Theoretical Computer Science 112 (1993) 53–97
Elsevier

A sheaf-theoretic approach to pattern matching and related problems

Yellamraju V. Srinivas

Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304, USA

Abstract

Srinivas, Y.V., A sheaf-theoretic approach to pattern matching and related problems, Theoretical Computer Science 112 (1993) 53–97.

We present a general theory of pattern matching by adopting an extensional, geometric view of patterns. Representing the geometry of the pattern via a Grothendieck topology, the extension of the matching relation for a constant target and varying pattern forms a sheaf. We derive a generalized version of the Knuth–Morris–Pratt string-matching algorithm by gradually converting this extensional description into an intensional description, i.e., an algorithm. The generality of this approach is illustrated by briefly considering other applications: Earley's algorithm for parsing, Waltz filtering for scene analysis, matching modulo commutativity, and the n-queens problem.

Contents

1. The geometry of matching 54
   1.1. The Knuth–Morris–Pratt algorithm 55
   1.2. Outline 56
   1.3. Background 56
2. Topologies, sites, sheaves 57
   2.1. Sites 57
   2.2. Sheaves 59
3. A specification for pattern matching 63
   3.1. The specification 64
4. Derivation of a pattern-matching algorithm 66
   4.1. Decomposing the pattern 66
   4.2. Decomposing the target 67
   4.3. Building occurrence arrows from pieces 68
   4.4. The abstract pattern-matching problem 68
   4.5. Enumerating compatible families 70
   4.6. Decomposition of covers 71

Correspondence to: Y.V. Srinivas, Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304, USA. Email: srinivas@kestrel.edu.

0304-3975/93/$06.00 © 1993—Elsevier Science Publishers B.V. All rights reserved

4.7. Building compatible families from pieces . . . 73
4.8. An incremental algorithm . . . 74
4.9. Improving the incremental algorithm . . . 77
4.10. Generation of elementary occurrences . . . 79
4.11. Matching the pattern against itself . . . 80
4.12. Computing the pattern–pattern-functor . . . 81
4.13. An algorithm using subsumption . . . 82
4.14. Instantiation for strings . . . 83
5. Related algorithms . . . 86
5.1. Multiple patterns . . . 86
5.2. Patterns with variables . . . 86
5.3. Commutative/associative matching . . . 87
5.4. Nonlocal properties, approximate matching . . . 87
5.5. Context-free parsing: Earley's algorithm . . . 88
5.6. Constraint propagation: Waltz filtering . . . 90
5.7. Enumerating functions . . . 90
5.8. The n-queens problem . . . 91
6. Concluding remarks . . . 92
6.1. In defense of abstract nonsense . . . 93
6.2. Why Grothendieck topologies? . . . 93
Appendix. A formal basis for KMP-style algorithms . . . 94
Acknowledgment . . . 96
References . . . 96

1. The geometry of matching

The pattern-matching problem consists of finding *occurrences* of a *pattern* in a *target*. A pattern is usually given by a constant entity (e.g., the string "*Charlie*"), an exemplar (e.g., the expression $E \times E + E$, with the variable E matching any expression), or, in general, a predicate (e.g., a connected graph with a prime number of edges). A target consists of an entity which is usually much larger than the pattern – hence, the possibility of multiple occurrences of the pattern – and which may spread out in space and time. Corresponding to the patterns above, some possible targets are a file representing a document, a syntax tree produced during compilation, and a graph representing a network. Usually, the pattern and the target are the "same kind" of entities: strings, graphs, bitmaps, etc. An occurrence is a piece of the target together with a correspondence with the pattern. If the pattern is a constant, this piece of the target should be the same as the pattern; if the pattern is an exemplar, the piece should have the same shape as the pattern; if the pattern is a predicate, the piece should satisfy the predicate.

Pattern matching in graphs, and in any data structure more complex than graphs, is NP-complete. However, in most practical situations, and when data structures such as strings and trees are used, more efficient algorithms are possible. In particular, occurrences of a constant pattern string in a target string can be enumerated in linear time, as shown by the Knuth–Morris–Pratt string-matching algorithm [22] (hereafter abbreviated as KMP). This algorithm uses some clever tricks to achieve this bound.

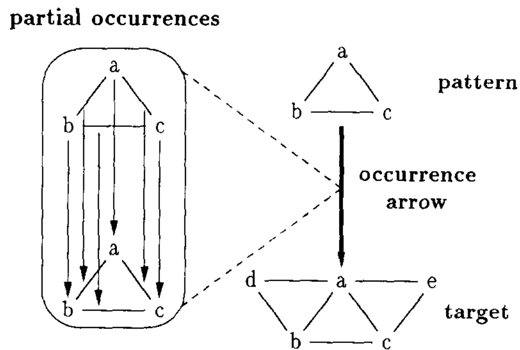


Fig. 1. Anatomy of an occurrence: example with graphs.

In this paper, we will analyze this algorithm by providing a derivation of a generalized version of the algorithm which works for any data structure (but not necessarily in linear time).

In generalizing KMP to data structures other than strings, the feature which acquires prominence is the piecing together of an occurrence from partial occurrences. We show an example of this phenomenon, using graphs, in Fig. 1: an occurrence arrow $p \to t$ is obtained by gluing together smaller arrows $p_i \to t_j$. The notion of building an occurrence arrow by "gluing" together or "sewing" together smaller arrows has a decidedly geometric flavor. The rest of this paper is devoted to formalizing and exploiting this geometric nature of the pattern-matching problem.

KMP has been generalized to data structures other than strings, such as trees [20, 9], and two-dimensional arrays [3, 7]. However, these generalizations are ad hoc in the sense that they do not provide a systematic way of obtaining a version of KMP for other data structures. This lack of generality arises from the lack of focus on the geometry of the problem.

1.1. The Knuth–Morris–Pratt algorithm

The KMP algorithm [22] is a fast pattern-matching algorithm for finding occurrences of a constant pattern in a target string. It is linear in the sum of the sizes of the pattern and the target strings. KMP reduces the complexity of the naive algorithm for string matching (check for a match at every position in the target string) by avoiding comparisons whose results are already known (from previous comparisons). In particular, given a character mismatch after the pattern is partially matched, the next possible position in the target where the pattern can match can be computed by using the knowledge of the partial match. This "sliding" of the pattern on a mismatch is the most well-known aspect of KMP. We show below an example where there is

a mismatch at the last character of the pattern and the pattern can be slid three positions to the right:

pattern	$a\ b\ c\ a\ b\ a$
target	$a\ b\ c\ a\ b\ c\ a\ b\ c$
matches	$\surd\ \surd\ \surd\ \surd\ \surd\ \times$
slide	$\longrightarrow a\ b\ c\ a\ b\ a$

The amounts by which to slide the pattern on possible mismatches can be precomputed in time proportional to the size of the pattern. Thus, all occurrences can be enumerated in a single left-to-right scan of the target string without backing up.

The table assigning the amount of sliding to each mismatch is called the *failure function*. We attack the problem of rigorously deriving such a function from a specification of pattern matching. There are several derivations of KMP in the literature [12, 8, 38, 25, 27]. However, all these derivations consider only pattern matching on strings. It is not apparent how to generalize these derivations because they crucially depend on properties of strings. We follow a more general approach of describing a match in terms of sub-matches; this description depends only on the geometry of the underlying data structure. We also explain the failure function as an instance of backtracking, a general strategy for searching.

1.2. Outline

In Section 2, we give definitions and examples of topologies and sheaves. In Section 3, we characterize the extension of the occurrence relation as a sheaf. In Section 4, we derive a generalized version of KMP starting from an occurrence sheaf. In Section 5, we show that the same derivation provides explanations for a variety of other algorithms. The appendix is devoted to results from category theory and sheaf theory, which form the formal basis of the derivation.

1.3. Background

The reader is assumed to have a working knowledge of category theory. The level of category theory required for a thorough understanding of the formal basis of the derivation in this paper precludes a short introduction here. However, the derivation of the pattern-matching algorithm can be understood at an intuitive level, by thinking of a category as a partially ordered collection of entities, a set-valued functor (and also a sheaf) as a multifunction, and a natural transformation as an indexed family of maps. We will extensively use natural transformations to represent compatible families of partial occurrences.

Relevant concepts of category theory can be found in mathematics textbooks [23, 17, 30] or computer-science-oriented introductions to category theory [28, 29, 5]. The notation used in this paper closely follows that used by Mac Lane [23].

2. Topologies, sites, sheaves

We formalize the geometry of patterns via Grothendieck topologies, which are more suited than general topology (point-set topology) to the finite structures which arise in computer science.[1] Normally, a topology is a collection of open sets which is closed under arbitrary unions and finite intersections. A Grothendieck topology is a generalization in which the poset of open sets is replaced by a category. The topology itself is captured in the notion of a "cover," which is a generalization of open covers. The definitive reference for Grothendieck topologies and sheaf theory is SGA4 [2, Exposés I–IV]. Several other books have brief descriptions [16, 24, 30, 4, 21].

2.1. Sites

Definition 2.1 (*Sieve*). A sieve S on an object a in a category $\mathscr{C}$ is a collection of arrows with codomain a which is closed under right composition, i.e., if $f: b \to a$ is in S, then for any arrow $g: c \to b$, the composite $f \circ g: c \to a$ is in S.

Definition 2.2 (*Grothendieck topology*). A Grothendieck topology J on a category $\mathscr{C}$ is an assignment to each object a of $\mathscr{C}$, a set $J(a)$ of sieves on a, called *covering sieves* (or just *covers*), satisfying the following axioms:

(1) *Identity cover.* For any object a, the maximal sieve $\{f \mid \text{codomain}(f) = a\}$ is in $J(a)$;

(2) *Stability under change of base.* If $R \in J(a)$ and $b \xrightarrow{f} a$ is an arrow of $\mathscr{C}$, then the sieve $f^*(R) = \{c \xrightarrow{g} b \mid f \circ g \in R\}$ is in $J(b)$;

(3) *Local character.* If $R \in J(a)$ and S is a sieve on a such that for each arrow $b \xrightarrow{f} a$ in R we have $f^*(S) \in J(b)$, then $S \in J(a)$.

Definition 2.3 (*Site*). A site is a category along with a Grothendieck topology. The site formed by a topology J on a category $\mathscr{C}$ will be denoted by $\langle \mathscr{C}, J \rangle$.

Explanation of axioms. The axioms of a topology[2] are closure conditions on the collection of covers. Axiom 1 states that the sieve generated by the identity arrow is a cover. Axiom 2 states that, given a cover of an object and a sub-structure of that object, the restriction of the cover to the sub-structure is a cover of the sub-structure. Axiom 3 states that covers of covers are also covers. Specifically, given a cover of an object, and given a cover for each of the objects[3] in the cover, the composed cover is

[1] See Section 6.2.

[2] From now on, we will drop the adjective "Grothendieck" when referring to Grothendieck topologies.

[3] Although, strictly speaking, the elements of a cover are arrows, we will frequently treat the *domains* of these arrows as the elements of the cover.

a (finer) cover of the original object. The axioms also imply that any sieve containing a covering sieve is itself a covering sieve.

We now give a series of examples of topologies on data structures induced by considering the sub-structure relationship. Other examples of sites are given in Section 5. We will normally specify a covering sieve by providing a family of arrows which generates it; such a family is called a *covering family*. Similarly, we will specify a topology by giving a collection of covers which generates it – the generated topology is the least topology which contains the given covers and satisfies the closure axioms of Definition 2.2.

Example 2.4 (*Sets*). Sets and functions form a category **Set**. A cover of a set S is a family of subsets of S, $\{S_i \subsetneq S \mid i \in I\}$, whose union is S, i.e., $\bigcup_{i\in I} S_i = S$.

Example 2.5 (*Connected graphs*). A graph is a pair of sets $\langle N, E \subseteq N \times N \rangle$ called nodes and edges. A path from the node a_1 to the node a_k in the graph G is a sequence of nodes $a_1, a_2, \ldots, a_k$ such that each $\langle a_i, a_{i+1} \rangle$ is an edge in G, for all $1 \leqslant i < k$. A graph is connected if there is a path between any two nodes in the graph. The nodes and edges of a graph G will be denoted by $N(G)$ and $E(G)$.

A graph morphism $f: G \to H$ is a pair of functions

$$\langle f_N : N(G) \to N(H),\ f_E : E(G) \to E(H) \rangle$$

which map nodes and edges compatibly, i.e.,

$$\forall \langle a, b \rangle \in E(G)\ f_E(\langle a, b \rangle) = \langle f_N(a), f_N(b) \rangle.$$

Connected graphs and their morphisms form a category **CGraph**. A subgraph of a graph H is a graph G such that $N(G) \subseteq N(H)$ and $E(G) \subseteq E(H)$. A cover of a graph G is a family of subgraphs $\{G_i \subsetneq G \mid i \in I\}$ such that

$$\bigcup_{i \in I} N(G_i) = N(G) \quad \text{and} \quad \bigcup_{i \in I} E(G_i) = E(G).$$

Example 2.6 (*Trees*). A *tree* is an undirected, connected, acyclic graph. The definitions of morphisms, subtrees, inclusions, and covers carry over from those of connected graphs. We, thus, have a subcategory of **CGraph** called **Tree**.

Example 2.7 (*Strings*). A *string* (unlabeled) is a pair $\langle s, <_s \rangle$ consisting of a set s and a linear order $<_s$ on that set (i.e., a total, irreflexive, and transitive relation). A subset $r \subseteq s$ of a string $\langle s, <_s \rangle$ is said to be *contiguous* if, for all elements a, b in r and for all elements x in s, $a <_s x <_s b \Rightarrow x \in r$. A morphism of strings is an order-preserving map whose image is contiguous. Strings and string morphisms form a category, **String**. A *substring* of $\langle t, <_t \rangle$ is a string $\langle s, <_s \rangle$ such that $s \subseteq t$ is a contiguous subset of t and $<_s$ is the restriction of $<_t$ to s. A cover[4] for a string $\langle s, <_s \rangle$ is a collection of substrings,

[4] The covers used for KMP are different from these; see Definition 3.2.

the union of whose images is equal to s. For example, the families {"a", "bc"} and {"ab", "b", "c"} are covers for the string "abc".

Example 2.8 (*Labeled structures*). Strings, trees, and graphs can be labeled. A labeling for a structure S is a function $l_S: U(S) \to L$ assigning labels from a fixed set L to each element of the underlying set $U(S)$ of the structure (for trees and graphs, we assume this set to be the nodes, thus yielding node-labeled trees and graphs). A labeled morphism $f: \langle S, l_S \rangle \to \langle T, l_T \rangle$ is an ordinary morphism $f: S \to T$ which preserves labels, i.e., $\forall s \in U(S)\ l_T(f(s)) = l_S(s)$. These definitions yield the categories **LString** (labeled strings), **LTree** (labeled trees), and **LCGraph** (labeled, connected graphs), for which the definition of covers is as before.

2.2. Sheaves

Sheaf theory studies the global consequences of locally defined properties [31, 37, 15]. The notion of "local" is characterized using a topology. A map which assigns a set (e.g., a set of occurrences, a set of functions, etc.) to each object of a topology is called a *sheaf* if the map is defined "locally", i.e., the value of the map on an object can be uniquely obtained from its values on any cover of that object.

Besides mapping each object to a set, a sheaf maps each arrow in the topology to a "restriction" function in the opposite direction. In most of the examples we will consider, the objects in the topology are constraints of some kind, and the sets to which these objects are mapped are sets of entities satisfying the constraints (i.e., the denotations of the constraints). The contravariance of the sheaf arises from the fact that for any inclusion (in the topology) of a weaker constraint into a stronger constraint, there will be more entities satisfying the weaker constraint than the stronger one, thus inducing an inclusion of denotations in the opposite direction. For the case of pattern matching, an occurrence of a pattern can be treated as an entity that satisfies the constraint of "looking like the pattern". A partial occurrence satisfies the weaker constraint of looking like a piece of the pattern. In general, there will be more partial occurrences than full occurrences, since not all partial occurrences need be extendible to full ones.

The transition from locally defined properties to global consequences happens via a compatible family of elements over a cover of an object. A cover of an object can be viewed as providing a decomposition of that object into simpler objects. The sheaf assigns a set to each element of the cover (i.e., each piece of the original object). A choice of elements from these sets, one for each piece, forms a compatible family if the choice respects the mappings by the restriction functions and if the elements chosen agree whenever two pieces of the cover overlap. If such a locally compatible choice induces a unique choice for the object being covered (a global choice), then the condition for being a sheaf is satisfied. For pattern matching, a compatible family of partial occurrences uniquely extends to a full occurrence.

To formalize the intuitive description of sheaves given above, we need some preliminary definitions.

Definition 2.9 (*Contravariant hom-functor*). For any object a of $\mathscr{C}$, the contravariant hom-functor associated with a, $\hom_{\mathscr{C}}(-,a):\mathscr{C}^{\mathrm{op}}\to\mathbf{Set}$, is defined by the following assignments:
for any object b in $\mathscr{C}$,
$\hom_{\mathscr{C}}(-,a)(b)=\hom_{\mathscr{C}}(b,a)=$ the set of arrows from b to a in the category $\mathscr{C}$;
for any arrow $f:b\to c$ in $\mathscr{C}$,
$\hom_{\mathscr{C}}(-,a)(f):\hom_{\mathscr{C}}(c,a)\to\hom_{\mathscr{C}}(b,a)$ is defined by $g\mapsto g\circ f$.

In Definition 2.1, we defined a sieve as a collection of arrows closed under right composition. A sieve has additional structure; this structure is highlighted by representing a sieve as a collection of arrows indexed by their domains, i.e., as a functor. A sieve becomes a sub-functor of a hom-functor when we retain only those arrows which are present in the sieve.

Definition 2.10 (*Sieve: functor representation*). A sieve $R=\{a_i\xrightarrow{f_i}a\mid i\in I\}$ on an object a of a category $\mathscr{C}$ can be represented as a sub-functor of the hom-functor $\hom_{\mathscr{C}}(-,a)$ as follows:

$$b\mapsto\{f\mid f\in R \text{ and } \mathrm{domain}(f)=b\},$$

$$c\xrightarrow{g}b\mapsto g^*,\quad \text{where } g^*:f\mapsto f\circ g.$$

The underlying structure of a sheaf is that of a multifunction.

Definition 2.11 (*Presheaf*). A presheaf on a category $\mathscr{C}$ is a contravariant functor from $\mathscr{C}$ to the category of sets **Set**.

A sheaf is a presheaf that satisfies an additional "gluing" condition, i.e., "local" information is sufficient to uniquely define the values in the codomain.

Definition 2.12 (*Sheaf*). A sheaf on a site $\langle\mathscr{C},J\rangle$ is a presheaf $F:\mathscr{C}^{\mathrm{op}}\to\mathbf{Set}$ such that, for every object a of $\mathscr{C}$ and every covering sieve $R\in J(a)$, each morphism $R\to F$ in $\mathbf{Set}^{\mathscr{C}^{\mathrm{op}}}$ has exactly one extension to a morphism $\hom_{\mathscr{C}}(-,a)\to F$.

The definition above implicitly treats the sieve R as a functor. A morphism $\tau:R\to F$ in the functor category $\mathbf{Set}^{\mathscr{C}^{\mathrm{op}}}$ is a natural transformation, and is a concise way of representing a *compatible family* of elements on a cover R. Translating into elementary language, for a sieve $R=\{a_i\xrightarrow{f_i}a\mid i\in I\}$ on the object a, a compatible family of elements of F on the sieve R is a collection of elements $\{s_i\in F(a_i)\mid i\in I\}$, one for each arrow in the sieve R, which are compatible in the sense that, for any arrow $u:a_i\to a_j$ in $\mathscr{C}$ for which $f_i=f_j\circ u$, the function $F(u)$ maps s_j onto s_i.

Morphisms of the form $\hom_{\mathscr{C}}(-, a) \to F$ bijectively correspond, by the Yoneda lemma (see, for example, Mac Lane [23, Section III.2]), to elements of the set $F(a)$. Thus, the sheaf condition states that there is a bijection between compatible families on any cover of a – families of locally defined entities – and elements of $F(a)$ – globally defined entities.

We give below a few examples of sheaves to illustrate locally defined properties and the sheaf condition. Example 3.3 gives an example of a sheaf of occurrences for trees. Other examples of sheaves are given in Section 5.

Example 2.13 (*Books in a library*). Consider a site $\mathscr{T}$ in which objects are time intervals[5] and arrows are inclusions. An interval $[s, t]$ is covered by a family of intervals $\{[s_i, t_i] \mid i \in I\}$ if $\bigcup_{i \in I} [s_i, t_i] = [s, t]$. With respect to a particular library, define a contravariant functor $B: \mathscr{T}^{\mathrm{op}} \to \mathbf{Set}$ as follows:

For any interval $[s, t]$,

$B([s, t])$ is the set of books which are present in the library throughout the interval $[s, t]$.

For any inclusion of intervals $f: [s, t] \hookrightarrow [u, v]$,

$B(f)$ is the restriction function which maps each book onto itself. A book present in the library throughout the larger interval $[u, v]$ is obviously present during the sub-interval $[s, t]$.

This functor is a sheaf because, if $\{[s_i, t_i] \mid i \in I\}$ covers $[s, t]$, and if a book is present in the library throughout each of the intervals $[s_i, t_i]$, then it is also present throughout $[s, t]$.

The sheaf of library books illustrates the slogan

if a property is locally true over a cover of an object,
then it is true over the entire object,

and shows how sheaves connect local and global properties. In the graph-coloring sheaf (Example 2.14), it is possible to connect the chromatic number of a graph (a global property) with colorings of subgraphs (local properties).

Example 2.14 (*Graph coloring*). Consider the site of undirected, connected graphs described in Example 2.5. Let us confine our attention to a sub-category $\mathbf{UCGraph}_{\hookrightarrow}$ of **UCGraph** which contains all the objects but only inclusion arrows. Consider the task of coloring such graphs with at most k colors. Define a contravariant functor $C: \mathbf{UCGraph}_{\hookrightarrow}^{\mathrm{op}} \to \mathbf{Set}$ as follows:

For any graph G,

$C(G)$ is the set of all k-colorings of the graph G.

For any graph inclusion $f: G \hookrightarrow H$,

$C(f)$ is the function which restricts the colorings of H to G. If a graph H has a k-coloring, then each of its subgraphs also has a k-coloring.

[5] It does not matter whether these intervals are open, closed, or any mixture of these.

This functor is a sheaf because, if $\{G_i \mid i \in I\}$ covers G, and if $\{c_i \in C(G_i) \mid i \in I\}$ is a family of colorings such that the colorings agree on intersections among the graphs G_i, then the c_i's induce a unique coloring of the entire graph G.

Example 2.15 (*Sheaf of functions*). Let D be a set (the domain). The powerset $\mathscr{P}(D)$ forms a category with objects being subsets of D and arrows being inclusions. We obtain a site by defining a cover of a set X to be a family of sets $\{X_i \mid i \in I\}$ such that $\bigcup_{i \in I} X_i = X$. Let R be another set (the range). Define a contravariant functor $F : \mathscr{P}(D)^{\mathrm{op}} \to \mathbf{Set}$ as follows:
For any set $X \subseteq D$,
$F(X)$ is the set of all functions with domain X and range R.
For any inclusion $f : X \hookrightarrow Y$,
$F(f)$ is the map $g \mapsto g|_X (= g \circ f)$ which restricts the domain of a function.
This functor is a sheaf because, extensionally, a function is defined by specifying its value for each element of the domain. Thus, if the family $\{X_i \mid i \in I\}$ covers the set X, and $\{f_i : X_i \to R \mid i \in I\}$ is a family of functions such that

$$f_i|_{X_i \cap X_j} = f_j|_{X_i \cap X_j} \quad \text{for } i, j \in I,$$

then there is a unique function $f : X \to R$ such that

$$f(x) = f_i(x) \quad \text{for any } i \text{ such that } x \in X_i.$$

Example 2.16 (*Nonsheaves*). To help the reader understand the mechanics of the sheaf condition, here are two examples of functors which are not sheaves. The examples show that sometimes local properties alone are not sufficient to determine global properties.

For any site $\langle \mathscr{C}, J \rangle$, the topology J can be viewed as a functor as follows (see axiom 2 of a Grothendieck topology, Definition 2.2, for a definition of f^*):

$$a \mapsto J(a) \quad \text{for } a \in \mathrm{Obj}(\mathscr{C}),$$

$$f \mapsto f^* \quad \text{for } f \in \mathrm{Arr}(\mathscr{C}).$$

This functor is not a sheaf because, given a cover $\{a_i \xrightarrow{f_i} a \mid i \in I\}$ of an object a, and a compatible family of covers $\{c_i \in J(a_i) \mid i \in I\}$, there may be several covers on a which extend this family.

For another example, consider the site of sets defined in Example 2.4, but only with finite sets and inclusion arrows. Denoting this sub-category by $\mathbf{FinSet}_{\hookrightarrow}$, define a functor $P : \mathbf{FinSet}_{\hookrightarrow}^{\mathrm{op}} \to \mathbf{Set}$ which maps each set to the set of all its permutations, and each inclusion arrow to a restriction function on permutations (e.g., the inclusion $\{a, b\} \hookrightarrow \{a, b, c\}$ induces the restriction $bca \mapsto ba$). This functor is obviously not a sheaf.

3. A specification for pattern matching

We will model an occurrence of a pattern p in a target t as an arrow $p \to t$ in some site. For the pattern-matching problem to be computationally tractable, and to simplify the derivation, we make some additional assumptions:

(1) All objects in the site are finite.

(2) A finest cover[6] exists for each object and is finite.

(3) Every arrow in the site is an occurrence arrow.

(4) All covers are strict epimorphic families.

The last assumption needs some explanation. An epimorphic family of arrows is a generalization of a family of functions which are collectively surjective. A *strict* epimorphic family satisfies the additional condition that all the information about the codomain is contained in the family: arrows defined on elements of a cover of an object determine a unique arrow on the object [2, Exposé I, 10].

Definition 3.1 (*Strict epimorphic family*). Let $F = \{a_i \xrightarrow{f_i} a \mid i \in I\}$ be a family[7] of arrows. A family of arrows $G = \{a_i \xrightarrow{g_i} b \mid i \in I\}$ is said to be *compatible with* F if, for every object c, every pair of indices $i, j \in I$, and every pair of arrows $u: c \to a_i$, $v: c \to a_j$,

$$f_i \circ u = f_j \circ v \;\Rightarrow\; g_i \circ u = g_j \circ v.$$

The family F is said to be a *strict epimorphic family* (Fig. 2) if, for every family of arrows G which is compatible with F, there is a unique arrow $h: a \to b$ through which G factors, i.e.,

$$g_i = h \circ f_i \quad \text{for all } i \in I.$$

The assignment of the unique arrow h to G, provided by the definition of a strict epimorphic family given above, will be called a "gluing" operation in the rest of this paper. Intuitively, a cover $\{a_i \xrightarrow{f_i} a \mid i \in I\}$ provides a decomposition of the object a,

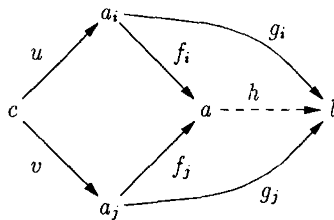


Fig. 2.

[6] The collection of covers of any object is ordered by inclusion. The finest cover is the least element, i.e., a cover none of whose elements can be further decomposed via a cover.

[7] Strict epimorphic implies epimorphic. Hence, there is no need to add this condition in the definition.

and the arrow h can be viewed as being obtained by gluing together the domains of the g_i's.

The notion of a strict epimorphic family is similar to, but weaker than, the notion of a colimit. Whereas a colimit requires universality with respect to a given diagram, the definition of a strict epimorphic family requires universality only with respect to the entire sieve generated by the family.

Example 3.2 (*Strict covers for strings*). In all the sites defined in Section 2.1 (except the site of strings, Example 2.7), the covers are strict epimorphic families. We now redefine the covers for strings so as to make them strict.

Given a string $\langle s, <_s \rangle$ and two elements $x, y \in s$, we say that x and y are *adjacent* in s if the string "xy" is a substring of s. A cover of a string s is a family of substrings $\{s_i \subsetneq s \mid i \in I\}$ such that, for any pair of elements x, y which are adjacent in s, there is a substring s_i in the cover in which x and y are adjacent. Thus, for the string "*abcd*", the following families are covers: {"*abcd*"}, {"*abc*", "*bcd*"}, {"*ab*", "*bc*", "*cd*"}. However, {"*ab*", "*cd*"} is not a cover because b and c are adjacent in "*abcd*" but they are not adjacent in any element of the cover. To understand the requirement about adjacency, observe that the cover has to not only cover the elements of a string but also cover the total order of the string.

A string can alternatively be considered to be a simple, acyclic path in a graph (i.e., all the nodes in the path are distinct). The definition of cover for strings is then a specialization of that for graphs.

3.1. The specification

Using the vocabulary introduced until now, we can abstractly characterize pattern matching as follows.

A *pattern-matching problem* consists of the following:

A data structure

> A site $\langle \mathscr{C}, J \rangle$ satisfying the assumptions outlined at the beginning of Section 3.

The input–output relation

> Given a pattern p and a target t, with $p, t \in \mathrm{Obj}(\mathscr{C})$,
> find the set of occurrences of p in t, i.e., compute $\hom_{\mathscr{C}}(p, t)$.

We will represent the occurrence relation as a collection of sheaves of the form $\hom_{\mathscr{C}}(-, t)$, one for each target t. The fact that the hom-functors are sheaves trivially follows from our assumption that all covers are strict epimorphic families. The sheaf condition for these sheaves allows us to find occurrences of a pattern by decomposing the pattern, finding occurrences of the pieces, and gluing the partial occurrences.

In Section 5, we will consider variants of our derivation which can be applied to sheaves other than hom-functors. The definition of such sheaves then becomes part of the problem specification.

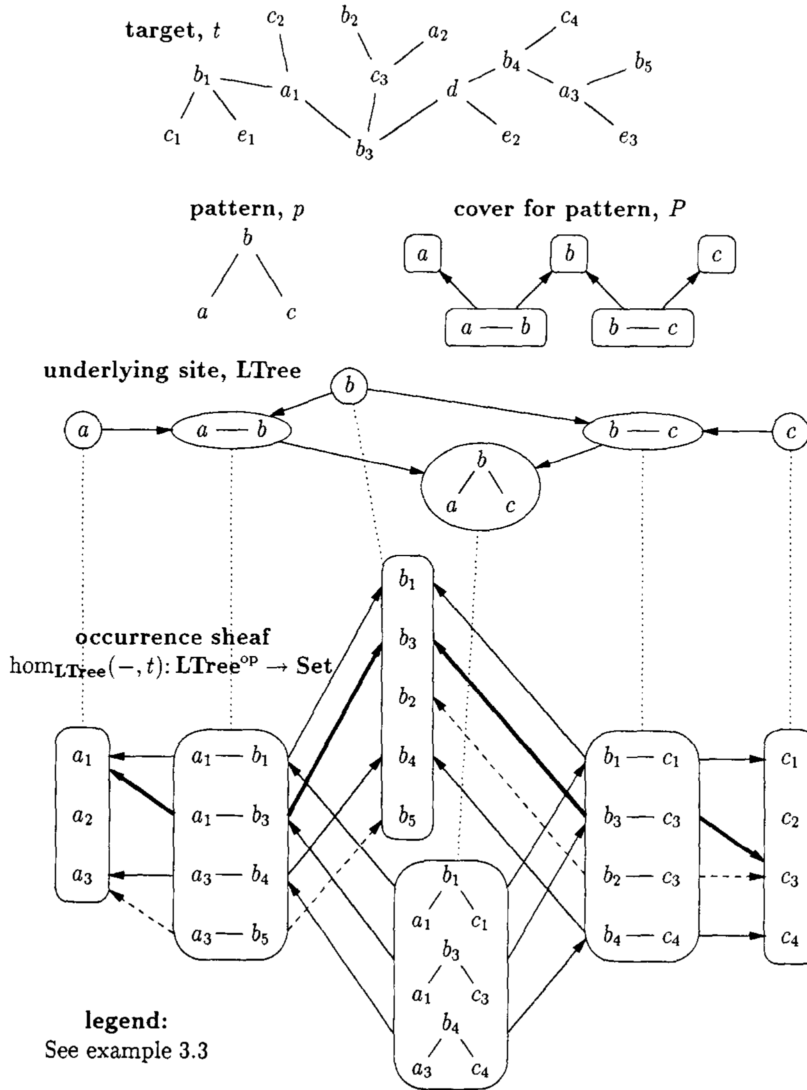


Fig. 3. Sheaf of occurrences: example with labeled trees.

Given that hom-functors are automatically sheaves, to specify a pattern-matching problem in this framework, we only have to specify the topology of the data structure involved. This entails defining the underlying category and the collection of covers satisfying the assumptions of finiteness, strictness, etc. A constructive proof that covers are strict will yield the gluing operation used to build a full occurrence from a compatible family of partial occurrences.

Example 3.3 (*An occurrence sheaf*). In Fig. 3, we show an example of the sheaf of occurrences of a pattern tree in a target tree. The underlying site **LTree**, of labeled

trees, is defined in Examples 2.6 and 2.8. In the figure, we show a specific target tree t, a specific pattern tree p, the finest cover P for the pattern, and a part of the sheaf of occurrences, $\hom_{\mathrm{LTree}}(-,t)$. The pattern cover is represented as a functor, but only the codomain of this functor is shown. The subscripts on the labels in the target tree are not part of the tree; they are a convenient notation for representing occurrence arrows. A sample-compatible family of partial occurrences is shown with bold arrows. Dotted arrows indicate partial occurrences which do not give rise to any compatible families.

4. Derivation of a pattern-matching algorithm

We now present a derivation of a generalized version of the KMP algorithm. We follow the general heuristic of converting the extensional description of the occurrence relation as a sheaf into an intensional description (an algorithm). The algorithm results from a synergy of four very general program synthesis/transformation techniques:

(1) *Divide and conquer.* Exploit the sheaf condition; assemble a full occurrence by gluing together partial occurrences.

(2) *Finite differencing.* Collect and update partial occurrences incrementally while traversing the target.

(3) *Backtracking.* Instead of saving all partial occurrences, save just one; when this partial occurrence cannot be extended, fail back to another.

(4) *Partial evaluation.* Precompute pattern-based (and, therefore, constant) computations.

The formal basis of the implementation strategies and transformations we use is given in the appendix. A rigorous derivation, with the theories represented using algebraic specification, is given in the author's dissertation [36].

Notation. For the rest of this section, we assume a site $\langle \mathscr{C}, J \rangle$ satisfying the assumptions of Section 3. The hom-functor $\hom_{\mathscr{C}}(-,t)$ will be written sometimes as h^t. The category of presheaves on the category $\mathscr{C}$ will be denoted by $\mathbf{PreShv}(\mathscr{C})$, and the category of sheaves on the site $\langle \mathscr{C}, J \rangle$ by $\mathbf{Shv}\langle \mathscr{C}, J \rangle$. When a cover is denoted by a family of arrows such as $\{p_i \to p\}$, the index varies over the arrows; hence, if there are two different arrows $q \rightrightarrows p$ from q into p, these will be denoted by $p_i \to p$ and $p_j \to p$, with $i \neq j$ and $p_i = p_j = q$.

4.1. Decomposing the pattern

We first exploit the sheaf condition on the occurrence sheaf $\hom_{\mathscr{C}}(-,t)$ to produce a problem reduction strategy [35] for enumerating the occurrences of a pattern in a target:

(1) *Decompose.* Choose a cover $\{p_i \to p\}$ for the pattern p.

(2) *Solve sub-problems.* Find occurrences of elements of the cover, i.e., find partial occurrences $p_i \to t$ of the pattern.

(3) *Compose.* Glue together partial occurrences to obtain full occurrences $p \to t$.

(4) *Base case.* For indecomposable pieces p_i of the pattern, decompose the target; see Section 4.2.

We can simplify solving the sub-problems by using the stability of covers under refinement (the third axiom of a Grothendieck topology) and by choosing the finest cover for the decomposition. This choice eliminates the recursion in step 2; other choices are possible, provided the topology is "nice" [36]. The gluing operation of step 3 is the constructive version of the bijection given by the sheaf condition (the notation $\tilde{\to}$ indicates an arrow which is an isomorphism),

$$\mathrm{Nat}(P, h^t) \tilde{\to} h^t(p),$$

where P is the chosen cover of the pattern. Since we have chosen the occurrence sheaf to be a hom-functor, the above gluing operation can be obtained from the definition of a strict epimorphic family. The unique arrow provided by the latter definition is ultimately obtained from the definition of the data structure which forms the site.

4.2. Decomposing the target

The problem has been reduced to enumerating occurrences of pieces of the pattern. We solve this problem by decomposing the target; however, we have to be careful, because occurrences may be split among pieces of the target (consider the occurrence "*bcd*" $\subsetneq$ "*abcde*" when the target is split into "*abc*" and "*cde*"). Thus, the appropriate structure to decompose is the entire sheaf. We can decompose the sheaf $\hom_{\mathscr{C}}(-, t)$ using a problem reduction strategy, based on the fact that the functor

$$\varepsilon_{\langle \mathscr{C}, J \rangle} : \mathscr{C} \xrightarrow{y} \mathbf{PreShv}(\mathscr{C}) \xrightarrow{sh} \mathbf{Shv}\langle \mathscr{C}, J \rangle$$

carries covers to epimorphic families (Theorem A.3); y is the Yoneda embedding and sh is the sheafification functor. Intuitively, a decomposition of the target induces a decomposition of the corresponding sheaves. Here is the strategy:

(1) *Decompose.* Choose a cover $\{t_j \to t\}$ for the target t.

(2) *Solve sub-problems.* Build the occurrence sheaves $\hom_{\mathscr{C}}(-, t_j)$ for each piece of the target.

(3) *Compose.* Combine these sheaves to produce the occurrence sheaf $\hom_{\mathscr{C}}(-, t)$.

(4) *Base case.* For indecomposable pieces t_j of the target, see Section 4.3.

The above composition operation can be expressed as

$$\hom_{\mathscr{C}}(-, t) = \underrightarrow{\mathrm{Colim}}\, \hom_{\mathscr{C}}(-, t_j),$$

provided we choose arrows in the cover $\{t_j \to t\}$ to be monics (see Theorem A.4 and the discussion before it). The colimit above is taken in the category of sheaves $\mathbf{Shv}\langle \mathscr{C}, J \rangle$.

Using the fact that the sheafification functor is a left adjoint (see Theorems A.2 and A.5), we can reduce the colimit to

$$\hom_{\mathscr{C}}(-,t) = sh \circ \underrightarrow{\mathrm{Colim}}\, \hom_{\mathscr{C}}(-,t_j),$$

where the colimit is now in the category of presheaves **PreShv**($\mathscr{C}$), and can be computed pointwise (roughly, by computing the union $\hom_{\mathscr{C}}(p_i,t_1) \cup \hom_{\mathscr{C}}(p_i,t_2) \cup \cdots$ for each p_i).

4.3. Building occurrence arrows from pieces

Combining the schemes for decomposing the pattern and the target, we see that we only need the values of $\hom_{\mathscr{C}}(p_i,t)$, which in turn depend on the values of $\hom_{\mathscr{C}}(p_i,t_j)$.[8] We can now compute $\hom_{\mathscr{C}}(p,t)$, which is our goal, in two ways: (i) sheafify the colimit of the $\hom_{\mathscr{C}}(-,t_j)$'s obtained above and read off the value $\hom_{\mathscr{C}}(p,t)$, or (ii) only compute the part of the colimit which gives the values of $\hom_{\mathscr{C}}(p_i,t)$, and pass via the sheaf condition to the value of $\hom_{\mathscr{C}}(p,t)$. We will follow the (apparently simpler) latter approach here; we will return to the former in Section 4.8.

(1) *Decompose.* Choose the finest cover $\{p_i \to p\}$ for the pattern p, and any monic cover $\{t_j \rightarrowtail t\}$ for the target t.

(2) *Solve sub-problems.* Find the elementary occurrences $p_i \to t_j$, thus filling in the values of $\hom_{\mathscr{C}}(p_i,t_j)$ in the functors $\hom_{\mathscr{C}}(-,t_j)$.

(3) *Compose.*

(a) *Codomain.* Combine the partially filled functors $\hom_{\mathscr{C}}(-,t_j)$ (via a colimit of presheaves) to obtain a partially filled functor $\hom_{\mathscr{C}}(-,t)$, i.e., fill in the values of $\hom_{\mathscr{C}}(p_i,t)$ by composing the elementary occurrences $p_i \to t_j$ with the arrows in the cover $t_j \rightarrowtail t$.

(b) *Domain.* Glue together the partial occurrences $p_i \to t$ [via the sheaf condition on $\hom_{\mathscr{C}}(-,t)$] to obtain full occurrences $p \to t$.

(4) *Base case.* Generation of elementary occurrences $p_i \to t_j$ is dependent on the underlying site/data structure; see also Section 4.10.

This strategy for computing occurrences is a formalization of the intuitive description of matching given in Section 1 and Fig. 1. Figure 4 informally represents the process for strings.

4.4. The abstract pattern-matching problem

The computing of occurrences by decomposing the pattern and the target has not changed the computational aspect of the problem significantly (enumerating

[8] Since colimits in **Preshv**($\mathscr{C}$) are computed pointwise, and because we have assumed that the pattern pieces p_i come from the finest cover, sheafification will not modify the values of $\hom_{\mathscr{C}}(p_i,t_j)$.

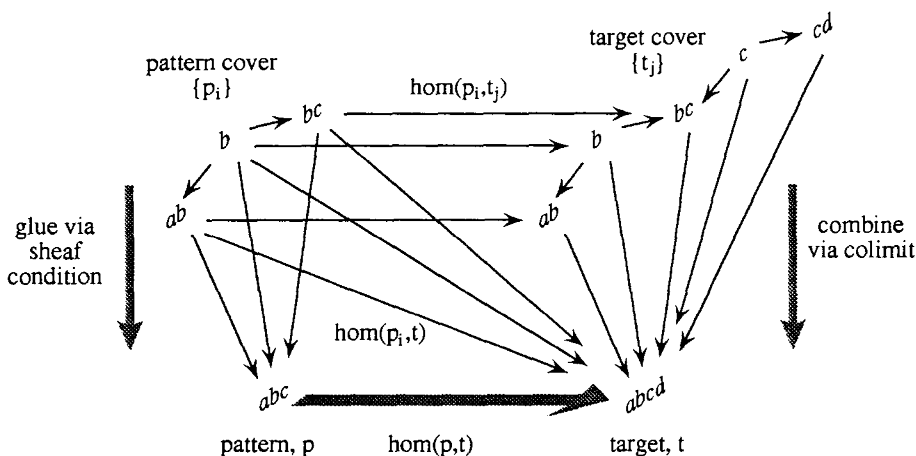


Fig. 4. Building occurrence arrows from pieces.

compatible families of partial occurrences is the real problem; see Section 4.5). However, the strategy has a *descriptive* advantage: we have abstracted away the specific data structures involved by using the device of Grothendieck topologies. The problem has now been recast as follows:

> Find occurrences of the graph of the pattern cover in the graph of the occurrence sheaf.

This description is now in the language of categories, functors, and natural transformations, abstract mathematical structures which have been studied extensively. Moreover, this description has the advantage that such graphs can be decomposed and manipulated in more convenient ways than Grothendieck topologies (see Example 4.2). Thus, our approach can be seen as a generic method for pattern matching, parameterized by a Grothendieck topology.

Example 4.1. We show how the strategy of Section 4.3 can be instantiated for the tree occurrence sheaf shown in Fig. 2. The pattern p is decomposed into the finest cover P as shown; this cover consists of single nodes or single edges. The target t is decomposed likewise; this decomposition occurs implicitly when the target is traversed and each new edge is encountered. Traversing the target generates elementary occurrences – $\mathrm{hom}_{\mathrm{LTree}}(p_i, t_j)$ – such as a_1–b_1, b_3–c_3, etc. The colimit of the functors $\mathrm{hom}_{\mathscr{C}}(-, t_j)$ is just a pointwise union; so, these elementary occurrences are used to populate the part of the sheaf shown in the figure. Once the requisite part of the sheaf is filled, we obtain full occurrences – $\mathrm{hom}_{\mathrm{LTree}}(p, t)$ – using the sheaf condition. This is done by enumerating compatible families on the pattern cover – a sample family is shown with bold arrows – and gluing them together. The gluing operation for trees is straightforward: combine a collection of partial functions on p to obtain a total function on p. The reader may note that compatible families are just occurrences of the graph of the pattern cover in the graph of the occurrence sheaf.

4.5. Enumerating compatible families

In the problem reduction strategy described above, finding elementary occurrences, $p_i \to t_j$, is the "base case", and is usually trivial (e.g., identity arrows), or dependent on the particular data structures and occurrence relation used. We will consider certain data-structure-independent aspects in Section 4.10.

The most complex part of the strategy above is using the sheaf condition, and consists of two steps:

(1) Enumerate compatible families of partial occurrences, $\mathrm{Nat}(P, h^t)$, where P is the chosen cover of the pattern.

(2) Glue together such families to obtain full occurrences.

The gluing operation is dependent on the site and we do not consider it further. There are several ways to compute compatible families. A simple procedure is to use the definition of a natural transformation; this yields a generate-and-test algorithm. Another way is to exploit the connection with limits (see SGA4 [2, Exposé I, Section 3.5]):

$$\mathrm{Nat}(P, h^t) \cong \varprojlim_{\langle p_i \to p \rangle \in \mathscr{C}/P} h^t(p_i),$$

where $\mathscr{C}/P$ is the cover P represented as a comma category. The above limit can be computed via products and equalizers [23, p. 109, 29, p. 82] (the latter has an algorithm).

Since we are interested in arriving at the KMP algorithm, we follow a different strategy: just as the matching problem has been "lifted" to that of graph matching, so can the topology be lifted to a topology on the covers themselves. This allows us to decompose the pattern cover into pieces and build compatible families piecemeal.

Compatible families $\mathrm{Nat}(P, h^t)$ form a functor as P varies:[9]

$$\mathrm{Nat}(-, h^t) : \mathbf{PreShv}(\mathscr{C})^{\mathrm{op}} \to \mathbf{Set}$$

$$\begin{array}{ccc} F & \mapsto & \mathrm{Nat}(F, h^t) \\ \tau \downarrow & & \uparrow _ \circ \tau \\ G & \mapsto & \mathrm{Nat}(G, h^t) \end{array}$$

For the case of pattern covers, τ is usually an inclusion of a piece of the pattern cover, and $_ \circ \tau$ is the restriction of a compatible family (over the entire pattern cover) to that piece.

Let the pattern cover be decomposed via a colimit ("shared union")

$$P = \varinjlim P_x.$$

[9] A cover is a sieve and, hence, a functor and an object of $\mathbf{PreShv}(\mathscr{C})$.

Since the functor $\mathrm{Nat}(-, h^t)$ carries colimits to limits,[10] we have

$$\mathrm{Nat}(P, h^t) = \underleftarrow{\mathrm{Lim}}\ \mathrm{Nat}(P_x, h^t).$$

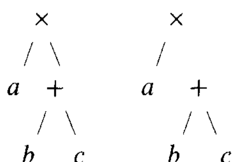
If we choose the topology on covers to be that defined in Section 4.6, then the functor $\mathrm{Nat}(-, h^t)$ actually forms a sheaf:

The situation at the base level is mirrored in the functor category.

4.5.1. Why ascend to functor categories?

One may ask why we should be dealing with functor categories and topologies on covers; why not use an appropriate topology at the base level? This is indeed possible, if the underlying site provides enough decompositions to enable occurrences to be computed easily. Sometimes, intermediate covers may not exist, thus not providing "stepping stones" in between elementary occurrences and full occurrences. Trees are an example.

Example 4.2. Consider the tree pattern on the left below, in the site **LTree** (see Example 2.8):



When the target is traversed depth-first, and occurrences are assembled bottom-up, it is possible to have a partial occurrence of the kind shown on the right above, which is obviously not a tree. A similar situation arises when assembling occurrences of connected graphs: partial occurrences during intermediate stages need not be connected.

In examples such as those above, we can rectify the lack of decomposition opportunities for objects by expanding the underlying category of the site and altering the topology accordingly. Rather than do this on a site-by-site basis, our approach using a topology on covers shows a systematic and general way of accomplishing this.

4.6. Decomposition of covers

We will now define a topology on covers which makes $\mathrm{Nat}(-, h^t)$ a sheaf, and which provides more flexible decompositions than those given by the covers in the underlying site.

[10] This is true of all contravariant hom-functors [23, p. 112]; "Nat" is just the name of the homsets in the category $\mathbf{Preshv}(\mathscr{C})$ of functors and natural transformations.

Definition 4.3 (*A topology on sieves*). Given two functors $F, G : \mathscr{C}^{\mathrm{op}} \to \mathbf{Set}$ on a category $\mathscr{C}$, an inclusion of functors $F \dot{\subsetneq} G$ is a natural transformation, each of whose components is an inclusion arrow, i.e., for each object $c \in \mathscr{C}$, $F(c) \subseteq G(c)$. Consider the category of all sieves in $\mathscr{C}$ together with the sub-functor relation. A cover for F is a family $\{F_x \dot{\subsetneq} F\}$ of sub-functors such that

$$\bigcup_x F_x = F, \text{ i.e., } \forall c \in \mathscr{C} \bigcup_x F_x(c) = F(c).$$

The collection of all sub-functors of a sieve S (or sub-sieves of S) forms a poset. The union of two sub-sieves, written as $X \cup Y$, is just the pointwise union of the two functors, or the union of the arrows in each of the sieves. The intersection $X \cap Y$ is similarly defined pointwise.

Definition 4.4 (*Prime arrow, prime sieve*). Given a sieve S in a category $\mathscr{C}$, an arrow $f \in S$ is said to be prime (in S) if it cannot be factored, i.e., $\nexists (h \in S, g \in \mathrm{Arr}(\mathscr{C}))\, f = h \circ g$. A sub-sieve $R \subseteq S$ is said to be prime (in S) if it is generated by a prime arrow $f \in S$, i.e., $R = \{f \circ g \mid g \in \mathrm{Arr}(\mathscr{C}) \text{ and codomain}(g) = \text{domain}(f)\}$.

The intent of Definition 4.4 is to specify the lifting of covers from a site to its functor category. Let $\langle \mathscr{C}, J \rangle$ be a site, p a pattern, and P its finest cover. Let $\{p_i \to p\}$ be the collection of prime arrows in P. Then the collection of prime sieves generated from the arrows p_i (the corresponding prime sieves will be denoted by P_i) covers the sieve P in the functor category (in fact, it is the finest cover, if P is).

The following definition of complement will be useful for decomposing covers into two pieces.

Definition 4.5 (*Complement of a sieve*). Given a pair of sieves $R \subseteq S$, the complement of R with respect to S, written as $S \sim R$, is defined to be the sieve generated by $S - R$ (set difference). When the sieve S is clear from the context, the complement of R will be written as R'.

Note that the intersection of complementary sieves need not be empty. Complementary sieves yield the following pushout:

$$\begin{array}{ccc} R \cap R' & \longrightarrow & R' \\ \downarrow & & \downarrow \\ R & \longrightarrow & S \end{array}$$

The purpose of lifting covers to the functor category was to allow decompositions of the pattern cover. The following definition provides the pieces.

Definition 4.6 (*Sub-pattern*). Given a pattern p and its finest cover P, a sub-pattern is the union of any collection of prime sieves in P.

Note that, since sub-patterns are defined at the functor category level, there need not exist an actual object corresponding to the sub-pattern in the underlying site (cf. the disconnected tree of Example 4.2).

4.7. *Building compatible families from pieces*

The derivation until now has left us with the task of computing compatible families $\mathrm{Nat}(P, h^t)$. Using the topology on covers described above, this reduces to computing a limit:

$$P = \underrightarrow{\mathrm{Colim}}\, P_x \;\Rightarrow\; \mathrm{Nat}(P, h^t) = \underleftarrow{\mathrm{Lim}}\, \mathrm{Nat}(P_x, h^t).$$

We do not seem to have made any progress, because we again have to compute a limit, which is equivalent to the problem of enumerating compatible families (see the beginning of Section 4.5). However, we can now simplify this limit to a pullback if the decomposition of P into P_x's consists of only two pieces (along with their intersection); the topology defined on covers guarantees that this can always be done (see Section 4.6). Pullbacks in **Set** can be easily computed: the pullback of $A \xrightarrow{f} C \xleftarrow{g} B$ is given by

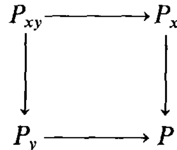
$$A \underset{C}{\times} B = \{\langle x, y\rangle \in A \times B \mid f(x) = g(y)\} = \bigcup_{c \in C} (f^{-1}(c) \times g^{-1}(c)),$$

with the arrows into A and B being the two projections from the product.

4.7.1. *Binary decomposition: pyramid algorithms*

At each stage, we decompose the pattern cover into two pieces, yielding a binary tree of decompositions. We obtain the following divide-and-conquer strategy [32]:

(1) *Decompose*. Split the pattern cover P into two pieces, P_x, P_y, and their intersection P_{xy}, as in the following pushout:



(2) *Recursive invocation*. Compute compatible families on the pieces, $\mathrm{Nat}(P_x, h^t)$ and so on, together with the restriction functions corresponding to the inclusions $P_{xy} \subsetneq P_x$ and $P_{xy} \subsetneq P_y$.

(3) *Compose.* Join compatible families via the following pullback:

$$\begin{array}{ccc} \mathrm{Nat}(P_{xy}, h^t) & \longleftarrow & \mathrm{Nat}(P_x, h^t) \\ \uparrow & & \uparrow \\ \mathrm{Nat}(P_y, h^t) & \longleftarrow & \mathrm{Nat}(P, h^t) \end{array}$$

(4) *Base case.* For indecomposable pieces of the cover, i.e., prime sieves, P_i, the compatible families are obtained from $\hom_{\mathscr{C}}(p_i, t)$ and their restrictions. These values are, in turn, computed from elementary occurrences $\hom_{\mathscr{C}}(p_i, t_j)$ by decomposing the target (Section 4.3). Ultimately, elementary occurrences depend on the underlying site.

The recursive invocation terminates because the pattern cover P is finite and, hence, the decomposition tree is finite. We, thus, obtain a top-down nonincremental algorithm. Such an algorithm could be implemented on a SIMD-style machine, a tree machine, or a data flow machine. The key observation is that the locality in the definition of a sheaf can be fruitfully mapped into the locality required by parallel machines.

Complexity. The worst-case complexity of the above algorithm for strings, with pattern length $|p|$ and target length $|t|$, is $\mathcal{O}(|p| \times |t|)$. The reason is that pieces of the pattern may occur within the pattern itself, thus necessitating multiple copies of elementary occurrences. We will later see how to avoid the generation of multiple copies.

4.8. An incremental algorithm

The binary decomposition algorithm works when the partial occurrences $p_i \to t$ are all available at once. We now investigate how to make this algorithm incremental, i.e., compute occurrences assuming that the target is traversed sequentially. This choice will ultimately lead us to the KMP algorithm. We will assume that traversing the target produces increments which are compatible families over some prime sieve of the pattern cover (see Section 4.10 for how these increments are generated).

Considering the expression for compatible families again,

$$P = \underrightarrow{\mathrm{Colim}}\ P_x \;\Rightarrow\; \mathrm{Nat}(P, h^t) = \underleftarrow{\mathrm{Lim}}\ \mathrm{Nat}(P_x, h^t),$$

and choosing a binary decomposition operation (pushout), we see that, to compute compatible families on P, given families on P_x, we need the families on the complement P'_x. Compatible families on P'_x can be computed by the same expression, but this time we need families on the complements with respect to P'_x. Continuing this reasoning, and considering that an increment can be a compatible family on *any* prime sieve, we see that we need families on all combinations of prime sieves. Using Definition 4.6, this means that we need compatible families on every sub-pattern.

To update the compatible families $\mathrm{Nat}(\Pi, h^t)$ for any sub-pattern Π, given increments on prime sieves P_i, we will use the technique of finite differencing [26].

Notation. To simplify the presentation, we will use the notation $_ \oplus _$ for pushouts and $_ \otimes _$ for pullbacks. This notation omits the third object and the two arrows required for pushouts and pullbacks. These should be evident from the context: the third object is usually the intersection, and the two arrows are inclusions. Also, we will assume the result of such an operation is not just an object, but also a pair of arrows; these arrows are required to compute subsequent pushouts or pullbacks!

Consider the following expression for the occurrence sheaf h^t (see Section 4.2):

$$\mathrm{hom}_{\mathscr{C}}(-, t) = sh \circ \underrightarrow{\mathrm{Colim}}\ \mathrm{hom}_{\mathscr{C}}(-, t_j),$$

where $\{t_j \rightarrowtail t\}$ is a monic cover for the target t, and sh is the sheafification functor. Now, assume that we already have computed the sheaf for the portion t of the target, and we incrementally traverse a little more, δt, such that we have the following pushout for the traversed portion of the target:

$$\begin{array}{ccc} t \cap \delta t & \longrightarrow & \delta t \\ \downarrow & & \downarrow \\ t & \longrightarrow & t \oplus \delta t \end{array}$$

The new portion of the target, δt, yields new elementary occurrences of the form $\mathrm{hom}_{\mathscr{C}}(p_i, \delta t)$ for some pieces p_i of the pattern. Applying the above equation, we get

$$\mathrm{hom}_{\mathscr{C}}(-, t \oplus \delta t) = sh \circ (\mathrm{hom}_{\mathscr{C}}(-, t) \oplus \mathrm{hom}_{\mathscr{C}}(-, \delta t)).$$

The standard formula for sheafification involves a colimit over covers (see Theorem A.2). Since we are not interested in the entire sheaf, we can use the following isomorphism[11] given by the sheaf condition:

$$\mathrm{hom}_{\mathscr{C}}(-, t \oplus \delta t)(p) \cong \mathrm{Nat}(P, h^t \oplus h^{\delta t}),$$

where p is the pattern, and P is the chosen pattern cover. The problem now is to update the compatible families $\mathrm{Nat}(P, h^t)$ as h^t changes. $h^t \oplus h^{\delta t}$ is a colimit of functors, which is computed pointwise; therefore, values of $h^t \oplus h^{\delta t}$ are given as set unions. Hence, we can use the distributive law of Lemma A.6.

The sheaf $\mathrm{Nat}(-, h^t)$ is given by the following:

(1) *Base case.* For a prime sieve P_i, $\mathrm{Nat}(P_i, h^t)$ is derived from the values $h^t(p_i)$ and their restrictions.

(2) *Induction.* For a sub-pattern Π which is not a prime sieve, let it be decomposed as the pushout $\Pi_x \oplus \Pi_y$. Then,

$$\mathrm{Nat}(\Pi, h^t) = \mathrm{Nat}(\Pi_x, h^t) \otimes \mathrm{Nat}(\Pi_y, h^t).$$

[11] To get an equality, rather than an isomorphism, the compatible families on the right have to be glued.

We can now distribute the increment $h^{\delta t}$ over the definition above, yielding the following:

(1) *Base case.* For a prime sieve P_i, $\mathrm{Nat}(P_i, h^t \oplus h^{\delta t})$ is derived from $(h^t \oplus h^{\delta t})(p_i)$, which is given by

$$(h^t \oplus h^{\delta t})(p_i) = h^t(p_i) \cup h^{\delta t}(p_i).$$

(2) *Induction.* For a sub-pattern Π which is decomposed as the pushout $\Pi_x \oplus \Pi_y$,

$$\begin{aligned}\mathrm{Nat}(\Pi, h^t \oplus h^{\delta t}) &= \mathrm{Nat}(\Pi_x, h^t \oplus h^{\delta t}) \otimes \mathrm{Nat}(\Pi_y, h^t \oplus h^{\delta t}) \\ &= (\mathrm{Nat}(\Pi_x, h^t) \otimes \mathrm{Nat}(\Pi_y, h^t)) \cup (\mathrm{Nat}(\Pi_x, h^t) \otimes \mathrm{Nat}(\Pi_y, h^{\delta t})) \\ &\quad \cup (\mathrm{Nat}(\Pi_x, h^{\delta t}) \otimes \mathrm{Nat}(\Pi_y, h^t)) \\ &\quad \cup (\mathrm{Nat}(\Pi_x, h^{\delta t}) \otimes \mathrm{Nat}(\Pi_y, h^{\delta t})).\end{aligned}$$

As discussed at the beginning of this section, the entire sheaf $\mathrm{Nat}(-, h^t)$ has to be updated, because we can get new elementary occurrences for any piece p_i of the pattern and, therefore, new compatible families on any prime sieve P_i; the constraint of a binary decomposition then requires all partial occurrences to be maintained.

Figure 5 shows the incremental maintenance of a sheaf as new elements are added to it. A sheaf spreads out in two dimensions, with increments in one dimension, and the pattern pieces in another. The distributive law on which the incremental maintenance is based exploits the fact that additions to the sheaf in the two dimensions are independent.

4.8.1. Binary decomposition: incremental version

The expression given above for computing $\mathrm{Nat}(\Pi, h^t \oplus h^{\delta t})$ can be simplified (some of the four terms on the right-hand side become empty) if we systematically update the sheaf bottom-up. We, thus, obtain the following algorithm, in which, for each increment, say on a prime sieve P_i, we view the pattern cover as decomposed into P_i and P_i' and compute compatible families via a pullback. (Note the difference between this and the pyramid algorithm of Section 4.7.1: there, a roughly equal split of the cover would minimize the height of the decomposition tree.) For each increment, we also update all partial families which may be affected.

(1) *Initial condition.* Start with an empty cache of partial occurrences.

(2) *Update.* Let the increment be over the prime sieve P_i.

(a) Install this increment into $\mathrm{Nat}(P_i, h^t)$:

$$\mathrm{Nat}(P_i, h^t \oplus h^{\delta t}) = \mathrm{Nat}(P_i, h^t) \cup \mathrm{Nat}(P_i, h^{\delta t}).$$

(b) For each sub-pattern Π which includes P_i, let P_i' be the complement. Compute $\mathrm{Nat}(\Pi, h^t \oplus h^{\delta t})$ by

$$\mathrm{Nat}(\Pi, h^t \oplus h^{\delta t}) = \mathrm{Nat}(\Pi, h^t) \cup (\mathrm{Nat}(P_i', h^t) \otimes \mathrm{Nat}(P_i, h^{\delta t})).$$

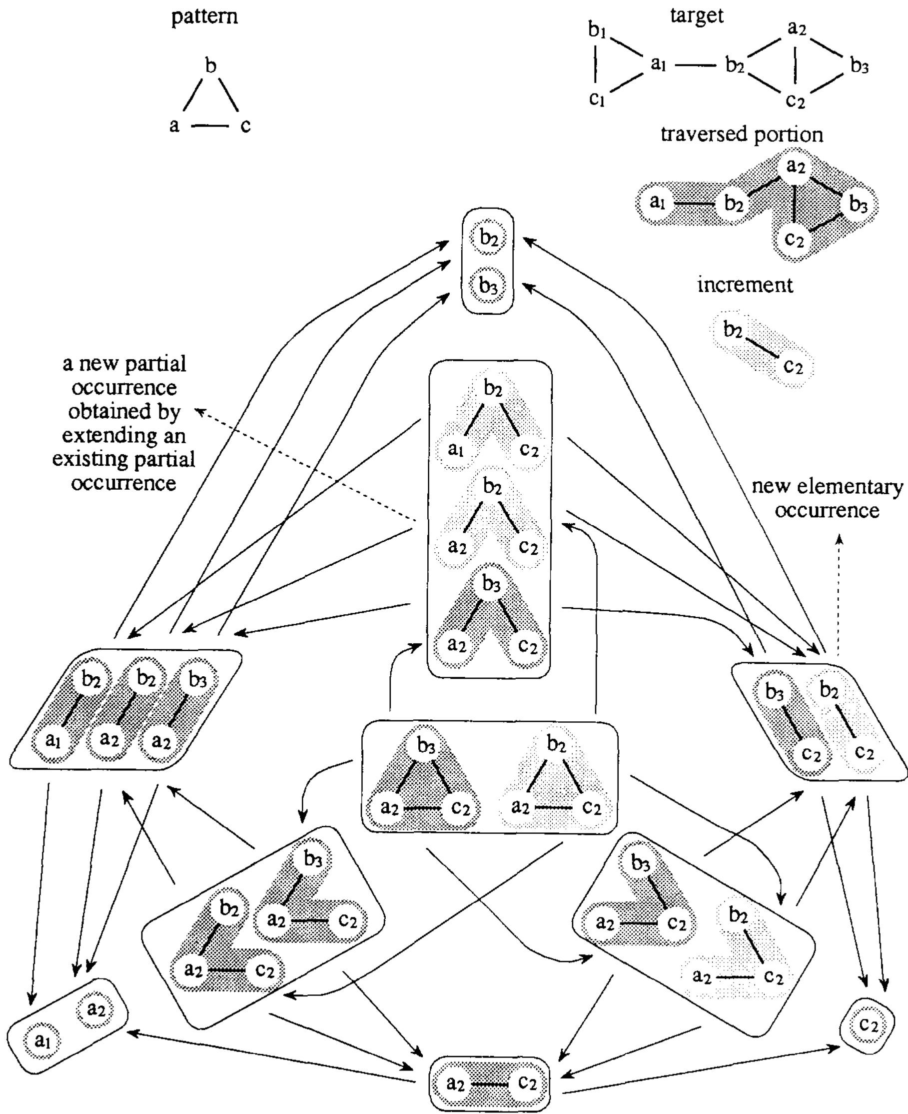
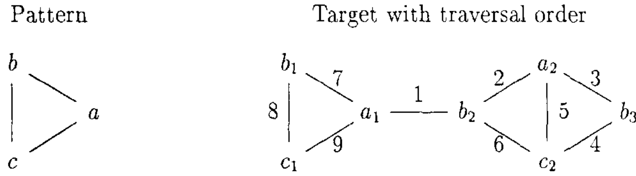


Fig. 5. Incremental maintenance of a sheaf.

Figure 6 shows a trace of the above incremental algorithm on graphs; the maintenance of the sheaf for the same example (step 6) is shown in Fig. 5.

4.9. Improving the incremental algorithm

The incremental algorithm given above is not very efficient: it saves and maintains all partial occurrences. In particular, for strings, with pattern length $|p|$ and target



States of the cache of partial occurrences:

1. a_1b_2
2. $a_1b_2,\ a_2b_2$
3. $a_1b_2,\ a_2b_2,\ a_2b_3$
4. $a_1b_2,\ a_2b_2,\ a_2b_3,\ b_3c_2,\ a_2b_3c_2$
5. $a_1b_2,\ a_2b_2,\ a_2b_3,\ b_3c_2,\ a_2b_3c_2,\ a_2c_2,\ a_2b_3c_2a_2,\ b_3c_2a_2,\ c_2a_2b_3,\ c_2a_2b_2$
6. $a_1b_2,\ a_2b_2,\ a_2b_3,\ b_3c_2,\ a_2b_3c_2,\ a_2c_2,\ a_2b_3c_2a_2,\ b_3c_2a_2,\ c_2a_2b_3,\ c_2a_2b_2,$ $b_2c_2,\ c_2a_2b_2c_2,\ b_2c_2a_2,\ c_2b_2a_2,\ c_2b_2a_1$
7. $a_1b_2,\ a_2b_2,\ a_2b_3,\ b_3c_2,\ a_2b_3c_2,\ a_2c_2,\ a_2b_3c_2a_2,\ b_3c_2a_2,\ c_2a_2b_3,\ c_2a_2b_2,$ $b_2c_2,\ c_2a_2b_2c_2,\ b_2c_2a_2,\ c_2b_2a_2,\ c_2b_2a_1,\ a_1b_1$
8. $a_1b_2,\ a_2b_2,\ a_2b_3,\ b_3c_2,\ a_2b_3c_2,\ a_2c_2,\ a_2b_3c_2a_2,\ b_3c_2a_2,\ c_2a_2b_3,\ c_2a_2b_2,$ $b_2c_2,\ c_2a_2b_2c_2,\ b_2c_2a_2,\ c_2b_2a_2,\ c_2b_2a_1,\ a_1b_1,\ b_1c_1,\ a_1b_1c_1$
9. $a_1b_2,\ a_2b_2,\ a_2b_3,\ b_3c_2,\ a_2b_3c_2,\ a_2c_2,\ a_2b_3c_2a_2,\ b_3c_2a_2,\ c_2a_2b_3,\ c_2a_2b_2,$ $b_2c_2,\ c_2a_2b_2c_2,\ b_2c_2a_2,\ c_2b_2a_2,\ c_2b_2a_1,\ a_1b_1,\ b_1c_1,\ a_1b_1c_1,$ $c_1a_1,\ a_1c_1b_1,\ c_1a_1b_1,\ c_1a_1b_2,\ a_1b_1c_1a_1$

Fig. 6. Trace of incremental algorithm on graphs (cf. Fig. 5).

length $|t|$, the complexity is $\mathcal{O}(2^{|p|} \times |t|)$. To improve the algorithm, we have to reduce the size of the cache of partial occurrences (i.e., the size of the sheaf $\mathrm{Nat}(-, h^t)$).

One way to reduce the size of the cache is to remove all partial occurrences in the cache which have no potential of being expanded, as indicated by the traversal of the target, e.g., reaching a leaf in a tree. For strings, assuming a left-to-right traversal, there can be at most $|p|-1$ partial matches which are potentially expandable (as opposed to $2^{|p|}$), thus reducing the complexity to $\mathcal{O}(|p| \times |t|)$. This optimization depends on the data structure involved and on the traversal mechanism; we do not consider it further here.

Another way to reduce the size of the cache is to exploit dependencies between partial occurrences: if a partial occurrence can be "derived" from another, then it can be removed from the cache (and regenerated later on, if necessary). Such an optimization conforms to our overall heuristic of converting an extensional representation into an intension: replace the extension of the cache by a generator.

These dependencies arise when a piece of the pattern occurs in the pattern itself. For example, given a partial occurrence $p_i \to t$ and an arrow $p_j \to p_i$ between pieces of the

pattern, we can immediately generate the partial occurrence $p_j \to p_i \to t$ by composition. We say that the latter occurrence is *subsumed* by the former. In Section 4.11, we will extend this subsumption relation to occurrences represented as compatible families. This induces a partial order on occurrences. Using this partial order, we will represent the cache of partial occurrences by its maximal elements; other elements are generated by composition when required.

4.10. Generation of elementary occurrences

We now consider the base case of the recursive algorithms presented until now, namely, the generation of elementary occurrences of the form $\hom_{\mathscr{C}}(p_i, t_j)$. Normally, there is a procedure which traverses the target and produces the pieces t_j. The specifics of generating elementary occurrences from these pieces depend on the particular data structure forming the site and the definition of the occurrence relation. What we are interested in here is a data-structure-independent feature, the generation of *multiple* occurrences from the same target piece t_j. In other words, a piece of the target can be "parsed" in several ways as a piece of the pattern. Figure 7 shows an example using labeled trees (the pattern pieces are numbered to distinguish them). In general, the multiple parses so generated may be independent. However, for common cases of pattern matching, the following property is true.

Subsumption property. If an atomic piece t_j of the target generates multiple elementary occurrences $\{p_x \to t_j \mid x \in X\}$, then there is an occurrence $p_i \to t_j$ in this set through which each of the other occurrences factors: $p_x \to p_i \to t_j$.

If this property is true of a site, then we can generate alternative occurrences by using information obtained by matching the pattern against itself. If not, we have to rely on the particular features of the site; for example, in Waltz filtering [39] (see also Section 5.6), the possible labelings of a junction are obtained from the physical properties of three-dimensional space.

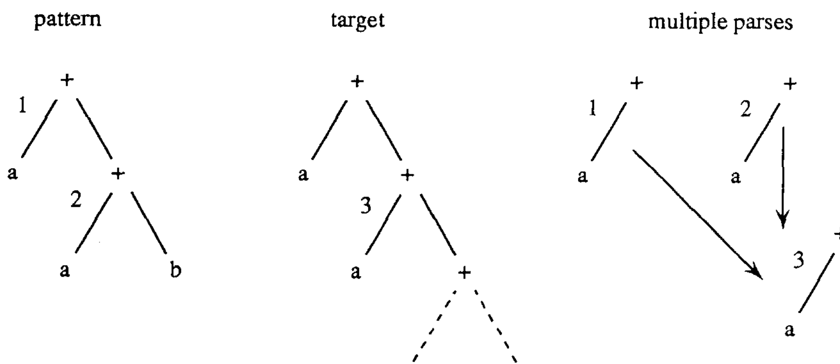


Fig. 7. Multiple parses of a piece of the target.

4.11. Matching the pattern against itself

The generation of multiple parses for a piece of the target can be extended to larger pieces of the target. For example, if $\mathcal{F}$ is a compatible family on a sub-pattern Π, then we can derive other compatible families by using other parses for portions of $\mathcal{F}$. Such derived families are said to be *subsumed* by $\mathcal{F}$. Just as we have lifted the topology of the underlying data structure to the level of covers, so can the subsumption relation informally defined in Section 4.9 be lifted to sub-patterns and partial occurrences represented by compatible families. This subsumption relation generates a structure called the *pattern–pattern functor*, which is a collection of sheaves, one for each sub-pattern, and is defined below (this functor is very similar to the Yoneda embedding of a category into its functor category).

Definition 4.7 (*Subsumption*). Given two sub-patterns Π and Π', a subsumption arrow $\Pi \to \Pi'$ is a natural transformation,[12] which is obtained as a compatible family of occurrences of pieces of Π in Π'.

Given two sub-patterns Π and Π', the sub-pattern Π' is said to be subsumed by Π, written as $\Pi' \preceq \Pi$, if there is a subsumption arrow $\Pi' \to \Pi$.

This relation can be extended to occurrences: a partial occurrence given by the composition $\Pi' \to \Pi \to h^t$ is said to be subsumed by the partial occurrence $\Pi \to h^t$.

Notation. Given a covering sieve P represented as a functor, we will denote by $\mathrm{Sub}(P)$ the category of all sub-functors of P, with the arrows being inclusion natural transformations.

Definition 4.8 (*Pattern–pattern functor*). The subsumption sheaf for a sub-pattern Π of a pattern cover P is the functor $\mathcal{S}^{\Pi}:\mathrm{Sub}(P)^{\mathrm{op}} \to \mathbf{Set}$ defined by

$$\Pi' \mapsto \{\Pi' \xrightarrow{s} \Pi \mid s \text{ is a subsumption arrow}\},$$

$$f:\Pi'' \to \Pi' \mapsto f^*:\mathcal{S}^{\Pi}(\Pi') \to \mathcal{S}^{\Pi}(\Pi''), \text{ where } f^* = _\circ f.$$

Let J_P denote the topology on sieves (Definition 4.3) restricted to $\mathrm{Sub}(P)$. The pattern–pattern functor for a pattern cover P is the functor $\Phi:\mathrm{Sub}(P) \to \mathbf{Shv}\langle \mathrm{Sub}(P), J_P \rangle$ defined by

$$\Pi \mapsto \mathcal{S}^{\Pi}, \text{ the subsumption sheaf for } \Pi,$$

$$f:\Pi \to \Pi' \mapsto \Phi(f):\mathcal{S}^{\Pi} \dot{\to} \mathcal{S}^{\Pi'}, \text{ with each component given by } f_* = f \circ _.$$

The pattern–pattern functor allows us to generate all the subsumed occurrences of a given occurrence. Such subsumed occurrences will be used to reduce the size of the cache in the algorithm of Section 4.13.

[12] Remember that sub-patterns are sub-functors of the pattern cover represented as a functor.

4.12. Computing the pattern–pattern functor

The pattern–pattern functor captures the occurrence relation among all sub-patterns of a given pattern. In trying to find an algorithm for the pattern-matching problem – find occurrences of a *single* pattern in a *single* target – we have reached the problem of finding occurrences of *multiple* patterns in *multiple* targets (note the similarity to the "RETE" algorithm [14]). This shows that precomputation for a problem may be more complex than the problem itself: the trade-off is between a complex precomputation (done once per pattern) and a simple matching algorithm.[13] This also explains why precomputation is exponential for trees [20].

Here is a nonincremental, divide-and-conquer scheme to compute the pattern–pattern functor. This is a reapplication of our derivation so far: each subsumption sheaf $\mathcal{S}^{\Pi}$ is built by passing via the sheaf condition from smaller arrows to bigger ones (domain decomposition); each such sheaf is "seeded" by taking a colimit of smaller sheaves (codomain decomposition).

(1) *Base case.* Seed the subsumption sheaf for any prime sieve P_i by installing the values of $\hom_{\mathscr{C}}(p_j, p_i)$ for all atomic pieces p_j of the pattern. Sheafify.

(2) *Induction.* For any sub-pattern Π which is not a prime sieve, choose a binary decomposition $\Pi = \Pi_x \oplus \Pi_y$ (pushout). Then the subsumption sheaf for Π is given by

$$\mathcal{S}^{\Pi} = sh \circ (\mathcal{S}^{\Pi_x} \oplus \mathcal{S}^{\Pi_y}).$$

The pattern–pattern functor can also be generated incrementally by traversing the pattern. In this case, we can also build the pattern cover along the way. This scheme is obtained by distributing an increment to the pattern over the above nonincremental scheme.

(1) *Initial condition.* Start with an empty pattern–pattern functor.

(2) *Update.* Let p_i be the increment to the pattern, and let P_i be the corresponding sieve.

- (a) *Subsumption sheaf for P_i.* Seed the sheaf with values of $\hom_{\mathscr{C}}(p_j, p_i)$ for known atomic pattern pieces p_j. Sheafify, again for known sub-patterns Π.
- (b) *Install new domains.* Expand Sub(P) by adding P_i, and $\Pi \cup P_i$ for all known sub-patterns Π. Install these new sub-patterns in each sheaf. Sheafify.
- (c) *Install new codomains.* Install the new sub-patterns (computed above) in the (domain category of the) pattern–pattern functor. For each new sub-pattern, compute the subsumption sheaf as the colimit of the sheaves on a cover.

There are several optimizations possible (by recursively applying parts of the KMP derivation) for computing the pattern–pattern functor and the subsumption sets required by the algorithm in Section 4.13. We do not pursue these here, because our primary goal is to show that the failure function of KMP can be explained as an instance of backtracking.

[13] It is frequently mentioned in the literature (e.g., [38]) that the precomputation and the matching parts of KMP are fundamentally the same. Our generalization shows that this similarity is only apparent; it is an accidental feature which is special to strings.

4.13. An algorithm using subsumption

In the previous few sections, we computed the dependencies between partial occurrences. We will now exploit these dependencies to reduce the size of the cache of partial occurrences maintained by the incremental matching algorithm. The subsumption relation is a preorder on partial occurrences; by taking the quotient under mutual subsumption, we obtain a partial order. The cache can then be represented by its maximal elements (i.e., by removing all subsumed occurrences). Now, we have to simulate the effect of the incremental algorithm of Section 4.8.1, which works on the entire cache. The problem can be visualized as filling in the following diagram:

$$\begin{array}{ccc} \mathrm{Nat}(-,h^t) & \xrightarrow{\text{representation by maximal elements}} & R(-,h^t) \\ \Big\downarrow{\scriptstyle \text{update}} & & \Big\downarrow{\scriptstyle ??} \\ \mathrm{Nat}(-,h^t \oplus h^{\delta t}) & \xrightarrow{\text{representation by maximal elements}} & R(-,h^t \oplus h^{\delta t}) \end{array}$$

To deduce the update function on the optimized cache, we exploit the fact that expanding a partial occurrence preserves subsumption, i.e.,

$$(\pi \preceq \pi' \ \wedge \ \phi \preceq \phi') \Rightarrow (\pi \otimes \phi \preceq \pi' \otimes \phi'),$$

where π, π', ϕ, and ϕ' are partial occurrences, and the expansions on the right are assumed to exist.

Thus, given an elementary occurrence, we try to expand each partial occurrence in the cache; for any partial occurrence which cannot be expanded, we generate its immediately subsumed occurrences and try to expand them, and so on. This procedure guarantees that all partial occurrences which would have been generated in the unoptimized cache are represented in the optimized cache as actual partial occurrences or as partial occurrences subsumed by others. In other words, we extend the representation of the cache just enough to accommodate all partial occurrences generated by the new increment.

The technique above is succinctly expressed by Hirschberg and Larmore [19]:

> The principle of failure functions is disarmingly simple: when searching for an extremal value within a sequence, it suffices to consider only the subsequence of items, each of which is the first feasible alternative of its predecessor.

Dijkstra formalizes the same principle in his linear search theorem [12]: to search for the largest element in a linear order which satisfies a given predicate, start from the maximum and search in decreasing order. Our update function is a generalization which works for any poset. Given some value, its immediately subsumed values ("first feasible alternatives") are the least upperbounds of the connected components in the poset of all subsumed values.

4.13.1. *Binary decomposition: incremental version with subsumption*

Here is the algorithm which represents the cache by its maximal elements, with a modified update function to handle subsumed occurrences. The modified update function can be systematically obtained from the algorithm which updates the entire cache (Section 4.8.1) by using the definition of subsumption; we omit the details because they are tedious.

(1) *Initial condition.* Start with an empty cache of partial occurrences.

(2) *Update.* Let the increment to the target be t_j. For each partial occurrence π (i.e., a compatible family on some sub-pattern Π) in the cache, do the following:

(a) *Expand.* For each elementary occurrence v (over the sieve P_i) generated by the increment, if v is compatible with π (i.e., the restrictions of v and π to $\Pi \cap P_i$ are equal), then add the expanded occurrence $\pi \otimes v$ to the partial occurrences on $\Pi \cup P_i$, and continue with the next π; otherwise, backtrack.

(b) *Backtrack.* Generate all immediately subsumed occurrences of π. Repeat the expansion procedure above for each. If there are no subsumed occurrences, goto 3.

(3) *Unconsumed increments.* If the increment t_j did not participate in any expansion in the previous item, add all unsubsumed elementary occurrences generated by t_j to the cache.

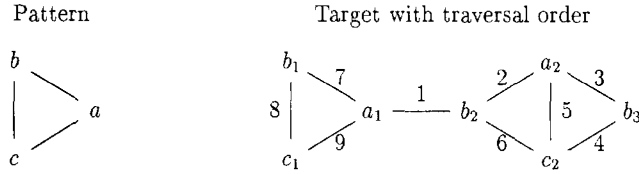
Replacing occurrences by subsumed occurrences is picturesquely called "sliding" the pattern. If we view the algorithm of Section 4.8.1 as search in a space where the states are partial occurrences, and the operations are expansion by elementary occurrences arising from traversing the target, the above scheme is an instance of dependency-directed backtracking. See [36] for a formal representation of the above algorithm as search.

Figure 8 shows a trace of the above algorithm on graphs; the example is the same as in Figs. 6 and 5. The trace incorporates two further optimizations: (1) a full occurrence can be discarded (after outputting it), and (2) if the traversal of the target indicates that there will be no further opportunities to expand a particular partial occurrence, then that occurrence can be discarded (the "traversal dead-end filter"). In both cases, to maintain the invariant that the cache contains the maximal elements of the full cache in the corresponding state, discarded occurrences have to be replaced by immediately subsumed ones.

Complexity. The above algorithm is a generalized version of the KMP algorithm. The complexity of the nonincremental algorithm of Section 4.7 arose from the fact that each piece of the target can generate multiple occurrence arrows. If we use the subsumption relation, then only one occurrence arrow need be generated; the others are subsumed and can be regenerated later, if necessary. Thus, for strings, the above algorithm for enumerating occurrences is linear in the size of the target.

4.14. *Instantiation for strings*

We briefly outline the instantiation of our derivation of a generalized KMP algorithm to strings. The underlying category, **LString**, of labeled strings is defined in



States of the cache of partial occurrences:

1. a_1b_2
2. a_1b_2, a_2b_2
3. a_1b_2, a_2b_2, a_2b_3
4. a_1b_2, a_2b_2, $a_2b_3c_2$ *—expand* a_2b_3
5. a_1b_2, $c_2a_2b_2$, $a_2b_3c_2a_2$ *—expand* a_2b_2 *and* $a_2b_3c_2$
   —filter the full occurrence $a_2b_3c_2a_2$
   —replace it by subsumed occurrences: $a_2b_3c_2$, $b_3c_2a_2$, $c_2a_2b_3$
6. a_1b_2, $a_1b_2c_2$, $c_2a_2b_2c_2$, $a_2b_3c_2$, $b_3c_2a_2$, $c_2a_2b_3$ *—expand* $c_2a_2b_2$ *and* a_1b_2
   —eliminate $c_2a_2b_2c_2$, $a_2b_3c_2$, $b_3c_2a_2$, $c_2a_2b_3$, $a_1b_2c_2$
   —using traversal dead-end filter
7. a_1b_2, a_1b_1
8. a_1b_2, $a_1b_1c_1$ *—expand* a_1b_1
9. a_1b_2, $a_1b_1c_1a_1$, $c_1a_1b_2$ *—expand* $a_1b_1c_1$ *and* a_1b_2
   —end of algorithm
   —traversal dead-end filter removes all partial occurrences

Fig. 8. Trace of generalized KMP on graphs (cf. Figs. 5 and 6).

Examples 2.7 and 2.8. We impose the additional restrictions that all the strings in the site are finite, and all arrows are monic. We choose the topology defined in Example 3.2 to make the covers strict epimorphic families. This site satisfies the assumptions of Section 3 and enables us to instantiate the derivation for strings.

For the incremental algorithm, we choose a left-to-right traversal of the target string (other choices are possible; this choice leads to the standard KMP algorithm). Observe that, in view of our definition of string covers, this traversal has to enumerate *pairs* of elements of the target string.[14] The left-to-right traversal has the property that

[14] A cover is a *set*; the requirement of strictness forces the representation of the linear order of a string via overlapping pairs. The standard KMP algorithm can be modeled by adding a linear order to covers, which provides a unique gluing with the simpler cover consisting of single elements of the string. Rather than generalize the definition of cover, we prefer to add this detail as an optimization to the final algorithm.

we need only save those partial occurrences which touch the right edge of the portion of the target already traversed; other partial occurrences have no potential of being expanded and can be deleted. Such partial occurrences are just occurrences of all prefixes of the pattern. This reduces the number of partial occurrences, in the worst case, for a pattern p of length $|p|$, from $2^{|p|}$ to $|p|$. The incremental algorithm of Section 4.8.1 is then the naive $\mathcal{O}(|p| \times |t|)$ algorithm.

Next, we can eliminate all but one partial occurrence, using the subsumption relation. When specialized to strings, subsumption is just the substring relation on the domains of occurrence arrows: $p \to t$ subsumes $q \to t$ if and only if q is a substring of p. When combined with the optimization above of retaining only expandable partial occurrences, the subsumption relation can be further specialized to the suffix relation on the domains of occurrence arrows. Here is the appropriate picture:

```
pattern   a b c a b c a b c
                          a   (subsumed occurrence)
                    a b c a   (subsumed occurrence)
              a b c a b c a   (partial occurrence)
target    a b c a b c a b c a a a b
```

The precomputation of the pattern–pattern functor is also simplified. Given that we are saving only occurrences of prefixes of the pattern, the subsumption relation assigns to each prefix of the pattern all prefixes which are also suffixes (this is the prefix–suffix problem). The subsumption relation (let us denote this by Σ) can be incrementally computed using the following distributive law:

$$\Sigma(x.i) = \{y.i \mid y \in \Sigma(x) \ \wedge \ \text{prefix}(y.i, p)\},$$

where the dot denotes concatenation of strings, p is the pattern, x is a sub-pattern (a prefix of p), and i is the increment produced by traversing the pattern. This computation can be further optimized to yield only the immediately subsumed string for each sub-pattern (this is the maximal prefix–suffix problem).

The above computation captures the essence of the standard KMP algorithm. Once the subsumption relation has been precomputed, we can optimize the incremental algorithm which updates the cache of partial occurrences. There is only one active partial occurrence at any time, because all other potentially expandable partial occurrences (which are suffixes of the current partial occurrence) are subsumed. If the current partial occurrence cannot be expanded, the algorithm backtracks to the next subsumed partial occurrence, and continues doing so until the current increment has been consumed.

5. Related algorithms

Our derivation of a generalized KMP algorithm works for sheaves other than occurrence sheaves, because we have not used properties which are specific to matching; in fact, we only defined occurrences to be arrows in some category. Here are some other problems which can be described using sheaves, and for which the KMP derivation (or parts of it) can be applied. Some of these problems are closely related to matching; others, such as the n-queens problem, are quite remote. However, all these problems share the common characteristic of attempting to satisfy a *locally* defined collection of constraints (see, especially, Waltz filtering and the n-queens problem), a concept which is nicely captured by a sheaf.

5.1. Multiple patterns

Given two patterns p and q, we say that the pattern $p \vee q$ occurs in a target if either p occurs or q occurs. If the two patterns do not intersect with each other, then finding occurrences of $p \vee q$ is straightforward: take the union of the occurrences of p and occurrences of q. If the two patterns do intersect, we can obtain the sheaf of occurrences for $p \vee q$ as a pushout (more generally, a colimit, if there are more than two patterns):

$$\begin{array}{ccc} \hom_{\mathrm{Sub}(p\cap q)}(-,t) & \longrightarrow & \hom_{\mathrm{Sub}(p)}(-,t) \\ \downarrow & & \downarrow \\ \hom_{\mathrm{Sub}(q)}(-,t) & \longrightarrow & \hom_{\mathrm{Sub}(p\vee q)}(-,t) \end{array}$$

In the diagram, Sub(–) denotes the poset of sub-patterns of a pattern. The above scheme essentially corresponds to the algorithm of Aho and Corasick for multiple string patterns [1].

5.2. Patterns with variables

Labeled data structures in which some of the labels are variables are easy to handle: just define the site appropriately. Here is an example with variable labels in trees.

Example 5.1 (*Expression trees*). Let Σ be a signature, i.e., a collection of sort names and a collection of operation names defined on these sorts. Each operation f is associated with a rank, $s_1, s_2, \ldots, s_n \to s$. The collection of expressions over the signature Σ is defined inductively as follows:

(1) The distinguished symbol v (for "variable") is an expression of sort s, for every sort s in Σ.

(2) If $c\colon \to s$ is a constant of sort s, then c is an expression of sort s.

(3) If $f: s_1, s_2, \ldots, s_n \to s$ is an operation, and $e_1, e_2, \ldots, e_n$ are expressions of sorts $s_1, s_2, \ldots, s_n$, then $f(e_1, e_2, \ldots, e_n)$ is an expression of sort s.

Expressions can be represented as rooted, ordered, labeled trees. A morphism of expression trees is a tree morphism (see Example 2.6) which is injective on nodes, preserves the order on the children of a node, and preserves labels (except v, which may map on to a node with any label).

A cover of an expression e is a family of morphisms $\{e_i \to e \mid i \in I\}$ such that for each sub-expression e' of e, there is at least one e_i such that $\text{root}(e_i) = \text{root}(e')$. The intent of this restriction is to ensure that not all of the e_i's are variables, which essentially contain no information about covering. For example, the expression $f(a, g(b, c))$ is covered by $f(a, v)$ and $g(b, c)$.

Variable patterns are somewhat more interesting. Consider the site of strings (Example 2.7). We can postulate a pattern V which matches any string; thus, this pattern is an initial object in the site. We can also have *typed* variable patterns, e.g., a pattern L_{10} may match only strings whose length is 10. Variables may be embedded in strings, yielding patterns such as aXb which matches any string starting with the letter a and ending with the letter b. Such patterns can be handled by changing the site appropriately (e.g., add an arrow from aXb to each string it matches). However, they complicate the generation of elementary occurrences (which is the base case of the divide-and-conquer step of our derivation). This is to be expected because variables are entities which are *algebraically* defined: sites only capture the geometry of the patterns, the algebra is represented in the codomain of a sheaf.

5.3. *Commutative/associative matching*

Matching modulo commutativity or associativity, or both, can be handled, again by using an appropriate site. The procedure for generating elementary occurrences has to account for these axioms.

Example 5.2 (*Expression trees, morphisms modulo commutativity*). The site defined in the previous example can be extended to incorporate commutativity. Let f be an operation in the signature which is commutative, i.e., $\forall a, b\ f(a, b) = f(b, a)$. A morphism is defined as before, except that the ordering on the children of a node labeled by f need not be preserved. The definition of cover is the same. This definition of morphism represents matching modulo commutativity. The definition can be obviously extended to several commutative operations, to associative operations, and to any mixture of these.

5.4. *Nonlocal properties, approximate matching*

Occurrence arrows may be constrained by some nonlocal properties, such as being a monic, or containing at most k mismatches. The former is a "moderately" nonlocal

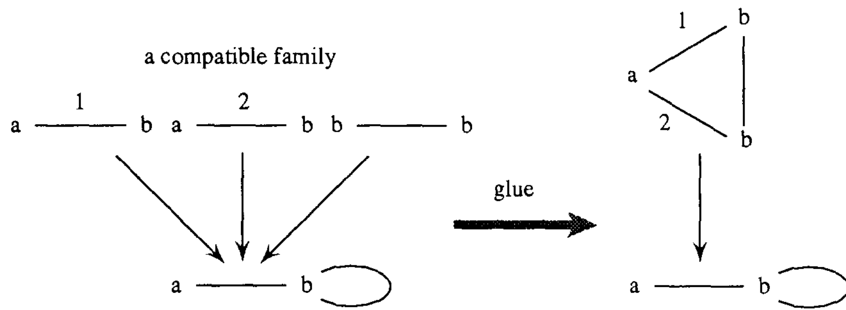


Fig. 9. An example showing that being monic is a nonlocal property.

property and can be handled by making the topology a little coarser. The latter is a truly global property of an occurrence and yields only a separated presheaf;[15] hence, compatible families have to be filtered (by a predicate which determines whether the number of mismatches is $\leqslant k$) before they can be glued.

Why is being a monic a nonlocal[16] property? Consider the example in the site of graphs shown in Fig. 9. The problem is that the topology (in which a graph can be covered by its set of edges) is too fine to detect the incompatibility between two maps whose domains are equivalent. This can be fixed by altering the topology so that two edges which can potentially be mapped onto the same edge are never split up in a cover. We can also solve the problem by only building compatible families which yield a monic after gluing.

5.5. *Context-free parsing: Earley's algorithm*

Earley's algorithm for context-free parsing [13] scans the input string from left-to-right, accumulating partial parses (left contexts) of the input seen so far. For each input increment, some partial parses in the current set of parses are expanded, and some are discarded, depending upon the compatibility of the increment with the parses. We will represent this by a sheaf in which a parse is obtained by gluing together partial parses. Here are the definitions.

Definition 5.3 (*Partial parse*). A partial parse of a string w is a derivation of the form $S \overset{*}{\Rightarrow} \alpha \hookleftarrow w$, where S is the start symbol of the grammar, $\overset{*}{\Rightarrow}$ is the reflexive, transitive closure of the derivation relation, and α is a sentential form. Thus, a partial parse, in addition to deriving w, may derive other elements to its left and to its right.

[15] In a separated presheaf, the correspondence $F(a) \rightarrow \mathrm{Nat}(R, F)$ is an injection, rather than a bijection, as in a sheaf.

[16] The proper phrase is "not obviously local", or not local on the obvious topology. Locality is always defined with respect to a topology. We later define a coarser topology which makes the property of being a monic local.

Example 5.4 (*Parse sheaf*). Partial parses produce a sheaf. The underlying site is the site of labeled strings with inclusion arrows, and strict epimorphic families as covers. The functor Φ assigns to each string the set of its partial parses:

$$\Phi : \mathbf{LString}^{\mathrm{op}} \to \mathbf{Set}$$

$$\begin{array}{l} w \mapsto \{S \overset{*}{\Rightarrow} \alpha \hookleftarrow w\} \\ i\Big\downarrow \quad \Big\uparrow _ \circ i, \quad \text{i.e., } (S \overset{*}{\Rightarrow} \beta \hookleftarrow v) \mapsto (S \overset{*}{\Rightarrow} \beta \hookleftarrow v \hookleftarrow w) \\ v \;\mapsto \{S \overset{*}{\Rightarrow} \beta \hookleftarrow v\} \end{array}$$

We can get an exact parse of a target t using the above sheaf, by enclosing the target in two new symbols distinct from the vocabulary of the grammar, $\dashv t \vdash$, and adding a new start symbol S' with the production $S' \to \dashv S \vdash$.

We can obtain a divide-and-conquer algorithm from the above parse sheaf by choosing a cover for the target, enumerating partial parses of the pieces, and gluing the partial parses. However, there can be an infinite number of partial parses for each piece; hence, this set has to be represented intensionally. The grammar provides the requisite intension: we say that a parse π subsumes a parse ϕ if ϕ can be obtained by applying additional productions to π (see [6] for a more sophisticated scheme which uses equivalence classes of derivations).

Earley's algorithm uses a left-to-right traversal of the target. This restricts partial parses to only those which start with terminals. Now consider a grammar which generates the language $(a+b)^+$:

$$S \to X \mid XS,$$

$$X \to a \mid b.$$

Consider a target string "*aaa*", covered by {"*aa*", "*aa*"}. The first "*aa*" will generate the following partial parses (all other parses being subsumed):

$$S \overset{*}{\Rightarrow} aa, \qquad S \overset{*}{\Rightarrow} aaS.$$

The second "*aa*" also generates the same partial parses. Now, when we try to find compatible families of parses, we do not succeed, and we have to backtrack (i.e., generate subsumed parses). Here are the subsumed parses for the second "*aa*":

$$S \overset{*}{\Rightarrow} aaa, \qquad S \overset{*}{\Rightarrow} aaaS.$$

Next, we enumerate compatible families of partial parses, thus eliminating some parses (e.g., $S \overset{*}{\Rightarrow} aa$). The resultant parses, after gluing the compatible families, are:

$$S \overset{*}{\Rightarrow} aaa, \qquad S \overset{*}{\Rightarrow} aaaS.$$

The key observation here is that, for any increment to the target, we *always* have to backtrack until we produce parses which extend the left context. Thus, we might as well expand the current left context rather than try to parse the new increment: this

corresponds to the "predict" step of Earley's algorithm (the collections of unsubsumed partial parses are the "items"). Here are possible predictions, given parses for the first "*aa*":

$$S \overset{*}{\Rightarrow} aaa, \qquad S \overset{*}{\Rightarrow} aaaS,$$

$$S \overset{*}{\Rightarrow} aab, \qquad S \overset{*}{\Rightarrow} aabS.$$

Some of these predictions have to be rejected because they are not parses of the second "*aa*": this is the "verify" step of Earley's algorithm. This characterization of Earley's algorithm shows its close connection with the KMP algorithm, and also the connection between KMP and LR-parsing.

5.6. *Constraint propagation: Waltz filtering*

Relaxation algorithms for constraint propagation can be described by sheaves when the constraints are local. An example is Waltz filtering [39], an algorithm which assigns three-dimensional interpretations to two-dimensional line drawings of scenes. The underlying site is that of undirected, connected graphs with a different notion of cover.

Example 5.5 (*Graphs, junction covers*). We now define a coarser topology on graphs (cf. Example 2.5) which is suitable for Waltz filtering. A junction in a graph is a subgraph consisting of a node together with all its adjacent nodes (two nodes are adjacent if they are connected by an edge). A junction cover of a graph is the collection of all junctions in that graph; these are the finest covers in the site.

These graphs represent (parts of) line drawings of three-dimensional scenes. The algorithm assigns labels to each edge in the drawing, labels such as shadow edge, concave edge, convex edge, obscuring edge, etc. The possible combinations of labels for commonly occurring junctions, such as L-junctions, T-junctions, forks, etc., are precomputed by using physical properties of three-dimensional space.

The algorithm works by choosing the junction cover for the given line drawing, assigning the precomputed label combinations to each junction, and eliminating inconsistent combinations of labels: when two junctions share an edge, the edge should be assigned a unique label. Thus, the Waltz filtering algorithm can be obtained by applying the KMP derivation to the sheaf of labelings of graphs (this sheaf is similar to the graph-coloring sheaf of Example 2.14).

5.7. *Enumerating functions*

Consider the sheaf of functions of Example 2.15, and consider the problem of enumerating all functions of the form $f: D \to R$. Such functions can be built from pieces by decomposing the domain and using the sheaf condition. If we choose the finest

cover for D, the base case for the problem reduction strategy of Section 4.1 is to enumerate all functions from a singleton set $\{x\} \subseteq D$ to the range R. By the definition of function, there is one such function for each element of R. This algorithm has been encoded as a "global-search" theory by Smith [33], and has been used to derive several algorithms in the KIDS program synthesis/transformation system [34], for example, the n-queens problem [normally, the set $\mathrm{hom}(D, R)$ is filtered after enumeration to satisfy the conditions imposed by a specific problem]. We give below a different derivation for the n-queens problem.

5.8. The n-queens problem

The n-queens problem is to place n queens on an $n \times n$ chessboard such that no two queens are in conflict, i.e., no two queens share the same row, column, or diagonal. This problem is typically solved by a backtracking algorithm [40, 11, 34]. We show how to use a divide-and-conquer strategy. The obvious scheme of decomposing the chessboard does not work. The correct entity to decompose is the conjunction defining the problem: this conjunction can be realized by achieving each conjunct separately. Here is the appropriate sheaf.

Example 5.6 (*Sheaf for n-queens*). The underlying site is the poset of subgraphs of a complete (i.e., fully connected) graph with n nodes. This graph is the constraint graph, representing the potential interaction between the queens. A configuration of queens corresponding to a graph G is a function assigning a position on an $n \times n$ chessboard to each node in the graph G. We define a contravariant functor assigning the set of valid configurations of queens to each graph: a valid configuration is one in which queens do not conflict along the edges of the graph.

$$\mathbf{Config}: \mathrm{Sub}(K_n)^{\mathrm{op}} \rightarrow \mathbf{Set}$$

$$\begin{array}{l} G \mapsto \{N(G) \xrightarrow{c} \mathrm{Board} \mid \forall e \in E(G)\ \mathrm{ok}(c, e)\} \\ i \big\downarrow \qquad \big\uparrow {}_{-\circ i} \\ H \mapsto \{N(H) \xrightarrow{d} \mathrm{Board} \mid \forall e \in E(H)\ \mathrm{ok}(d, e)\} \end{array}$$

K_n is the complete graph with n nodes, $\mathrm{Sub}(K_n)$ is the poset of subgraphs of K_n, $N(-)$ and $E(-)$ denote the nodes and edges of a graph, $\mathrm{Board} = \{\langle x, y\rangle \mid 1 \leqslant x \leqslant n,\ 1 \leqslant y \leqslant n\}$, and $\mathrm{ok}(c, e)$ is a predicate stating that the positions in the configuration c of the two queens connected by the edge e are not in conflict. The sheaf condition is satisfied because pieces of a cover always intersect in at least one node, and the queen represented by that node will rule out conflicts between subconfigurations.

We can apply the KMP derivation to this sheaf. We start with a problem reduction strategy, choosing the finest cover for the graph K_n. The base case is to enumerate all

pairs of queens which do not conflict. We can switch to an incremental algorithm which generates queen-pairs incrementally and maintains partial configurations. We can add a subsumption relation: a configuration c_1 subsumes a configuration c_2 if $c_2 \subseteq c_1$. Now, if an increment cannot extend a configuration, we have to test it against subsumed configurations: this is the backtracking step.

6. Concluding remarks

We have given a sheaf-theoretic characterization of pattern matching. We defined an occurrence to be an arrow in a category. This category is equipped with a Grothendieck topology to allow the decomposition of patterns and targets. The extension of the occurrence relation can be described by a sheaf on this topology. We derived a generalized version of the Knuth–Morris–Pratt pattern-matching algorithm from such a sheaf. The derivation uses limits and colimits to decompose various parts of the problem:

(1) The pattern is decomposed via a cover in the topology. Correspondingly, partial occurrences are glued together via a limit.

(2) The target is decomposed via another cover. The sheaves generated by the pieces of the target are combined via a colimit.

(3) When increments to the target are given sequentially, the collection of partial occurrences is updated via sheafification (the process of completing a functor into a sheaf). This can be viewed as computing a colimit in time.

We explained the failure function of KMP in terms of a subsumption relation. Using subsumption, the cache of partial occurrences can be minimally represented by deleting all subsumed occurrences. This representation makes the algorithm efficient, by reducing the amount of work to be done. Updating such a cache sometimes necessitates the regeneration of subsumed occurrences; this regeneration corresponds to backtracking in the space of partial occurrences.

In view of the minimal assumptions we have made regarding the underlying data structures and the occurrence relation, our derivation works for sheaves other than those generated by an occurrence relation. Hence, we have a general derivation for the problem of

enumerating a collection of locally defined morphisms.

Besides extensions to KMP, such as multiple patterns, patterns with variables, and commutative matching, several other algorithms fit this characterization: Earley's algorithm for context-free parsing, Waltz filtering for scene analysis, enumerating functions, and the n-queens problem. Common to these algorithms is the satisfaction of *locally* defined constraints. Moreover, these algorithms are a combination of some form of decomposition applied to the domain (geometry) and some form of search applied to the codomain (algebra). Such a combination of geometry and algebra is indeed the purpose of sheaf theory.

6.1. In defence of abstract nonsense

In this paper, we have attempted to separate the "abstract nonsense" of pattern matching from the specific features of a data structure or an occurrence relation. Such a separation is inevitable in any field into which category theory is introduced. We quote Peter Hilton [18]:

> Category theory provides a language for discussing very significant mathematical ideas, it provides a unifying medium, and it isolates the "general abstract nonsense" – in Saunders Mac Lane's vivid phrase – from the hard "concrete" mathematics.

The abstract nonsense of pattern matching is the notion that occurrences are built out of pieces, with elementary occurrences either being trivially defined or dependent on the particular data structure. Such a characterization, together with general implementation strategies such as divide-and-conquer, finite differencing, and search (these could be called the abstract nonsense of algorithm design), yields the structure for a general class of algorithms, with a surprising variety of examples. This leads to the speculation that category theory, together with a little geometry, provides a good foundation for studying the abstract structure of algorithms.

6.2. Why Grothendieck topologies?

The arrows in the covers of all the sites defined in this paper are monics. Hence, it may be argued that the machinery of Grothendieck topologies is too general for describing the problem of pattern matching. So, we give below an example of a site in which covers contain nonmonic arrows.

Example 6.1 (*Expressions with sharing*). Consider the site of expression trees defined in Example 5.1. We can modify this site so that expressions are represented by directed, acyclic graphs, thus allowing sharing of some sub-expressions. The morphisms are accordingly modified to correspond to morphisms of graphs (see Example 2.5). We define a cover to be an epimorphic family of arrows in which the domain of every arrow is a *tree*.

Such nonmonic covers also arise when we consider matching modulo a set of equations (e.g., commutativity and identity).

Quite apart from allowing nonmonic covers, Grothendieck topologies have the advantage of defining the topology directly in terms of covers; it is this feature that makes them convenient for our derivation. Most of the data structures used in pattern matching are not closed under union or intersection (e.g., strings, connected graphs, etc.), thus precluding a direct characterization in terms of ordinary topologies.

Appendix. A formal basis for KMP-style algorithms

We present some results from category theory and sheaf theory, mostly from SGA4 [2], which form the basis of the derivation of the pattern-matching algorithm given in the paper. These results arise purely from the definitions of categories, topologies, and sheaves and, thus, are not specific to a particular problem or data structure. Yet, they yield enough information to synthesize an abstract algorithm.

Every presheaf can be minimally "completed" into a sheaf. This process is called "sheafification", and is carried out in two stages, converting a presheaf into a separated presheaf and then into a sheaf. Roughly speaking, for a presheaf $F:\mathscr{C}^{op}\to\mathbf{Set}$, this is done by comparing the set $F(x)$ and the set of compatible families over covers of x: extra elements in $F(x)$ are deleted, and distinct elements corresponding to the same compatible family are identified; in all, the correspondence between compatible families and members of $F(x)$ is forced to be bijective.

Lemma A.1 (Ordering on covers, Artin et al. [2, Exposé II, Section 1.1.1]). *In any site $\langle\mathscr{C},J\rangle$, for any object x, the collection of covering sieves $J(x)$ forms a cofiltered poset under inclusion of covers.*

Theorem A.2 (Sheafification, Artin et al. [2, Exposé II, Section 3]). *Given a presheaf $F:\mathscr{C}^{op}\to\mathbf{Set}$, the functor $LF:\mathscr{C}^{op}\to\mathbf{Set}$ is defined by setting*

$$LF(x)=\underset{R\in J(x)}{\underrightarrow{\mathrm{Colim}}}\ \mathrm{Nat}(R,F),$$

with the colimit taken over the poset $J(x)$. For an arrow $g:x\to y$, $LF(g)$ is defined using the change-of-base functor $g^*:J(y)\to J(x)$, which transfers covers from y to x. The assignment above of $F\mapsto LF$ can be extended to arrows, yielding a functor

$$L:\mathbf{PreShv}(\mathscr{C})\to\mathbf{Preshv}(\mathscr{C}).$$

For any presheaf F, the presheaf LF is always a separated presheaf. If F itself is a separated presheaf, then LF is a sheaf. These properties of L produce an adjunction

$$\mathbf{Preshv}(\mathscr{C})\underset{i}{\overset{sh}{\rightleftarrows}}\mathbf{Shv}\langle\mathscr{C},J\rangle,$$

where i is the inclusion, and sh is the "associated-sheaf" or "sheafification" functor, defined by $i\circ sh\cong L\circ L$.

For the next result, we need a definition. Let $\varepsilon_{\langle\mathscr{C},J\rangle}$ denote the composition of the following functors:

$$\varepsilon_{\langle\mathscr{C},J\rangle}:\mathscr{C}\xrightarrow{\ y\ }\mathbf{Preshv}(\mathscr{C})\xrightarrow{\ sh\ }\mathbf{Shv}\langle\mathscr{C},J\rangle,$$

where y is the Yoneda embedding and sh is the sheafification functor. The subscript in $\varepsilon_{\langle\mathscr{C},J\rangle}$ will be dropped when it is evident from the context.

Theorem A.3 (Lifting of covers, Artin et al. [2, Exposé II, Section 4.4]). *If* $\{f_i : x_i \to x\}$ *is a covering family for* x *in* $\langle \mathscr{C}, J \rangle$, *then* $\{\varepsilon(f_i) : \varepsilon(x_i) \to \varepsilon(x)\}$ *is an epimorphic family in* **Shv** $\langle \mathscr{C}, J \rangle$.

The result follows from the facts that covers can be lifted into **Preshv**($\mathscr{C}$), and the above adjunction.

We would like to use the above result to decompose the target into pieces $t_j \to t$, build the occurrence sheaves $\hom(-, t_j)$ for these pieces, and combine these sheaves to produce the occurrence sheaf $\hom(-, t)$ for the target. To do this, we need a constructive method to build the codomain of an epimorphic family. This is possible in any topos.

Theorem A.4 (Epimorphic families in a topos). *In any topos, if* $\{g_i : y_i \to y\}$ *is an epimorphic family, then*

$$y \cong \underrightarrow{\text{Colim}}\ \text{image}(g_i),$$

where "image" denotes the image of an arrow, and the colimit is taken over the poset of subobjects of y.

We can apply this result to **Shv** $\langle \mathscr{C}, J \rangle$ because it is a topos. Moreover, if we choose a cover $t_j \to t$ for the target which has only monic arrows, then the two results above combine to give

$$\hom(-, t) \cong \underrightarrow{\text{Colim}}\ \hom(-, t_j).$$

We next exploit the properties of an adjunction to obtain a nice expression for the colimit of sheaves required for decomposing the target. The sheafification functor, being a left adjoint, preserves colimits.

Theorem A.5 (Colimits of sheaves, Artin et al. [2, Exposé II, Section 4.1]). *Let* $F : \mathscr{T} \to$ **Shv** $\langle \mathscr{C}, J \rangle$ *be a diagram of sheaves. Then, the colimit of this diagram is given by*

$$\underset{\mathscr{T}}{\underrightarrow{\text{Colim}}}\ F = sh\left(\underset{\mathscr{T}}{\underrightarrow{\text{Colim}}}\ i \circ F\right),$$

where i *is the inclusion functor from sheaves to presheaves, and* sh *is the sheafification functor.*

Recalling the definition of sheafification, the above expression allows us to compute occurrences using two colimits: one over pieces of the target, and one over the set of covers of the pattern. The latter translates to incremental computation of partial occurrences, when combined with the distributive law below.

Lemma A.6 (Distributing limits, Mac Lane [23, p. 212]). *In* **Set**, *colimits are universal, i.e., the pullback of a colimit cone is a colimit cone. In particular, for coproducts we get*

$$a \underset{x \coprod y}{\times} (b \coprod c) = (a \underset{x}{\times} b) \coprod \left(a \underset{x \coprod y}{\times} c \right).$$

This can be extended to several coproducts (the subscripts for the pullbacks are omitted):

$$(a \coprod b) \times (c \coprod d) = (a \times c) \coprod (a \times d) \coprod (b \times c) \coprod (b \times d).$$

Acknowledgment

This work was done as part of my Ph.D. dissertation at the University of California, Irvine, and was supported in part by the National Science Foundation CER grant CCR-8521398. I thank my advisor, Peter Freeman, for encouraging me to pursue this research; David Rector, for introducing me to sheaf theory and answering many questions about category theory, topology, and algebraic specification; a fellow graduate student, Ira Baxter, for listening to my half-baked ideas, asking probing questions, and pointing out the connection of my work to Waltz filtering; and Doug Smith, for providing constructive comments on a previous version of this paper. The preparation of this paper was supported by the Office of Naval Research grant N00014-91-J-1924.

References

[1] A.V. Aho and M.J. Corasick, Efficient string matching: an aid to bibliographic search, *Comm. ACM* **18** (1975) 333–340.

[2] M. Artin, A. Grothendieck and J.L. Verdier, *Théorie des Topos et Cohomologie Etale des Schémas*, Lecture Notes in Mathematics, Vol. 269 (Springer, Berlin, 1972); SGA4, Séminaire de Géométrie Algébrique du Bois-Marie, 1963–1964.

[3] T. Baker, A technique for extending rapid exact string matching to arrays of more than one dimension, *SIAM J. Comput.* **7** (1978) 533–541.

[4] M. Barr and C. Wells, *Toposes, Triples and Theories* (Springer, New York, 1985).

[5] M. Barr and C. Wells, *Category Theory for Computing Science* (Prentice-Hall, New York, 1990).

[6] D.B. Benson, The basic algebraic structures in categories of derivations, *Inform. and Control* **28** (1975) 1–29.

[7] R. Bird, Two dimensional pattern matching, *Inform. Process. Lett.* **6** (1977) 168–170.

[8] R.S. Bird, J. Gibbons and G. Jones, Formal derivation of a pattern matching algorithm, *Sci. Comput. Programming* **12** (1989) 93–104.

[9] J. Burghardt, A tree pattern matching algorithm with reasonable space requirements, in: *Proc. CAAP '88*, Lecture Notes in Computer Science, Vol. 299 (Springer, Berlin, 1988) 1–15.

[10] M. Demazure, Topologies et faisceaux, in: M. Demazure and A. Grothendieck, eds., *Propriétés Générales des Schémas en Groupes*, Lecture Notes in Mathematics, Vol. 151 (Springer, Berlin, 1970); Exposé IV of SGA 3 (Séminaire de Géométrie Algébrique du Bois-Marie, 1962/64).

[11] E.W. Dijkstra, Notes on structured programming, in: O.J. Dahl, E.W. Dijkstra and C.A.R. Hoare, eds., *Structured Programming* (Academic Press, New York, 1972) 1–81.

[12] E.W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).

[13] J. Earley, An efficient context-free parsing algorithm, *Comm. ACM* **13** (1970) 94–102.
[14] C.L. Forgy, Rete: a fast algorithm for the many pattern/many object pattern match problem, *Artificial Intelligence* **19** (1982) 17–37.
[15] M.P. Fourman, C.J. Mulvey and D.S. Scott, *Applications of Sheaves*, Lecture Notes in Mathematics, Vol. 753 (Springer, Berlin, 1979).
[16] R. Goldblatt, *Topoi: The Categorial Analysis of Logic* (North-Holland, Amsterdam, 1984).
[17] H. Herrlich and G.E. Strecker, *Category Theory: An Introduction* (Allyn & Bacon, Boston, 1973).
[18] P. Hilton, *Lectures on Category Theory* (Colgate University, Hamilton, NY, 1972).
[19] D.S. Hirschberg and L.L. Larmore, New applications of failure functions, *J. Assoc. Comput. Mach.* **34** (1987) 616–625.
[20] C.M. Hoffmann and M.J. O'Donnell, Pattern matching in trees, *J. Assoc. Comput. Mach.* **29** (1982) 68–95.
[21] P.T. Johnstone, *Topos Theory* (Academic Press, London, 1977).
[22] D.E. Knuth, J.H. Morris, Jr. and V.R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* **6** (1977) 323–350.
[23] S. Mac Lane, *Categories for the Working Mathematician* (Springer, New York, 1971).
[24] M. Makkai and G.E. Reyes, *First Order Categorical Logic*, Lecture Notes in Mathematics, Vol. 611 (Springer, Berlin, 1977).
[25] J.M. Morris, Programming by expression refinement: the KMP algorithm, in: W.H.J. Feijen et al., eds., *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra* (Springer, New York, 1990) 327–338.
[26] R. Paige and S. Koenig, Finite differencing of computable expressions, *ACM Trans. Prog. Lang. Systems* **4** (1982) 402–454.
[27] H.A. Partsch and N. Völker, Another case study on reusability of transformational developments: pattern matching according to Knuth, Morris, and Pratt, Tech. Report, KU Nijmegen, 1990.
[28] B.C. Pierce, *Basic Category Theory for Computer Scientists* (MIT Press, Cambridge, MA, 1991).
[29] D. Rydeheard and R.M. Burstall, *Computational Category Theory* (Prentice-Hall, Englewood Cliffs, NJ, 1988).
[30] H. Schubert, *Categories* (Springer, Berlin, 1972).
[31] J.A. Seebach, Jr., L.A. Seebach and L.A. Steen, What is a sheaf?, *Amer. Math. Monthly* **77** (1970) 681–703.
[32] D.R. Smith, Top-down synthesis of divide-and-conquer algorithms, *Artificial Intelligence* **27** (1985) 43–96.
[33] D.R. Smith, The structure and design of global search algorithms, Tech. Report KES.U.87.12, Kestrel Institute, Palo Alto, California, 1988; *Acta Inform.* (to appear).
[34] D.R. Smith, KIDS: a semiautomatic program development system, *IEEE Trans. Software Engrg.* **16** (1990) 1024–1043.
[35] D.R. Smith, Structure and design of problem reduction generators, in: B. Möller, ed., *Constructing Programs from Specifications* (North-Holland, Amsterdam, 1991) 91–124.
[36] Y.V. Srinivas, Pattern matching: a sheaf-theoretic approach, Ph.D. Dissertation, Univ. of California, Irvine, 1991; Tech. Report 91-41, Dept. of ICS.
[37] B.R. Tennison, *Sheaf Theory* (Cambridge University Press, Cambridge, 1975).
[38] J. van der Woude, Playing with patterns, searching for strings, *Sci. Comput. Programming* **12** (1989) 177–190.
[39] D. Waltz, Understanding line drawings of scenes with shadows, in: P.H. Winston, ed., *The Psychology of Computer Vision* (McGraw Hill, New York, 1975) 19–91.
[40] N. Wirth, Program development by stepwise refinement, *Comm. ACM* **14** (1971) 221–227.