$.
Now let us establish that the conjunction of the following judgements is contradictory, i.e., the derivation of one of them implies that the other cannot be derived.

$$\langle \overline{p}', g \cdot f' \rangle \twoheadrightarrow \sigma_1 \qquad \text{where } g \neq [\,] \tag{A.33}$$

$$\langle \overline{p}', f' \rangle \twoheadrightarrow \sigma'' \tag{A.34}$$

The rules ELIM and BIND, read deductively (i.e., the premises before the conclusion), increment the length of the pattern whilst rule UN-

PAR keeps it unchanged. The derivations are lists starting with the same axiom END. Because of that and since (A.33) and (A.34) share the same pattern $\overline{p}'$, it is sufficient to prove that (A.33) cannot derive (A.34) and, reciprocally, that (A.34) cannot derive (A.33). Moreover, we only need to consider derivations by means of UNPAR, since it is the sole rule which keeps the pattern invariant.

(i) The *length* $|f|$ of a forest f is defined as

$$\begin{aligned} |[\,]| &= 0 \\ |[t \mid f]| &= 1 + |f| \end{aligned}$$

The rule UNPAR, read deductively (i.e, premise first, conclusion next), shortens the length of the forest or keeps it invariant (if $A(c) = 1$), since we have

$$\begin{aligned} |f_1 \cdot f_2| = |f_1| + |f_2| &\triangleq A(c) + |f_2| \\ |[c(f_1) \mid f_2]| &= 1 + |f_2| \end{aligned}$$

Therefore (A.34) cannot derive (A.33), because $g \neq [\,]$, so $g \cdot f_2$ is strictly longer than f_2, i.e., $|f_2| < |g \cdot f_2|$.

(ii) Let us consider now the derivation from (A.33) by applying, perhaps repeatedly, the inference rule UNPAR. The effect of the deduction on the forest consists in replacing a prefix of at least one tree of $f_1 \cdot f_2$ by a single tree of root c, i.e., $[c(f_1) \mid f_2]$. In particular, one tree can be replaced by one tree, but at one point at least two trees must be rewritten into one, because the derivation must try to reach (A.34), which contains a forest strictly shorter. Thus, the start of the derivation has the following shape

$$\dfrac{\dfrac{\langle \overline{p}', g \cdot f_2 \rangle \twoheadrightarrow \sigma_1}{\vdots}\,\textsf{UNPAR}}{\langle \overline{p}', [t_1 \mid f_2] \rangle \twoheadrightarrow \sigma_1}\,\textsf{UNPAR}$$

The prefix g has been rewritten into a single tree t_1 (by the way, this process is called *parsing*). Let us assume that one more usage of the rule UNPAR rewrites at least *two* more trees from $[t_1 \mid f_2]$ into one. (If not, it means that the size of the forest would remain constant, i.e., $A(c) = 1$, and strictly greater than f_2, as mentioned above.) But this shortening means that the resulting forest differs from f_2 by, at least, the first tree. Therefore (A.33) cannot derive (A.34).

This achieves to prove that the sub-case $g \neq [\,]$ is impossible. □

References

[1] D. Engler, B. Chelf, A. Chou, and 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 and D. Engler. MJ–A System for constructing Bug-finding Analyses for Java. Technical report, Stanford University (USA), September 2003.

[3] H. Chen and 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, and W. Weimer. Temporal-safety Proofs for Systems Code. In *14th Int. Conf. Computer-aided Verif. (CAV)*, volume LNCS 2404. Springer-Verlag, 2002.

[5] W. Griswold, D. Atkinson, and 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, and D. Taylor, editors, *Conf. of the Centre For Advanced Studies on Collaborative Research*, volume 1, Toronto (Canada), November 1992. IBM Centre for Advanced Studies, IBM Press.

[7] K. Olender and L. Osterweil. Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation. *IEEE Trans. Softw. Eng.*, 16(3), March 1990.

[8] A. Sellink and C. Verhoef. Native Patterns. In *Working Conf. on Reverse Eng. (WCRE)*. IEEE Computer Society, October 1998.

[9] C. De Roover, T. D'Hondt, J. Brichau, C. Noguera, L., and Duchien. Behavioral Similarity Matching using Concrete Source Code Templates in Logic Queries. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, Nice (France), January 2007.

[10] S. Burson, G.B. Kotik, and 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, and R. Majumdar. The BLAST Query Language for Software Verification. In *6th ACM SIGPLAN Int. Conf. Principles and Practice of Declarative Prog. (PPDP)*, Verona (Italy), August 2004. ACM Press.

[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, and J. Rekers. *The Syntax Definition Formalism SDF*. SIGPLAN Notices, 1992.

[14] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling Language Definitions: The ASF+SDF Compiler. *ACM Trans. Prog. Lang. Syst.*, 24:334–368, 1999.

[15] Elco Visser. Meta-programming with Concrete Object Syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Prog. and Component Eng. (GPCE)*, volume 2487 of *LNCS*, pages 299–315, Pittsburgh (USA), October 2002. Springer-Verlag.

[16] E. Scott, A. Johnstone, and R. Economopoulos. BRNGLR: A Cubic Tomita-style GLR parsing Algorithm. *Acta Informatica*, 44(6), 2007.

[17] S. McPeak and 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] Nic 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), July 2006. ACM Press.

[20] Jan Van Leeuwen, editor. *Formal Models and Semantics*, volume B, chapter Rewrite Systems, pages 243–320. Elsevier, 1990.

[21] Elco Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, University of Amsterdam, Programming Research Group (Netherlands), 1997.

[22] M. S. Peterson and M. N. Wegman. Linear Unification. *J. Comput. Syst. Sci.*, 16(2), April 1978.

[23] F. Luccio, A. M. Enriquez, P. O. Rieumont, and L. Pagli. Exact Rooted Subtree Matching in Sublinear Time. Technical Report TR-01-14, University of Pisa (Italy), 2001.

[24] D. Beyer, G. Castagna, and A. Frisch. CDuce: an XML-centric General-purpose Language. In *8th ACM SIGPLAN Int. Conf. Functional Prog. (ICFP)*, Uppsala (Sweden), August 2003. ACM Press.

[25] D. Zook, S. S. Huang, and Y. Smaragdakis. Generating AspectJ Programs with Meta-AspectJ. In *Conf. Generative Prog. and Component Eng. (GPCE)*, volume LNCS 3286, Vancouver (Canada), October 2004. Springer Verlag.

[26] M. Bravenboer, R. Vermaas, J. Vinju, and E. Visser. Generalized Type-based Disambiguation of Meta Programs with Concrete Object Syntax. In R. Glück and M. Lowry, editors, *4th Int. Conf. Generative Prog. and Component Eng. (GPCE)*, volume LNCS 3676. Springer-Verlag, September 2005.