Pattern Matching: A Sheaf-Theoretic Approach

Yellamraju V. Srinivas
Department of Information and Computer Science
University of California, Irvine, USA
srinivas@ics.uci.edu

Ph.D. Dissertation
Technical Report 91-41, May 1991

UNIVERSITY OF CALIFORNIA

IRVINE

Pattern Matching: A Sheaf-Theoretic Approach

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Yellamraju Venkata Srinivas

Dissertation Committee:

Professor Peter Freeman, Chair

Professor Thomas A. Standish

Professor David Rector

1991

DEDICATION

To my parents,

for always believing in me,

and

to my friend, C. D. Sharma,

whose memory will always live with me.

Contents

List of Figures

Curriculum Vitae

1961 Born in Karaikudi, TamilNadu, India

1983 B.Tech. in Electronics and Communication Engineering,
Regional Engineering College, Warangal, India

1985 M.Tech. in Computer Science and Engineering,
Indian Institute of Technology, Kanpur, India

1988 M.S. in Information and Computer Science,
University of California, Irvine, USA

1991 Ph.D. in Information and Computer Science,
University of California, Irvine, USA
Dissertation:
Pattern Matching: A Sheaf-Theoretic Approach

Acknowledgements

A dissertation is as much the product of an individual as it is of the environment around that individual. I would like to thank the following people for creating an excellent environment, and for their explicit and implicit influence on this dissertation:

Prof. Peter Freeman, my advisor, for constant encouragement, for arranging financial support, for allowing me to pursue an abstract research topic, for establishing an intellectually and culturally diverse research group, and for his sensitivity to cultural differences;

Prof. Thomas A. Standish, for encouragement and advice about passing the hurdles along the route to a doctoral degree, for handling details in the absence of my advisor, and for highlighting geometric ideas in computer science;

Prof. David Rector, for introducing me to sheaf theory, for teaching me styles of usage of abstract mathematical tools (styles which can only be learnt from an experienced teacher and not from any textbook), and for participating in endless discussions about category theory, algebraic specification, topology, and other unrelated subjects;

Members of the Advanced Software Engineering Project, Guillermo Arango, Ira Baxter, Albert Feroldi, Julio Leite, Chris Pidgeon, and Veikko Seppänen, for providing an exceptional environment for both intellectual ruminations and social diversions, and for making me feel at home in a foreign country;

Ira and Linda Baxter, for being understanding and generous during the difficult time of completing a dissertation; and

Dr. Manavala M. Desu, who enabled me to come to the United States with his promise of financial support.

This work was partially supported by the National Science Foundation CER/ grant CCR-8521398, by the University of California Micro Grant #87-055 in conjunction with the Software Productivity Consortium, and the Department of Information and Computer Science, which provided teaching assistantships and tuition fellowships.

Abstract of the Dissertation

Pattern Matching: A Sheaf-Theoretic Approach

by
Yellamraju Venkata Srinivas
Doctor of Philosophy in Information and Computer Science
University of California, Irvine, 1991
Professor Peter Freeman, Chair

A general theory of pattern matching is presented by adopting an extensional, geometric view of patterns. The extension of the matching relation consists of the occurrences of all possible patterns in a particular target. The geometry of the pattern describes the structure of the pattern and the spatial relationships among parts of the pattern. The extension and the geometry, when combined, produce a structure called a sheaf. Sheaf theory is a well developed branch of mathematics which studies the global consequences of locally defined properties. For pattern matching, an occurrence of a pattern, a global property of the pattern, is obtained by gluing together occurrences of parts of the pattern, which are locally defined properties.

A sheaf-theoretic view of pattern matching provides a uniform treatment of pattern matching on any kind of data structure—strings, trees, graphs, hypergraphs, and so on. Such a parametric description is achieved by using the language of category theory, a highly abstract description of commonly occurring structures and relationships in mathematics.

A generalized version of the Knuth-Morris-Pratt pattern matching algorithm is derived by gradually converting the extensional description of pattern matching as a sheaf into an intensional description. 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 match by gluing together partial matches; (2) Finite differencing: collect and update partial matches incrementally while traversing the target; (3) Backtracking: instead of saving all partial matches, save just one; when this partial match cannot be extended, fail back to another; (4) Partial evaluation: precompute pattern-based (and therefore constant) computations.

The derivation is carried out in a general framework using Grothendieck topologies. By appropriately instantiating the underlying data structures and topologies, the same scheme results in matching algorithms for patterns with variables and with multiple patterns. Slight variations of the derivation result in Earley's algorithm for context-free parsing, and Waltz filtering, a relaxation algorithm for providing 3-D interpretations to 2-D images.

Other applications of a geometric view of patterns are briefly considered: rewrites, parallel algorithms, induction and computability.

Introduction

If your thesis will cause great dissension
Strive a Grothendieck topos to mention
This device categorical
Like the great Delphic oracle
Ensures restful incomprehension
— David Rector (1989)

It is becoming increasingly clear that formal methods are essential for better controlling software development. Formal methods enable us to understand and systematize the technical aspects of the process of software development, to understand and capture the domain-specific knowledge involved in a particular program, and to understand and capture the design and implementation knowledge involved in converting a specification into an implementation. My work within the Advanced Software Engineering group at Irvine is based on a combination of two approaches: (1) the use of a transformational approach to software construction, and (2) the systematic capture and use of domain-specific knowledge to control the transformation process.

The Draco transformation tool, invented by Neighbors [Neighbors 84, Freeman 87], is a domain-based transformation system for systematically converting a formal specification (written in an application-specific language) into an efficient implementation. The tool is semi-automatic and works by bridging the gap between a specification and an implementation via a series of intermediate domains which progressively introduce more detail. Domains thus serve to decompose descriptive knowledge into manageable chunks. Similarly, we need a theory of decomposing transformations (which implement domains in terms of lower-level domains) so that we may record implementation knowledge in a modular fashion and synthesize implementations out of pieces.

The work reported in this dissertation is a solution to a part of the larger problem of transformational construction of software. The sheaf-theoretic approach lays the foundation for a geometric treatment of patterns, rewrites, and transformations. The dissertation also serves as an example of a domain description. The analysis of the pattern matching domain is deep, in the sense that, it not only provides a language for describing concepts related to pattern matching, but also charts the design space of converting a specification of a pattern matching problem into an efficient implementation.

We comment about the connection of this dissertation to reuse, another guiding research theme in our group. It is a common complaint that people do not tend to use reusable components. This dissertation illuminates the reason for this phenomenon. This entire dissertation is a description of a single reusable component: the implementation of pattern matching. The level of generality and abstractness of the description is such that it covers most of pattern matching, and a significant portion of the design space for implementing pattern matching.

This dissertation shows that code is just a miniscule part of the description of a reusable component. A large amount of subsidiary information is necessary to describe a reusable component: the design, the design space, criteria for choosing between alternatives, relations between different parts of the space, etc. Thus, each reusable component is an elaborate theory about a miniscule world; and theory formation is intrinsically hard. Non-realization of this fact is the root cause of the failure of current approaches to reuse in solving the software construction problem.

To motivate reuse in a larger context, we recapitulate Dijkstra's maxim that programming is hard. The philosophy of reuse is to simplify the task of programming by transferring the burden to the domain analyst. Dijkstra's maxim can be extended as follows: programming is hard; reusable programming is *harder.*

Domain analysis, in its pure form, is just theory formation, and therefore very hard. The variety of domains used in computer science is comparable (barely!) to the variety of domains (i.e., axiomatic systems) in mathematics. However, the crucial difference is that the computer has opened up the possibility of handling domains of hitherto unknown (to mathematicians) complexity. We quote from Dijkstra:

> Coping mathematically with the programming problem obviously implies that we regard the programming language we use as some sort of formal system, and each program we consider as some sort of formal object. As long as we ignore problems of scale, there is nothing novel in that approach, for it would be rather similar to what all sorts of logicians do. But programs are big! Admittedly, most formal experiments have been carried out with rather small programs, but sizes are increasing, and how to push the barrier still further is becoming an explicitly stated research topic.
>
> E. W. Dijkstra [Dijkstra 82]

Thus the essential problem in computer science is complexity. Even the most mundane domains in computer science require the most sophisticated techniques and theories of mathematics. As this dissertation demonstrates, the simple problem of pattern matching requires sheaf theory and category theory to formalize it properly. The essential complexity of computer science demands the best complexity control devices. We use category theory in this dissertation to control the complexity of the description in the presence of genericity.

Chapter 1
The Geometry of Pattern Matching

Alice laughed. "There's no use trying," she said:
"one can't *believe in impossible things."*
"I daresay you haven't had much practice," said the Queen.
"When I was you age, I always did it for half-an-hour a day.
Why, sometiems I've believed as many as six impossible things before breakfast."
— Lewis Carroll, *Through the Looking-Glass* (1871)

Pattern matching is an interesting problem with applications in unification, rewriting, image analysis, DNA sequencing, etc. 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. In most of this dissertation, only exact matching (as opposed to approximate matching) and constant patterns are considered.

Pattern matching 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, a constant pattern string can be matched in a target string in linear time, as shown by the Knuth-Morris-Pratt string matching algorithm [Knuth et al. 77]. This algorithm uses some clever tricks to achieve this bound. In this dissertation, we will analyze this algorithm by providing a formal derivation of a generalized version of the algorithm which works for any data structure (but not necessarily in linear time).

1.1 The Knuth-Morris-Pratt pattern matching algorithm

The Knuth-Morris-Pratt algorithm [Knuth et al. 77] (hereafter abbreviated as "KMP") 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.

The naive quadratic algorithm for string matching tests for an occurrence of the pattern at every position in the string. For a pattern string p, a target string t, with $|s|$ denoting the length of the string s, and with $s[i]$ denoting the i^{th} character of the string s, the naive algorithm can be written as

for $i = 1, |t|$ **do**
  if match(p, t, i) **then** output(i)
where
match$(p, t, i) = \forall 1 \leq j \leq |p| \cdot p[j] = t[i + j - 1]$

KMP reduces the complexity of the naive algorithm by avoiding comparisons whose results are already known (from previous comparisons). In particular, when a character does not match 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. Here is 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.

slide	$\longrightarrow$ *a b c a b a*
pattern	*a b c a b a*
matches	✓✓✓✓✓×
target	*a b c a b c a b c*

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-ro-right scan of the target string without backing up.

Less apparent, but equally important in KMP is that an occurrence of the pattern is built by piecing together individual occurrences of each character in the pattern. This feature will acquire prominence when we generalize the algorithm. Moreover, the left-to-right traversal of the target strings is not crucial. The pattern can also be slid towards the left as shown below:

slide	*a b c a b a* $\longleftarrow$
pattern	*a b c a b a*
matches	✓✓✓✓✓×
target	*a b c a b a b c a b c a*

1.2 The anatomy of an occurrence

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 two examples of this phenomenon, using strings and graphs, in figures 1.1 and 1.2: an occurrence arrow $p \rightarrow t$ is obtained by gluing together smaller arrows $p_i \rightarrow t_i$. The notion of building an occurrence arrow by "gluing" together or "sewing" together smaller arrows has a decidedly geometric flavor. The rest of this dissertation 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 [Hoffmann and O'Donnell 82, Burghardt 88], and two-dimensional arrays [Baker 78, Bird 77]. 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.

1.3 Problem definition

The problem attacked in this dissertation is the following:

> *To rigorously derive a generalized version of the Knuth-Morris-Pratt pattern matching algorithm which works for any data structure.*

A solution to this problem entails

1. a general definition of pattern matching suitable for any data structure, and
2. a description of the features of the Knuth-Morris-Pratt algorithm (e.g., sliding) in this general setting.

Although the focus of this dissertation is on the Knuth-Morris-Pratt algorithm, the theory of pattern matching developed here has more general applicability. Not only does the theory cover pattern matching for any data structure, but it also presents a unified picture of such diverse algorithms as Earley's algorithm for context-free parsing [Earley 70], and the Waltz filtering algorithm for image analysis [Waltz 75].

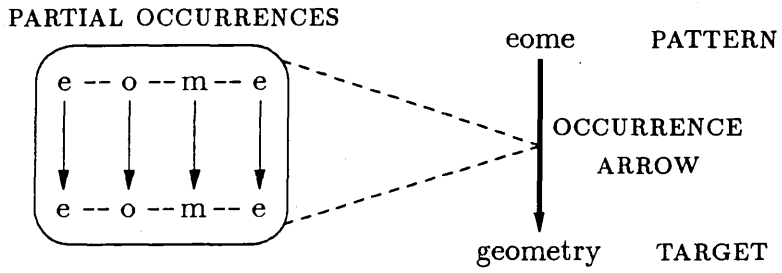


Figure 1.1: Anatomy of an occurrence: example with strings

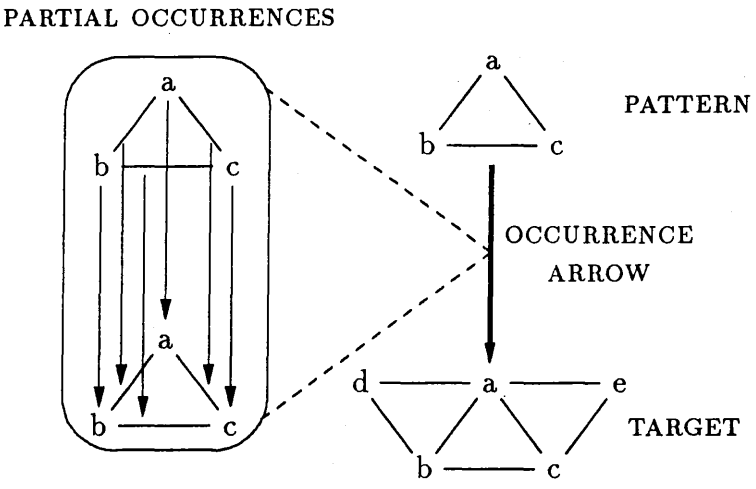


Figure 1.2: Anatomy of an occurrence: example with graphs

1.4 Approach

A general theory of pattern matching is developed by focussing on its geometric aspects. Thus, the approach adopted in this dissertation is extensional, geometric, parametric, and sheaf-theoretic.

Extensional: Rather than consider the occurrences of a specific pattern in a specific target, we consider the extension of the occurrence relationship. The extension captures the essential properties of pattern matching in an unbiased way, and brings the geometry of the problem to the fore.

Geometric: The geometry of pattern matching is emphasized. As shown in section 1.2, the occurrence relation is intimately dependent on the spatial interconnection of the data structures involved. The geometry is axiomatized using Grothendieck topologies. A Grothendieck topology formalizes the notion of "covering" an object by a collection of smaller objects.

Parametric: The specific properties of the data structures involved are abstracted away by formalizing the geometry of the data structures using the language of category theory. Only certain well-defined geometric properties, such as the notion of "covering," are assumed of the data structures involved.

Sheaf-theoretic: Once the pattern matching problem has been described extensionally and parametrically, the real work consists of gluing together partial occurrences. This inductive definition is carried out in the language of sheaf theory. Sheaf theory studies the global consequences of locally defined properties.

1.5 Results

The main results of the work described in this dissertation are

1. A deep theory (domain analysis) of pattern matching applicable to any data structure. This theory provides the appropriate primitive concepts to describe pattern matching and related problems such as rewriting.

2. A rigorous derivation of a generalized version of the Knuth-Morris-Pratt pattern matching algorithm. Exploration of some alternatives along the main derivation path provide explanations for a related family of algorithms, such as Earley's parsing algorithm.

1.6 Outline

Chapter 2 provides some background material on category theory and concepts from sheaf theory which are necessary for generalizing KMP. The reader who is familiar with categories, Grothendieck topologies, and sheaves may skip this chapter. Chapter 3 provides an extensional view of pattern matching by considering the occurrence relation for a fixed target and for all possible patterns. The resulting structure is shown to be a sheaf.

Chapters 4 and 5 form the core of the dissertation. Chapter 4 succinctly represents the material of Chapters 2 and 3 in the language of algebraic specification. The result is a formal specification of the pattern matching problem. Starting from this specification, a rigorous derivation of a generalized version of KMP is presented in Chapter 5. The design space around the main derivation path is also explored to show alternatives and related algorithms.

Chapter 6 shows the generality of the theory by outlining other potential applications of a sheaf-theoretic view of pattern matching: multiple patterns, patterns with variables, parsing, etc. Chapter 7 is devoted to related work: other derivations of KMP, and other applications of sheaf theory in computer science. Chapter 8 provides a summary of the dissertation and outlines contributions of the work.

The development of the ideas in this dissertation is supplemented by three running examples of pattern matching in strings, trees, and graphs (shown in figures 3.1–3.3). Given the highly abstract nature of category theory and sheaf theory, the reader may find it useful to connect all abstract concepts introduced back to these simple examples.

Presentation

The presentation technique chosen in this dissertation is somewhat unorthodox, because the subject material is neither pure mathematics nor pure computer science. Papers in mathematics are intended to be read by other mathematicians and hence focus on high-level ideas, leaving the details to be inferred by the reader. In this dissertation, the intent is not only to present high-level ideas, but also to develop enough details of the KMP derivation so that a good transformation system can semi-automatically follow the steps. The ideal presentation would be to describe the high-level ideas and code up the details in a transformation system. However, current transformation systems are not powerful enough to incorporate some of the higher-order techniques used herein. Rather than submit to the idiosyncrasies of a particular transformation system at the expense of clarity, we have adopted the compromise of giving an algebraic specification of the derivation in parallel with the text: the algebraic specification captures the details of the derivation, the text explains the ideas involved.

Chapter 2
Background: Sites and Sheaves

"When I use a word," Humpty Dumpty said, in rather a scornful tone,
"it means just what I choose it to mean—neither more nor less."
"The question is," said Alice,
"whether you can make words mean so many different things."
"The question is," said Humpty Dumpty, "which is to be master—that's all."
— Lewis Carroll, *Through the Looking-Glass* (1871)

The mathematical concepts required for the derivation of the Knuth-Morris-Pratt pattern matching algorithm are described in this chapter. The vocabulary of category theory is briefly introduced in section 2.1. Sites, which are categories along with topologies, are defined in section 2.4. Sites are useful for capturing the geometric properties of data structures. Several examples are provided to help the reader gain an intuitive understanding of sites and the notion of "covering." Sheaves are defined in section 2.5. The definition is somewhat abstract because of its generality. Some examples and non-examples are given to show how sheaves connect local and global properties, and how the sheaf condition is a formalization of "gluing."

2.1 Category theory

The reader is assumed to have a working knowledge of category theory. The level of category theory required for a thorough understanding of the material in this dissertation precludes a short introduction here. However, to the extent possible, intuitive explanations are included so that the mythical "mathematically mature reader" can follow the arguments. Although the language of category theory is abstract, the instantiation of categorical concepts for pattern matching is simpler and closer to computer science. The derivation of the pattern matching algorithm can be understood at an intuitive level without resorting to category theory. Hence, to help the reader who is less than expert in category theory, we briefly introduce the vocabulary of category theory and show some examples.

Relevant concepts of category theory can be found in mathematics textbooks such as [Mac Lane 71, Herrlich and Strecker 73, Schubert 72] or computer-science-oriented introductions such as [Pierce 88, Rydeheard and Burstall 88, Barr and Wells 90].

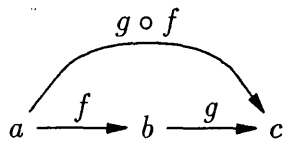
The notation used in this dissertation closely follows [Mac Lane 71]. For reference, figure 2.1 summarizes the notation used.

2.1.1 Definition of a category

DEFINITION 2.1: *Category.* A category is a collection of *objects* (written a, b, c, etc.) and a collection of *arrows* (written f, g, h, etc.). Each arrow connects two objects, called its *domain* and *codomain.* To explicitly indicate the domain and codomain, an arrow f with $\mathrm{dom}(f) = a$ and $\mathrm{cod}(f) = b$ will be written as

$$f: a \to b \quad \text{or} \quad a \xrightarrow{f} b.$$

Two arrows f and g can be composed provided they share a common object, i.e., if $\mathrm{cod}(f) = \mathrm{dom}(g)$, as shown below:



The composition operation "$\circ$" is associative, i.e., $f \circ (g \circ h) = (f \circ g) \circ h$, whenever these compositions are defined. Moreover, each object a is associated with an identity arrow $\mathrm{id}_a: a \to a$, which acts as an identity for the composition operation, i.e.,

$$\text{given } a \xrightarrow{f} b, \quad f \circ \mathrm{id}_a = f = \mathrm{id}_b \circ f.$$

□

Categories will usually be denoted by calligraphic letters, $\mathcal{C}$, $\mathcal{D}$, etc., or by boldface words, **Set**, **Graph**, etc. The collection of objects of a category $\mathcal{C}$ will be denoted by $|\mathcal{C}|$ or $\mathrm{Obj}(\mathcal{C})$; the collection of arrows by $\mathrm{Arr}(\mathcal{C})$. Arrows are sometimes called "morphisms" or "maps."

EXAMPLE 2.2: *The category of sets and functions.* The prototypical example of a category is the category **Set** whose objects are all sets and whose arrows are all functions between sets.[1] Composition is the familiar composition of functions defined as $(g \circ f)(x) = g(f(x))$. Associated with each set A is an identity function id_A, defined as $\mathrm{id}_A(x) = x$. Evidently, the associative and identity laws are satisfied. □

[1]Strictly speaking, the arrows in this category are triples $\langle a, f, b \rangle$ consisting of a function along with its domain and codomain. This is so that all the arrows in the category are distinct, and to differentiate between inclusion functions, $a \hookrightarrow b$, and identity functions, id_b. For readers interested in foundations, **Set** is the category of *small* sets (see also section 4.2).

Notation	Meaning
$\mathcal{C}, \mathcal{D}, \ldots$	names of categories
$a, b, c, \ldots$	names of objects
$f, g, h, \ldots$	names of arrows
$F, G, \ldots$	names of functors
$\mathcal{J}, \mathcal{J}_x, \mathcal{J}_y, \ldots$	names of diagram categories
$\tau, \nu, \ldots$	names of natural transformations
$\mathrm{Obj}(\mathcal{C})$, or $\lvert\mathcal{C}\rvert$	collection of all objects of the category $\mathcal{C}$
$\mathrm{Arr}(\mathcal{C})$	collection of all arrows of the category $\mathcal{C}$
$\rightarrow$	an arrow
id_a	identity arrow on the object a
$\hookrightarrow$	inclusion arrow
$\rightarrowtail$	monomorphism or monic
$\twoheadrightarrow$	epimorphism or epic
$\dot{\rightarrow}$	natural transformation
$\dot{\hookrightarrow}$	inclusion natural transformation (sub-functor)
$\cong$, or $\xrightarrow{\sim}$	isomorphism
$\mathcal{D}^{\mathcal{C}}$	functor category of functors from $\mathcal{C}$ to $\mathcal{D}$
$\mathcal{C}^{\mathrm{op}}$	opposite category
$\mathcal{C}_{\hookrightarrow}, \mathcal{C}_{\rightarrowtail}, \ldots$	subcategory of $\mathcal{C}$ with arrows of kind $\hookrightarrow, \rightarrowtail, \ldots$
$\mathrm{Fun}(\mathcal{C}, \mathcal{D})$	collection of functors from $\mathcal{C}$ to $\mathcal{D}$
$\mathrm{Nat}(F, G)$	collection of natural transformations from F to G
Δc	constant functor
$\nu\colon \Delta c \dot{\rightarrow} F$	cone from c to F
$\underset{\mathcal{J}}{\mathrm{Cones}}(c, F)$	collection of cones from c to F over diagram $\mathcal{J}$
$\underset{\mathcal{J}}{\varprojlim} F$	limit of functor F over diagram $\mathcal{J}$
$\underset{\mathcal{J}}{\varinjlim} F$	colimit of functor F over diagram $\mathcal{J}$
1	terminal object
$a \underset{c}{\times} b$	pullback of $a \xrightarrow{f} c \xleftarrow{g} b$ (arrows f and g implicit)
$a \underset{c}{+} b$	pushout of $a \xleftarrow{f} c \xrightarrow{g} b$ (arrows f and g implicit)
$f\lvert_X$	restriction of the domain of a function $f\colon D \to R$ to $X \subseteq D$
$f\colon x \mapsto y$	alternative notation for $f(x) = y$

Figure 2.1: Table of categorical notation

EXAMPLE 2.3: *Pre-orders as categories.* A binary relation R on a set P (i.e., $R \subseteq P \times P$) is called a *pre-order* if it is reflexive (i.e., for each $p \in P$, we have $p\ R\ p$) and transitive (i.e., whenever $p\ R\ q$ and $q\ R\ r$, we have $p\ R\ r$). Such a pre-order can be considered to be a category as follows. The objects are the elements of the set P. The arrows are the pairs $\langle p, q\rangle$ for which $p\ R\ q$, with $\mathrm{dom}\langle p, q\rangle = p$ and $\mathrm{cod}\langle p, q\rangle = q$. Given a composable pair of arrows

$$p \xrightarrow{\langle p,q\rangle} q \xrightarrow{\langle q,r\rangle} r,$$

we define

$$\langle q, r\rangle \circ \langle p, q\rangle = \langle p, r\rangle.$$

The arrow $\langle p, r\rangle$ exists because the relation R is transitive. Since R is reflexive, the arrow $\langle p, p\rangle$ always exists. We define $\mathrm{id}_p = \langle p, p\rangle$. The associative and identity laws are satisfied by virtue of transitivity. □

Other examples of categories will be given in the section on sites (section 2.4).

2.1.2 Remarks about the categorical approach

The central philosophy of category theory is that arrows are accorded equal status with objects.[2] Not only is the structure of an object relevant, but also its behavior with respect to other objects (defined via arrows) is relevant. An analogous situation in computer science is the importance of operations in a data type: e.g., the same underlying collection of cells can be viewed as an array, stack, queue, or deque depending on the operations provided.

Another characteristic of category theory is that concepts are frequently defined only up to isomorphism. An arrow $a \xrightarrow{f} b$ in a category is an isomorphism if there is an arrow $a \xleftarrow{f^{-1}} b$ such that $f^{-1} \circ f = \mathrm{id}_a$ and $f \circ f^{-1} = \mathrm{id}_b$. Isomorphic objects are considered to be "abstractly the same." This is another way of saying that all the relevant properties of the objects are contained in the arrows. For example, in the category **Set**, isomorphic objects have the same cardinality. The particular elements present in the sets are not relevant; only the cardinality is relevant. Similarly, in the category **Graph** of graphs and graph morphisms (see example 2.19 for a definition), isomorphic graphs have the same "shape"; the particular nodes and vertices used to build the graph are not relevant.

Category theory is a powerful abstraction device because we can control the amount of relevant structure by defining the arrows appropriately. Category theory was invented as an abstract language for describing certain structures and constructions which repeatedly occur in many branches of mathematics, such as topology,

[2]Some authors adopt the more radical approach of treating arrows as primary, and objects as secondary.

Categorical Concept	Instantation	
	Posets	Sets
arrow	sub-structure	function
monomorphism	sub-structure	injection
epimorphism	—[a]	surjection
isomorphism	equality	bijection
initial object	minimum, or bottom	empty set
terminal object	maximum, or top	any singleton set
product	intersection	cartesian product
coproduct	union	disjoint union
pullback	intersection	fibered product
pushout	union	shared union
limit	greatest lowerbound	compatible families
colimit	least upperbound	shared union

[a] All arrows in a poset category are monomorphisms.

Figure 2.2: Categorical concepts instantiated for posets and sets

algebra, and logic. Hence it is a good language for parameterization: a base category with appropriate properties is assumed, and all the constructions of a theory are defined in this category; theorems proved in the general language can then be specialized to any structure satisfying the axioms of the base category.

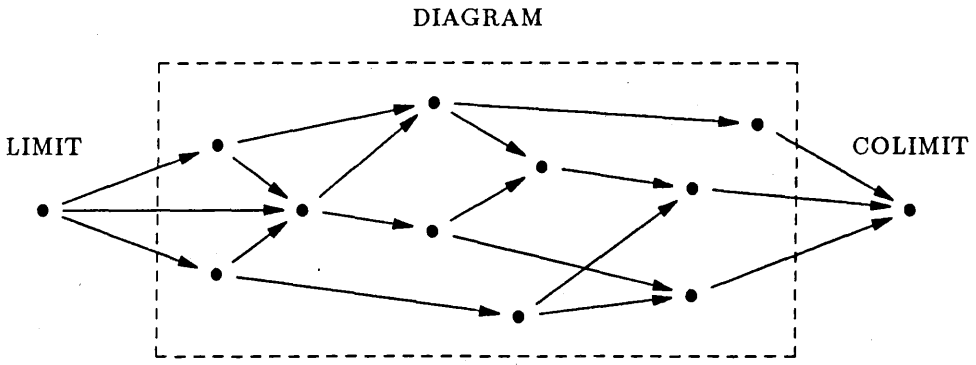
2.1.3 Some constructions in category theory

The axioms for categories are quite weak, in the sense that a large number of mathematical structures satisfy them. Much of the utility of category theory comes from defining more complex constructions using arrows: products, pullbacks, colimits, etc. Properties determined and proved in the general setting then specialize to any category in which the construction is possible.

Many of the categories used in this dissertation are partially ordered collections, e.g., the collection of all strings together with the substring relation, or the collection of all graphs together with the subgraph relation. Figure 2.2 lists some common concepts and constructions from category theory as instantiated for partially ordered collections (posets). Also included are instantiations for the category of sets. Two frequently used constructions, limits and colimits, are described in detail in the next section.

2.1.4 Limits and colimits

A standard method for analyzing or reasoning about a complex entity is by expressing it in terms of simpler, regularly defined entities. For example, a real number can be expressed as the limit of a sequence of rational numbers. In category theory, this notion of limit is generalized by defining it in terms of arrows; and since arrows have a direction, there arise two kinds of limits, called "limit" and "colimit." These concepts can be informally explained as follows. Consider a diagram consisting of some objects and some arrows all pointing in the same direction, say, left to right. Then, a limit and a colimit for this diagram are objects which can be used to "universally complete" the diagram on the left and the right, respectively, as shown in the picture below.

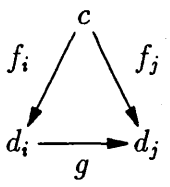


The adjective "universal" means that the limit and the colimit are better than any other object which can be used to complete the diagram. For example, in poset categories, all the arrows are inclusions, and the limit and colimit of a diagram are just the greatest lowerbound and least upperbound. The greatest lowerbound is universal, in the sense that it is greater than any other lowerbound, i.e., it is "*just* to the left of the diagram." Here are formal definitions of these concepts.

DEFINITION 2.4: *Diagram.* A diagram in a category $\mathcal{C}$ is a collection of objects in $\mathcal{C}$ and a collection of arrows between these objects. □

Here is what it means to "complete" a diagram on the left. Once it is recognized that objects, arrows, and composition are the only primitives in category theory, the definition below is somewhat inevitable. A similar definition, with the arrows reversed, applies for completing a diagram on the right.

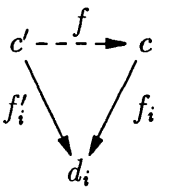
DEFINITION 2.5: *Cone.* Given a diagram D in a category $\mathcal{C}$ and an object c in $\mathcal{C}$, a cone *from the object c to the base D* is a collection of arrows $\{ f_i: c \to d_i \mid d_i \in D \}$, one for each object d_i in the diagram D, such that for any arrow $g: d_i \to d_j$ in D, the following triangle commutes,



i.e., $g \circ f_i = f_j$. □

A limit (or, dually, a colimit) is the best, or universal, way to complete a diagram. Universal definitions such as the one below are the key technique of manufacturing new arrows when using the language of category theory. In the case of pattern matching, this is how an occurrence arrow is built out of pieces.

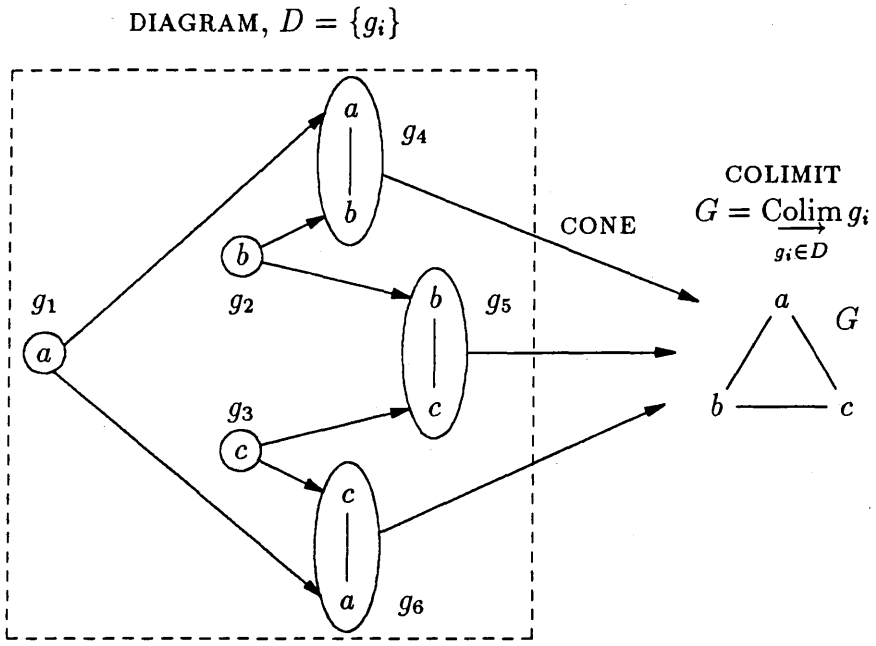
DEFINITION 2.6: *Limit.* A limit for a diagram D in a category $\mathcal{C}$ is an object c in $\mathcal{C}$ along with a cone $\{ f_i: c \to d_i \mid d_i \in D \}$ from c to D such that for any other cone $\{ f'_i: c' \to d_i \mid d_i \in D \}$ from an object c' to D, there is a unique arrow $f: c' \to c$ such that for every object d_i in D, the following diagram commutes



i.e., $f_i \circ f = f'_i$. □

Limits and colimits are useful because they allow us to break up a complex structure into an ascending or descending collection of simpler, more regular structures. We can then reason about these simpler structures and transfer properties from them to the complex structure by using universality. For example, we can decompose a pattern as a colimit of smaller pieces, find occurrences of the smaller pieces, and then compose them to get an occurrence of the pattern.

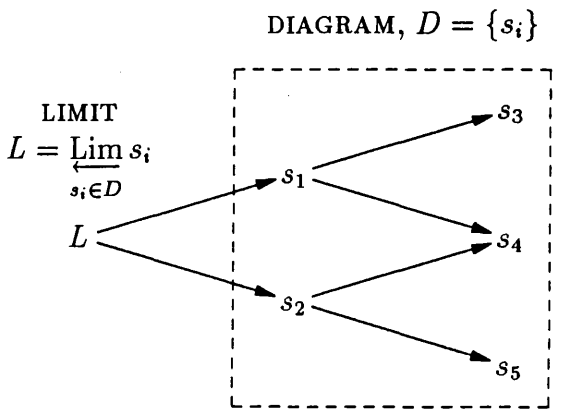
EXAMPLE 2.7: *Colimits in the category of graphs.* Intuitively, it is clear that a graph can be defined by specifying all its nodes and edges. This fact is formally captured by expressing a graph as a colimit of its nodes and edges, as shown in the picture below: the edges are glued together at shared nodes. The category of graphs is described in example 2.20.

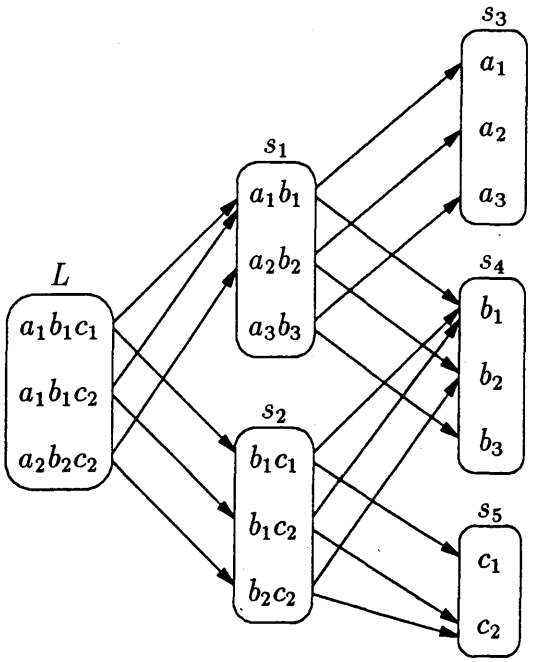


Any graph which contains G can also be used to complete the diagram above. However, G is minimal in the sense that it contains just the information necessary to complete the diagram and no more.

A colimit can be thought of as "shared union," with the diagram specifying the components and the parts which are shared among these components. □

EXAMPLE 2.8: *Limits in the category of sets.* As we saw in section 1.2, an occurrence of a pattern string can be expressed in terms of occurrences of substrings of the pattern. The picture below shows how the occurrences of the string "*abc*" are built as a limit of the occurrences of "*ab*" and "*bc*". The picture consists of two parts: the first shows an overview of the limit; the second shows the details of the individual arrows. The category of sets of is described in example 2.2.

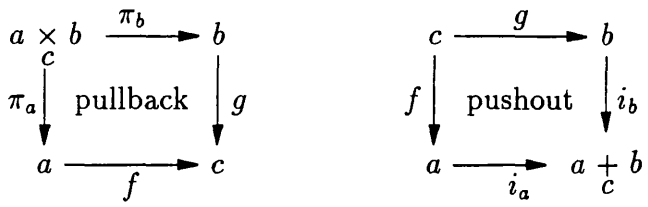




Any subset of L can also be used to complete the diagram above. However, L is maximal in the sense that it contains *all* compatible families of partial occurrences.

A limit can be thought of as enumerating compatible families, or as a generalized intersection, with the diagram specifying how the pieces interconnect. □

When there are only two arrows in the diagram, as shown below, the limit and colimit are called "pullback" and "pushout," respectively. If the arrows are clear from the context, the pullback and pushout are written as $a \underset{c}{\times} b$ and $a \underset{c}{+} b$, respectively.



In **Set**, the pullback of $A \xrightarrow{f} C \xleftarrow{g} B$ is given by

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

with the arrows into A and B being the two projections from the product. It can be seen that the pullback forms the products $f^{-1}(c) \times g^{-1}(c)$ for each element $c \in C$; hence the name "fibered" product ($f^{-1}(c)$ is the "fiber" of f over c).

In **Set**, the pushout of $A \xleftarrow{f} C \xrightarrow{g} B$ is given by

$$A \underset{C}{+} B = (A \amalg B)/\{ \langle f(c), g(c) \rangle \mid c \in C \},$$

where "$\amalg$" denotes disjoint union and "/" denotes the quotient operation. The arrows from A and B to the pushout object are given by the obvious injections.

2.1.5 Some facts about functor categories

Sheaf theory frequently involves manipulations in functor categories, i.e., categories in which the objects are functors and the morphisms are natural transformations. In the case of pattern matching, compatible families of partial matches are succinctly described in functor categories. The results below allow us to unravel such descriptions, look at the graphs of such families, and decompose the graphs into pieces.

A fundamental tool for reasoning in functor categories is the Yoneda lemma, which allows us to "lift" objects and arrows of a category $\mathcal{C}$ into the functor category $\mathbf{Set}^{\mathcal{C}^{\mathrm{op}}}$. Before stating the contravariant form of the Yoneda lemma, we need some definitions.

The collection of all the arrows into an object forms a functor as follows.

DEFINITION 2.9: *Contravariant hom-functor.* For any object a of $\mathcal{C}$, the contravariant hom-functor associated with a, $\hom_{\mathcal{C}}(-,a)\colon \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$, is defined by the following assignments:

1. for any object b in $\mathcal{C}$,
   $\hom_{\mathcal{C}}(-,a)(b) = \hom_{\mathcal{C}}(b,a)$ (the set of arrows from b to a in the category $\mathcal{C}$);
2. for any arrow $f\colon b \to c$ in $\mathcal{C}$,
   $\hom_{\mathcal{C}}(-,a)(f)\colon \hom_{\mathcal{C}}(c,a) \to \hom_{\mathcal{C}}(b,a)$ is defined by $g \mapsto g \circ f$.

□

It is clear that a category can be completely specified by providing the hom-functors associated with all the objects in the category. Thus, instead of objects, we have hom-functors, and instead of arrows, natural transformations between hom-functors. The following definition precisely defines this transformation.

DEFINITION 2.10: *Yoneda functor.* Let $\mathcal{C}$ be a category with small hom-sets. The Yoneda functor $Y\colon \mathcal{C} \to \mathbf{Set}^{\mathcal{C}^{\mathrm{op}}}$ is defined by the following assignments:

1. for any object a of $\mathcal{C}$, $Y(a) = \hom_{\mathcal{C}}(-,a)$;
2. for any arrow $f\colon a \to b$ of $\mathcal{C}$,
   $Y(f)\colon \hom_{\mathcal{C}}(-,b) \to \hom_{\mathcal{C}}(-,a)$ is a natural transformation,
   with the component $Y(f)_c$ at c (an object of $\mathcal{C}$) given by $g \mapsto f \circ g$.

□

The transformation defined by the Yoneda functor is an embedding, i.e., we have a copy of the base category $\mathcal{C}$ in the functor category $\mathbf{Set}^{\mathcal{C}^{\mathrm{op}}}$; the Yoneda functor "lifts" the base category into the functor category.

LEMMA 2.1: *Yoneda embedding.* The Yoneda functor $Y\colon \mathcal{C} \to \mathbf{Set}^{\mathcal{C}^{\mathrm{op}}}$ defined above is full and faithful, i.e., an embedding.

PROOF. The fact follows from the Yoneda lemma (lemma 2.2 below). For a direct proof, see [Barr and Wells 90, page 98]. □

The correspondence between arrows and natural transformations given by the Yoneda functor remains valid even when one of the functors is replaced by an arbitrary set-valued functor (rather than a hom-functor).

LEMMA 2.2: *Yoneda.* If $\mathcal{C}$ is a category with small hom-sets, $F: \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$ is a functor, and c is an object of $\mathcal{C}$, then there is a bijection between natural transformations $\hom_{\mathcal{C}}(-, c) \to F$ and elements of $F(c)$:

$$\mathrm{Nat}(\hom_{\mathcal{C}}(-, c), F) \cong F(c).$$

PROOF. (Sketch) The bijection above arises because each natural transformation $\tau: \hom_{\mathcal{C}}(-, c) \to F$ is uniquely and completely determined by the image under τ_c (the component of τ at c) of the identity arrow on c. Thus the isomorphism of the lemma is given in one direction by

$$\tau \mapsto \tau_c(\mathrm{id}_c),$$

and in the other direction by

$$c \mapsto \sigma^c, \text{ where } \sigma^c: f \mapsto F(f)(c).$$

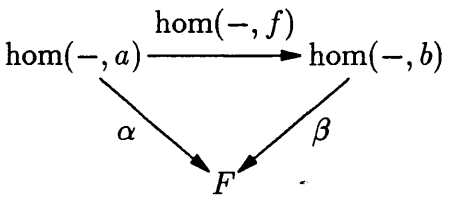
□

Just as a category can be described using hom-functors, so can a set-valued functor on that category be described using hom-functors. The following construction essentially builds pieces of the graph of a functor, and defines that functor to be the colimit of all such pieces.

DEFINITION 2.11: *The category $\mathcal{C}/F$* [SGA4, Exposé I, §3.4.0]. Let $\mathcal{C}$ be a category and $F: \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$ a functor. Then the category $\mathcal{C}/F$ is defined as follows:

1. The objects are pairs $\langle a, \alpha \rangle$ with a an object of $\mathcal{C}$ and $\alpha: \hom_{\mathcal{C}}(-, a) \to F$ a natural transformation.

2. The arrows from $\langle a, \alpha \rangle$ to $\langle b, \beta \rangle$ are arrows $f \in \mathrm{Arr}(\mathcal{C})$ such that the following diagram commutes:



□

LEMMA 2.3: *Decomposition into hom-functors* [SGA4, Exposé I, §3.4]. Every functor in $\mathbf{Set}^{\mathcal{C}^{op}}$ can be expressed as the colimit of a diagram whose vertices are hom-functors.

PROOF. (Sketch) Consider the "forgetful" functor $U: \mathcal{C}/F \to \mathbf{Set}^{\mathcal{C}^{op}}$ which maps each object $\langle a, \alpha \rangle$ into the hom-functor $\hom_{\mathcal{C}}(-, a)$ and each arrow $\hom_{\mathcal{C}}(-, f)$ into itself. Then

$$F = \varinjlim_{\mathcal{C}/F} U.$$

Each object $\langle a, \alpha \rangle$ in $\mathcal{C}/F$ corresponds, by the Yoneda lemma, to an element of $F(a)$. Each arrow $\langle a, \alpha \rangle \to \langle b, \beta \rangle$ in $\mathcal{C}/F$ corresponds to a piece of the graph of the set map $F(f)$. The colimit of all such pieces is just the graph of F, and hence isomorphic to F.

For a full proof which verifies universality, see [Schubert 72, section 10.2]. □

We finally come to the result which is necessary for the derivation of the pattern matching algorithm in Chapter 5:

LEMMA 2.4: *Natural transformations as a limit* [SGA4, Exposé I, §3.5]. If $F, G: \mathcal{C}^{op} \to \mathbf{Set}$ are two functors, then there exists an isomorphism

$$\mathrm{Nat}(F, G) \cong \varprojlim_{\langle a,\alpha \rangle \in \mathcal{C}/F} G(a).$$

PROOF. Using the decomposition of a functor into hom-functors (lemma 2.3 above), we have

$$\mathrm{Nat}(F, G) = \mathrm{Nat}\Big(\varinjlim_{\langle a,\alpha \rangle \in \mathcal{C}/F} \hom_{\mathcal{C}}(-, a), G\Big).$$

Now, $\mathrm{Nat}(F, G)$ is just another name for $\hom_{\mathbf{Set}^{\mathcal{C}^{op}}}(F, G)$. Since (contravariant) hom-functors carry colimits to limits [Mac Lane 71, page 112], we have

$$\mathrm{Nat}\Big(\varinjlim_{\langle a,\alpha \rangle \in \mathcal{C}/F} \hom_{\mathcal{C}}(-, a), G\Big) = \varprojlim_{\langle a,\alpha \rangle \in \mathcal{C}/F} \mathrm{Nat}(\hom_{\mathcal{C}}(-, a), G).$$

By the Yoneda lemma (lemma 2.2 above), the expression $\mathrm{Nat}(\hom_{\mathcal{C}}(-, a), G)$ in the right-hand side is isomorphic to $G(a)$, and hence

$$\mathrm{Nat}(F, G) \cong \varprojlim_{\langle a,\alpha \rangle \in \mathcal{C}/F} G(a).$$

□

2.2 Geometry, topology, and flavors of topology

Geometry is the study of the spatial relationships of structures. Topology is the study of geometric properties via particular axiomatizations.

There are several kinds of topology used in mathematics: the point-set topology used in analysis (also called general topology) and the simplicial and Grothendieck topologies used in algebraic geometry.

The objects of study in general topology are collections of open sets which are closed under arbitrary union and finite intersection. The primary goal is to study continuity, especially properties of spaces which are invariant under continuous deformation. For example, a doughnut can be deformed into a coffee cup; since the number of holes is an invariant, the hole in the doughnut persists as the hole in the handle of the cup. General topology is not very useful for our purposes because many structures of interest in computer science do not satisfy the axioms. For example, if we treat all the substrings of a string as open sets, then this collection is not closed under union. Hence the results and techniques of general topology do not immediately apply to strings. Usually, the general topology of finite structures is uninteresting.

On the other hand, Grothendieck topologies, one of the kinds of topologies used in algebraic geometry, are based on the concept of a cover, which can be defined for any category. A cover captures the spatial interconnection between the parts of a structure. For example, the set of substrings {"*abrac*", "*acada*", "*dabra*"} covers the string "*abracadabra*", because we can glue together the substrings (at the overlapping parts, "*ac*" and "*da*") to obtain the original string. The axioms for a Grothendieck topology are somewhat milder than those of general topology, and thus well suited to the structures which arise in computer science.

2.3 Sheaf theory

Sheaf theory studies the global consequences of locally defined properties. The notion of "local" is characterized using a topology. This topology associates a collection of "covers" with each object in an underlying category. A collection of sets, together with a map from the objects of the 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.

For example, consider a complicated three-dimensional surface, such that on any small enough region it can be defined by a polynomial in a three-dimensional Euclidean coordinate system. Thus, much as in calculus, the whole surface is broken up into small pieces, a coordinate system is erected on each piece, and each piece is described by a polynomial. For such a description to be well-defined, it should be the case that whenever two pieces intersect, the descriptions in the two coordinate systems attached to the two pieces should agree on the intersection. Then, the

surface is uniquely defined everywhere. Using the same technique, any transformation of the surface (e.g., a continuous deformation) can be defined by specifying local transformations which agree on the intersections.

[Tennison 75] provides a gentle introduction to sheaf theory, concentrating on basic definitions and results; the later chapters provide a taste of some applications. [Seebach et al. 70] is a short introduction at the level of undergraduate topology. The authors show three examples to provide an overview of what a sheaf is.

As opposed to classical sheaf theory which uses general topology as its basis, Grothendieck generalized the notion of topology to any category. Grothendieck topologies and the resulting sheaf theory are described in detail in [SGA4, Exposé I–IV]. However, the description is highly abstract, terse, and is characterized by frequent use of functorial descriptions and universes; hence it is not easy reading. [Schubert 72, Chapter 21] is a concise description based on [SGA4]. [Goldblatt 84, Chapter 14] provides basic definitions.

2.4 Sites

In classical sheaf theory, the base of a sheaf consists of a topological space, i.e., a collection of open sets. Grothendieck generalized sheaf theory by removing the restriction that the base be formed of open sets and inclusion arrows between these sets. When generalized to an arbitrary category, the points in a topological space disappear; only the open sets exist as objects in the category. The inclusion relation between open sets is replaced by arbitrary arrows in a category. The topology itself is captured in the notion of a "cover," which is a generalization of open covers in general topology. One crucial difference is that open sets (now objects in a category) need not be closed under union (i.e., coproducts need not exist in the category), nor under intersection (i.e., pullbacks need not exist).[3]

We define below the notion of a Grothendieck topology on an arbitrary category [SGA4, Exposé II]. We then describe several computer science examples illustrating this notion.

DEFINITION 2.12: *Sieve.* A sieve S on an object a is a collection of arrows with codomain a which is closed under composition on the right, 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. □

[3]It is somewhat inappropriate to compare topologies of open sets and Grothendieck topologies at this level; a more meaningful comparison is at the level of closures. Each Grothendieck topology uniquely corresponds to a closure operator (on functors). The axioms satisfied by this operator are similar to the Kuratowski axioms for closure (see, for example, [Johnstone 77]).

DEFINITION 2.13: *Grothendieck topology.* A Grothendieck topology J on a category $\mathcal{C}$ is an assignment to each object a of $\mathcal{C}$, 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 $\mathcal{C}$, then the sieve $f^*(R) = \{ c \xrightarrow{g} b \mid f \circ g \in R \}$ is in $J(b)$;

3. (Stability under refinement)
   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 2.14: *Site.* A site is a category along with a Grothendieck topology. The site formed by a topology J on a category $\mathcal{C}$ will be denoted by $\langle \mathcal{C}, J \rangle$. □

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

Explanation of axioms. Axiom 1 for a topology states that the sieve generated by the identity arrow is a cover. Axiom 2 may be interpreted as in figure 2.3. Roughly speaking, 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 (figure 2.4). Specifically, given a cover of an object, and given a cover for each of the objects in the cover, the composed cover is a (finer) cover of the original object.

DEFINITION 2.15: *Covering family.* A collection of arrows with common codomain $\{ a_i \xrightarrow{f_i} a \mid i \in I \}$ is said to be a covering family for a topology J, if the sieve generated by the family is a covering sieve for the topology J. □

It is clear that every sieve is generated by a collection of "prime" arrows, i.e., arrows which are not factors of any other arrow of the sieve. In view of this, we will usually display a sieve by providing its prime arrows. Topologies can also be defined using covering families: the concept is called a *pretopology.* Such a definition requires the existence of pullbacks in the underlying category. We prefer a definition which does not rely upon pullbacks, because some of the categories in which we are interested do not have pullbacks. Definitions of a pretopology can be found in [SGA4, Exposé II] and [Johnstone 77, Goldblatt 84, Schubert 72].

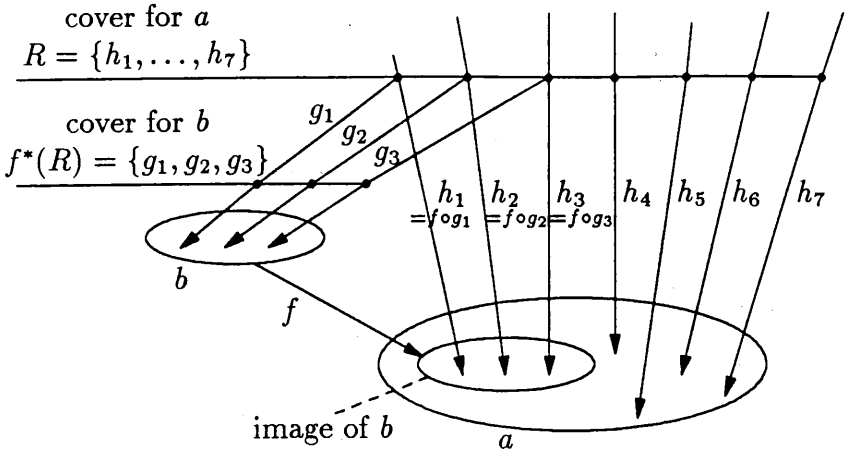


Figure 2.3: Restriction of a cover along an arrow $b \xrightarrow{f} a$

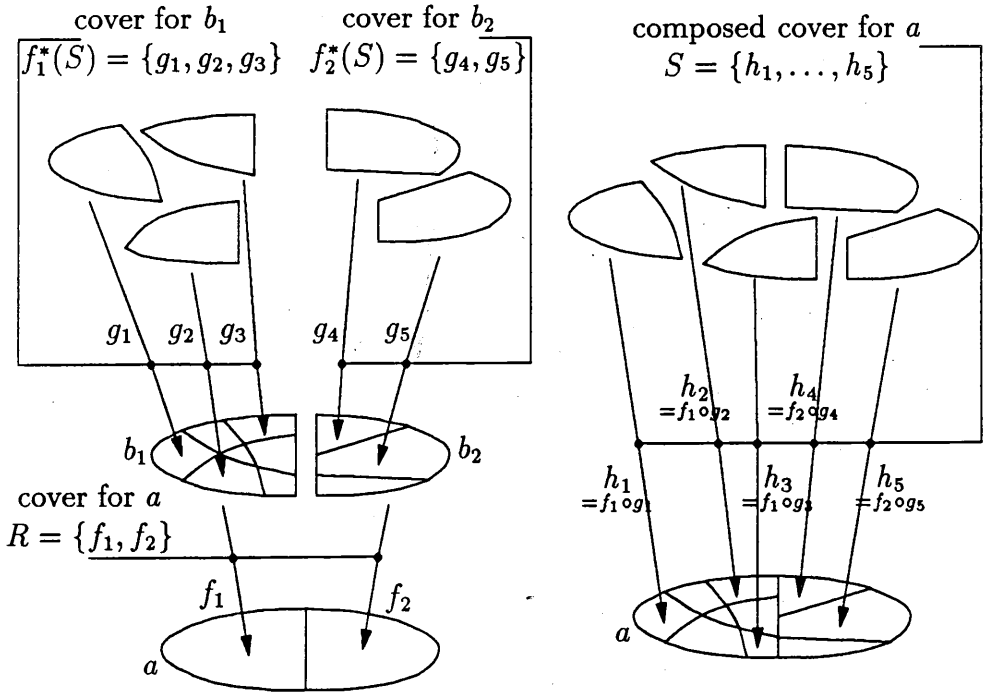


Figure 2.4: Stability of a cover under refinement

2.4.1 Examples of sites

We start with two trivial examples. These form two extremes within which all other topologies will fall: the finest grained topology and the coarsest grained topology.

EXAMPLE 2.16: *Chaotic topology.* The chaotic topology on a category $\mathcal{C}$ is the topology in which every sieve is a covering sieve. □

EXAMPLE 2.17: *Discrete topology.* The discrete topology on a category $\mathcal{C}$ is the topology in which the only covering sieves are those generated by the identity arrows. □

We now give a series of examples of topologies on data structures induced by considering the sub-structure relationship.

EXAMPLE 2.18: *Sets.* Sets and functions form a category **Set** (see example 2.2). A cover of a set S is a family of subsets of S, $\{ S_i \hookrightarrow S \mid i \in I \}$, whose union is S, i.e.,

$$\bigcup_{i \in I} S_i = S.$$

□

EXAMPLE 2.19: *Directed graphs.* A *graph* (unlabeled, directed, multigraph) is a 4-tuple $\langle N, E, s, t \rangle$ of two sets called nodes (N) and edges (E) and two functions called source (s) and target (t) which assign nodes to edges. The relations between these are captured in the following diagram:

$$E \underset{t}{\overset{s}{\rightrightarrows}} N$$

The nodes and edges of a graph G are 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 to nodes and edges to edges, such that the functions are compatible with source and target maps, i.e., for all edges $e \in E(G)$

$$s_H(f_E(e)) = f_N(s_G(e))$$
$$\text{and } t_H(f_E(e)) = f_N(t_G(e)).$$

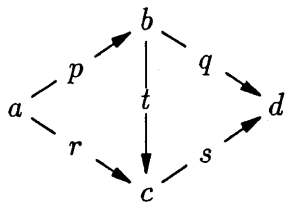
Graphs and graph morphisms form a category, **Graph**. A *subgraph* of a graph H is a graph G such that

$$N(G) \subseteq N(H), E(G) \subseteq E(H)$$
$$\text{and } s_G = s_H|_{N(G)}, t_G = t_H|_{N(G)}.$$

Corresponding to a subgraph G of H, we have an inclusion arrow $G \hookrightarrow H$ in the category **Graph**. A graph is concisely represented by drawing the nodes as points and edges as arrows, with the tail of arrow indicating its source and the head its target. The graph

$$\left\langle \begin{array}{l} \{a,b,c,d\}, \\ \{p,q,r,s,t\}, \\ \{p \mapsto a, q \mapsto b, r \mapsto a, s \mapsto c, t \mapsto b\}, \\ \{p \mapsto b, q \mapsto d, r \mapsto c, s \mapsto d, t \mapsto c\} \end{array} \right\rangle$$

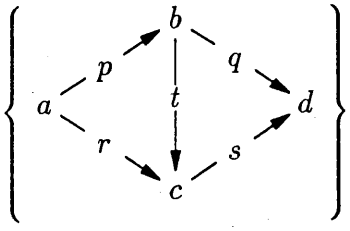
is represented by the following picture:

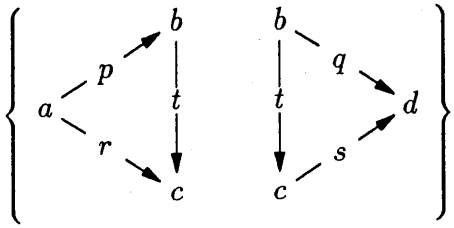


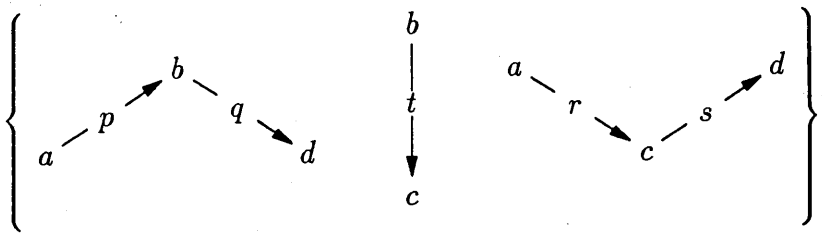
A cover of a graph G is a family of subgraphs $\{ G_i \hookrightarrow G \mid i \in I \}$ such that

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

For example, the following sets are some of the covers of the graph shown above:

1. 

2. 

3. 

□

EXAMPLE 2.20: *Undirected, connected graphs.* An undirected 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 $1 \leq i < k$. A graph is connected if there is a path between any two nodes in the graph.

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.,

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

Undirected, connected graphs and their morphisms form a category **UCGraph**. A subgraph of a graph H is a graph G such that $E(G) \subseteq E(H)$. The fact that $N(G) \subseteq N(H)$ follows because G is connected. A cover of a graph G is a family of subgraphs $\{ G_i \hookrightarrow G \mid i \in I \}$ such that

$$\bigcup_{i \in I} E(G_i) = E(G).$$

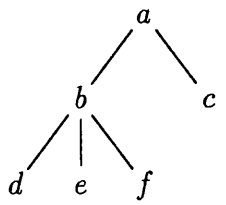
A similar fact about nodes follows from connectedness. These definitions are similar to those for directed multigraphs in the example above. □

EXAMPLE 2.21: *Trees.* A *tree* (rooted, unlabeled) is an undirected, connected, acyclic graph with a distinguished node called the "root." The definitions of morphisms, subtrees, inclusions, and covers carry over from those of graphs, with appropriate substitutions. The definition of *subtree* is different from the conventional one: a subtree is a subgraph which is also a tree. We thus have a subcategory of **UCGraph** called **Tree**.

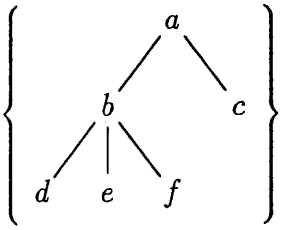
A tree is pictorially represented by drawing its root at the top, the children of the root at the next level, their children at the next level, and so on. The tree

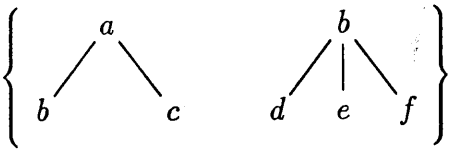
$$\left\langle \begin{array}{l} \{a, b, c, d, e, f\}, \\ \{\langle a, b\rangle, \langle a, c\rangle, \langle b, d\rangle, \langle b, e\rangle, \langle b, f\rangle\} \end{array} \right\rangle$$

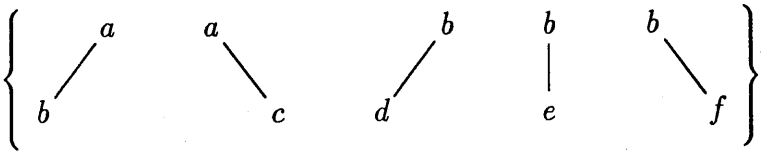
is represented by the following picture:



The following sets are some of the covers of the tree shown above:

1. 

2. 

3. 

□

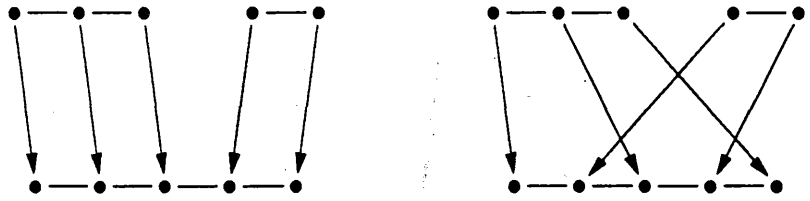
EXAMPLE 2.22: *Strings.* A *string* (unlabeled) is a pair $\langle s, <_s \rangle$ consisting of a set s and a linear order $<_s$ (i.e., a total, irreflexive, and transitive relation). A subset $r \subseteq s$ of a string $\langle s, <_s \rangle$ is said to be *contiguous* if for all elements a, b in 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. Corresponding to a substring s of t, we have an inclusion arrow $s \hookrightarrow t$ in the category **String**. A string is concisely represented by enumerating its elements in order, e.g., the string

$$\left\langle \begin{array}{l} \{a, b, c\}, \\ \{\langle a < b \rangle, \langle a < c \rangle, \langle b < c \rangle\} \end{array} \right\rangle$$

is represented by "abc".

A cover for a string $\langle s, <_s \rangle$ is a collection of substrings 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 2.23: *Strings, alternative morphisms.* Another category can be built using the strings of the previous example but different morphisms. A string morphism is an order-preserving map; there is no contiguity requirement. A substring is a monic (with respect to the new definition of morphism); in detail, a substring of $\langle t, <_t \rangle$ is a string $\langle s, <_s \rangle$ such that $s \subseteq t$ and $<_s$ is the restriction of $<_t$ to s. The definition of covers carries over. The difference between the two kinds of morphisms is illustrated in the picture below.



□

In all the examples above, the arrows were relatively simple and "structurally" defined. Sites can accommodate more complex arrows, and arrows with semantics attached to them. Here is a site in which the arrows are a little more complex.

EXAMPLE 2.24: *Expression trees.* Let Σ be a signature, i.e., a collection of sort names and a collection of operation names defined on these sorts. Here is an example, the signature for stacks:

signature STACK =
sorts ELEM, STACK
operations

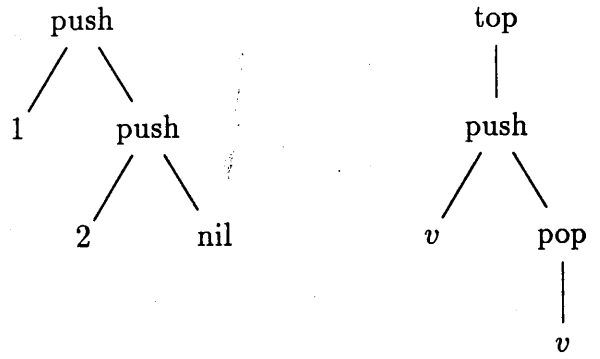
nil :	$\rightarrow$ STACK
push : ELEM, STACK	$\rightarrow$ STACK
pop : STACK	$\rightarrow$ STACK
top : STACK	$\rightarrow$ ELEM

end

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: $\rightarrow s$ is a constant of sort s, then c is an expression of sort s.
3. If f: $s_1, s_2, \ldots, s_n \rightarrow s$ 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. Here are some examples using the signature STACK-SIG (1 and 2 are constants of sort ELEM):



The matching relation between expressions (generated from a signature Σ) is inductively defined as follows:

1. For any expression e, the "variable" expression v matches e.
2. Any constant c matches itself.

3. Given matches of e_i with e'_i, for $i = 1, \ldots, n$, and if $f(e_1, \ldots, e_n)$ is a valid expression, then $f(e_1, \ldots, e_n)$ matches $f(e'_1, \ldots, e'_n)$.

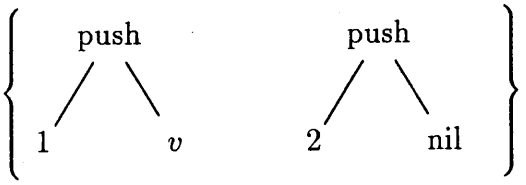
Sub-expressions are defined inductively as follows:

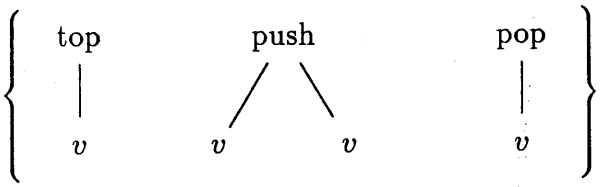
1. Every expression is a sub-expression of itself.
2. The expressions $e_1, \ldots, e_n$ are sub-expressions of $f(e_1, \ldots, e_n)$.
3. If e_1 is a sub-expression of e_2, and e_2 a sub-expression of e_3, then e_1 is a sub-expression of e_3.

A morphism of expressions, $e \rightarrow e'$, is a match of e with some sub-expression of e'. Given a signature Σ, expression and their morphisms form a category called $\mathbf{Expr}(\Sigma)$.

A cover of an expression e is a family of morphisms $\{ e_i \rightarrow e \mid i \in I \}$ 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.

Here is a cover for each of the expressions shown above:





□

Several other examples of sites, in the spirit of the examples above, suggest themselves: hypergraphs, labeled strings, trees, and graphs, sets of strings, trees, and graphs (useful for parallel matching). The variety in these examples shows the generality of Grothendieck topologies and the fact that many data structures in computer science have interesting topologies.

2.5 Sheaves

As mentioned in section 2.3, a sheaf consists of a collection of sets associated with objects of a topology. In addition, consistent with the spirit of category theory, corresponding to each arrow, there is a restriction function on the sets as shown in

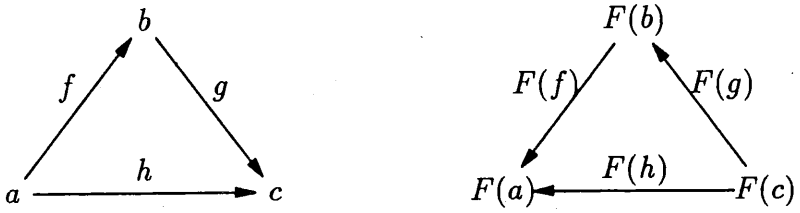
figure 2.5. These assignments determine a contravariant functor on the base category; the functor is called a presheaf (to be a sheaf, it has to satisfy the sheaf condition, described in sections 2.5.1 and 2.5.3).

DEFINITION 2.25: *Presheaf.* A presheaf on a category $\mathcal{C}$ is a contravariant functor from $\mathcal{C}$ to the category of sets **Set**. □

Explicitly, a presheaf $F: \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$ assigns to each object a of $\mathcal{C}$ a set $F(a)$ and to each arrow $f: a \to b$ of $\mathcal{C}$ a "restriction" function $F(f): F(b) \to F(a)$ (note the reversal of direction) such that

(i) $F(\mathrm{id}_a) = \mathrm{id}_{F(a)}$, and

(ii) if $h = g \circ f$ in $\mathcal{C}$ then $F(h) = F(f) \circ F(g)$.



EXAMPLE 2.26. For any set K, the constant presheaf $\overline{K}: \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$ is defined by $\overline{K}(a) = K$ for all objects a in $\mathcal{C}$, and $\overline{K}(f) = \mathrm{id}_K$ for all arrows f in $\mathcal{C}$. □

EXAMPLE 2.27. Any contravariant hom-functor $\mathrm{hom}_{\mathcal{C}}(-, a): \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$ (see definition 2.9), with a an object of $\mathcal{C}$, is a presheaf. □

EXAMPLE 2.28. Let "sieves(a)" denote the collection of all sieves on an object a of a category $\mathcal{C}$. The presheaf $\Omega: \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$ is defined by $\Omega(a) = \mathrm{sieves}(a)$. For any arrow $f: b \to a$ in $\mathcal{C}$, $\Omega(f) = f^*$, where $f^*: \Omega(a) \to \Omega(b)$ maps sieves on a to sieves on b by $s \mapsto \{\, c \xrightarrow{g} b \mid f \circ g \in s \,\}$. The name Ω is derived from the fact that Ω is the truth-values object in the topos $\mathbf{Set}^{\mathcal{C}^{\mathrm{op}}}$. □

2.5.1 The sheaf condition: Simple form

A contravariant functor only provides the basic structure needed for a sheaf. To be a sheaf, such a functor F has to satisfy the additional condition that elements of $F(a)$ can be obtained by "gluing" together elements of $\{\, F(a_i) \mid i \in I \,\}$ where $\{\, a_i \xrightarrow{f_i} a \mid i \in I \,\}$ is a cover of a (see figure 2.6). In other words, such presheaves are "locally" defined, a notion which is formally described in the definition of "sheaf" below.

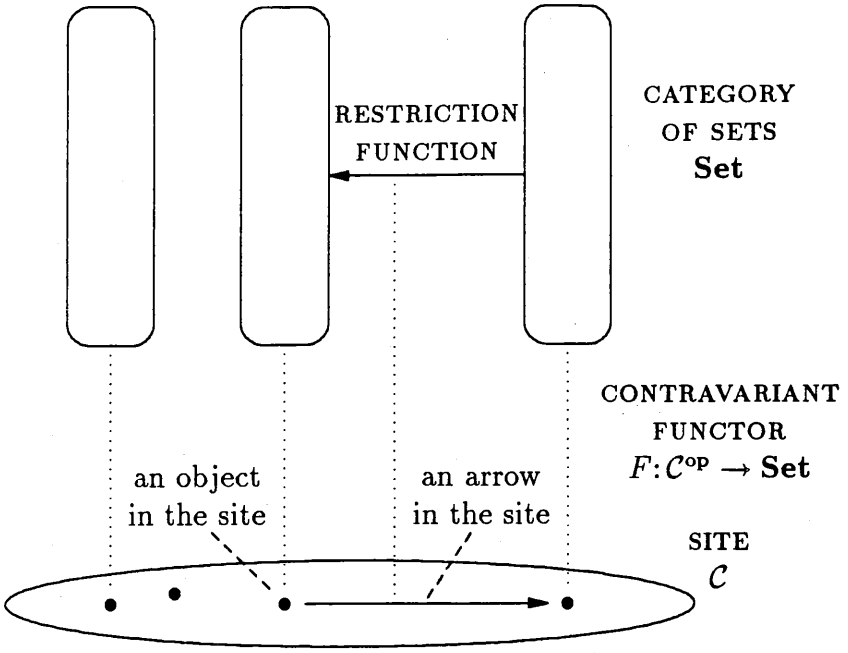


Figure 2.5: Parts of a sheaf

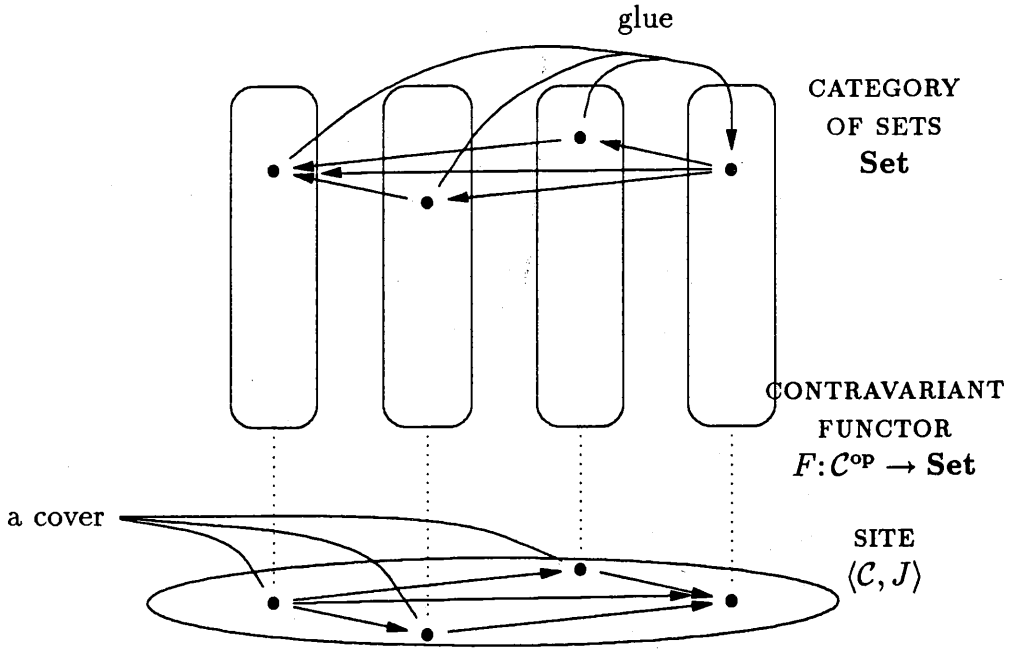
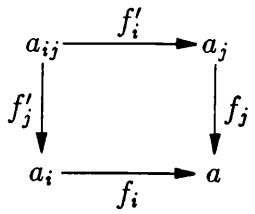


Figure 2.6: The sheaf condition

Before defining a sheaf using covering sieves, we give a simpler definition, using covering families, which is valid if pullbacks exist in the underlying site. We first introduce some notation. Let $F:\mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$ be a presheaf and $\{ a_i \xrightarrow{f_i} a \mid i \in I \}$ be a covering family for a. Form the pullback shown below:



The images of these arrows by the functor F are written as follows: $F_i = F(f_i)$, $F_j^i = F(f_j')$, and $F_i^j = F(f_i')$.

DEFINITION 2.29: *Sheaf* (using pullbacks). A sheaf on a on a site $\langle \mathcal{C}, J \rangle$ is a presheaf $F:\mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$ which satisfies the following "gluing" condition: given any covering family $\{ a_i \xrightarrow{f_i} a \mid i \in I \}$ of an object a in $\mathcal{C}$, and any selection of elements $s_i \in F(a_i)$ for all $i \in I$ which are pairwise compatible, i.e., $F_j^i(s_i) = F_i^j(s_j)$ for all $i, j \in I$, then there is exactly one element $s \in F(a)$ such that $F_i(s) = s_i$ for all $i \in I$. □

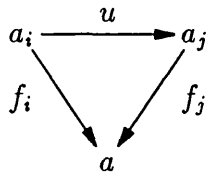
The gluing condition in the definition above can be concisely stated as requiring the following diagram of sets to be an equalizer:

$$F(a) \longrightarrow \prod F(a_i) \rightrightarrows \prod F(a_i \underset{a}{\times} a_j)$$

2.5.2 More about sieves

To define a sheaf using sieves (i.e., without using pullbacks), we need some basic facts about sieves: how they can be treated as comma categories or as sub-functors of a hom-functor. The simple definition of a sieve (definition 2.12) defines a sieve to be a *set* of arrows closed under composition on the right. This definition is more set-theoretic than category-theoretic. To capture the extra structure which is present in a sieve, we can represent it as a comma category.

DEFINITION 2.30: *Sieve: comma category representation.* A sieve $R = \{ a_i \xrightarrow{f_i} a \mid i \in I \}$ on an object a of a category $\mathcal{C}$ can be represented as a sub-category of the comma category $\mathcal{C}\downarrow a$. The objects are arrows $a_i \xrightarrow{f_i} a$ contained in R. The arrows are given by commutative triangles of the following form:



□

The structure of the comma category defined above can be succinctly captured in a functor as shown next.

DEFINITION 2.31: *Sieve: functor representation.* A sieve $R = \{ a_i \xrightarrow{f_i} a \mid i \in I \}$ on an object a of a category $\mathcal{C}$ can be represented as a sub-functor of the hom-functor $\hom_{\mathcal{C}}(-, a)$ as follows:

$$\begin{aligned} b &\mapsto \{ f \in \hom_{\mathcal{C}}(b, a) \mid f \in R \} \\ c \xrightarrow{g} b &\mapsto g^*, \text{ where } g^* : f \mapsto f \circ g \end{aligned}$$

□

By abuse of notation, both a sieve R and its associated representations as a comma category or as a functor (as defined above) will be denoted by the same symbol, the context serving to resolve the ambiguity.

Figure 2.7 shows the different representations of a sieve on an example graph.

In the functor representation of a sieve, we notice that, although the functor is defined over the entire category $\mathcal{C}$, the values of the functor are empty for all those objects for which there no arrows belonging to the sieve with the objects as domains. The subcategory over which the functor is non-empty is called the *base* of the sieve. This notion will be useful in subsequent chapters, and is formally defined below.

DEFINITION 2.32: *Base of a sieve.* Given a sieve R represented as a functor $R: \mathcal{C}^{\mathrm{op}} \rightarrow$ **Set**, the base of the sieve, written base(R), is defined to be the full subcategory of $\mathcal{C}$ induced by the collection of objects for which the values of R are non-empty. □

We are now ready to give a definition of a sheaf which does not require pullbacks to exist in the underlying site.

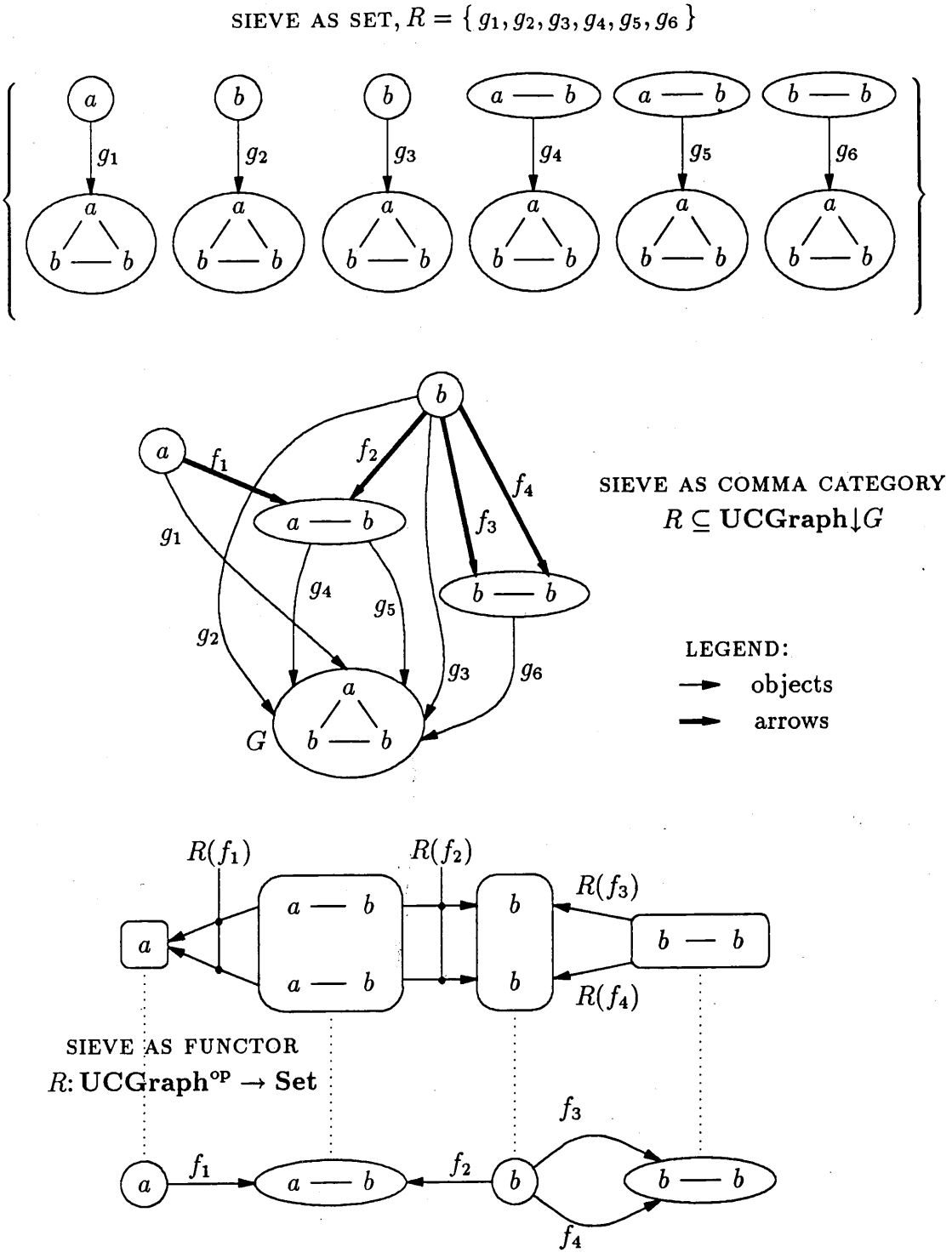


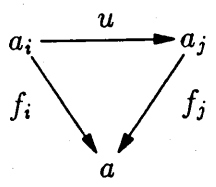
Figure 2.7: Representations of a sieve

2.5.3 Definition of a sheaf

DEFINITION 2.33: *Sheaf* (using sieves). A sheaf on a site $\langle \mathcal{C}, J \rangle$ is a presheaf $F: \mathcal{C}^{\mathrm{op}} \to$ **Set** such that for every object a of $\mathcal{C}$ and every covering sieve $R \in J(a)$, each morphism $R \to F$ in $\mathbf{Set}^{\mathcal{C}^{\mathrm{op}}}$ has exactly one extension to a morphism $\hom_{\mathcal{C}}(-, a) \to F$. □

Formally, the condition above can be described as follows. Let $i_R: R \hookrightarrow \hom_{\mathcal{C}}(-, a)$ be the inclusion of the sieve R, considered as a functor, in the hom-functor on a. This inclusion induces a map of natural transformations, $\mathrm{Nat}(\hom_{\mathcal{C}}(-a,), F) \to \mathrm{Nat}(R, F)$, defined by $\tau \mapsto \tau \circ i_R$. The sheaf condition above requires that this map be a bijection.

Let us now try to understand the definition of a sheaf in elementary terms. Let $R = \{ a_i \xrightarrow{f_i} a \mid i \in I \}$ be a covering sieve of a. A natural transformation $\kappa: R \to F$ (with R being considered as a functor) corresponds to a compatible family of elements $\{ s_i \in F(a_i) \mid i \in I \}$. We see this by unrolling the definition of a natural transformation: $\kappa: R \to F$ maps each arrow $f_i: a_i \to a$ of the sieve R onto to an element $\kappa_{a_i}(f_i) \in F(a_i)$. Further, for any commutative triangle in the sieve (considered as a comma category), we have the following assignments:

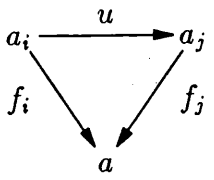


$$f_i \xmapsto{\kappa_{a_i}} s_i \in F(a_i)$$
$$f_j \xmapsto{\kappa_{a_j}} s_j \in F(a_j)$$
$$s_j \xmapsto{F(u)} s_i$$

Thus, each morphism $R \to F$ is a compatible family of elements on R. By the Yoneda lemma, each morphism $\hom_{\mathcal{C}}(-, a) \to F$ corresponds to an element $s \in F(a)$. When translated, the sheaf condition of definition 2.33 says that corresponding to each compatible family of elements $\{ s_i \in F(a_i) \mid i \in I \}$ over the sieve $R = \{ a_i \xrightarrow{f_i} a \mid i \in I \}$, there is a unique element $s \in F(a)$ such that $s_i = F(f_i)(s)$ for all $i \in I$. This can be seen to be equivalent to definition 2.29, when pullbacks exist in the underlying site.

The concept of "compatible family" of elements will arise frequently in the rest of this dissertation. Hence, for reference, we record the definition here.

DEFINITION 2.34: *Compatible family of elements.* Given a presheaf $F: \mathcal{C}^{op} \rightarrow \mathbf{Set}$, an object $a \in \mathrm{Obj}(\mathcal{C})$, and a sieve $R = \{ a_i \xrightarrow{f_i} a \mid i \in I \}$ on 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, such that for any arrow $u: a_i \rightarrow a_j$ in R for which the triangle below commutes, the function $F(u)$ maps s_j onto s_i.



$s_j \xmapsto{F(u)} s_i$

□

2.5.4 Examples of sheaves

The sheaves with which this dissertation is mostly concerned are sheaves of occurrences of a pattern in a target. Chapter 3 provides three simple examples of such sheaves (figures 3.1–3.3). The reader may wish to glance at these sheaves to understand the definition of a sheaf. We provide below a few other examples.

EXAMPLE 2.35: *Books in a library.* Consider a site $\mathcal{T}$ in which objects are time intervals[4] and arrows are inclusions. An interval $[s,t]$ is covered by a collection 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 $B: \mathcal{T}^{op} \rightarrow \mathbf{Set}$ 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 forms 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 $[s,t]$. □

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

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 below, it is possible to connect the chromatic number of a graph (a global property) with colorings of subgraphs (local properties).

EXAMPLE 2.36: *Graph coloring.* Consider the site of undirected, connected graphs described in example 2.20. 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 $C\colon \mathbf{UCGraph}_{\hookrightarrow}^{\mathrm{op}} \to \mathbf{Set}$ as follows:

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

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

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 there is a unique coloring of the entire graph G. □

EXAMPLE 2.37: *Sheaf of functions.* Let D be a set (the domain), and $\mathcal{P}(D)$ its powerset. $\mathcal{P}(D)$ 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 $F: \mathcal{P}(D)^{\mathrm{op}} \to \mathbf{Set}$ 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 $g \mapsto g|_X$ which restricts the domain of a function.

This functor forms a sheaf because, extensionally, a function is defined by specifying its value for each element of the domain. Thus, if $\{ X_i \mid i \in I \}$ covers the set X, and $\{ f_i: X_i \to R \mid i \in I \}$ is a family of functions such that

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

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

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

The sheaf above is typical of the sheaves used in real and complex analysis. Usually, a restricted class of functions, such as continuous functions, analytic functions, or polynomials, is considered. Each set $F(X)$ then has an algebra associated with it, e.g., polynomials form a ring. The sheaf relates the geometry in the base with the algebra in these sets. □

To help the reader understand the mechanics of the sheaf condition, here is an example of a functor which is not a sheaf. The example shows that sometimes local properties alone are not sufficient to determine global properties.

For any site $\langle \mathcal{C}, J \rangle$, the topology J forms a functor as follows (a sub-functor of Ω, example 2.28):

$$\begin{array}{rcll} a & \mapsto & J(a) & \text{for } a \in \mathrm{Obj}(\mathcal{C}) \\ f & \mapsto & f^* & \text{for } f \in \mathrm{Arr}(\mathcal{C}) \end{array}$$

This functor does not form a sheaf because, given a cover $\{ a_i \xrightarrow{f_i} a \mid i \in I \}$ of an object a, and a family of covers $\{ c_i \in J(a_i) \mid i \in I \}$, there may be several covers on a which extend this family. Also, the constant presheaf of example 2.26 is not, in general, a sheaf. Contravariant hom-functors are sheaves only if the covers in the underlying site are strict epimorphic families (see section 3.2).

Chapter 3
Pattern Matching: An Extensional View

Now the ideal aim of design is to remove from the object,
be it an automobile or a bedroom,
every detail, every moulding, every variation of surface,
every extra part except that which induces to its effective function.
— Lewis Mumford, *Drama of the Machines* (1930)

We first adopt a simple view of pattern matching in section 3.1: constant patterns and simple topologies on the structures involved. We will later consider some generalizations in Chapter 6. In section 3.2, we add an assumption to the axioms for sites to eliminate "covers" which do not cover the entire object. In section 3.3, we consider the extension of the occurrence relation and show that it is a sheaf. Section 3.4 is devoted to three simple examples of occurrence sheaves; these will be used as running examples throughout the dissertation. Section 3.5 summarizes the chapter with a specification of pattern matching.

3.1 A simple view of the occurrence relation

Matching is a binary relation. The two entities involved will be called the *pattern* and the *target*. A simple view of matching is that a specified sub-structure of the target has a shape, and possibly other attributes, such as labels, which are the same as those of the pattern. We will call such a relation an *occurrence* relation, preferring the word *match* for more general patterns, e.g., patterns with variables, predicates, etc.

An occurrence is formally represented as an arrow $p \rightarrow t$ in some category. We assume that both the pattern and the target are the same kind of entities and belong to the same category. It is possible to define a matching relation between heterogeneous objects, e.g., a string matching a path in a graph. This added generality is not needed in this dissertation.

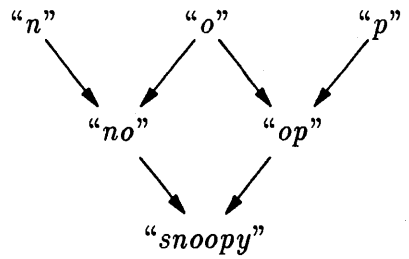
We further assume that every arrow in the category is an occurrence arrow. We thus arrive at the following specification for pattern matching (with constant patterns):

> Given a category $\mathcal{C}$ of structures, a pattern $p \in \mathrm{Obj}(\mathcal{C})$, and a target $t \in \mathrm{Obj}(\mathcal{C})$, the collection of occurrences of the pattern p in the target t is the collection of all arrows $p \rightarrow t$ in $\mathcal{C}$, i.e.,
>
> $$\text{occurrences of } p \text{ in } t = \mathrm{hom}_{\mathcal{C}}(p, t).$$

3.2 Strict epimorphic families

To examine the geometric aspects of patterns and pattern matching, we need a topology on the structures involved. A simple topology can be obtained by considering the sub-structure relation, which is represented by monic arrows. To obtain a site from the base category $\mathcal{C}$, we have to define covers for each object. Looking back at the examples of topologies in section 2.4, we see that the axioms for Grothendieck topologies are not strong enough to rule out certain non-intuitive "covers" such as those which do not cover the entire target.

EXAMPLE 3.1. An example of non-intuitive "covers" is the topology in which every sieve is a covering sieve (see example 2.16). In the category of strings, the following sieve is a cover for this topology, although the image of this sieve is smaller than the target string.



□

To rule out such non-intuitive cases, we use covers which are *strict epimorphic families* [SGA4, Exposé I], [Demazure 70]. An epimorphic family is a formalization of the idea that a cover is surjective,[1] i.e., it covers the *entire* target. The following definition is a straightforward generalization of the notion of epimorphic from a single arrow to a family of arrows.

[1] As usual, with respect the arrows in the category. For a surprising epimorphic family, see example 3.6.

DEFINITION 3.2: *Epimorphic family.* A collection of arrows with common codomain $\{ a_i \xrightarrow{f_i} a \mid i \in I \}$ is said to be an epimorphic family, if for any parallel pair of arrows $g, h: a \rightrightarrows b$, we have

$$\big(\forall i \in I \cdot (g \circ f_i = h \circ f_i)\big) \Rightarrow g = h.$$

□

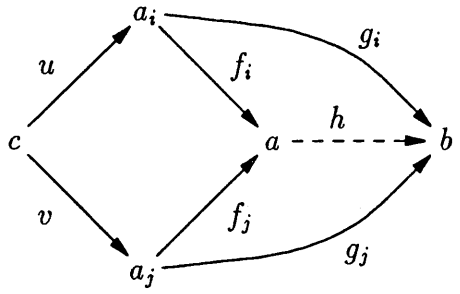
An epimorphic family $\{ a_i \xrightarrow{f_i} a \mid i \in I \}$ is strict if there is an effective way of "gluing" together arrows defined on pieces a_i of the family to obtain an arrow on the codomain a.

DEFINITION 3.3: *Strict epimorphic family.* Let $F = \{ a_i \xrightarrow{f_i} a \mid i \in I \}$ be an epimorphic family. 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 if for every family of arrows G which is compatible with F, there is a unique arrow $h: a \to b$ such that

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



□

DEFINITION 3.4: *Gluing operation.* Given a strict epimorphic family F, and a family G of arrows compatible with F, the assignment of the unique arrow h to G (provided by the definition of a strict epimorphic family above) is called a gluing operation. □

The motivation for this terminology will become clear when, in the next section, we consider sheaves of occurrences of a pattern in a target.

We now add the assumption about strict[2] epimorphic families to a site to make it suitable as a basis for pattern matching:

[2]Normally, one would choose *effective* epimorphic families. However, the definition of this concept requires the existence of pullbacks. When pullbacks do exist, the two notions are equivalent.

ASSUMPTION: All covers in the base topology J of the site $\langle \mathcal{C}, J \rangle$ are strict epimorphic families.

Here are some examples which illustrate the notion of strictness. Roughly speaking, all the information about an object can be obtained from any of its covers.

EXAMPLE 3.5: *Strict vs. non-strict.* Consider the site of example 2.22. The cover $\{$"*ab*", "*cd*"$\}$ of the string "*abcd*" is not strict, since it also covers the string "*cdab*". The cover $\{$"*abc*", "*cd*"$\}$ of the string "*abcd*" is strict, since there is only one way to glue the pieces of the cover to obtain the original string back. A cover which is strict contains all the essential information for constructing the object it covers. We formally define below the site of strings along with covers which are strict epimorphic families.

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 $\{ s_i \hookrightarrow s \mid i \in I \}$ 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 the letters 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 sources and targets of the edges in the path are distinct). The definition of cover for strings is then a specialization of that for graphs. □

EXAMPLE 3.6: *Strict $\not\Rightarrow$ surjective.* Here is an example which shows strictness does not mean that everything in the object should be present in the cover. The only requirement is that all the information about the object be systematically recoverable from the cover.

In the site of example 2.24, the variables contain no essential information. Thus we can prune away variables from the trees (variables can only occur as leaves). To do so, we extend the underlying category $\mathbf{Expr}(\Sigma)$ with certain "malformed" objects satisfying the following rule:

any expression with some variables removed is also an expression.

No essential information is lost because we can always complete an expression tree by examining the signature. In the same spirit, we add new covers:

the collection $\{ e_i \mid i \in I \}$ covers the expression e, if it covers e with some variables removed.

These covers are strict although their images may be smaller than the objects they cover. □

3.3 The extension of the occurrence relation

Using a topology in which covers are strict epimorphic families, we can give a more detailed description of the occurrence function $\hom_{\mathcal{C}}(p,t)$. We get an extensional description by considering the values of this function for all patterns p. Such an extension is not just a set; there is a functorial relationship between the sets $\hom_{\mathcal{C}}(p,t)$ as p varies. If $i: q \to p$ is an arrow (i.e., q occurs in p), then every occurrence $m: p \to t$ of p in t induces an occurrence $m \circ i: q \to t$ of q in t by restriction. Thus the extension of the occurrence function is given by the contravariant hom-functor $\hom_{\mathcal{C}}(-,t): \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$, defined as follows:

For an object $p \in \mathcal{C}$,
$\hom_{\mathcal{C}}(-,t)(p) = \hom_{\mathcal{C}}(p,t) =$ the set of occurrences of p in t.

For an arrow $i: q \mapsto p$ of $\mathcal{C}$,
$\hom_{\mathcal{C}}(-,t)(i): \hom_{\mathcal{C}}(p,t) \to \hom_{\mathcal{C}}(q,t)$ is given by $g \mapsto g \circ i$
(the restriction of occurrences of p to occurrences of q).

There is more structure to the extension than this. The functor is a sheaf on the topology on the underlying site. If $\{ p_i \to p \mid i \in I \}$ is a cover of the pattern p, the sheaf condition states that an occurrence of p can be obtained by gluing together occurrences of the pieces p_i. An occurrence of p_i will be called a partial occurrence.

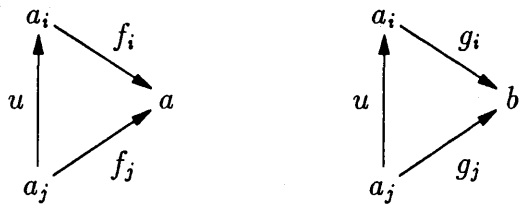
DEFINITION 3.7: *Partial occurrence.* A partial occurrence of a pattern p in a target t is an occurrence of a sub-structure of p in the target t, i.e., a pair of arrows $p \leftarrowtail q \to t$. □

It is evident, from the definition of a strict epimorphic family, that every hom-functor is a sheaf.[3] However, it will be instructive to explicitly go through the details.

[3]The alert reader with some background in category theory will have noticed that, since hom-functors translate colimits into limits [Mac Lane 71, page 112], we could have taken the much simpler approach of assuming that a pattern is a colimit of each of its covers. We do not do so because this is a stronger assumption than the one that covers are strict epimorphic families. Moreover, colimits need not generally exist in the sites we work with.

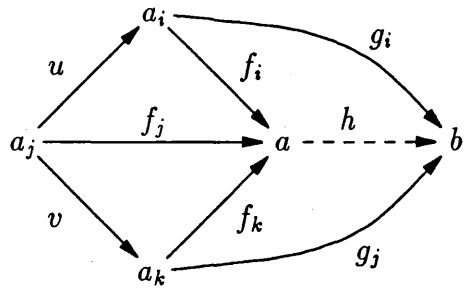
LEMMA 3.1: *Hom-functors as sheaves.* Given a site $\langle \mathcal{C}, J \rangle$ such that every covering sieve in J is a strict epimorphic family, for any object $b \in \mathcal{C}$, the contravariant hom-functor $\hom_{\mathcal{C}}(-, b): \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$ is a sheaf.

PROOF. We will show that $\hom_{\mathcal{C}}(-, b)$ satisfies the sheaf condition. Let a be any object of $\mathcal{C}$, and let $R = \{ a_i \xrightarrow{f_i} a \mid i \in I \}$ be a covering sieve for a. Let $G = \{ a_i \xrightarrow{g_i} b \mid i \in I \}$ be a selection of elements from $\hom_{\mathcal{C}}(a_i, b)$ which is a compatible family of elements on the sieve R. Compatibility means that whenever the diagram on the left commutes for any arrow $u: a_j \to a_i$ in the sieve, then the diagram on the right also commutes, i.e., we have $\hom_{\mathcal{C}}(-, u): g_i \mapsto g_j$.



By juxtaposing two such pairs of diagrams, we have the following diagram in which

$$f_i \circ u = f_k \circ v \Rightarrow g_i \circ u = g_k \circ v.$$



Since we have assumed that all covers are strict epimorphic families, it follows from the definition of a strict epimorphic family that there is a unique arrow $h: a \to b$ (i.e., an element of $\hom_{\mathcal{C}}(a, b)$) such that $h \circ f_i = g_i$ for all $i \in I$, i.e., $\hom_{\mathcal{C}}(-, f_i): h \mapsto g_i$. □

We thus see that, given a pattern p, a target t, and a cover $\{ p_i \to p \mid i \in I \}$ of p, a compatible family of partial occurrences $\{ p_i \to t \mid i \in I \}$ on a cover of the pattern p can be glued together to obtain an occurrence $p \to t$ of p in t.

3.4 Examples of occurrence sheaves

We now give three examples of sheaves of occurrences, figures 3.1–3.3, for trees, graphs, and strings. The underlying sites are as follows.

LTree (labeled trees):
A labeled tree $\langle T, \ell_T \rangle$ is a tree T (see example 2.21), along with a function $\ell_T: N(T) \to L$ assigning labels to nodes, from a fixed label set L. A labeled tree morphism $f: \langle T, \ell_T \rangle \to \langle U, \ell_U \rangle$ is a tree morphism $f: T \to U$ (see example 2.21) which preserves labels, i.e., $\ell_U(f(n)) = \ell_T(n)$, for all nodes $n \in N(T)$. Covers are as defined in example 2.21. All arrows in the site are occurrence arrows.

LUCGraph (labeled, undirected, connected graphs):
A labeled, undirected, connected graph $\langle G, \ell_G \rangle$ is an undirected, connected graph G (see example 2.20), along with a function $\ell_G: N(G) \to L$ assigning labels to nodes, from a fixed label set L. A labeled graph morphism $f: \langle G, \ell_G \rangle \to \langle H, \ell_H \rangle$ is a graph morphism $f: G \to H$ (see example 2.20) which preserves labels, i.e., $\ell_H(f(n)) = \ell_G(n)$, for all nodes $n \in N(G)$. Covers are as defined in example 2.20. All arrows in the site are occurrence arrows.

LString (labeled strings):
A labeled string $\langle s, <_s, \ell_s \rangle$ is a string $\langle s, <_s \rangle$ (see example 2.22), along with a function $\ell_s: s \to L$ assigning labels to elements of the string, from a fixed label set L. A labeled string morphism $f: \langle s, <_s, \ell_s \rangle \to \langle t, <_t, \ell_t \rangle$ is a string morphism $f: \langle s, <_s \rangle \to \langle t, <_t \rangle$ (see example 2.22) which preserves labels, i.e., $\ell_t(f(x)) = \ell_s(x)$, for all elements $x \in s$. Covers are as defined in example 3.5. Occurrence arrows are monics.

3.4.1 Guide to the figures of occurrence sheaves

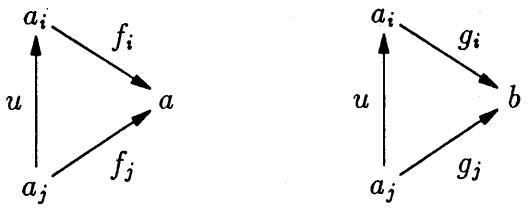
Each of the figures of occurrence sheaves, figures 3.1–3.3, consists of:

1. a specific target t,
2. a specific pattern p, along with a specific cover P, and
3. the portion of the occurrence sheaf, $\hom_{\mathcal{C}}(-, t): \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$, related to the pattern cover, where $\mathcal{C}$ is one of the sites defined above (trees/graphs/strings).

The subscripts and superscripts used in the figures are not part of the labels; they are just used to disambiguate the various structures involved without explicitly drawing the inclusion arrows. Thus, in figure 3.1, a_1 —— b_3 represents an inclusion arrow $(a \text{ —— } b) \hookrightarrow t$ (where t is the target shown) with a being mapped to the particular a in the target with subscript 1 and so on.

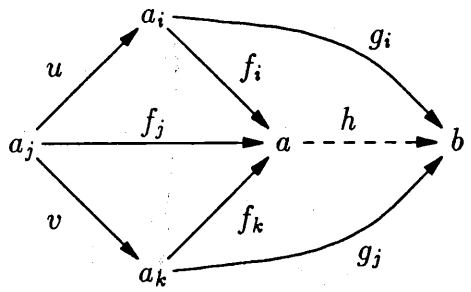
LEMMA 3.1: *Hom-functors as sheaves.* Given a site $\langle \mathcal{C}, J \rangle$ such that every covering sieve in J is a strict epimorphic family, for any object $b \in \mathcal{C}$, the contravariant hom-functor $\hom_{\mathcal{C}}(-, b): \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$ is a sheaf.

PROOF. We will show that $\hom_{\mathcal{C}}(-, b)$ satisfies the sheaf condition. Let a be any object of $\mathcal{C}$, and let $R = \{ a_i \xrightarrow{f_i} a \mid i \in I \}$ be a covering sieve for a. Let $G = \{ a_i \xrightarrow{g_i} b \mid i \in I \}$ be a selection of elements from $\hom_{\mathcal{C}}(a_i, b)$ which is a compatible family of elements on the sieve R. Compatibility means that whenever the diagram on the left commutes for any arrow $u: a_j \to a_i$ in the sieve, then the diagram on the right also commutes, i.e., we have $\hom_{\mathcal{C}}(-, u): g_i \mapsto g_j$.



By juxtaposing two such pairs of diagrams, we have the following diagram in which

$$f_i \circ u = f_k \circ v \Rightarrow g_i \circ u = g_k \circ v.$$



Since we have assumed that all covers are strict epimorphic families, it follows from the definition of a strict epimorphic family that there is a unique arrow $h: a \to b$ (i.e., an element of $\hom_{\mathcal{C}}(a, b)$) such that $h \circ f_i = g_i$ for all $i \in I$, i.e., $\hom_{\mathcal{C}}(-, f_i): h \mapsto g_i$. □

We thus see that, given a pattern p, a target t, and a cover $\{ p_i \to p \mid i \in I \}$ of p, a compatible family of partial occurrences $\{ p_i \to t \mid i \in I \}$ on a cover of the pattern p can be glued together to obtain an occurrence $p \to t$ of p in t.

3.4 Examples of occurrence sheaves

We now give three examples of sheaves of occurrences, figures 3.1–3.3, for trees, graphs, and strings. The underlying sites are as follows.

LTree (labeled trees):
A labeled tree $\langle T, \ell_T \rangle$ is a tree T (see example 2.21), along with a function $\ell_T: N(T) \to L$ assigning labels to nodes, from a fixed label set L. A labeled tree morphism $f: \langle T, \ell_T \rangle \to \langle U, \ell_U \rangle$ is a tree morphism $f: T \to U$ (see example 2.21) which preserves labels, i.e., $\ell_U(f(n)) = \ell_T(n)$, for all nodes $n \in N(T)$. Covers are as defined in example 2.21. All arrows in the site are occurrence arrows.

LUCGraph (labeled, undirected, connected graphs):
A labeled, undirected, connected graph $\langle G, \ell_G \rangle$ is an undirected, connected graph G (see example 2.20), along with a function $\ell_G: N(G) \to L$ assigning labels to nodes, from a fixed label set L. A labeled graph morphism $f: \langle G, \ell_G \rangle \to \langle H, \ell_H \rangle$ is a graph morphism $f: G \to H$ (see example 2.20) which preserves labels, i.e., $\ell_H(f(n)) = \ell_G(n)$, for all nodes $n \in N(G)$. Covers are as defined in example 2.20. All arrows in the site are occurrence arrows.

LString (labeled strings):
A labeled string $\langle s, <_s, \ell_s \rangle$ is a string $\langle s, <_s \rangle$ (see example 2.22), along with a function $\ell_s: s \to L$ assigning labels to elements of the string, from a fixed label set L. A labeled string morphism $f: \langle s, <_s, \ell_s \rangle \to \langle t, <_t, \ell_t \rangle$ is a string morphism $f: \langle s, <_s \rangle \to \langle t, <_t \rangle$ (see example 2.22) which preserves labels, i.e., $\ell_t(f(x)) = \ell_s(x)$, for all elements $x \in s$. Covers are as defined in example 3.5. Occurrence arrows are monics.

3.4.1 Guide to the figures of occurrence sheaves

Each of the figures of occurrence sheaves, figures 3.1–3.3, consists of:

1. a specific target t,
2. a specific pattern p, along with a specific cover P, and
3. the portion of the occurrence sheaf, $\hom_{\mathcal{C}}(-, t): \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$, related to the pattern cover, where $\mathcal{C}$ is one of the sites defined above (trees/graphs/strings).

The subscripts and superscripts used in the figures are not part of the labels; they are just used to disambiguate the various structures involved without explicitly drawing the inclusion arrows. Thus, in figure 3.1, a_1 —— b_3 represents an inclusion arrow $(a \text{ —— } b) \hookrightarrow t$ (where t is the target shown) with a being mapped to the particular a in the target with subscript 1 and so on.

The pattern cover is represented as a functor, but only the image of the functor is shown. For each occurrence sheaf, a sample compatible family of occurrences is indicated with bold arrows. Elements of the sheaf which do not form part of a compatible family are indicated with dashed arrows.

Circles and ellipses denote objects of the category $\mathcal{C}$ of the underlying site. Boxes with rounded corners denote sets, i.e., objects in the category **Set**. Arrows in $\mathcal{C}$ are denoted by arrows (!). Arrows in **Set** (i.e., functions) are exploded: the mapping of each element in the domain is shown by an arrow.

The reader who is unfamiliar with sheaf theory will find it instructive to carefully study figures 3.1–3.3, and

1. verify the definitions and axioms of section 2; and
2. see how occurrences of a pattern can be obtained by gluing together occurrences of pieces of the pattern.

The reader may note that a compatible family of partial occurrences is just an instance of the pattern cover in the graph of the sheaf. This intuitive concept will be formalized using "cones" in Chapter 5; see figures 5.3–5.4.

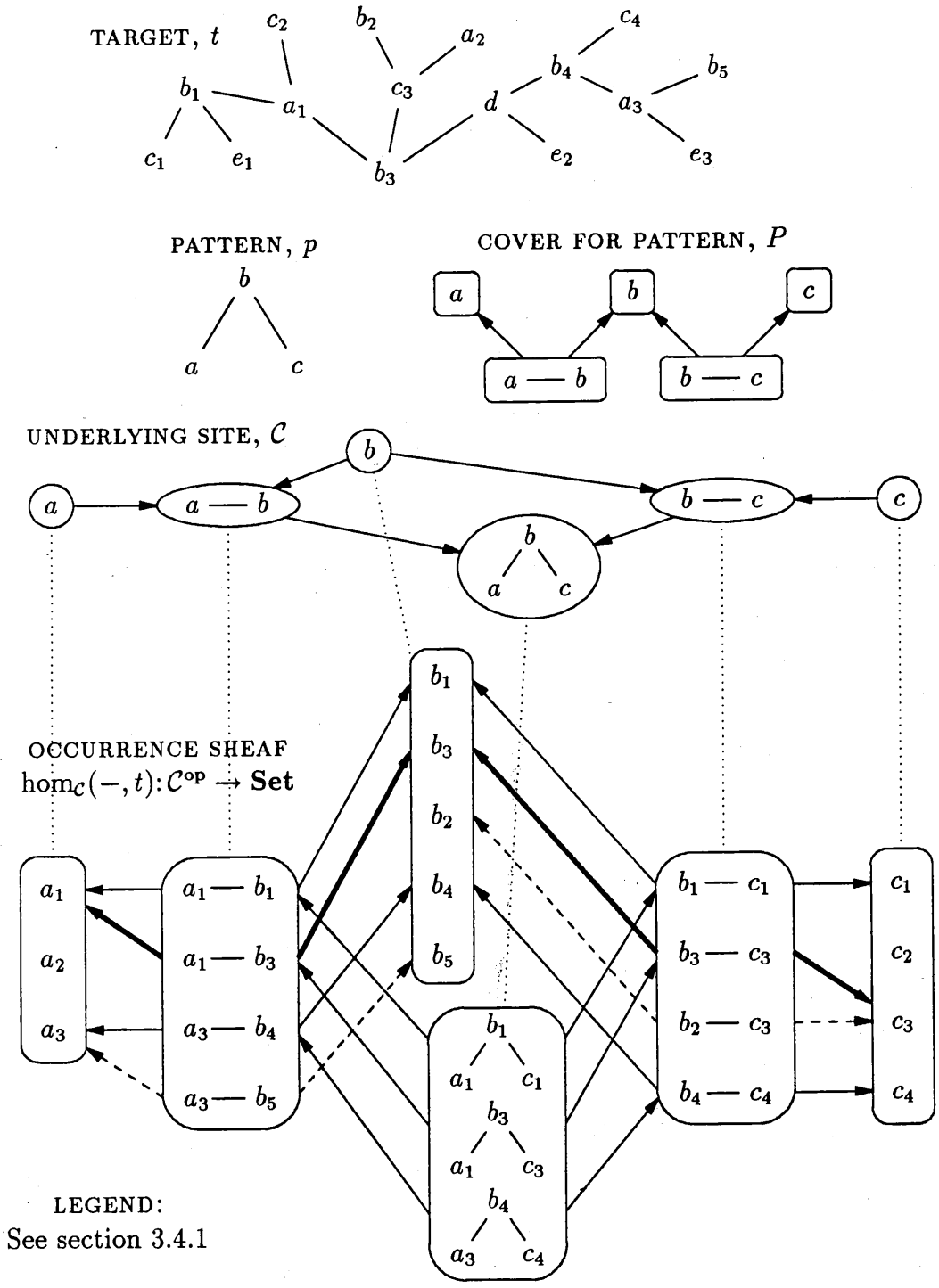


Figure 3.1: Sheaf of occurrences: example with trees

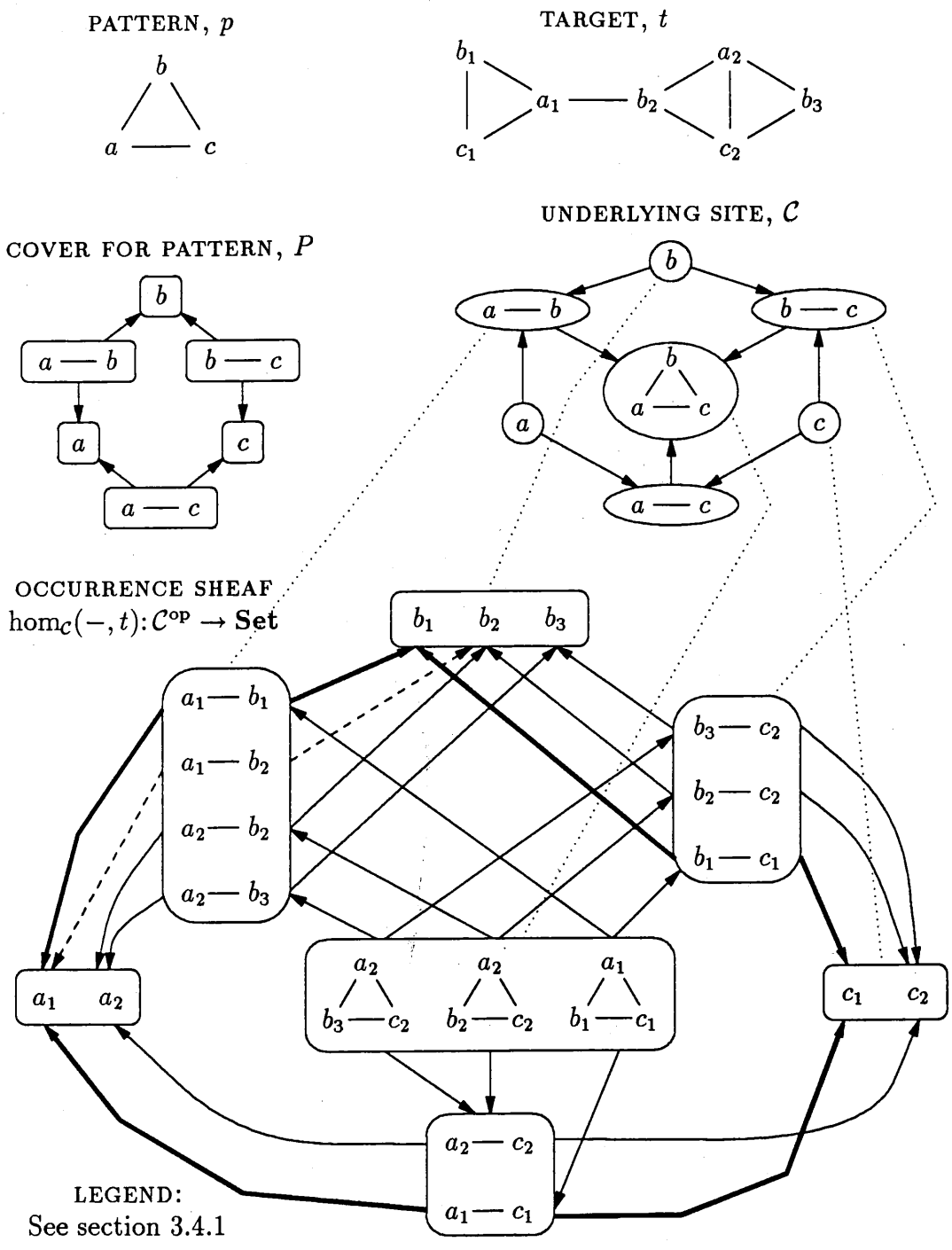


Figure 3.2: Sheaf of occurrences: example with graphs

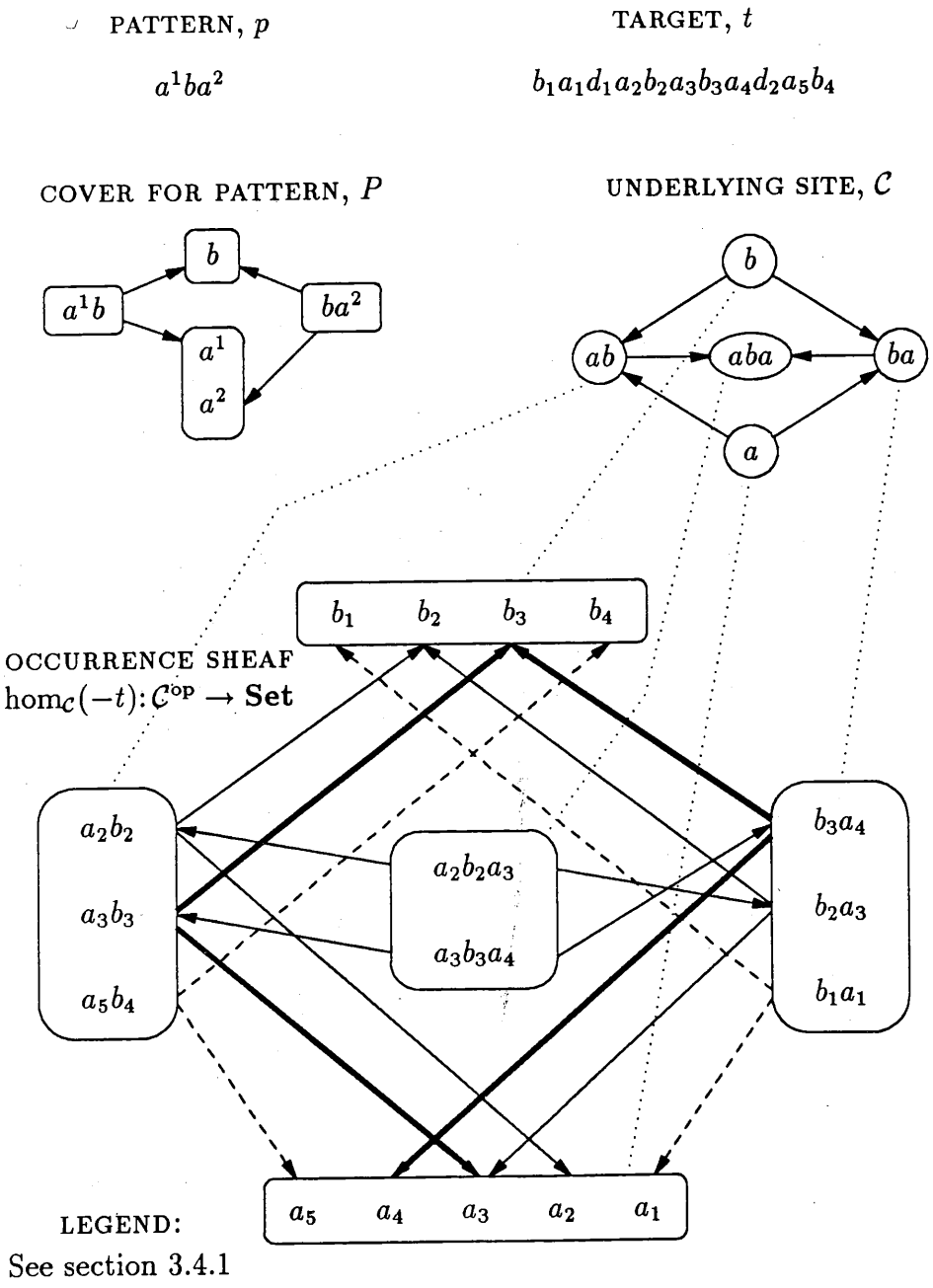


Figure 3.3: Sheaf of occurrences: example with strings

3.5 Summary: A simple specification of pattern matching

We now collect the details and assumptions about pattern matching discussed in this chapter. A more formal version is presented in the next chapter.

A pattern matching problem consists of

The context:
A site $\langle \mathcal{C}, J \rangle$ such that all covers in the topology J are strict epimorphic families.

The occurrence relation:
Some of the arrows, $p \rightarrow t$, represent occurrences of a pattern p in a target t.

Under the additional assumption that all arrows are occurrence arrows, it trivially follows that the extension of the occurrence relation, $\hom_{\mathcal{C}}(-, t)$, for a fixed target, and all possible patterns, forms a sheaf.

Admittedly, the characterization above does not constrain the problem enough, because it is too general. However, even with such weak assumptions, the derivation in Chapter 5 can proceed quite a few steps before needing additional assumptions.

The sheaf characterization of pattern matching shows the power of category theory to concisely represent information about a problem. None of the information in the sheaf is new or surprising; such information is either evident or implicitly known to people familiar with the problem. Categorical language is an excellent vehicle for articulating such implicit knowledge.

Chapter 4
An Algebraic Specification of the Pattern Matching Problem

First-order logic is surely an artifice,
albeit one of the most important inventions in human thought.
But none of us thinks in a first-order language.
The predicates of natural dialectics are order-insensitive
(one moment's individuals are another's equivalence classes)
and our appreciation of mathematics depends on our ability to
interpret *the words of mathematics.*
The interpretation itself is not first-order.
— Peter Freyd, *Aspects of Topoi* (1972)

We need a formal specification method for rigorously deriving a pattern matching algorithm in Chapter 5. We have chosen algebraic specification to encode the specification of pattern matching for several reasons:

1. The theory of algebraic specification is well developed: see, for example, the surveys [Srinivas 90, Wirsing 90], or the textbooks [Ehrig and Mahr 85, Ehrig and Mahr 90].
2. The axiomatic methods of algebraic specification are well suited to the kind of pattern matching theory developed in this dissertation.
3. Algebraic specification provides several structuring operations for making a specification modular: see, for example, [Sannella and Tarlecki 88a].
4. Formal transformations and implementations can be explicitly encoded using the language of algebraic specification: see, for example, [Sannella and Tarlecki 88b].

4.1 Algebraic specification

We do not adopt any particular language for algebraic specification, so as not to constrain the exposition. The syntax and operations used are based on the institution-independent work of Sannella and Tarlecki [Sannella and Tarlecki 88a], and the kernel

language ASL [Astesiano and Wirsing 86, Wirsing 86]. Along with these, we freely use higher-order functions, dependent types, and subsorts, treat algebras as first-class objects, and use constraints such as "small" and "finite" for constraining the possible models of a specification. In general, such a mixture may lead to inconsistencies; but, in a concrete situation such as sheaf theory, there is little danger of the theory collapsing.

The notation for algebraic specification used in this dissertation is summarized in figure 4.1. We frequently use dependent sorts to specify the domains of operations. A dependent sort is a sort which is constructed from another sort via a given function; this is a convenient way of specifying subsorts. The structuring operations we use to build specification are **extend**, conservative extension, **include**, syntactic inclusion, and parameterization. The semantics of all specifications is loose, i.e., there is no restriction on the models, unless otherwise indicated. For implementing one specification by another, we use the constructors defined in [Sannella and Tarlecki 88b]. A specification S is implemented by a specification T via a constructor κ, written $S \rightsquigarrow \kappa(T)$, if κ converts every model of T into a model of S. We expect T to have fewer models than S, since some implementation decisions would have been made in T.

4.2 Foundations

The semantics of an algebraic specification is defined using category theory. In this dissertation, we need a specification for categories themselves. Hence, some care needs to be exercised in using algebraic specification so as to avoid foundational paradoxes.

Another problem is that algebraic specification is an axiomatic specification method. It works most of the time, but sometimes it is difficult to axiomatically define some entity; it is easier to exhibit its construction. For example, it is difficult to axiomatically specify the category of graphs, whereas it very easy to construct the category of graphs. The category of graphs is the model category of the following specification.

spec GRAPH =
sorts NODES, EDGES
operations
  source : EDGES → NODES
  target : EDGES → NODES
end

We will assume an ambient category theory, e.g., a first-order theory of "meta-categories" as defined in [Mac Lane 71]. We will also assume an ambient set theory with three levels of universes: small, large, and very large.[1] Here are some specific entities in the ambient category theory:

[1] It does not matter whether the theories of categories and sets are separate or one is defined within the other.

spec SPEC-NAME =
sorts DOMAIN, RANGE, SPECIAL
—*the subsort relationship; modeled as a subset*
subsorts SPECIAL $\prec$ DOMAIN
operations
—*a total function*
tot-fun : DOMAIN, DOMAIN $\rightarrow$ RANGE
—*a partial function; partiality indicated by a comment*
par-fun : DOMAIN $\rightarrow$ RANGE —*partial*
—*a function defined on a subsort*
sub-fun : SPECIAL $\rightarrow$ RANGE
—*dependent sorts; domains of variables indicated at the right*
dep-fun : v, some-fun(v) $\rightarrow$ RANGE **for** $v \in$ DOMAIN
—*a higher-order function; $\rightarrow$ associates to the right*
hi-fun : DOMAIN $\rightarrow$ DOMAIN $\rightarrow$ RANGE
axioms
—*functions need not be completely specified*
—*the variables x and y are implicitly universally quantified*
tot-fun(x, y) = tot-fun(x) + tot-fun(y)
—*definedness predicate*
Defined(par-fun(x)) **if** tot-fun$(x, x) = 0$
—*characteristic function for subsort*
—*signature of χ_{SPECIAL} is* DOMAIN $\rightarrow$ BOOL
$\chi_{\text{SPECIAL}} = \lambda x \cdot$ tot-fun$(x, x) < 0$
—*convention: χ_{SPECIAL} is sometimes written as* "special"
—*i.e., the subsort name in roman font*
special$(x) \Rightarrow$ Defined(par-fun(x)) $\wedge$ par-fun$(x) \in [-1, 1]$
—*notation for Curried functions*
tot-fun$_x = \lambda y \cdot$ tot-fun(x, y) —*Currying on first argument*
tot-fun$^y = \lambda x \cdot$ tot-fun(x, y) —*Currying on second argument*
—*notation for application for higher-order functions*
hi-fun$(x)(y)$ = hi-fun$_x(y)$ = hi-fun$^y(x)$
end

Figure 4.1: Summary of notation for algebraic specification

Cat	the category of small categories
CAT	the category of large categories
$\mathrm{Fun}(\mathcal{C}, \mathcal{D})$	the collection of functors between the categories $\mathcal{C}$ and $\mathcal{D}$
$\mathrm{Nat}(F, G)$	the collection of natural transformations between the functors F and G

The semantics of any algebraic specification is defined in terms of the ambient category theory. Models of a specification form a category, which is a model of the ambient theory. In particular, the model category of the specification CAT below is the category of small categories, **Cat**.

For cases where it is easier to construct models rather than use axiomatic specification, we assume a meta-construct **Mod**, which returns the model category of any specification; thus, $\mathbf{Mod}(\mathrm{CAT}) = \mathbf{Cat}$.

4.3 Categories, sites, and sheaves

The specifications dealing with category theory and sheaf theory below are for reference, so that we can proceed with the derivation rigorously. They tend to be compact, and frequently not suited for easy understanding. Detailed explanations are given in the text. References to relevant portions of the text are attached to the specifications.

spec CAT =
—*section 2.1.1, definition 2.1*
sorts OBJ, ARR
subsorts COMPOSABLE $\prec$ ARR $\times$ ARR —*composable arrows*
constraints small(ARR) —*models are small categories*
operations
dom : ARR $\rightarrow$ OBJ —*domain of an arrow*
cod : ARR $\rightarrow$ OBJ —*codomain of an arrow*
id_ : OBJ $\rightarrow$ ARR —*identity arrow on an object*
_ $\circ$ _ : COMPOSABLE $\rightarrow$ ARR —*composition*
axioms
$\chi_{\mathrm{COMPOSABLE}} = \lambda f, g \cdot \mathrm{dom}(f) = \mathrm{cod}(g)$
$f \circ (g \circ h) = (f \circ g) \circ h$ —*associative law*
$\forall a \xrightarrow{f} b \cdot \mathrm{id}_a \circ f = f = f \circ \mathrm{id}_b$ —*identity law*
end

Notation: In the specifications below, the expression CAT **named** $\mathcal{C}$ denotes a renaming operation on the specification CAT with the assignments

$$\mathrm{OBJ} \mapsto \mathrm{Obj}(\mathcal{C}), \text{ and } \mathrm{ARR} \mapsto \mathrm{Arr}(\mathcal{C}).$$

The composition operator and identity arrows are not renamed; the context usually determines the category with which they are associated.

spec SIEVE-SPEC =
—*section 2.4, definition 2.12*
extend CAT **named** $\mathcal{C}$
sorts SIEVE
subsorts SIEVE $\prec$ SET(Arr($\mathcal{C}$))
operations
sieves : Obj($\mathcal{C}$) $\rightarrow$ SET(SIEVE)
axioms
—*a sieve is a set of arrows closed under right composition*
$\chi_{\text{SIEVE}} = \lambda S \cdot f \in S \Rightarrow (\forall g \in \text{Arr}(\mathcal{C}) \cdot \text{composable}(f, g) \Rightarrow f \circ g \in S)$
—*the set of sieves on an object*
$\text{sieves}(a) = \{ S \in \text{SIEVE} \mid \exists b \xrightarrow{f} a \in S \}$
—*derived fact: elements of a sieve have the same codomain*
$\forall S \in \text{SIEVE}, \exists a \in \text{Obj}(\mathcal{C}) \cdot (\forall f \in S \cdot \text{cod}(f) = a)$
end

spec SIEVE-AS-FUNCTOR =
—*section 2.5.2, definition 2.31*
extend SIEVE-SPEC
—*the operation ϕ below converts a sieve into a functor*
—*this conversion is usually implicit, and ϕ will be omitted*
operations
ϕ : SIEVE $\rightarrow$ Fun($\mathcal{C}^{\text{op}}$, **Set**)
axioms
—*a sieve is a sub-functor of a contravariant hom-functor*
$\phi_S(b) = \{ f \in \text{hom}_{\mathcal{C}}(b, a) \mid f \in S \}$
$\phi_S(c \xrightarrow{g} b) = g^*$
where $a = \text{cod}(S)$ —*codomain of the arrows in S*
and $g^*: f \mapsto f \circ g$
end

In the specifications below, the sort SIEVE will be written as SIEVE($\mathcal{C}$) to explicitly indicate the category in which the sieves are defined.

spec SITE =
—*section 2.4, definitions 2.13 and 2.14*
extend SIEVE-SPEC
operations
$J : a \to \text{SET}(\text{sieves}(a))$ **for** $a \in \text{Obj}(\mathcal{C})$
axioms
—*axioms for a Grothendieck topology*
—*maximal sieve (generated by the identity arrow) is a cover*
$\{ f \mid \text{cod}(f) = a \} \in J(a)$
—*stability of covers under change of base*
$R \in J(a) \Rightarrow \forall b \xrightarrow{f} a \in \text{Arr}(\mathcal{C}) \cdot f^*(R) \in J(b)$
where $f^*(R) = \{ c \xrightarrow{g} b \mid f \circ g \in R \}$
—*stability of covers under refinement*
$R \in J(a) \wedge \exists S \in \text{sieves}(a) \cdot (\forall b \xrightarrow{f} a \in R \cdot f^*(S) \in J(b))$
$\Rightarrow S \in J(a)$
end

Notation: The notation $\langle \mathcal{C}, J \rangle$ in the specification for a sheaf below just assigns names to relevant parts of the site.

spec SHEAF-SPEC($\langle \mathcal{C}, J \rangle$ :: SITE) =
—*section 2.5, definition 2.33*
include SIEVE-AS-FUNCTOR —*for the sieve R below*
sorts SHEAF
subsorts SHEAF $\prec \text{Fun}(\mathcal{C}^{\text{op}}, \mathbf{Set})$
operations
glue : $F \to \text{Nat}(R, F) \to F(a)$ **for** $F \in$ SHEAF, $a \in \text{Obj}(\mathcal{C}), R \in J(a)$
axioms
—*the sheaf condition;*
—$\_ \circ i_R$ *is the restriction of natural transformations induced*
—*by the inclusion of the sieve R in the hom-functor* $\text{hom}_{\mathcal{C}}(-, a)$
$\chi_{\text{SHEAF}} =$
$\lambda F \cdot \forall a \in \text{Obj}(\mathcal{C}), \forall R \in J(a) \cdot \text{bijective}(\_ \circ i_R)$
where $i_R : R \hookrightarrow \text{hom}_{\mathcal{C}}(-, a)$
and $\_ \circ i_R : \text{Nat}(\text{hom}_{\mathcal{C}}(-, a), F) \to \text{Nat}(R, F)$
$\text{glue}_F = y \circ \gamma_F$
where $\gamma_F = (\_ \circ i_R)^{-1} : \text{Nat}(R, F) \to \text{Nat}(\text{hom}_{\mathcal{C}}(-, a), F)$
and $y : \text{Nat}(\text{hom}_{\mathcal{C}}(-, a), F) \xrightarrow{\sim} F(a)$ **for** $a \in \text{Obj}(\mathcal{C})$
—y *is the bijection given by the Yoneda lemma; lemma 2.2*
end

Notation: The sort SHEAF from the specification SHEAF-SPEC above will usually be written as SHEAF($\langle \mathcal{C}, J \rangle$) to explicitly indicate the underlying site.

spec STRICT-EPI-FAMILY =

—*section 3.2, definitions 3.2, 3.3, and 3.4*

extend CAT **named** $\mathcal{C}$

sorts CC-FAMILY, EPI-FAMILY, STRICT-EPI-FAMILY

subsorts STRICT-EPI-FAMILY $\prec$ EPI-FAMILY $\prec$ CC-FAMILY $\prec$ SET(Arr($\mathcal{C}$))

operations

compatible : CC-FAMILY, CC-FAMILY $\rightarrow$ BOOL

axioms

—*family of arrows with a common codomain*

$\chi_{\text{CC-FAMILY}} = \lambda F \cdot \exists a \in \text{Obj}(\mathcal{C}) \cdot (\forall f \in F \cdot \text{cod}(f) = a)$

—*any* $F \in$ CC-FAMILY *will be represented below as* $\{ a_i \xrightarrow{f_i} a \mid i \in I \}$

—*generalization of "epimorphic" to families of arrows*

$\chi_{\text{EPI-FAMILY}} = \lambda F \cdot \text{cc-family}(F) \wedge$
$\forall g, h : a \rightrightarrows b \cdot (\forall i \in I \cdot g \circ f_i = h \circ f_i) \Rightarrow g = h$

—*family of arrows compatible with a strict epimorphic family*

$\text{compatible}(\{ a_i \xrightarrow{f_i} a \mid i \in I \}, \{ a_i \xrightarrow{g_i} a \mid i \in I \}) =$
$\forall c \in \text{Obj}(\mathcal{C}), \forall i, j \in I, \forall c \xrightarrow{u} a_i, c \xrightarrow{v} a_j \in \text{Arr}(\mathcal{C}) \cdot$
$(f_i \circ u = f_j \circ v \Rightarrow g_i \circ u = g_j \circ v)$

—*strictness: the unique assembly property*

$\chi_{\text{STRICT-EPI-FAMILY}} = \lambda F \cdot \text{epi-family}(F) \wedge$
$\forall G \in \text{CC-FAMILY} \cdot \text{compatible}(F, G) \Rightarrow (\exists! a \xrightarrow{h} b \cdot (\forall i \in I \cdot h \circ f_i = g_i))$

end

4.4 Pattern matching

Finally, we provide a specification for pattern matching from which an algorithm is derived in Chapter 5. The simplifying assumptions made in this specification were discussed in Chapter 3.

spec PM-SITE =
—*site suitable for pattern matching, sections 3.1 and 3.5*
extend SITE **named** $\langle \mathcal{C}, J \rangle$, STRICT-EPI-FAMILY
operations
glue : $R \rightarrow$ (CC-FAMILY $\rightarrow$ Arr($\mathcal{C}$)) **for** $R \in J(a), a \in \text{Obj}(\mathcal{C})$
axioms
—*all covers are strict epimorphic families*
$\forall a \in \text{Obj}(\mathcal{C}) \cdot \forall S \in J(a) \cdot$ strict-epi-family(S)
—*gluing operation associated with a* PM-SITE
Defined($\text{glue}_R(F)$) **if** compatible(R, F)
—*the function below is obtained by Skolemizing the existential*
—*quantifier in the definition of a strict epimorphic family*
$\text{glue}_R(F) = h$ —*see the definition of* $\chi_{\text{STRICT-EPI-FAMILY}}$
end

spec PATTERN-MATCH($\langle \mathcal{C}, J \rangle$ :: PM-SITE) =
sorts OCCURRENCE
subsorts OCCURRENCE $\prec$ Arr($\mathcal{C}$) —*some arrows are occurrences*
operations
occurrences : Obj($\mathcal{C}$), Obj($\mathcal{C}$) $\rightarrow$ SET(OCCURRENCE)
axioms
—*all arrows are occurrence arrows*
—*simplifying assumption (section 3.1)*
$\chi_{\text{OCCURRENCE}} = \lambda x \cdot$ true
—*derived fact; since all arrows are occurrence arrows*
occurrences(p, t) = $\text{hom}_{\mathcal{C}}(p, t)$
end

Chapter 5
Derivation of a Pattern Matching Algorithm

"Would you tell me, please, which way I ought to go from here?"
"That depends a good deal on where you want to get to," said the Cat.
— Lewis Carroll, *Alice's Adventures in Wonderland* (1865)

5.1 Overview of the derivation

A generalized version of the Knuth-Morris-Pratt pattern matching algorithm [Knuth et al. 77] is derived by gradually converting the extensional description of pattern matching as a sheaf into an intensional description. The algorithm results from a synergy of four very general program synthesis/transformation techniques (figure 5.1):

1. Divide and conquer: exploit the sheaf condition; assemble a full match by gluing together partial matches;
2. Finite differencing: collect and update partial matches incrementally while traversing the target;
3. Backtracking: instead of saving all partial matches, save just one; when this partial match cannot be extended, fail back to another;
4. Partial evaluation: precompute pattern-based (and therefore constant) computations.

The derivation is carried out in a general framework using the axioms of a site (suitable for pattern matching) and the sheaf condition. All the theories and transformations involved are formally described using the techniques of algebraic specification [Srinivas 90, Wirsing 90] (also see section 4.1). In addition to obtaining the Knuth-Morris-Pratt pattern matching algorithm for strings, trees, graphs, etc., by appropriately instantiating the underlying data structures and topologies, the same program derivation results in matching algorithms for patterns with variables and with multiple patterns. Pursuing other alternatives along the path of the main derivation results in algorithms such as Earley's algorithm for context-free parsing, and Waltz filtering, a relaxation algorithm for providing 3-D interpretations to 2-D images. These other algorithms will be discussed in Chapter 6.

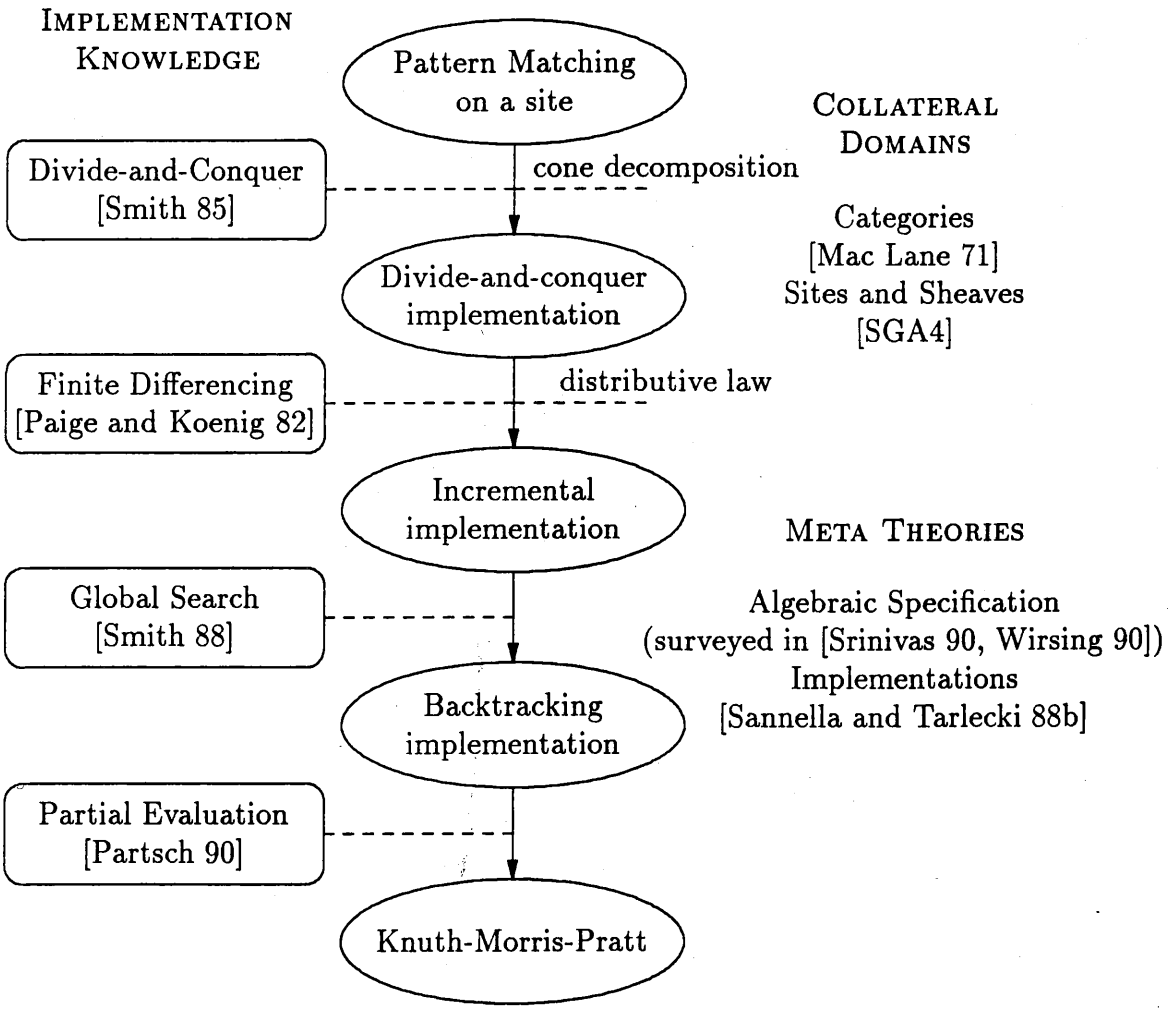


Figure 5.1: Overview of the derivation and classification of domain knowledge

5.2 Decision sequence and design space

Before presenting the detailed derivation, we show an informal picture of the relevant decisions made during the derivation. Alternatives at each decision point are also briefly described, thus showing the part of the design space around the main derivation.

The sheaf condition immediately provides a divide-and-conquer strategy for enumerating the occurrences of a pattern in a target:

1. (Decomposition step) Choose a cover for the pattern.
2. (Recursive invocation) Find occurrences of elements of the cover, i.e., find partial occurrences of the pattern.
3. (Composition step) Compose partial occurrences to obtain occurrences of the pattern.

Termination of the algorithm above requires the following assumption

ASSUMPTION. A finest cover exists for each pattern.

In sites where finest covers may not exist, e.g., context-free parsing, other means of termination have to be employed.

On scrutiny, we see that the recursive invocation in step 2 above will apply the algorithm using covers for each element of the original cover. Since covers are stable under refinement (the third axiom of a Grothendieck topology), we can eliminate the recursive call by choosing the composite cover (a finer cover) in step 1.

If we choose the finest cover in step 1, then step 2 becomes the primitive step of a divide-and-conquer algorithm. This primitive step is usually trivial, e.g., in the sheaves of section 3, this step consists of enumerating the identity arrows on the finest cover of the target. Hence, we do not consider this step further.

Having disposed of steps 1 and 2, we look at step 3, which is the most complex part of the algorithm. There are two parts in this step:

3.1. Enumerate compatible families of partial occurrences on the chosen cover.

3.2. Glue together compatible partial occurrences to obtain total occurrences.

By our assumption that covers are strict epimorphic families, the gluing operation required in step 3.2 is provided by the underlying site, which is a parameter to the derivation.

After examining several characterizations of compatible families of elements, we select the representation of such families by cones. This representation reduces the problem of pattern matching on any site to that of graph matching, thus allowing us to exploit the properties of graphs in the rest of the derivation.

Cones over a diagram can be obtained by combining cones over parts of the diagram. This leads to a divide-and-conquer algorithm for enumerating the cones over a diagram. The relevant diagram here is that generated by the chosen cover of the pattern. We have chosen a binary decomposition so as to simplify the composition step.

1. (Decomposition step) Split the pattern cover P into two pieces.
2. (Recursive invocation) Find all the cones on each piece.
3. (Composition step) Combine the cones on pieces of the cover to obtain the cones on P.

The recursion stops when a diagram cannot be decomposed any more, i.e., it consists of a single object or a single arrow. Cones on objects are trivial; cones on arrows can be obtained by an image computation.

At this stage, we make a decision to sequentially traverse the target, rather than assuming that the extension of the sheaf is already given. Traversing the target inverts the computation of the cones; rather than actively assembling cones on pieces of the pattern cover, cones are assembled in the order given by the traversal. Such an incremental algorithm is obtained by applying finite differencing to the recursive cone assembly above.

1. Maintain a cache of partial occurrences; start with an empty cache.
2. Traverse the target, producing a stream of cones on pieces of the pattern cover.
3. For each increment, update all partial occurrences in the cache which are affected.

The incremental algorithm above is a naive algorithm which maintains all partial occurrences. To facilitate optimization, the problem is now reformulated as a search problem.

1. A state in the search space consists of a partial occurrence (a cone), and the unprocessed part of the stream of increments.
2. An operation in the search space consists of expanding the partial occurrence in a state using some increment in the stream.
3. The goal predicate tests for states which contain full occurrences.

The incremental algorithm above is equivalent to a breadth-first search of this state space. It can be improved by applying standard optimizations to the search. For example, we can prune away states which are not expandable. In the case of

strings, with a left-to-right traversal of the target, we need only save those partial occurrences which touch the right end of the part of the target already traversed. This reduces the number of partial occurrences which have to be updated on each cycle of the incremental algorithm.

We can further reduce the number of partial occurrences at any stage by employing our overall heuristic of converting an extension into an intension: we replace a set of partial occurrences by a generator for that set. Such sets can be precomputed, as shown below.

Examination of the definition of compatible families reveals that, in a partial occurrence, each piece of target is associated with a piece of the pattern. In other words, each piece of the target is "parsed" as a piece of the pattern. Now, if a piece of the pattern occurs more than once in the pattern, then a piece of the target can generate multiple parses. The relationship of the pattern with itself can be precisely represented as the pattern-pattern-sheaf, i.e., the sheaf of occurrences of the pattern in the pattern.

By applying the first few steps of our derivation to the pattern-pattern-sheaf (observe that our derivation applies to any sheaf, not just to sheaves of occurrences), we can precompute the following:

> for each partial occurrence ν, the set of partial occurrences which can be generated from ν by parsing its pieces differently.

Such alternative partial occurrences which are generated are said to be *subsumed* by the original partial occurrence. The subsumption relation forms a partial order. Using such precomputed sets, we can more efficiently search (breadth-first search with dependency-directed backtracking) the space of partial occurrences:

1. Try to expand the current set of partial occurrences using some parse of the current increment.

2. If some partial occurrence cannot be expanded, replace it with the largest partial occurrences subsumed by it and try again.

In the algorithm above, it may happen that for a particular increment, a partial occurrence has to be repeatedly replaced by subsumed partial occurrences until the increment can expand one. This repetition can be avoided by precomputing the appropriate subsumed partial occurrences for each partial occurrence and increment pair: this function is the failure function of the Knuth-Morris-Pratt algorithm. All these optimizations are then consolidated into a generalized Knuth-Morris-Pratt algorithm.

5.3 Detailed derivation

The starting point of our derivation is the algebraic specification of pattern matching given in the previous chapter. For convenience, we repeat that specification here.

spec PATTERN-MATCH($\langle \mathcal{C}, J \rangle$ :: PM-SITE) =
sorts OCCURRENCE
subsorts OCCURRENCE $\prec$ Arr($\mathcal{C}$) —*some arrows are occurrences*
operations
occurrences : Obj($\mathcal{C}$), Obj($\mathcal{C}$) $\rightarrow$ SET(OCCURRENCE)
axioms
—*all arrows are occurrence arrows*
—*simplifying assumption (section 3.1)*
$\chi_{\text{OCCURRENCE}} = \lambda x \cdot \text{true}$
—*derived fact; since all arrows are occurrence arrows*
occurrences$(p, t) = \text{hom}_{\mathcal{C}}(p, t)$
end

5.3.1 The extension of the occurrence relation

To derive an algorithm for finding occurrences, we will use the technique of converting the extension of the occurrence relation into an efficient intension (i.e., algorithm). The advantage of investigating the extension is that it captures only the essential properties of the occurrence relation, and is also impartial to sequential and parallel algorithms. Extensionally, the occurrence relation for a fixed target and variable pattern forms a sheaf, as shown by lemma 3.1. We include this fact in the specification below.

spec PATTERN-MATCH-1($\langle \mathcal{C}, J \rangle$ :: PM-SITE) =
extend PATTERN-MATCH($\langle \mathcal{C}, J \rangle$), SHEAF-SPEC($\langle \mathcal{C}, J \rangle$)
operations
occ-sheaf : Obj($\mathcal{C}$) $\rightarrow$ Fun($\mathcal{C}^{\text{op}}$, **Set**)
axioms
—*fixed target; variable pattern*
occ-sheaf$^t = \text{hom}_{\mathcal{C}}(-, t)$
—*hom-functors are sheaves when covers are strict epimorphic families*
—*Lemma 3.1*
occ-sheaf$^t \in$ SHEAF($\langle \mathcal{C}, J \rangle$)
end

5.3.1.Alt.1 Pattern matching as search: Geometry vs. algebra

Rather than choose to fix the target, we can choose to fix the pattern, and investigate the properties of $\hom_{\mathcal{C}}(p, -)$. This converts pattern matching into a search problem, and most derivations of the Knuth-Morris-Pratt algorithm in the literature start with such a characterization. For example, Dijkstra starts with the following specification for matching in strings [Dijkstra 76]:

> Given a pattern p and a target t, find all indices i, with $1 \leq i \leq |t|$, such that p matches t at i.

The collection of indices $1 \leq i \leq |t|$ defines a search space. The condition "match(p, t, i)" is the goal predicate.

We have not chosen the approach of matching as search because it is difficult to generalize this specification: the formulation depends on the properties of the particular data structure chosen. Also, this approach misses the geometric aspects of pattern matching, and hence results in a poorer theory.

Any treatment of pattern matching has to deal with aspects concerning both the pattern and the target. For example, later on in Dijkstra's derivation, the decomposition of the matching relation given by

$$\text{match}(p, t, i) = \forall 1 \leq j \leq |p| \cdot p[j] = t[i + j - 1]$$

is used to design the failure function by considering the relationship of the pattern with itself.

In our derivation, we postpone considering aspects concerning the target until section 5.3.17.

Whenever there is a problem described in terms of hom-sets, e.g., $\hom_{\mathcal{C}}(p, t)$, there are, in general, two ways of looking for solutions: explore properties of p, and explore properties of t. The former results in geometric concepts; the latter in algebraic or logical concepts. This duality is characteristic of category theory, arising from the fact that arrows are accorded a first-class status, and from the fact that the directionality of arrows differentiates objects which are domains and objects which are codomains.

The interaction between the two approaches comes to the fore in sheaf theory: the site captures the geometry, the stalk space captures the algebra. The power of sheaf theory comes from this ability to combine the two aspects. Our use of sheaf theory to describe pattern matching not only exploits well-known mathematical results, but also shows that many data structures in computer science possess interesting geometric properties.

5.3.2 A divide-and-conquer theory

The sheaf condition provides an opportunity for calculating occurrences(p, t) using partial occurrences on pieces of p (a piece of p is an element of a cover of p). Thus, we use a divide-and-conquer strategy to compute occurrences. To do so, we will first describe a theory of divide-and-conquer algorithms, and then see what additional assumptions are necessary by comparing this theory with the specification PATTERN-MATCH-1 above.

We reproduce below the divide-and-conquer theory invented by Smith [Smith 85], simplified and translated into the algebraic specification notation being used here. We are interested in divide-and-conquer program schemes[1] of the following kind:

$F(x) =$
  if
    $Primitive(x) \rightarrow Directly\_Solve(x)$ □
    $\neg\, Primitive(x) \rightarrow Compose \circ \alpha(F) \circ Decompose(x)$
  fi

Translated into our notation, the specification of F looks as in the specification DIVIDE-AND-CONQUER below. The specification SET-1 defines the higher-order function "α", which applies a given function to all the elements of a set, and builds the set of results.

spec SET-1 =
extend SET
operations
  $\alpha : (\mathrm{D} \rightarrow \mathrm{R}) \rightarrow (\mathrm{SET(D)} \rightarrow \mathrm{SET(R)})$
axioms
    —*apply a function to each element of a set*
    —*and return the set of the results*
  $\alpha(f)(\emptyset) = \emptyset$
  $\alpha(f)(\{x\}) = \{f(x)\}$
  $\alpha(f)(x \cup y) = \alpha(f)(x) \cup \alpha(f)(y)$
end

[1] The notation is similar to that of Dijkstra's guarded commands [Dijkstra 76].

spec DIVIDE-AND-CONQUER =
—*based on [Smith 85]*
extend SET-1
sorts D, R —*domain and range*
subsorts P $\prec$ D, NP $\prec$ D —*primitive and non-primitive elements of* D
operations
$\_ \succ \_ :$ D, D $\rightarrow$ BOOL —*well-founded order on* D
primitive : D $\rightarrow$ BOOL
directly-solve : P $\rightarrow$ R
decompose : NP $\rightarrow$ SET(D)
compose : SET(R) $\rightarrow$ R
axioms
—D *is the disjoint union of primitive and non-primitive elements*
D = P $\cup$ NP P $\cap$ NP = $\emptyset$
χ_{P} = primitive $\chi_{\text{NP}} = \neg$ primitive
well-founded($\succ$) —*no infinite descending chains*
—*the decompose operator preserves* $\succ$
$\forall x_i \in \text{decompose}(x) \cdot x \succ x_i$
end

The intent is to use the divide-and-conquer specification above to implement an arbitrary specification Π-SPEC,

spec Π-SPEC =
sorts D, R
operations
Π : D $\rightarrow$ R
end

via a constructor DC-IMP (see [Sannella and Tarlecki 88b] for a definition of constructor implementations)

$$\Pi\text{-SPEC} \rightsquigarrow \text{DC-IMP}(\text{DIVIDE-AND-CONQUER})$$

where the constructor DC-IMP defines a recursive function F in the specification DIVIDE-AND-CONQUER as shown below:

constructor DC-IMP =
derive from
extend DIVIDE-AND-CONQUER **with**
F : D $\rightarrow$ R
—*recursive definition of* F
primitive(x) $\Rightarrow F(x) =$ directly-solve(x)
$\neg$ primitive(x) $\Rightarrow F(x) =$ compose $\circ\ \alpha(F) \circ$ decompose(x)
by $\{\Pi \mapsto F, \text{D} \mapsto \text{D}, \text{R} \mapsto \text{R}\}$
end

The implementation above is correct if we can prove that

$$\forall x \in \text{D} \cdot \Pi(x) = F(x).$$

The key feature of a divide-and-conquer implementation is that the proof of the proposition above is carried out by structural induction over the domain D using the well-founded order $\succ$. This entails

1. proving primitive$(x) \Rightarrow \Pi(x) =$ directly-solve(x); and
2. proving $\neg$ primitive$(x) \Rightarrow \Pi(x) =$ compose$(\{F(x_i)\})$ with the assumption that $\forall x_i \in$ decompose$(x) \cdot \Pi(x_i) = F(x_i)$.

Given proofs of the two parts above, we can conclude (using Theorem 6.1 of [Smith 85]) that the implementation of Π by F is correct.

5.3.3 Exploiting the sheaf condition

Comparing the specifications Π-SPEC and PATTERN-MATCH-1, we get the following correspondence:

$\Pi \mapsto$ occurrences
D $\mapsto$ Obj$(\mathcal{C}) \times$ Obj$(\mathcal{C})$
R $\mapsto$ SET(OCCURRENCE)

Thus the problem now is to design appropriate decomposition and composition operators. Since we are interested in exploiting the sheaf condition, we postulate a decomposition operator which assigns a covering sieve to each non-primitive pattern.

decompose : Obj$(\mathcal{C}) \rightarrow$ SIEVE$(\mathcal{C})$ —*partial*
—*decomposition yields a covering sieve*
—*decision about which decomposition to pick is delayed*
decompose$(p) \in J(p)$
Defined(decompose(p)) **if** $\neg$ primitive(p)

The appropriate sheaf to use for computing occurrences is given by the following sequence of equations.

occurrences(p, t)
$=$ hom$_{\mathcal{C}}(p, t)$ —*definition from* PATTERN-MATCH
$=$ hom$_{\mathcal{C}}(-, t)(p)$ —*Curry on the second argument*
$=$ occ-sheaf$^t(p)$ —*definition from* PATTERN-MATCH-1

By the sheaf condition, using the cover for p given by the decomposition operator, we have

$$\text{occ-sheaf}^t(p) = \{\, \text{glue}_{\text{occ-sheaf}^t}(\tau) \mid \tau \in \text{Nat}(P, \text{occ-sheaf}^t) \,\}$$
$$\textbf{where } P = \text{decompose}(p)$$

The expression $\text{Nat}(P, \text{occ-sheaf}^t)$ above is based on the declarative/extensional view of the sheaf where the values of $\text{occ-sheaf}^t(x)$, for all arguments x, are all assumed to exist in advance. Since we are interested in an algorithm to compute $\text{occ-sheaf}^t(p)$, we do not have the luxury of this assumption. Thus, let us see for what x's the values of $\text{occ-sheaf}^t(x)$ are necessary to compute $\text{Nat}(P, \text{occ-sheaf}^t)$. By definition, a natural transformation $\tau \in \text{Nat}(P, \text{occ-sheaf}^t)$ assigns a function $\tau_{p_i}: P(p_i) \rightarrow \text{occ-sheaf}^t(p_i)$ for each $p_i \in \text{Obj}(\mathcal{C})$. These functions are trivial for all p_i for which $P(p_i)$ is empty. Thus, we need only values of $\text{occ-sheaf}^t(p_i)$ for each p_i for which $P(p_i)$ is not empty. This collection of p_i is precisely the base of cover P of p given by the decomposition operator (see definition 2.32).

Thus, for a non-primitive pattern p, we have the following steps for calculating $\text{occ-sheaf}^t(p)$:

decompose p, obtaining a covering sieve P with base $\{\, p_i \mid i \in I \,\}$
recursively invoke $\text{occ-sheaf}^t(p_i)|_{\text{base}(P)}$
compute $\text{Nat}(P, \text{occ-sheaf}^t)$ —*using values of* occ-sheaf^t *at* $p_i, i \in I$
$\forall \tau \in \text{Nat}(P, \text{occ-sheaf}^t) \cdot \text{glue}_{\text{occ-sheaf}^t}(\tau)$

In the sequence of steps above, the decomposition operator does not produce a set; it produces a sieve.[2] Thus, the signature of the DIVIDE-AND-CONQUER theory is not general enough to accommodate this situation. We will not, however, generalize the DIVIDE-AND-CONQUER theory. In the two instances where we use a functorial version of divide-and-conquer,[3] we validate the implementation: the implementation DC-PATTERN-MATCH in section 5.3.5 is justified by the sheaf condition; the implementation CONE-DECOMPOSITION in section 5.3.10 is justified by Theorem 5.1 on cone decomposition.

5.3.4 Existence of finest covers

To determine the particular cover assigned by the decomposition operator to a pattern, let us examine the recursive invocation of the divide-and-conquer algorithm. In this step, for each non-primitive pattern p_i, the decomposition operator is applied again to yield a cover for p_i. Since covers are stable under refinement (the third axiom of a Grothendieck topology), we can design the decomposition operator such that the composed, finer, cover is returned on the first invocation. By unfolding the

[2] It is not possible to treat the sieve here as a set; the structure of the sieve is essential for determining compatible families of partial occurrences which can be glued together.

[3] A functorial version of divide-and-conquer implements a functor. Both objects and arrows are decomposed. The decomposition is via a colimit; the composition is via a limit.

divide-and-conquer algorithm further, we can continue this process until the decomposition operator yields a cover none of whose pieces can be further decomposed. This necessitates the assumption that finest covers exist in the underlying site.

Moreover, for the divide-and-conquer algorithm to terminate, we require that the decomposition operator always yield a finite cover, and that primitive objects be finite, so that the base step, "directly-solve" will terminate. Together with the finest cover assumption, this implies that all objects in the site are finite.

spec FINITARY-NOETHERIAN-PM-SITE =
—PM-SITE *in which objects are finite and finest covers exist*
extend PM-SITE
axioms
—$\subseteq$ *is the inclusion relation on sieves*
—$S \subseteq R$ *means* S *is at least as fine as* R
—*a finest covering sieve exists for each object; additional assumption*
$\forall a \in \mathrm{Obj}(\mathcal{C}), \exists F \in J(a) \cdot (\forall R \in J(a) \cdot F \subseteq R)$
—*finiteness of objects*
$\forall a \in \mathrm{Obj}(\mathcal{C}) \cdot \mathrm{finite}(a)$
end

5.3.4.Alt.1 No finest covers

The assumption that finest covers exist need not always be true. For an example, consider the standard topology on complex numbers, with the problem of computing Mandelbrot sets on the unit circle. The computation can be split up by using a cover for the unit circle. A finest such split does not exist. Since the ordering imposed by splitting is not well-founded, a divide-and-conquer algorithm will not terminate. Hence we have to resort to other means to terminate the algorithm: e.g., impose restrictions on the precision, or on the time allocated for the process.

For another example, where finest covers need not exists, see section 6.2.

5.3.5 A divide-and-conquer algorithm

Using the existence of finest covers, we can eliminate the recursion in the divide-and-conquer algorithm by unfolding it completely. We thus have the following implementation for PATTERN-MATCH-1.

spec DC-PATTERN-MATCH($\langle \mathcal{C}, J \rangle$ :: FINITARY-NOETHERIAN-PM-SITE) =
sorts P-OBJ, NP-OBJ, P-ARR, NP-ARR
—*primitive and non-primitive objects and arrows*
subsorts P-OBJ $\prec$ Obj($\mathcal{C}$), NP-OBJ $\prec$ Obj($\mathcal{C}$),
P-ARR $\prec$ Arr($\mathcal{C}$), NP-ARR $\prec$ Arr($\mathcal{C}$)
operations
dc-occs : Obj($\mathcal{C}$) $\rightarrow$ Fun($\mathcal{C}^{\mathrm{op}}$, **Set**)
prim-obj : Obj($\mathcal{C}$) $\rightarrow$ BOOL
prim-arr : Arr($\mathcal{C}$) $\rightarrow$ BOOL
directly-solve : P-OBJ, Obj($\mathcal{C}$) $\rightarrow$ Obj(**Set**)
directly-solve : P-ARR, Obj($\mathcal{C}$) $\rightarrow$ Arr(**Set**) —*overloaded*
decompose : NP-OBJ $\rightarrow$ SIEVE($\mathcal{C}$)
—*a decomposition operator for non-primitive arrows is not necessary*[4]
—*the composition operator is implicit*
axioms
—*primitive and non-primitive objects are disjoint*
$\chi_{\text{P-OBJ}} = \text{prim-obj}$ $\chi_{\text{NP-OBJ}} = \neg\,\text{prim-obj}$
—*primitive and non-primitive arrows are disjoint*
$\chi_{\text{P-ARR}} = \text{prim-arr}$ $\chi_{\text{NP-ARR}} = \neg\,\text{prim-arr}$
—*primitive objects have no non-trivial covers*[5]
$\text{prim-obj}(p)$ **iff** $J(p) = \left\{ \{ f \mid \text{cod}(f) = p \} \right\}$
—*arrows are primitive if their domain and codomain are primitive objects*
$\text{prim-arr}(p \xrightarrow{f} q)$ **iff** $\text{prim-obj}(p) \wedge \text{prim-obj}(q)$
—*decomposition yields the finest covering sieve*
$\text{decompose}(p) \in J(p) \wedge \forall R \in J(p) \cdot \text{decompose}(p) \subseteq R$
—*details of "directly-solve" will be delayed*

—*definition of "dc-occs"*
$\text{prim-obj}(p) \Rightarrow \text{dc-occs}^t(p) = \text{directly-solve}^t(p)$
$\text{prim-arr}(f) \Rightarrow \text{dc-occs}^t(f) = \text{directly-solve}^t(f)$
—*the composition operator applies the sheaf condition*
—*to go from partial occurrences to full occurrences*
$\neg\,\text{prim-obj}(p) \Rightarrow \text{dc-occs}^t(p) = \{ \text{glue}_P(\tau) \mid \tau \in \text{Nat}(P|_{\text{base}(P)}, \text{dc-occs}^t\,|_{\text{base}(P)}) \}$
where $P = \text{decompose}(p)$ —*the sieve P is represented as a functor*
—*computation of "dc-occs" for non-primitive arrows in not necessary*
end

[4]Computing the sheaf for non-primitive arrows is done by choosing a cover for the codomain and pulling-back the cover to the domain using the second axiom of a Grothendieck topology.

[5]The trivial cover is the sieve generated by the identity arrow, and guaranteed to be a cover by the first axiom of a Grothendieck topology.

divide-and-conquer algorithm further, we can continue this process until the decomposition operator yields a cover none of whose pieces can be further decomposed. This necessitates the assumption that finest covers exist in the underlying site.

Moreover, for the divide-and-conquer algorithm to terminate, we require that the decomposition operator always yield a finite cover, and that primitive objects be finite, so that the base step, "directly-solve" will terminate. Together with the finest cover assumption, this implies that all objects in the site are finite.

spec FINITARY-NOETHERIAN-PM-SITE =
—PM-SITE *in which objects are finite and finest covers exist*
extend PM-SITE
axioms
—$\subseteq$ *is the inclusion relation on sieves*
—$S \subseteq R$ *means* S *is at least as fine as* R
—*a finest covering sieve exists for each object; additional assumption*
$\forall a \in \mathrm{Obj}(\mathcal{C}), \exists F \in J(a) \cdot (\forall R \in J(a) \cdot F \subseteq R)$
—*finiteness of objects*
$\forall a \in \mathrm{Obj}(\mathcal{C}) \cdot \mathrm{finite}(a)$
end

5.3.4.Alt.1 No finest covers

The assumption that finest covers exist need not always be true. For an example, consider the standard topology on complex numbers, with the problem of computing Mandelbrot sets on the unit circle. The computation can be split up by using a cover for the unit circle. A finest such split does not exist. Since the ordering imposed by splitting is not well-founded, a divide-and-conquer algorithm will not terminate. Hence we have to resort to other means to terminate the algorithm: e.g., impose restrictions on the precision, or on the time allocated for the process.

For another example, where finest covers need not exists, see section 6.2.

5.3.5 A divide-and-conquer algorithm

Using the existence of finest covers, we can eliminate the recursion in the divide-and-conquer algorithm by unfolding it completely. We thus have the following implementation for PATTERN-MATCH-1.

spec DC-PATTERN-MATCH($\langle \mathcal{C}, J \rangle$ :: FINITARY-NOETHERIAN-PM-SITE) =
sorts P-OBJ, NP-OBJ, P-ARR, NP-ARR
—*primitive and non-primitive objects and arrows*
subsorts P-OBJ $\prec$ Obj($\mathcal{C}$), NP-OBJ $\prec$ Obj($\mathcal{C}$),
P-ARR $\prec$ Arr($\mathcal{C}$), NP-ARR $\prec$ Arr($\mathcal{C}$)
operations
dc-occs : Obj($\mathcal{C}$) $\rightarrow$ Fun($\mathcal{C}^{\mathrm{op}}$, **Set**)
prim-obj : Obj($\mathcal{C}$) $\rightarrow$ BOOL
prim-arr : Arr($\mathcal{C}$) $\rightarrow$ BOOL
directly-solve : P-OBJ, Obj($\mathcal{C}$) $\rightarrow$ Obj(**Set**)
directly-solve : P-ARR, Obj($\mathcal{C}$) $\rightarrow$ Arr(**Set**) —*overloaded*
decompose : NP-OBJ $\rightarrow$ SIEVE($\mathcal{C}$)
—*a decomposition operator for non-primitive arrows is not necessary*[4]
—*the composition operator is implicit*
axioms
—*primitive and non-primitive objects are disjoint*
$\chi_{\text{P-OBJ}}$ = prim-obj $\chi_{\text{NP-OBJ}} = \neg$ prim-obj
—*primitive and non-primitive arrows are disjoint*
$\chi_{\text{P-ARR}}$ = prim-arr $\chi_{\text{NP-ARR}} = \neg$ prim-arr
—*primitive objects have no non-trivial covers*[5]
prim-obj(p) **iff** $J(p) = \left\{ \{ f \mid \mathrm{cod}(f) = p \} \right\}$
—*arrows are primitive if their domain and codomain are primitive objects*
prim-arr($p \xrightarrow{f} q$) **iff** prim-obj(p) $\wedge$ prim-obj(q)
—*decomposition yields the finest covering sieve*
decompose(p) $\in J(p) \wedge \forall R \in J(p) \cdot \mathrm{decompose}(p) \subseteq R$
—*details of "directly-solve" will be delayed*

—*definition of "dc-occs"*
prim-obj(p) $\Rightarrow$ dc-occs$^t(p)$ = directly-solve$^t(p)$
prim-arr(f) $\Rightarrow$ dc-occs$^t(f)$ = directly-solve$^t(f)$
—*the composition operator applies the sheaf condition*
—*to go from partial occurrences to full occurrences*
$\neg$ prim-obj(p) $\Rightarrow$ dc-occs$^t(p) = \{ \mathrm{glue}_P(\tau) \mid \tau \in \mathrm{Nat}(P|_{\mathrm{base}(P)}, \text{dc-occs}^t|_{\mathrm{base}(P)}) \}$
where P = decompose(p) —*the sieve P is represented as a functor*
—*computation of "dc-occs" for non-primitive arrows in not necessary*
end

[4]Computing the sheaf for non-primitive arrows is done by choosing a cover for the codomain and pulling-back the cover to the domain using the second axiom of a Grothendieck topology.

[5]The trivial cover is the sieve generated by the identity arrow, and guaranteed to be a cover by the first axiom of a Grothendieck topology.

The specification PATTERN-MATCH-1 is implemented by the specification DC-PATTERN-MATCH above via the following constructor which extends the specification with a function to return occurrences, renames some functions, and hides others:

$$\text{PATTERN-MATCH-1} \rightsquigarrow \kappa\text{-DC}(\text{DC-PATTERN-MATCH})$$

where

constructor κ-DC =
derive from
  extend DC-PATTERN-MATCH **with**
    operations
    occurrences : $\text{Obj}(\mathcal{C}), \text{Obj}(\mathcal{C}) \rightarrow$ SET(OCCURRENCE)
    axioms
      —*the main function being implemented*
    occurrences(p, t) = dc-occs$^t(p)$
by { occurrences $\mapsto$ occurrences, occ-sheaf $\mapsto$ dc-occs }
end

Note that, in the specification DC-PATTERN-MATCH, we not only have implemented the body of the specification PATTERN-MATCH-1, but also added an additional assumption to the parameter theory PM-SITE (that finest covers exist). For the specification above to be a valid implementation, we have to invoke the theorem about the compatibility of vertical implementation steps and horizontal structuring operations (parameterization) [Sannella and Tarlecki 88b].

EXAMPLE 5.1. Here is an example to illustrate the working of the algorithm in DC-PATTERN-MATCH using the occurrence sheaf for graphs shown in figure 3.2.

Primitive patterns in this site ($\mathbf{LUCGraph}_{\hookrightarrow}$) consist of graphs which have only one node or only one edge. The pattern p is not primitive. Hence it is decomposed into the finest cover P shown in the figure. Note that the pieces of the finest cover are primitive (i.e., nodes or edges), and cannot be decomposed further.

The operation "directly-solve" finds occurrences of the pieces in the pattern cover. Note that "directly-solve" not only has to enumerate occurrences of pattern pieces but also has to compute restrictions corresponding to any arrow in the pattern cover. Thus, "directly-solve" builds the part of the occurrence sheaf shown in the figure, except the occurrences of p.

Next, compatible families of partial occurrences are enumerated; an example is shown in bold arrows in the figure. Finally, the elements of such families are glued together to obtain occurrences of the pattern p. □

5.3.5.Alt.1 A complex-divide-simple-merge strategy

In the divide-and-conquer implementation DC-PATTERN-MATCH above, we chose a one-step decomposition operator, thus pushing all the work into the composition operator (see section 5.3.7 below). An alternative is to choose a simple composition operator, e.g., a binary decomposition of the pattern, so that the composition operator has to handle only two arguments at a time. In this case, the computation of $\mathrm{Nat}(P, \text{dc-occs}^t)$ can be reduced to a simple pullback (using the equivalence with limits in section 5.3.7 below). Such a strategy is useful when designing parallel pattern matching algorithms.

5.3.6 The gluing operation

In the implementation DC-PATTERN-MATCH above, in the expression

$$\{\, \mathrm{glue}_P(\tau) \mid \tau \in \mathrm{Nat}(P|_{\mathbf{base}(P)}, \text{dc-occs}^t|_{\mathbf{base}(P)}) \,\}$$

we used the gluing operation associated with the cover P. Strictly speaking, this gluing operation depends on the sheaf "occ-sheaf^t" in the specification PATTERN-MATCH-1. However, the natural transformation τ above corresponds to a family of arrows $F = \{\, p_i \to t \mid p_i \in P \,\}$ which is compatible with the covering sieve P (cf. Lemma 3.1). Thus, we can substitute for "$\mathrm{glue}_{\mathbf{occ\text{-}sheaf}^t}$", the gluing operation "glue_P" obtained from the underlying site: "glue_P" is the gluing operation associated with the cover P by virtue of its being a strict epimorphic family.

5.3.7 Compatible families of partial occurrences

The most complex step of the implementation DC-PATTERN-MATCH is the computation of the set of natural transformations $\mathrm{Nat}(P, \text{dc-occs}^t)$. Intuitively, any such natural transformation is a compatible family of partial occurrences of the pattern (see section 2.5.3). We now examine several characterizations of compatible families to see which is most amenable to efficient computation.

Let $F: \mathcal{C}^{\mathbf{op}} \to \mathbf{Set}$ be a sheaf on a site $\langle \mathcal{C}, J \rangle$, and let $R \in J(a)$ be a covering sieve of the object $a \in \mathrm{Obj}(\mathcal{C})$.

5.3.7.Alt.1 Computing natural transformations from first principles

We can compute the set of natural transformations $\mathrm{Nat}(R, F)$ by going back to first principles, using a generate-and-test algorithm, as shown in figure 5.2.

1. (Generate) For each object $a \in |\mathcal{C}|$, enumerate the set of functions from $R(a)$ to $F(a)$. Compute the product $\prod_{a \in |\mathcal{C}|} \mathbf{Set}(R(a), F(a))$ of all these sets.
2. (Test) For each tuple τ in the product, for each arrow $a \xrightarrow{f} b$ in $\mathcal{C}$, test the commutativity of the following diagram:

$$\begin{array}{ccc} R(a) & \xrightarrow{\tau_a} & F(a) \\ {\scriptstyle R(f)}\downarrow & \stackrel{?}{=} & \downarrow{\scriptstyle F(f)} \\ R(b) & \xrightarrow{\tau_b} & F(b) \end{array}$$

Retain only those tuples which pass the test.

Figure 5.2: Generate-and-test algorithm for $\mathrm{Nat}(R, F)$

5.3.7.Alt.2 Exploiting the limit theorem

An alternative characterization is given by the following bijection (lemma 2.4, section 2.1.5):

$$\mathrm{Nat}(R, F) \cong \varprojlim_{\mathcal{C}/R} F^{\#},$$

where the functor $F^{\#}: \mathcal{C}/R \to \mathbf{Set}$ is given by

$$F^{\#}(\langle a, \alpha \rangle) = F(a) \qquad \text{and} \qquad F^{\#}(f) = F(f^{\mathrm{op}}).$$

The diagram over which the limit is calculated is the graph of the sieve R. The limit above may be directly computed using the limit theorem [Mac Lane 71, page 109], which states that limits can be computed using products and equalizers. For readers who have the inclination for such details, it is interesting to note that the computation of the limit above by products and equalizers is identical to the generate-and-test scheme for natural transformations presented above, but restricted to the base of the sieve R. [Rydeheard and Burstall 88, page 82] shows a slightly different formulation of the dual of the limit theorem, and also shows how to convert such a constructive theorem into an algorithm.

Notation. In subsequent discussion, we will drop the superscript in $F^{\#}$ as it can be deduced from the context.

The chosen alternative

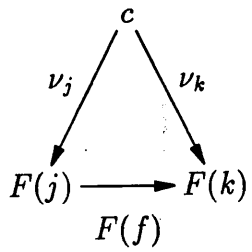
Another way to compute the limit, taking advantage of the fact that F is a set-valued functor, is given by the following bijection [Mac Lane 71, page 106]:

$$\varprojlim_{\mathcal{C}/R} F \cong \underset{\mathcal{C}/R}{\mathrm{Cones}}(1, F),$$

where "1" denotes any singleton set. We give below a functorial definition of "cone" (cf. definition 2.5), so as to connect the notion to that of a compatible family of partial occurrences.

DEFINITION 5.2: *Cone.* Let $\mathcal{C}$ and $\mathcal{J}$ be two categories, $\mathcal{J}$ being the "index" or "diagram" category (usually finite or small), and F a functor $F: \mathcal{J} \to \mathcal{C}$. Corresponding to any $\mathcal{C}$-object c, we have the constant functor Δc which maps all $\mathcal{J}$-objects onto c and all $\mathcal{J}$-arrows onto id_c. A cone from c to the functor $F: \mathcal{J} \to \mathcal{C}$ is defined to be a natural transformation $\Delta c \dot{\to} F$. □

Thus, a cone ν from the object c to the functor F assigns to each object $j \in \mathcal{J}$ an arrow $c \xrightarrow{\nu_j} F(j)$ such that for each arrow $j \xrightarrow{f} k$ in $\mathcal{J}$, the following diagram commutes:



The diagram above also shows the origin of the name "cone" for this construct.

We will now interpret the definition of a cone in the context of pattern matching. We are interested in cones of the form $\underset{\mathcal{C}/P}{\mathrm{Cones}}(1, F)$ with $F: \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$ an occurrence sheaf and P a covering sieve for a pattern $p \in |\mathcal{C}|$. As we saw in section 2.1.5, $\mathcal{C}/P$ is just the graph of the sieve P considered as a functor. The objects of $\mathcal{C}/P$ are pieces $p_i \to p$ of the pattern, and the arrows are of the form $p_i \xrightarrow{f} p_j$. A cone from "1" to F assigns to each piece $p_i \to p$ of the pattern an element $q_i \in F(p_i)$, such that for any arrow $p_i \xrightarrow{f} p_j$ in $\mathcal{C}/P$, we have $F(f): q_i \mapsto q_j$. Thus, it can be seen that each such cone is a compatible family of partial occurrences.

Two examples of cones using the example occurrence sheaves in figures 3.1 and 3.3 are shown in figures 5.3 and 5.4.

We choose the cone characterization of compatible families of occurrences to further proceed with the derivation, because it is a direct description of the intuitive

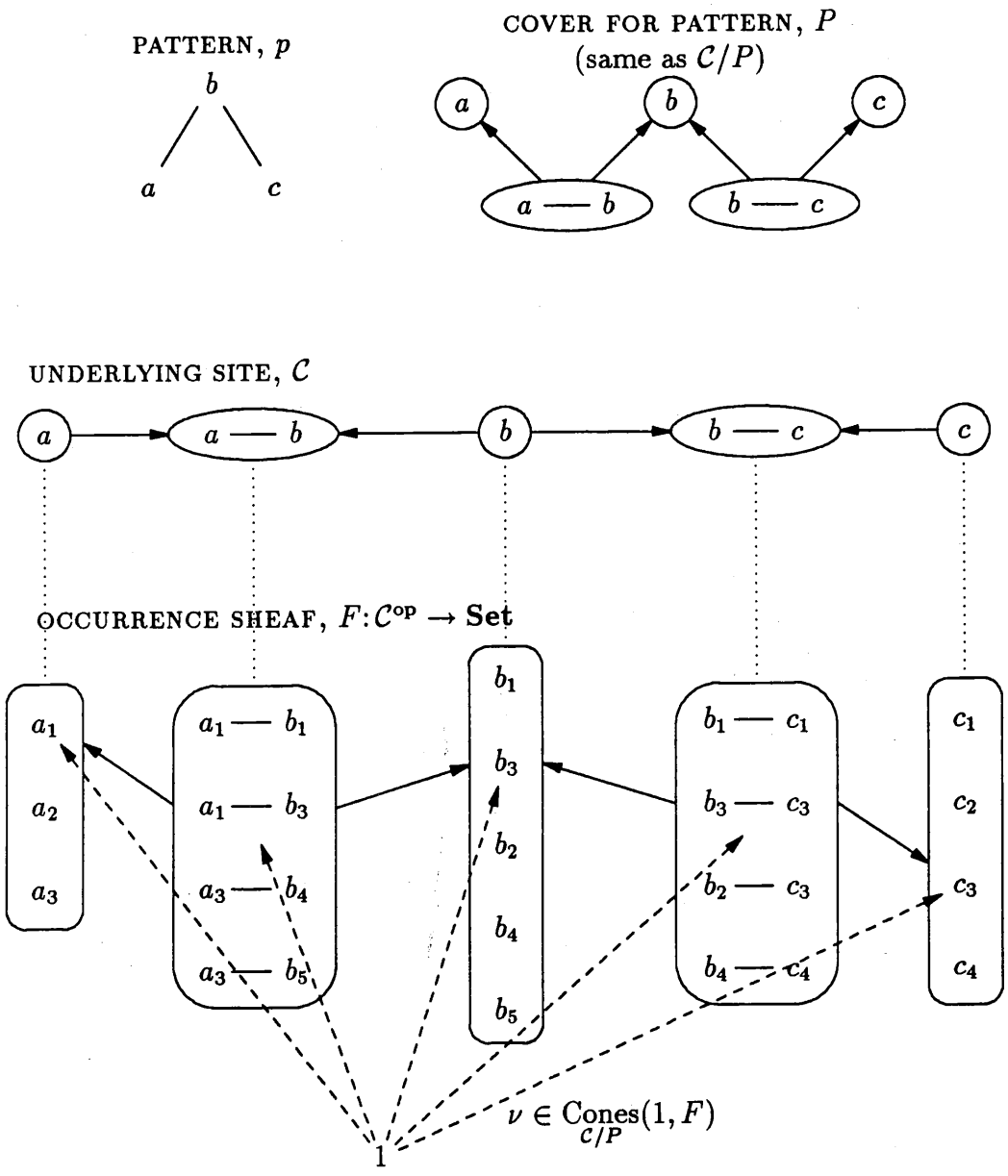


Figure 5.3: Cone on an occurrence sheaf: example with trees (cf. figure 3.1)

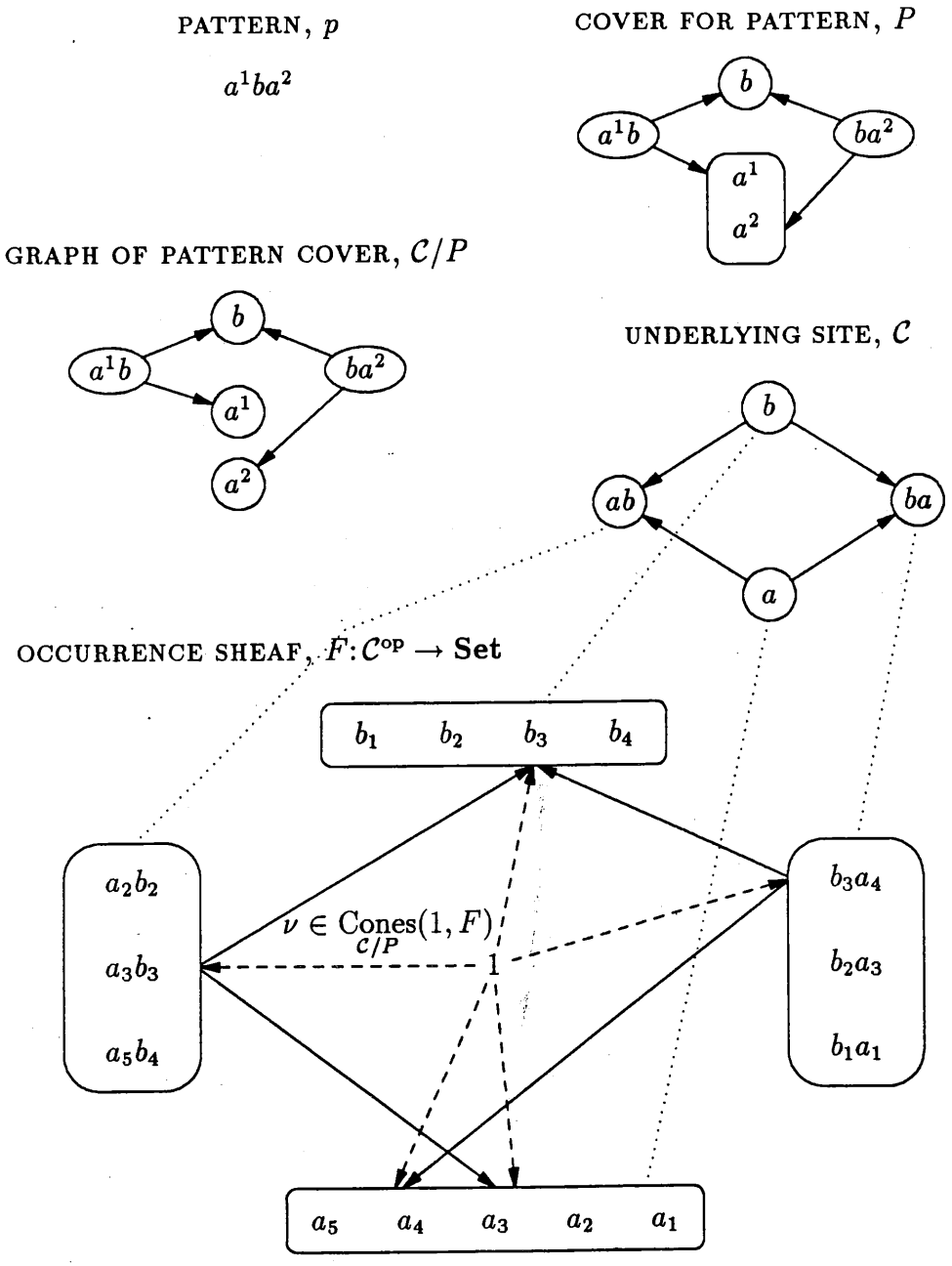


Figure 5.4: Cone on an occurrence sheaf: example with strings (cf. figure 3.3)

notion of compatible family of elements, and because it can be converted into an efficient algorithm. We record this decision in the specification in section 5.3.8 below, where the gluing operation is also accordingly modified.

5.3.8 Assessment

Consolidating the work of the past few sections, we have the following implementation for pattern matching.

spec DC-PATTERN-MATCH-1($\langle \mathcal{C}, J \rangle$ :: FINITARY-NOETHERIAN-PM-SITE) =
extend DC-PATTERN-MATCH
axioms
—*primitive part of "dc-occs" does not change*
prim-obj(p) $\Rightarrow$ dc-occs$^t(p)$ = directly-solve$^t(p)$
prim-arr(f) $\Rightarrow$ dc-occs$^t(f)$ = directly-solve$^t(f)$
—*non-primitive part of "dc-occs" is modified to use cones*
$\neg$ prim-obj(p) $\Rightarrow$ dc-occs$^t(p) = \{ \text{glue}_P(\nu) \mid \nu \in \underset{\mathcal{C}/P}{\text{Cones}}(1, \text{dc-occs}^t)|_{\text{base}(P)} \}$
where P = decompose(p)
end

Examination of the occurrence sheaves in figures 3.1–3.3 and the cone examples 5.3–5.4 reveals that enumerating compatible families of partial occurrences, $\underset{\mathcal{C}/P}{\text{Cones}}(1, \text{occ-sheaf}^t)$, is equivalent to finding occurrences of the graph of the pattern cover (represented as a functor) in the graph of the occurrence functor. Thus, we have converted the pattern matching problem into a graph matching problem. This seems circular, and a retrograde step, since our intention is to derive a pattern matching algorithm. However, this characterization is helpful because of several reasons.

First, it reduces pattern matching on any FINITARY-NOETHERIAN-PM-SITE to pattern matching on graphs. To realize the significance of this, consider the fact that the underlying site can be arbitrarily complex.[6] This reduction illustrates the power of category theory as a language for parameterization.

Second, it allows us to use specific properties of graphs to design a general purpose pattern matching algorithm. For example, the distributive law in section 5.3.15 crucially depends on the fact that strict epimorphic families in the category of graphs are colimits (this is generally not true, see footnote on page 46).[7]

Third, the richness of the topology on graphs provides opportunities for decomposing covers in many different ways. Such decomposition opportunities may not exist in the underlying site. Here are two examples.

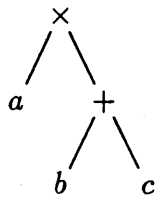
[6]This does not mean that the problem has been simplified. Graph matching is a hard problem: subgraph isomorphism is NP-complete.

[7]Strict epimorphic families are colimits in any topos. The category of graphs is a topos since it is equivalent to the functor category $\mathbf{Set}^{\rightrightarrows}$.

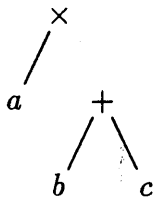
EXAMPLE 5.3. Consider the site of strings (example 2.22), in which the only non-identity covers are the finest covers. Hence, there are no intermediate decompositions possible. □

Lest the reader think the example above is a pathological case:

EXAMPLE 5.4. Some intermediate sieves between the identity cover and the finest cover are not covers in a tree. Consider the tree pattern



in the site **LTree**$_{\rightarrow}$ (see section 3.4). When the target is traversed depth-first, and occurrences are assembled bottom-up, it is possible to have a partial occurrence of the form



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 graphs. □

When such decomposition opportunities do not exist, a top-down complex-divide-simple-merge strategy (see section 5.3.5.Alt.1) becomes useless. This is the essential difference between the parallel algorithms generated by using the design alternative 5.3.5.Alt.1 and the alternative 5.3.13.Alt.1.

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

5.3.9 Reduction of pattern matching to graph matching

Before proceeding with the derivation, we slightly modify the divide-and-conquer implementation of pattern matching into an implementation solely in terms of cones. This formalizes the informal remarks in the previous section, and also simplifies the rest of the derivation.

In the divide-and-conquer implementation DC-PATTERN-MATCH-1, primitive objects and arrows are not implemented using cones. To fold the handling of primitive objects and arrows into cone computation, we use the identity cover on primitive objects. Thus, the partial function "decompose" can be extended into a total function. We have the following implementation by separating the primitive computations into a separate function:

spec DC-PATTERN-MATCH-2($\langle \mathcal{C}, J \rangle$ :: FINITARY-NOETHERIAN-PM-SITE) =
enrich DC-PATTERN-MATCH-1
operations
  decompose : $\mathrm{Obj}(\mathcal{C}) \to$ SIEVE($\mathcal{C}$)
  prim-occs : $\mathrm{Obj}(\mathcal{C}) \to \mathrm{Fun}(\mathcal{C}^{\mathrm{op}}, \mathbf{Set})$
axioms
  —*decomposing a primitive object yields the identity cover*
  $\text{prim-obj}(p) \Rightarrow \text{decompose}(p) = \{ f \mid \mathrm{cod}(f) = p \}$
  —"prim-occs" *is the restriction of* "dc-occs"
  —*to primitive objects and arrows*
  $\text{prim-obj}(p) \Rightarrow \text{prim-occs}^t(p) = \text{directly-solve}^t(p)$
  $\text{prim-arr}(f) \Rightarrow \text{prim-occs}^t(f) = \text{directly-solve}^t(f)$
  —*definition of* "dc-occs" *just in terms of cones*
  $\text{dc-occs}^t(p) = \{ \text{glue}_P(\nu) \mid \nu \in \underset{\mathcal{C}/P}{\text{Cones}}(1, \text{prim-occs}^t)|_{\text{base}(P)} \}$
    where $P = \text{decompose}(p)$
end

5.3.10 Piecewise assembly of cones

To compute cones over the diagram generated by the pattern cover P, we again use a divide-and-conquer algorithm. This time, however, we will use a decomposition of the diagram given by a colimit, rather than a cover. This is permissible since we will be working in the category of (small) diagrams; or, more precisely, the site generated by this category along with the standard topology given by inclusion arrows.[8]

[8]This category is similar to the category of graphs in example 2.19, except that the composition operation is absent for graphs. The essential structure is that of graphs; the composition operation is not used. Different books on category theory adopt one of the two approaches to diagrams: treat them as categories or treat them as graphs. Treating them as categories is simpler in the context of this dissertation.

DEFINITION 5.5: *The site of diagrams.* The category of small diagrams with inclusion arrows, denoted $\mathbf{Diag}_{\hookrightarrow}$, is the subcategory of **Cat** (the category of small categories) containing all the objects but only inclusion arrows. A sieve $\{ d_i \hookrightarrow d \mid i \in I \}$ is a covering sieve for the diagram d if $\bigcup_{i \in I} d_i = d$. □

We can define a well-founded order on diagrams by $d \succ d'$ if there is an inclusion $d' \hookrightarrow d$. This order is well-founded because a finest cover always exists for every diagram; it consists of the collection of objects and the collection of arrows in the diagram.

The existence of finest covers guarantees termination of a divide-and-conquer computation of cones on finite diagrams. The basis step of the divide-and-conquer algorithm is also easy. Cones on a diagram consisting of a single object are trivial. Cones on a diagram consisting of a single arrow are obtained by taking the image of the set function corresponding to the arrow.

Finally, to complete the divide-and-conquer algorithm, we need a proof of the strong problem reduction principle (i.e., structural induction for diagrams). Here is a theorem which states that cones on a diagram can be obtained by composing cones on sub-diagrams.

THEOREM 5.1: *Decomposition of cones.* Let $F: \mathcal{J} \to \mathcal{C}$ be a functor, with $\mathcal{J}$ a small "diagram" category. Let the category $\mathcal{J}$ be given by a union[9] over the diagram X:

$$\mathcal{J} = \bigcup_{x \in X} \mathcal{J}_x \qquad \text{with inclusions } i_x: \mathcal{J}_x \hookrightarrow \mathcal{J}.$$

For each $\mathcal{J}_x$, we have a restriction of F given by

$$F_x = F \circ i_x: \mathcal{J}_x \to \mathcal{C}.$$

and, for $c \in \mathrm{Obj}(\mathcal{C})$, a projection of cones $\pi_x: \underset{\mathcal{J}}{\mathrm{Cones}}(c, F) \to \underset{\mathcal{J}_x}{\mathrm{Cones}}(c, F)$ given by

$$\{\, c \xrightarrow{\nu_j} F(j) \mid j \in \mathcal{J} \,\} \mapsto \{\, c \xrightarrow{\nu_{i_x(j)}} F(i_x(j)) \mid j \in \mathcal{J}_x \,\}.$$

Consider a functor $G: X^{\mathrm{op}} \to \mathbf{Set}$ defined over the diagram X as follows (see figure 5.5). It assigns to each element of X a set of cones, and to each inclusion arrow a projection function on cones (note the change in direction).

$$\begin{aligned} x &\mapsto \underset{\mathcal{J}_x}{\mathrm{Cones}}(c, F_x) \\ x \hookrightarrow y &\mapsto \pi_{yx}: \underset{\mathcal{J}_y}{\mathrm{Cones}}(c, F_y) \to \underset{\mathcal{J}_x}{\mathrm{Cones}}(c, F_x) \end{aligned}$$

The cones on F can then be obtained as a limit of the cones on F_x:

$$\underset{\mathcal{J}}{\mathrm{Cones}}(c, F) = \varprojlim_{X} G = \varprojlim_{x \in X} \underset{\mathcal{J}_x}{\mathrm{Cones}}(c, F_x).$$

PROOF. We will show that every compatible family of cones on the diagrams $\mathcal{J}_x$

$$\{\, \nu_x \in \underset{\mathcal{J}_x}{\mathrm{Cones}}(c, F_x) \mid x \in X \,\}$$

can be uniquely extended to a cone $\nu \in \underset{\mathcal{J}}{\mathrm{Cones}}(c, F)$ on the diagram $\mathcal{J}$. Let the cone ν_x be given by the set

$$\nu_x = \{\, c \xrightarrow{\nu_{xj}} F(j) \mid j \in \mathcal{J}_x \,\}.$$

To construct the cone $\nu = \{\, c \xrightarrow{\nu_j} F(j) \mid j \in \mathcal{J} \,\}$ on the diagram $\mathcal{J}$, choose the arrow ν_j as $\nu_j = \nu_{xj}$ for any x such that $j \in \mathcal{J}_x$. Such a choice is possible because, by assumption, the chosen collection of cones on the diagrams $\mathcal{J}_x$ is compatible, and hence for any j such that $j \in \mathcal{J}_x \cap \mathcal{J}_y$, the arrows ν_{xj} and ν_{yj} must be equal. The projections π_x are defined by mapping ν_j to ν_{xj}.

The choice of the arrows in the cone ν and the definition of the projections π_x is clearly unique, and hence the collection of all cones of the form ν is a limit. □

[9]This is a simplifying assumption. The theorem is valid for colimits of diagrams.

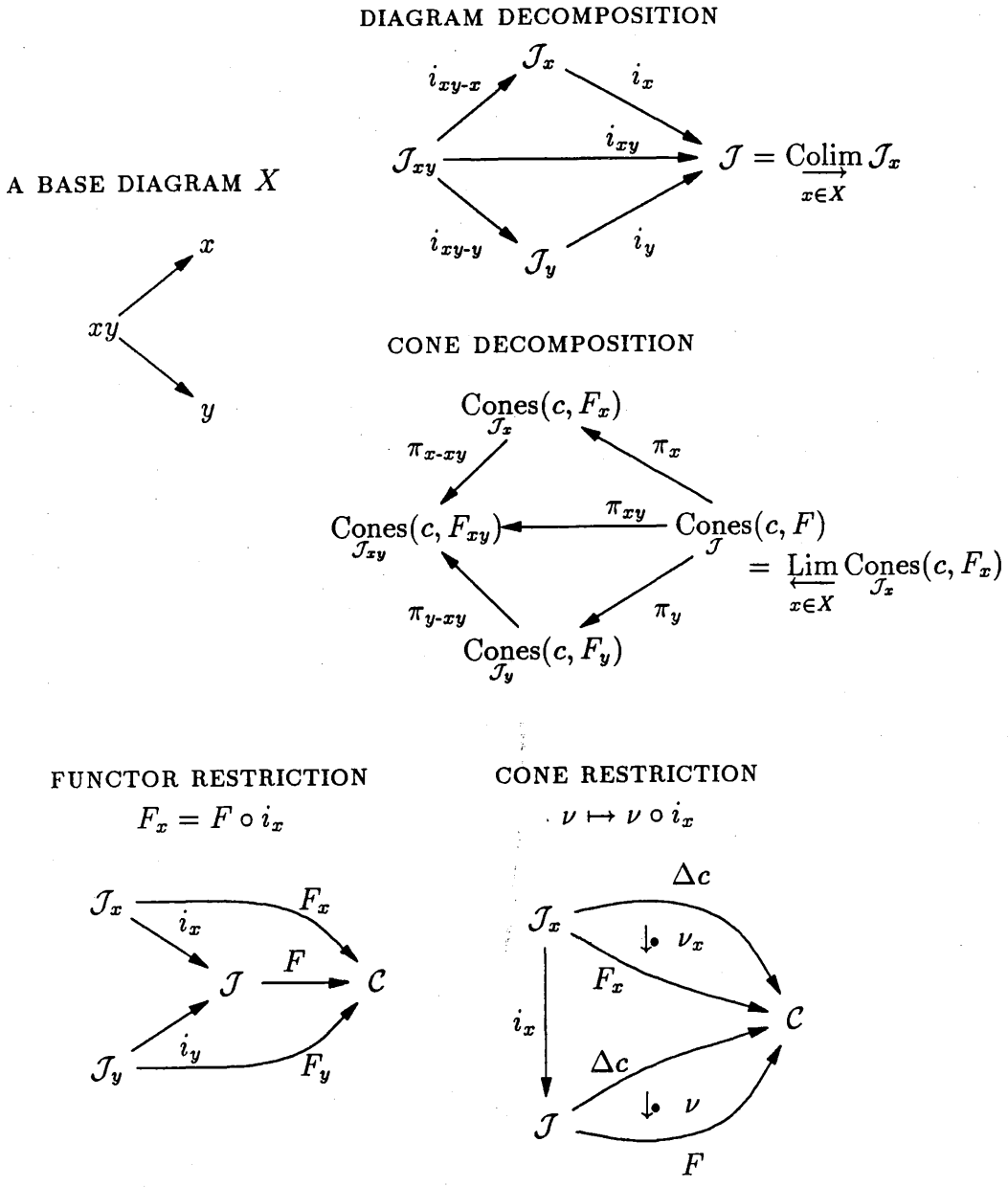


Figure 5.5: Diagrams and functors involved in Theorem 5.1

EXAMPLE 5.6: *Cone decomposition.* Figure 5.6 shows an example of cones on a diagram computed as a limit of cones on sub-diagrams. The sheaf is the occurrence sheaf of figure 3.1. The diagram $\mathcal{J}$ over which the cones are required is the graph of the pattern cover. This diagram is split up into two pieces $\mathcal{J}_x$ and $\mathcal{J}_y$ with a non-empty intersection $\mathcal{J}_{xy}$. The whole diagram can then be expressed as a pushout of the sub-diagrams.

Cones on each of the sub-diagrams are enumerated, along with the relevant projections. Cones on the diagram $\mathcal{J}$ are given by the limit of cones of $\mathcal{J}_x$, $\mathcal{J}_y$, and $\mathcal{J}_{xy}$; the limit, in this case, is a pullback. □

5.3.11 A functorial divide-and-conquer implementation of cones

In preparation for using cone decomposition in our derivation, we now translate Theorem 5.1 above into the language of algebraic specification. The cone decomposition theorem enables the implementation of the specification CONE-SPEC by the divide-and-conquer algorithm in the specification CONE-DECOMPOSITION.

In the specification CONE-SPEC below, we define cones functorially (cf. definitions 2.5 and 5.2), because the computation of cones via decomposition not only requires cones on a diagram but also restrictions of cones on one diagram to cones on another diagram.

spec CONE-SPEC =
extend DIAGRAM
operations
  cones : $c \to \mathrm{Fun}\big((\mathbf{Diag}{\downarrow}\mathcal{C})^{\mathrm{op}}, \mathbf{Set}\big)$ **for** $c \in \mathcal{C}, \mathcal{C} \in \mathbf{Cat}$
axioms
  $\mathrm{cones}(c, F) = \mathrm{Nat}(\Delta c, F)$
    where $F\colon \mathcal{J} \to \mathcal{C}$, $\Delta c\colon \mathcal{J} \to \mathcal{C}$
    and $\forall j \in \mathrm{Obj}(\mathcal{J}) \cdot \Delta c(j) = c$; $\forall f \in \mathrm{Arr}(\mathcal{J}) \cdot \Delta c(f) = \mathrm{id}_c$
  $\mathrm{cones}(c, \tau\colon F \dot{\to} G) = \_ \circ \tau$
end

Here is the specification which expresses cones on a diagram as a limit of cones on sub-diagrams. Note that the divide-and-conquer theory used here is more general than Smith's theory [Smith 85] (described in section 5.3.2). The specification below implements a *functor* rather than a *function.* Both objects and arrows are decomposed into colimit diagrams, and cones and their projections are computed as limits. The decomposition of arrows is induced by decompositions of their domains and codomains.

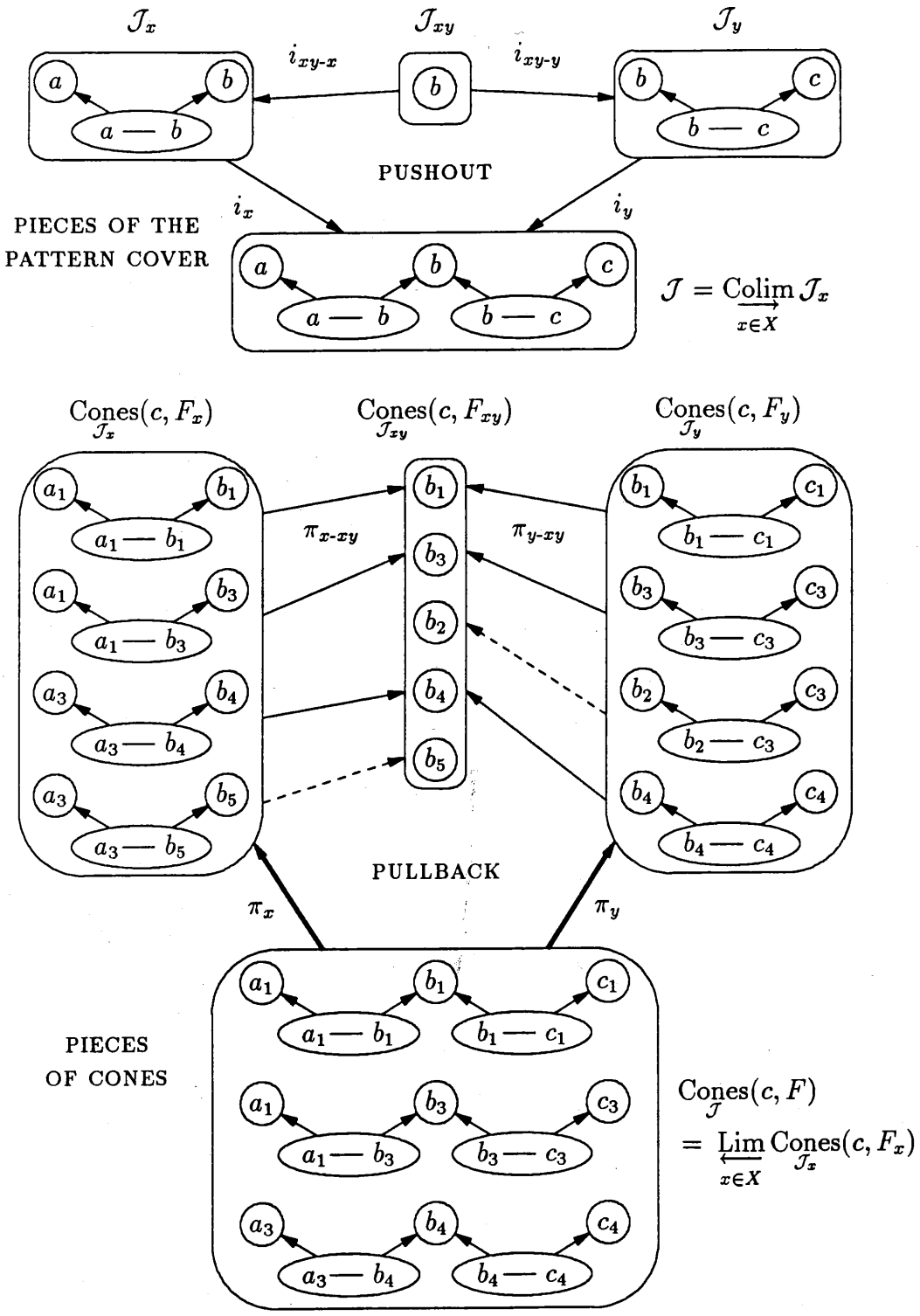


Figure 5.6: Example of cone decomposition (cf. figure 3.1)

spec CONE-DECOMPOSITION =
sorts P-OBJ, NP-OBJ, P-ARR, NP-ARR
—*primitive and non-primitive objects and arrows*
—*objects in* $\mathbf{Diag}{\downarrow}\mathcal{C}$ *are functors over diagrams*
—*arrows in* $\mathbf{Diag}{\downarrow}\mathcal{C}$ *are inclusions of functors*
subsorts P-OBJ $\prec$ Obj($\mathbf{Diag}{\downarrow}\mathcal{C}$), NP-OBJ $\prec$ Obj($\mathbf{Diag}{\downarrow}\mathcal{C}$),
P-ARR $\prec$ Arr($\mathbf{Diag}{\downarrow}\mathcal{C}$), NP-ARR $\prec$ Arr($\mathbf{Diag}{\downarrow}\mathcal{C}$)
operations
—*variable declarations for the signature below*
—$c \in \mathcal{C}$, $\mathcal{C} \in \mathbf{Cat}$, $X \in \mathbf{Diag}$
dc-cones : $c \to \mathrm{Fun}\big((\mathbf{Diag}{\downarrow}\mathcal{C})^{\mathrm{op}}, \mathbf{Set}\big)$
prim-obj : $\mathrm{Obj}(\mathbf{Diag}{\downarrow}\mathcal{C}) \to$ BOOL
prim-arr : $\mathrm{Arr}(\mathbf{Diag}{\downarrow}\mathcal{C}) \to$ BOOL
directly-solve : c, P-OBJ $\to \mathrm{Obj}(\mathbf{Set})$
directly-solve : c, P-ARR $\to \mathrm{Arr}(\mathbf{Set})$ —*overloaded*
decompose : NP-OBJ $\to \mathrm{Fun}(X, \mathbf{Diag}{\downarrow}\mathcal{C})$
compose : $\mathrm{Fun}(X, \mathbf{Set}^{\mathrm{op}}) \to \mathrm{Obj}(\mathbf{Set})$
axioms
—*primitive and non-primitive objects are disjoint*
$\chi_{\text{P-OBJ}} = \text{prim-obj}$ $\chi_{\text{NP-OBJ}} = \neg\,\text{prim-obj}$
—*primitive and non-primitive arrows are disjoint*
$\chi_{\text{P-ARR}} = \text{prim-arr}$ $\chi_{\text{NP-ARR}} = \neg\,\text{prim-arr}$
—*details of primitive objects will be delayed*
—*arrows are primitive if their domain and codomain are primitive*
$\text{prim-arr}(p \xrightarrow{f} q)$ **iff** $\text{prim-obj}(p) \wedge \text{prim-obj}(q)$
—*the decomposition operator produces a colimit diagram*
$F = \underrightarrow{\mathrm{Colim}}\ \text{decompose}(F)$
—*the composition operator is just a limit*
$\text{compose}(C) = \underleftarrow{\mathrm{Lim}}\ C$
—*details of* "directly-solve" *will be delayed*
—*recursive definition of* "dc-cones"
$\text{prim-obj}(F) \Rightarrow \text{dc-cones}(F) = \text{directly-solve}(F)$
$\text{prim-arr}(\tau) \Rightarrow \text{dc-cones}(\tau) = \text{directly-solve}(\tau)$
—*computation of cones as a limit; valid by Theorem 5.1*
$\neg\,\text{prim-obj}(F) \Rightarrow \text{dc-cones}(c, F) = \text{compose} \circ \text{dc-cones}(c, -) \circ \text{decompose}(F)$
—*computation of projections of cones*
$\neg\,\text{prim-arr}(\tau\colon F \to G) \Rightarrow \text{dc-cones}(\tau) = \underleftarrow{\mathrm{Lim}}_{y \in \tau^*(X)}\ \pi_y$
where $G = \underrightarrow{\mathrm{Colim}}_{x \in X}\, G_x$ **and** $i_x\colon G_x \to G$, $\pi_x = \text{dc-cones}(i_x)$
end

For the case of pattern matching, comparing the specification CONE-DECOMPOSITION and the expression $\underset{\mathcal{C}/P}{\text{Cones}}(1, \text{prim-occs}^t\,|_{\text{base}(P)})$ in the specification DC-PATTERN-MATCH-2, we get the following correspondence:

$\mathcal{J} \mapsto \mathcal{C}/P$
$F \mapsto \text{prim-occs}^t\,|_{\text{base}(P)}$
$\mathcal{C} \mapsto \textbf{Set}$
$c \mapsto 1$

and the following implementation of the specification DC-PATTERN-MATCH-2:

spec DC-PATTERN-MATCH-3($\langle \mathcal{C}, J \rangle$ :: FINITARY-NOETHERIAN-PM-SITE) =
extend DC-PATTERN-MATCH-2($\langle \mathcal{C}, J \rangle$), CONE-DECOMPOSITION
axioms
—*cones are computed by decomposition*
$\text{dc-occs}^t(p) = \{\, \text{glue}_P(\nu) \mid \nu \in \text{dc-cones}(1, \text{prim-occs}^t\,|_{\text{base}(P)}) \,\}$
where $P = \text{decompose}(p)$
end

5.3.11.Alt.1 Direct computation of cones

Cones can also be computed using data-flow techniques on the extension of the sheaf. Assign one process to each element of the sheaf. Processes communicate with each other by sending tokens via the arrows. When two processes connect to the same intermediate process (compatible on the intersection), they can be coalesced into one. The final processes which remains after iterating this represent the compatible families.

5.3.12 Cones on indecomposable diagrams

In the specification CONE-DECOMPOSITION above, we left the handling of primitive diagrams unspecified. A diagram is primitive if it consists of only one object or only one arrow. The computation of cones for such diagrams is shown below.

spec CONE-DECOMPOSITION-1 =
extend CONE-DECOMPOSITION
axioms
—*a primitive diagram consists of a single object or a single arrow*
prim-obj($F: \mathcal{J} \to \mathcal{C}$) **if** $\mathcal{J} \cong \bullet \vee \mathcal{J} \cong (\star \to \bullet)$
—*a cone on a single object is just a hom-set*
directly-solve($c, \bullet \xrightarrow{F} \mathcal{C}$) = $\hom_{\mathcal{C}}(c, F(\bullet))$
—*a cone on a single arrow is the graph of a map between hom-sets*
directly-solve($c, (\star \xrightarrow{f} \bullet) \xrightarrow{F} \mathcal{C}$) =
$\{ \langle x, y \rangle \in \hom_{\mathcal{C}}(c, F(\star)) \times \hom_{\mathcal{C}}(c, F(\bullet)) \mid y = F(f) \circ x \}$
—*projections for primitive arrows can be computed similarly*
end

5.3.13 Choice of decomposition for cones

On examining the specification CONE-DECOMPOSITION, we can notice that the composition operator is a limit, and hence may not be easy to calculate. To simplify this operator, we can assume that the pattern cover is always decomposed into two pieces. Thus the pattern cover is expressed as a pushout diagram, and the limit in the composition operator is reduced to a pullback. For an example, see figure 5.6. The choice of the two pieces into which the pattern is decomposed is dictated by the rest of the algorithm. The incremental algorithm discussed in section 5.3.18 below forces a particular choice: a big piece for the partial match, and a small piece for the increment.

5.3.13.Alt.1 Pyramid algorithms

If we choose a parallel algorithm, then a binary decomposition of the pattern (with the two pieces being roughly of the same size) minimizes the number of levels in the decomposition tree. This choice leads to the pyramid algorithms which are used in image processing [Rosenfeld 84].

5.3.14 Incremental computation

Consider the expression

$$\text{dc-occs}^t(p) = \{ \text{glue}_P(\nu) \mid \nu \in \text{dc-cones}(1, \text{prim-occs}^t|_{\text{base}(P)}) \}$$

in the specification DC-PATTERN-MATCH-3. There are essentially two strategies for computing this expression. We can proceed from right to left: computing primitive occurrences, and then cones, and then gluing them; this leads to the pyramid-style algorithms of section 5.3.13.Alt.1. We can also interleave the computation of the sub-expressions; this leads to the sequential algorithms considered below. In the case

of pattern matching, the population of the sheaf and the computation of cones can be interleaved. To justify the interleaving, we need a distributive law.

5.3.15 A distributive law for incremental computation

To interleave the population of the sheaf and the computation of cones, we will use a technique called finite differencing [Paige and Koenig 82, Paige 81]. Finite differencing is an optimizing transformation which efficiently updates the value of a function when the parameter to the function is changed, rather than recomputing the value of the function for the new parameter. Here are two simple examples of this technique.

EXAMPLE 5.7: *Multiplication by shift-and-add.* Let m and n be two decimal numbers, and suppose that their product $p = m \times n$ has already been computed. If we now change the multiplier by adding another digit,

$$n \times 10 + d,$$

the product can be updated by computing

$$p \times 10 + p \times d.$$

Defining a binary operation $\_ \bullet \_$ by

$$a \bullet b \stackrel{\text{def}}{=} a \times 10 + b,$$

the incremental update of the product can be described the following distributive law:

$$m \times (n \bullet d) = (m \times n) \bullet (m \times d).$$

□

EXAMPLE 5.8: *Generate and test.* A common computation where finite differencing can be profitably applied is when a set is built by filtering elements from another set. For example, we can enumerate all the odd multiples of a number n by filtering the odd numbers from the set of all multiples of n:

$$\text{odd-multiples}(n) = \text{filter-odd}(\text{multiples}(n)).$$

Suppose the multiples of n are generated incrementally. Then the set of odd multiples can be efficiently updated by using the following distributive law:

$$\text{filter-odd}(s \cup \{m\}) = \text{filter-odd}(s) \cup \text{filter-odd}(\{m\}).$$

□

As can be seen from the examples above, finite differencing works in the presence of a distributive law. Hence, we will try to formulate a distributive law to connect the population of the sheaf and the computation of cones. The two relevant operations, in the two dimensions in which a sheaf spreads out, are cone-combination and set-union (figure 5.7).

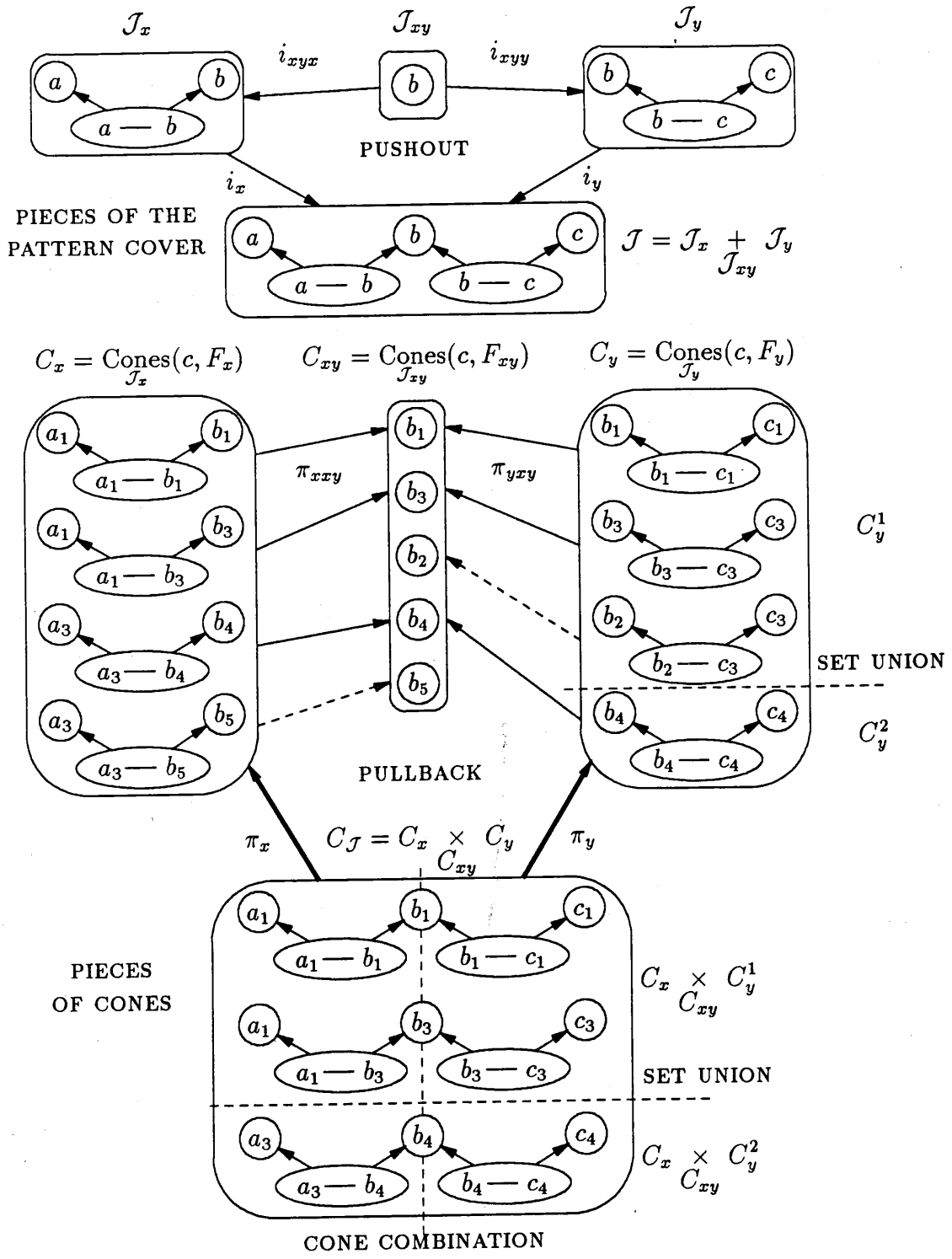


Figure 5.7: Distributive law for assembling cones (cf. figure 3.1)

Formally, the distributive law for computing cones over a diagram $\mathcal{J}$ can be expressed as below. Let the diagram $\mathcal{J}$ be built from two pieces, i.e., as the following pushout:[10]

$$\mathcal{J} = \mathcal{J}_x \underset{\mathcal{J}_{xy}}{+} \mathcal{J}_y.$$

$$\begin{array}{ccc} \mathcal{J}_{xy} & \longrightarrow & \mathcal{J}_y \\ \downarrow & \text{pushout} & \downarrow \\ \mathcal{J}_x & \longrightarrow & \mathcal{J} \end{array}$$

For any part $\mathcal{J}_x$ of the diagram, and a sheaf F, define the notation $C_x \stackrel{\text{def}}{=} \underset{\mathcal{J}_x}{\text{Cones}}(1, F)$. By theorem 5.1, the cones on P are given by the following pullback:

$$C_{\mathcal{J}} = C_x \underset{C_{xy}}{\times} C_y.$$

$$\begin{array}{ccc} C_{\mathcal{J}} & \longrightarrow & C_y \\ \downarrow & \text{pullback} & \downarrow \\ C_x & \longrightarrow & C_{xy} \end{array}$$

Assuming that the sheaf is populated incrementally, let the cones on $\mathcal{J}_y$ be given by the following set-union:

$$C_y = C_y^1 \cup C_y^2.$$

The distributive law which relates cone-combination and set-union is:

$$C_x \underset{C_{xy}}{\times} (C_y^1 \cup C_y^2) = (C_x \underset{C_{xy}}{\times} C_y^1) \cup (C_x \underset{C_{xy}}{\times} C_y^2),$$

$$\begin{array}{ccc} C_{\mathcal{J}}^1 & \longrightarrow & C_y^1 \\ \downarrow & \text{pullback} & \downarrow \\ C_x & \longrightarrow & C_{xy} \end{array} \qquad \begin{array}{ccc} C_{\mathcal{J}}^2 & \longrightarrow & C_y^2 \\ \downarrow & \text{pullback} & \downarrow \\ C_x & \longrightarrow & C_{xy} \end{array}$$

$$\begin{array}{ccc} C_{\mathcal{J}}^1 \cup C_{\mathcal{J}}^2 & \longrightarrow & C_y^1 \cup C_y^2 \\ \downarrow & \text{pullback} & \downarrow \\ C_x & \longrightarrow & C_{xy} \end{array}$$

where

$$C_{\mathcal{J}}^1 = (C_x \underset{C_{xy}}{\times} C_y^1) \quad \text{and} \quad C_{\mathcal{J}}^2 = (C_x \underset{C_{xy}}{\times} C_y^2).$$

For this distributive law to hold, we need the following theorem.

[10]The distributive law is true even if there are more than two pieces, i.e., the pattern cover is expressed a colimit (see section 5.3.19). However, for our purposes, pushouts are sufficient.

THEOREM 5.2: *Distributive law for cones.* In the category **Set**, coproducts commute with pullbacks, i.e., the pullback of a coproduct diagram is also a coproduct diagram.

PROOF. The distributive law follows from the fact that the category **Set** is a topos, since, in any topos colimits are universal (i.e., the pullback of a colimit diagram yields a colimit diagram) [Goldblatt 84, Freyd 72, Kock and Wraith 71].
However, it is more intuitive to do a direct calculation in the category **Set**. Let

$$B = a \underset{d}{\times} b \quad \text{and} \quad C = a \underset{d}{\times} c$$

$$\begin{array}{ccc} B & \longrightarrow & b \\ {\scriptstyle \pi_1^B}\downarrow & \text{pullback} & \downarrow{\scriptstyle p} \\ a & \xrightarrow[f]{} & d \end{array} \qquad \begin{array}{ccc} C & \longrightarrow & c \\ {\scriptstyle \pi_1^C}\downarrow & \text{pullback} & \downarrow{\scriptstyle q} \\ a & \xrightarrow[f]{} & d \end{array}$$

be the pullbacks along the arrow $a \xrightarrow{f} d$ of the two arrows $b \xrightarrow{p} d$ and $c \xrightarrow{q} d$.
We have to show that $a \underset{d}{\times} (b + c) = a \underset{d}{\times} b + a \underset{d}{\times} c$.
Pullbacks in **Set** are calculated as subsets of products. Thus,

$$B = a \underset{d}{\times} b = \{\, \langle x, y \rangle \in a \times b \mid f(x) = p(y) \,\}, \text{ and}$$

$$C = a \underset{d}{\times} c = \{\, \langle x, y \rangle \in a \times c \mid f(x) = q(y) \,\},$$

with the arrows into a given by the projections $(a \underset{d}{\times} b) \xrightarrow{\pi_1^B} a$ and $(a \underset{d}{\times} c) \xrightarrow{\pi_1^C} a$, both defined as $\langle x, y \rangle \mapsto x$. The coproduct of these two pullbacks is given by ("$\coprod$" is disjoint union, "/" is the quotient operator)

$$a \underset{d}{\times} b + a \underset{d}{\times} c = B \coprod C, \tag{1}$$

with projection $[\pi_1^B, \pi_1^C]$ into a. The coproduct of $b \xrightarrow{p} d$ and $c \xrightarrow{q} d$ is given by

$$b + c = b \coprod c \xrightarrow{[p,q]} d.$$

The pullback of $b + c$ along $a \xrightarrow{f} d$ is then given by

$$a \underset{d}{\times} (b + c) = \{\, \langle x, y \rangle \in a \times (b + c) \mid f(x) = [p, q](y) \,\} \tag{2}$$

$$\begin{array}{ccc} B + C & \longrightarrow & b + c \\ {\scriptstyle [\pi_1^B, \pi_1^C]}\downarrow & \text{pullback} & \downarrow{\scriptstyle [p,q]} \\ a & \xrightarrow[f]{} & d \end{array}$$

It is evident that the expressions (1) and (2) are equal, and the hence the distributive law is true in the category **Set**. □

Remarks.

1. Coproducts in **Set** are disjoint unions. In the informal description of the distributive law above, we have used union. The two are equivalent for pattern matching, because only new partial occurrences will be added to the sheaf.

2. We have assumed that when a new partial match is added, C_{xy} remains the same. This need not be true. However, new values added to C_{xy} cannot produce new cones. Hence, the pullbacks taken over the old C_{xy} and the new C_{xy} are the same.

5.3.16 A theory of incremental computation

Finite differencing is an optimization transformation; not an implementation strategy. We now combine finite differencing with recursion removal to provide an implementation scheme for functions whose arguments are incrementally provided as a stream of inputs. Just as a divide-and-conquer implementation requires the strong problem reduction principle, so does incremental implementation require a distributive law and an associative law.

The source specification consists of a domain equipped with an associative binary operation, and an arbitrary function f which satisfies the distributive law.

spec FD-SPEC =
sorts D, R
operations
f : D → R
0_R : → R
$\_ +_D \_$: D, D → D
$\_ +_R \_$: R, R → R
axioms
$r +_R 0_R = r$ —*right identity for* $+_R$
$(x +_D y) +_D z = x +_D (y +_D z)$ —*the associative law*
$f(x +_D y) = f(x) +_R f(y)$ —*the distributive law*
end

The function f is implemented by decomposing its input into a stream of increments, and feeding these increments one-by-one to an accumulating function g which exploits the distributive law to compute the value of f.

spec STREAM-SPEC(X :: ANY) =
sorts STREAM, NON-EMPTY-STREAM
subsorts NON-EMPTY-STREAM $\prec$ STREAM
constraints —*initial model*
operations
hd : NON-EMPTY-STREAM $\rightarrow$ ANY
tl : NON-EMPTY-STREAM $\rightarrow$ STREAM
nil : $\rightarrow$ STREAM
cons : ANY, STREAM $\rightarrow$ NON-EMPTY-STREAM
concat : STREAM, STREAM $\rightarrow$ STREAM
$/\_$: (ANY, ANY $\rightarrow$ R) $\rightarrow$ (STREAM $\rightarrow$ R)
axioms
STREAM = NON-EMPTY-STREAM $\coprod$ {nil}
hd(cons(x, s)) = x
tl(cons(x, s)) = s
cons(hd(s), tl(s)) = s
—*concatenation of streams*
concat(nil, t) = t
concat(cons(x, s), t) = cons(x, concat(s, t))
—*the "reduce" operator; it inserts a binary operator between*
—*the elements of a stream and evaluates the resulting expression*
$/f$(nil) = id_f
$/f$(cons(x, s)) = $f(x, /f(s))$
end

spec FD =
sorts D, R
include STREAM-SPEC(D)
operations
g : R, STREAM(D) $\rightarrow$ R
f : D $\rightarrow$ R
$\_ +_R \_$: R, R $\rightarrow$ R
axioms
—p *is the current partial result*
—*the second argument is the rest of the stream*
$g(p,$ nil$) = p$
$g(p,$ cons$(x, s)) = g(p +_R f(x), s)$
end

The constructor implementation below postulates a decomposition operator called "stream"[11], and invokes the iterative function g to implement f.

[11]The name "stream" should be interpreted as a verb, in the sense of "convert into a stream."

constructor FD-IMP =
derive from
  extend FD + FD-SPEC **with**
  operations
    stream : D $\rightarrow$ STREAM(D)
  axioms
    $f(x) = g(0_R, \text{stream}(x))$
    $/+_D(\text{stream}(x)) = x$
by σ: FD-SPEC $\hookrightarrow$ (FD + FD-SPEC)
end

The decomposition operator "stream" is left unspecified above. The choice depends on how the function f is specified. For example, if f is specified by a recursive description as in the specification DIVIDE-AND-CONQUER, then a good strategy is to design the decomposition operator such that it produces a stream of only primitive elements. This reduces the computation of f in the distributive law to "directly-solve," and hence converts the recursive definition of f into an iterative implementation.

5.3.17 Incremental population of the occurrence sheaf

Considering the specification DC-PATTERN-MATCH-3 again, we see that for an incremental computation of "dc-occs$^t(p)$," the stream of inputs should come from "prim-occs$^t(p_i)$," which populates the occurrence sheaf over the base of a cover P of p. Rather than execute "prim-occs$^t(p_i)$" for each piece $p_i \in \text{base}(P)$ of the pattern cover, we can "invert" the computation. We assume that the target is traversed just once, populating the sheaf at each p_i, and thus generating a stream of presheaves

$$O^{t_1} \subseteq O^{t_2} \subseteq O^{t_3} \subseteq \cdots \subseteq O^t,$$

where t_i is the part of the target traversed at stage i.

By traversing the target, we encounter primitive objects in the underlying site. However, the goal is to populate the sheaf, whose objects are occurrence arrows. To pass from objects to occurrence arrows, we need a definition of the occurrence relation, which, for our derivation is not completely specified (it is a parameter). The computation of occurrence arrows depends on the data structures involved and the specific form of the occurrence relation. We show below a few examples to illustrate that this computation can vary from simple enumeration to complex theorem-proving. We will not further consider this aspect because it is a parameter; the characterization above as a sequence of presheaves is sufficient.

EXAMPLE 5.9: *Strings, constant patterns.* Simple enumeration of inclusions. Each piece of the target matches at most one piece of the pattern. □

EXAMPLE 5.10: *Strings, patterns with variables.* Each piece of the target may match more than one piece of the pattern. □

EXAMPLE 5.11: *Strings, non-standard morphisms.* See example 2.23 for a definition of this site. The processing of each increment to the target needs the updating of all the elements of part of the sheaf populated until then. □

EXAMPLE 5.12: *A complex occurrence relation.* If the occurrence relation involves semantic conditions, theorem-proving may be required to compute occurrence arrows. □

5.3.18 Incremental assembly of occurrences

Using the theory of incremental computation presented in section 5.3.16 above, we now convert the recursive algorithm of DC-PATTERN-MATCH-3 into an iterative algorithm. For convenience, we repeat the specification DC-PATTERN-MATCH-3 here.

spec DC-PATTERN-MATCH-3($\langle \mathcal{C}, J \rangle$ :: FINITARY-NOETHERIAN-PM-SITE) =
extend DC-PATTERN-MATCH-2($\langle \mathcal{C}, J \rangle$), CONE-DECOMPOSITION
axioms
—*cones are computed by decomposition*
$\text{dc-occs}^t(p) = \{ \text{glue}_P(\nu) \mid \nu \in \text{dc-cones}(1, \text{prim-occs}^t|_{\text{base}(P)}) \}$
where $P = \text{decompose}(p)$
end

For the piecewise assembly of cones in "dc-cones", we have to choose a decomposition of the diagram corresponding to the cover P of the pattern. We can choose the finest decomposition (consisting of single objects and single arrows) as in the specification CONE-DECOMPOSITION-1. However, such a decomposition is too fine-grained. We instead adjust the decomposition so that it meshes with the incremental population of the occurrence sheaf.

Before proceeding, we need some definitions.

DEFINITION 5.13: *Prime arrow and prime sieve.* Given a sieve S, an arrow $f \in S$ is said to be prime (in S) if it is not a factor of any other arrow in the sieve, i.e., $\nexists g, h \in S \cdot h = g \circ f$. A sub-sieve $R \subseteq S$ is said to be prime (in S) if it is generated by a prime arrow $f \in S$. □

DEFINITION 5.14: *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'. □

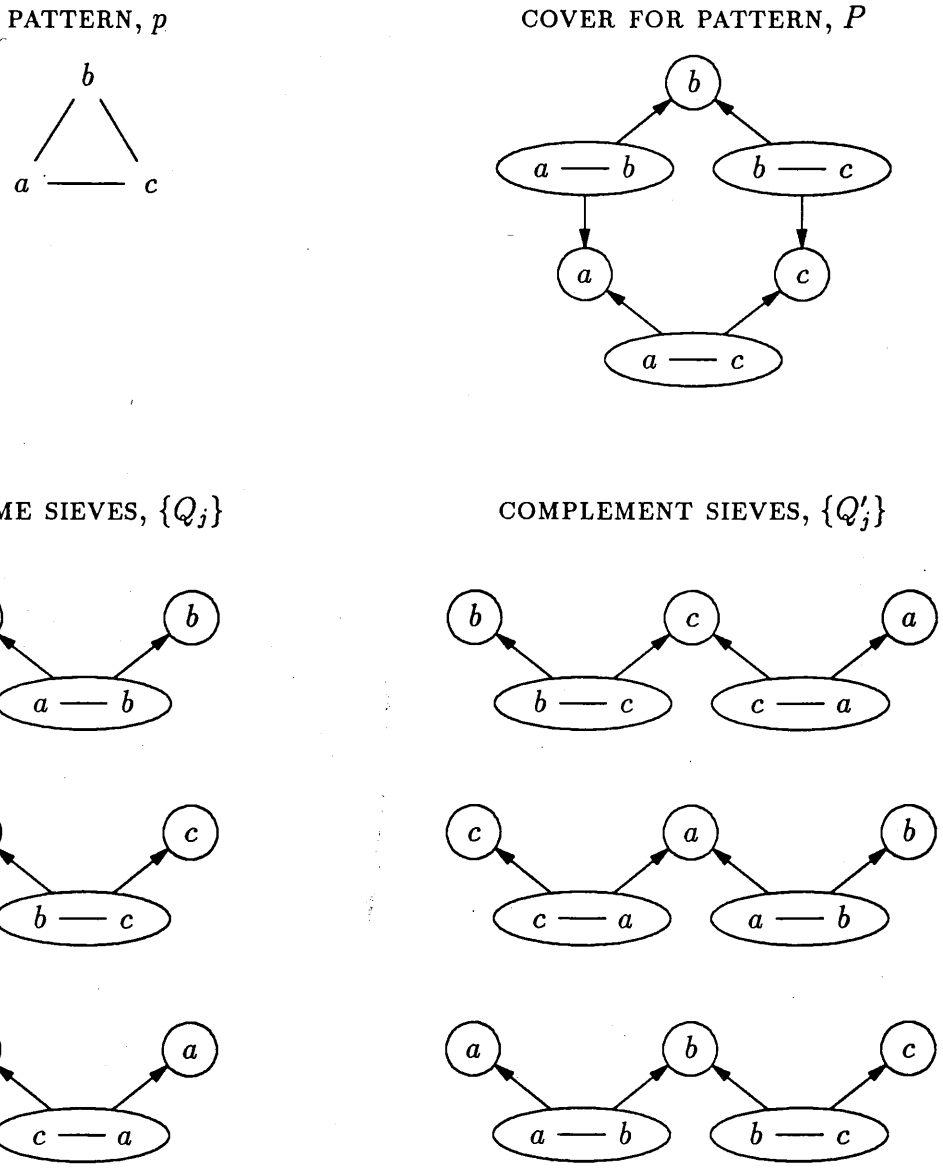


Figure 5.8: Prime sieves and complements: example with graphs (cf. figure 3.2)

Now, let P be the cover chosen by the divide-and-conquer algorithm in the specification DC-PATTERN-MATCH-3. Let $\{q_j \rightarrow p\}$ be the set of prime arrows in P and let $\{Q_j\}$ be the (prime) sieves generated by these arrows. It can be seen that whenever "prim-occs" finds an occurrence of q_j, it can also find, by restriction, occurrences of all elements of the sieve Q_j (since, if q_j is primitive, then all objects and arrows in Q_j must also be primitive). By inverting the computation done by "prim-occs," as discussed in section 5.3.17 above, we postulate an operation which traverses the target and produces increments to the occurrence sheaf over prime sieves:

$$\text{traverse: } \mathrm{Obj}(\mathcal{C}) \rightarrow \textsc{stream}(\mathrm{Fun}(\mathcal{C}/Q_j, \mathbf{Set}))$$
$$\textbf{for } Q_j \in \text{prime-sieves}(P), P \in J(p), p \in \mathrm{Obj}(\mathcal{C})$$

As discussed in section 5.3.16, to convert "dc-cones" into an iterative algorithm, we must choose the decomposition of cones such that one of the pieces is on $\mathcal{C}/Q_j$, the diagram over which increments are generated by the traversal operation above. Since the decomposition is binary (so as to reduce the limit to a pullback), the other piece of the cone must be on $\mathcal{C}/(P{\sim}Q_j) = \mathcal{C}/Q'_j$. Then the pattern cover can be expressed by the following pushout:

$$\begin{array}{ccc} \mathcal{C}/Q_j \cap \mathcal{C}/Q'_j & \longrightarrow & \mathcal{C}/Q'_j \\ \downarrow & \text{pushout} & \downarrow \\ \mathcal{C}/Q_j & \longrightarrow & \mathcal{C}/P \end{array}$$

Considering that the traversal of the target can produce increments over any piece $\mathcal{C}/Q_j$ of the pattern cover, it follows that cones on $\mathcal{C}/Q'_j$, for each j must be available on each cycle of the incremental computation. To keep cones on all $\mathcal{C}/Q'_j$'s available in the face of increments on any $\mathcal{C}/Q_k$, cones on $\mathcal{C}/(Q'_j{\sim}Q_k)$ should also be available. Continuing this process, and using the following identities about prime sieves,

$$P = \bigcup Q_j, \quad \text{where } Q_j \text{ is a prime sieve in } P, \text{ and}$$
$$Q'_j = \bigcup_{j \neq k} Q_k,$$

it follows that cones on any combination of $\mathcal{C}/Q_j$'s must be available on each cycle. Let us define

$$\text{subpatterns}(P) = \{\, \Pi_k = \bigcup_{x=1..n} \mathcal{C}/Q_{k_x} \mid Q_{k_x} \in \text{prime-sieves}(P) \,\},$$

where the notation Π_k is chosen to remind the reader that a cone on Π_k corresponds to a partial occurrence. Apart from cones on the Π_k's, cones on the intersection of any two Π_k's must also be available, so that the pushout diagram shown above can be built.

The informal reasoning presented here can be formalized by choosing an initial decomposition of the diagram $\mathcal{C}/P$ into the pair $\langle \mathcal{C}/Q_j, \mathcal{C}/Q'_j \rangle$ and then "differentiating" the computation of "dc-cones" with respect to the increments that are produced by the traversal of the target. For details about differentiation, see [Paige and Koenig 82, Paige 81]. We omit the formalization here and present the final iterative algorithm resulting from the particular decomposition we have chosen.

As discussed above, the iterative algorithm has to update the cones on each Π_k and each $\Pi_k \cap \Pi_l$ on each cycle. Thus we need one instance of the iterative algorithm in FD-SPEC for each of these. However, since all these algorithms feed off the same input stream (that produced by traversal of the target), they can be merged into a single algorithm as shown below. Again, we omit the details of this transformation.

Observe the use of the distributive law of Theorem 5.2 in the specification below.

spec FD-CONES($c :: \text{Obj}(\mathcal{C}), P :: \text{SIEVE}(\mathcal{C})$) =
—*incremental computation of cones*
sorts CACHE, INCREMENT
—*a cache associates a set of cones with each* $\Pi_k \in \text{subpatterns}(P)$
—*a cache is a functor because it contains projections of cones as well*
subsorts CACHE $\prec$ Fun($\mathbf{Diag}^{\text{op}}, \mathbf{Set}$)
—*increments are functors defined over diagrams of prime sieves*
subsorts INCREMENT $\prec$ Fun($\mathcal{C}/Q_j, \mathbf{Set}$) **for** $Q_j \in \text{prime-sieves}(P)$
operations
fd-cones : STREAM(INCREMENT) $\rightarrow$ SET(CONE)
update : CACHE, STREAM(INCREMENT) $\rightarrow$ CACHE
prim-cones : INCREMENT $\rightarrow$ SET(CONE)
axioms
—*start iterative algorithm with empty cache; return cones on* $\mathcal{C}/P$
fd-cones(s) = final-cache($\mathcal{C}/P$)
where final-cache = update($\emptyset, s$)
—*incremental update of the cache*
—*when the increment stream is exhausted, return the current cache*
update(cache, nil) = cache
—*process one element of the stream at a time*
update(cache, cons($\mathcal{C}/Q_j \xrightarrow{F} \mathbf{Set}, s$)) = update(cache$'$, s)
where cache$'$ = cache **except**
—*update cones on* $\mathcal{C}/Q_j$
cache$'$($\mathcal{C}/Q_j$) = cache($\mathcal{C}/Q_j$) $\cup$ prim-cones(F) **and**
—*update all cones which intersect with cones on* $\mathcal{C}/Q_j$
—*using the distributive law for cones; Theorem 5.2*
$\forall \Pi_k \in \text{subpatterns}(P) \cdot \mathcal{C}/Q_j \subseteq \Pi_k \Rightarrow$
cache$'$(Π_k) = cache(Π_k) $\cup$ cache($\Pi_k{\sim}\mathcal{C}/Q_j$) $\underset{\text{cache}(\Pi_\ell)}{\times}$ prim-cones(F)
where $\Pi_\ell = (\Pi_k{\sim}\mathcal{C}/Q_j) \cap \mathcal{C}/Q_j)$
—*the pullback above also updates projections of cones*
end

The expression "dc-cones(1, prim-occs^t $|_{\text{base}(P)}$) }" in the specification DC-PATTERN-MATCH-3 is implemented by "fd-cones(s)" with the following assignments:

$$c \mapsto 1$$
$$P \mapsto P$$
$$s \mapsto \text{traverse}(t)$$
$$\text{prim-cones}(F{:}\mathcal{C}/Q_j \rightarrow \mathbf{Set}) \mapsto \text{dc-cones}(1, \text{prim-occs}^t\,|_{\mathcal{C}/Q_j})$$

We consolidate the work of the past few sections into the incremental implementation of pattern matching below. Since the specification FD-CONES is parameterized,

the operations "fd-cones" and "prim-cones" are given subscripts to indicate the instance of FD-CONES to which they belong.

spec FD-PATTERN-MATCH($\langle \mathcal{C}, J \rangle$ :: FINITARY-NOETHERIAN-PM-SITE) =
extend FD-CONES —*see above for explanation of subscript notation*
extend CONE-DECOMPOSITION-1 **with**
decompose **renamed as** decompose . CONE
operations
fd-occs : Obj($\mathcal{C}$), Obj($\mathcal{C}$) → SET(OCCURRENCE)
traverse : Obj($\mathcal{C}$) → STREAM(INCREMENT)
decompose : Obj($\mathcal{C}$) → SIEVE($\mathcal{C}$)
axioms
fd-occs$(p, t) = \alpha(\text{glue}_P) \circ \text{fd-cones}_{1,P}(\text{traverse}(t))$
where $P = \text{decompose}(p)$
—*decomposition yields the finest covering sieve*
$\text{decompose}(p) \in J(p) \wedge \forall R \in J(p) \cdot \text{decompose}(p) \subseteq R$
—*computation of cones on $\mathcal{C}/Q_j$ is done by* "dc-cones"
$\text{prim-cones}_{c,P}(F) = \text{dc-cones}(c, F)$
—*the traversal operation for the target is left unspecified*
end

Here is the constructor which connects the specification PATTERN-MATCH of pattern matching with the incremental implementation above.

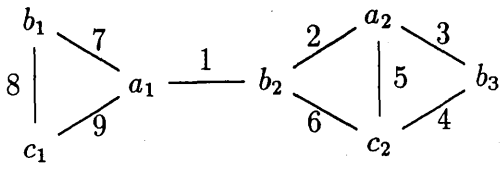
$$\text{PATTERN-MATCH} \rightsquigarrow \kappa\text{-FD}(\text{FD-PATTERN-MATCH})$$

where

constructor κ-FD =
derive from FD-PATTERN-MATCH
by { occurrences ↦ fd-occs }
end

To help the reader understand the operation of the incremental algorithm, we show a trace of this algorithm for graphs in figure 5.9.

Traversal order:



Partial occurrences (increments to the cache of cones):

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

Figure 5.9: Trace of incremental algorithm on graphs (cf. figure 3.2)

5.3.19 Sheafification: An alternative view of incremental cone assembly

The incremental algorithm of the previous section can be succinctly characterized as sheafification. The operation of computing cones over every sub-cover of the pattern cover corresponds to converting a presheaf of occurrences into a sheaf.

Sheafification of a presheaf F is defined by a universal property (the "minimal completion" of a presheaf into a sheaf by adding all cones which are absent), and is constructed as

$$\text{sheafify}(F) = (L \circ L)(F),$$

where L is a natural transformation defined by

$$L(F)(a) = \varinjlim_{R \in J(a)} \text{Nat}(R, F),$$

where the collection of covering sieves $J(a)$ is treated as a diagram by considering the partial order induced by inclusion of sieves.

The colimit above can be seen as computing cones over bigger and bigger sub-covers of the pattern until cones on the entire pattern are obtained. For the case of pattern matching, one application of L is sufficient because a presheaf of occurrences is already a separated presheaf.[12] When combined with the distributive law for sheafification,

$$\varinjlim \text{sheafify}(F_i) = \text{sheafify}(\varinjlim F_i),$$

sheafification can be seen to the same as the incremental computation defined in section 5.3.18. This is another instance of a universal construction such as sheafification yielding a naive algorithm.[13]

5.3.20 Assessment

The incremental algorithm in FD-PATTERN-MATCH is a general version of the naive algorithm for pattern matching. Was all this theoretical machinery necessary to derive it? The answer is "yes," because of the generality of the algorithm. It is easy to generate a naive generate-and-test algorithm for data structures for which there is a natural scheme for enumerating all "positions" in the data structure. For example, in a string the possible positions for an occurrence are given by the indices in the array representation. Thus, we can enumerate all the indices up to the length of the

[12]The first application of L converts a presheaf into a separated presheaf; the second application converts a separated presheaf into a sheaf.

[13]For a unification algorithm obtained from characterizing a most general unifier as an equalizer, see [Rydeheard and Burstall 88].

string and try to match the pattern at each of these positions. For data structures like graphs, it is not straightforward to enumerate all the positions where an occurrence may exist. Our derivation always produces a generate-and-test pattern matching algorithm for any data structure, given a traversal mechanism.

Most derivations start by assuming the naive algorithm. We do not. This difference results in our deeper explanation of the "sliding" operation and "failure" function of the Knuth-Morris-Pratt algorithm. Again, by treating the Knuth-Morris-Pratt algorithm in a general context, we expose its basic simplicity and elegance.

5.3.21 Optimizations

The incremental algorithm in FD-PATTERN-MATCH is not very efficient, because it saves all partial matches. In particular, for strings, with pattern p and target t, the complexity is $\mathcal{O}(2^{|p|} \times |t|)$. The complexity of the algorithm depends on the length of the stream generated by traversing the target and the complexity of the pullback operation in each cycle, which in turn depends on the size of the cache and the size of the increments. There are two conceptual sources of optimization for the algorithm: reducing the size of the cache, and reducing the length of the stream.

One way to reduce the size of the cache is to remove all partial occurrences in the cache which have no potential to be expanded, as indicated by the traversal of the target, e.g., reaching a leaf in a tree. For strings, assuming a linear left-to-right traversal, there can be at most $|p| - 1$ partial matches which are potentially expandable (as opposed to $2^{|p|}$ above), thus reducing the complexity to $\mathcal{O}(|p| \times |t|)$. In the example of figure 5.9, at step 7, all the elements of the cache can be removed because none of them can be expanded.

Since the decision to drop elements of the cache depends on the particular data structure and the traversal mechanism, we do not consider it further here.

To reduce the length of the stream generated by traversing the target, we re-examine the traversal operation to see how increments are generated. Recall that traversing the target initially produces a sequence of objects, which are then converted into occurrence arrows (see section 5.3.17). Now, when a piece of a pattern occurs in the pattern itself, each piece of the target may produce multiple occurrence arrows.

EXAMPLE 5.15. Here is an example with strings, for a pattern "$abab$" and a target "$ababab$". To disambiguate the various parts, we number them: $a^1b^1a^2b^2$ for the pattern and $a_1b_1a_2b_2a_3b_3$ for the target. When the piece a_2b_2 is encountered while traversing the target, since "ab" occurs twice in the pattern, we obtain two occurrence arrows: $a^1b^1 \mapsto a_1b_1a_2b_2a_3b_3$, and $a^2b^2 \mapsto a_1b_1a_2b_2a_3b_3$. □

Informally, this means that each piece of the target can be "parsed" in several ways as a piece of the pattern. The operation "directly-solve" generates occurrence arrows over pieces of pattern, p_i. These arrows produce cones over the diagrams Q_j

for all q_j such that $p_i \xrightarrow{q_j} p$ is an arrow in the pattern cover. Thus, the q_j's correspond to different ways of parsing the occurrence arrows on p_i's.

The assignment of q_j's representing parses to any p_i can be precomputed for a given pattern cover. Moreover, this computation can be extended to any collection of p_i's. The extension of this relation, called the pattern-pattern-sheaf, will be described in section 5.3.28.

Consistent with our overall direction of converting the extension of the pattern matching relation into an intension, we can replace the cache with a collection of generators: the generator associated with Π_k will produce cones which are built using alternative parses of the occurrence arrows making up Π_k. This reduces the number of elements in the cache, and hence the complexity of the matching algorithm (provided building occurrences by composing with arrows in the generators is more efficient than directly updating the cache).

The scheme above requires that, when there are multiple occurrence arrows generated by a piece of the target, some of the extra occurrence arrows are delayed in the stream. This is indeed valid, because the distributive law for cones (section 5.3.15) and, more generally, the sheafification operation (section 5.3.19), are insensitive to the order in which the increments to the occurrence sheaf are presented.

The net effect of these manipulations is that incrementally updating partial matches has been converted into a search problem: that of searching for compatible parses of pieces of the target. In the following sections, we will describe a theory of search, and precisely formulate the optimizations discussed above.

5.3.22 A theory of search

A search problem consists of a search space and a goal predicate. The search space is a collection of states which are explored; it is usually specified intensionally by providing a generator for states. The goal predicate indicates states which are acceptable as solutions.

We reproduce below a slightly modified version of Smith's characterization of the abstract structure of a global search problem [Smith 88]. The problem to be implemented is specified as a relation $O(x, z)$ between input and output elements.

```
spec O-SPEC =
sorts D, R
operations
  O : D, R → BOOL
end
```

The states in a state space are given by "descriptors" such as $\hat{r}$ and $\hat{s}$. The state space corresponding to an input x is generated as follows: $\hat{r}_0(x)$ produces the initial state, and Split$(\hat{r}, \hat{s})$ generates a new state $\hat{s}$ from an old state $\hat{r}$. Acceptable states are indicated by a predicate Extract$(z, \hat{r})$ which states that the output element z

can be immediately obtained from the state $\hat{r}$. The predicate Satisfies$(z, \hat{r})$ is the transitive closure of Extract$(z, \hat{r})$.

spec GLOBAL-SEARCH =
—*based on [Smith 88]*
sorts D, R —*domain and range*
—*state descriptors corresponding to input x form a dependent type*
sorts $\lambda x \in \mathrm{D} \cdot \mathrm{SD}(x)$
operations
—*$\hat{r}_0(x)$ is the initial state descriptor associated with input element x*
$\hat{r}_0$: D $\rightarrow$ SD
—*the next state generator*
—Split$(\hat{r}, \hat{s})$ *means that $\hat{s}$ is one possible next state for $\hat{r}$*
Split : SD, SD $\rightarrow$ BOOL
—Extract$(z, \hat{r})$ *means that the output element z is directly extractable*
—*from the state $\hat{r}$, i.e., without generating any more states*
Extract : R, SD $\rightarrow$ BOOL
—Satisfies$(z, \hat{r})$ *means that the output element z can be extracted*
—*from a state $\hat{s}$ generated from the state $\hat{r}$*
Satisfies : R, SD $\rightarrow$ BOOL
axioms
—*the denotation of an arbitrary state descriptor*
—*is the collection of all elements which are generated by*
—*a finite number of applications of* "Split"
$\mathrm{Satisfies}(z, \hat{r}) \iff \exists k \in \mathrm{NAT}, \hat{s} \in \mathrm{SD} \cdot (\mathrm{Split}^k(\hat{r}, \hat{s}) \wedge \mathrm{Extract}(z, \hat{s}))$
where
—Split^k *is the k-fold composition of* Split
$\mathrm{Split}^0(\hat{r}, \hat{t}) \iff \hat{r} = \hat{t}$
$\mathrm{Split}^{k+1}(\hat{r}, \hat{t}) \iff \exists \hat{s} \in \mathrm{SD} \cdot (\mathrm{Split}(\hat{r}, \hat{s}) \wedge \mathrm{Split}^k(\hat{s}, \hat{t}))$
end

Smith has an additional axiom which states that all solutions can be generated from the initial state associated with an input element:

$$\mathrm{O}(x, z) \Rightarrow \mathrm{Satisfies}(z, \hat{r}_0(x)).$$

This axiom is not necessary in our scheme because it will be contained in any constructor which is used in the implementation of a problem via the global search theory above.

5.3.23 Updating the cache as a search problem

To optimize the incremental algorithm in FD-PATTERN-MATCH, we reformulate it as a search problem. The state space consists of states of the cache together with the unprocessed part of the stream of increments. The next state is generated by picking off an increment from the stream of increments and updating the cache in the current state. The formal version is shown below. Observe that this view of the pattern matching problem as search is somewhat different from that in section 5.3.1.Alt.1. There the search space consists of positions in the target; here the search space consists of partial occurrences.

In our earlier description of the traversal operation (cf. section 5.3.17 and the specification FD-PATTERN-MATCH), we assumed that it produced increments over diagrams of prime sieves $\mathcal{C}/Q_j$, i.e., over pieces of the pattern cover. Since we want explicit control on how any piece of the target is parsed, we now change the traversal operation so that it produces increments over diagrams of the sieves P_i generated by objects p_i in the base of the pattern cover.

spec TARGET-TRAVERSAL(P :: SIEVE($\mathcal{C}$)) =
sorts INCREMENT, PARSE
—*increments are functors defined over diagrams* $\mathcal{C}/P_i$
—P_i *is the sieve generated by* p_i, *with* $p_i \in$ base(P)
subsorts INCREMENT $\prec$ Fun($\mathcal{C}/P_i$, **Set**)
—*a parse is an interpretation of an increment on* $\mathcal{C}/P_i$
—*as an occurrence associated with some prime sieve* Q_j
subsorts PARSE $\prec$ Fun($\mathcal{C}/Q_j$, **Set**) **for** $Q_j \in$ prime-sieves(P)
operations
traverse : Obj($\mathcal{C}$) $\rightarrow$ STREAM(INCREMENT)
parses : INCREMENT $\rightarrow$ SET(PARSE)
axioms
—*parsing of increments will be disussed in section 5.3.28*
end

spec GS-CONES($c :: \text{Obj}(\mathcal{C}), P :: \text{SIEVE}(\mathcal{C})$) =
—*incremental cone computation as search*
—*states are caches, arcs are parses*
—*import the definition and updating of a cache from* FD-CONES
include derive from FD-CONES(c, P) **by**
update $\mapsto$ update
CACHE $\mapsto$ CACHE
PARSE $\mapsto$ INCREMENT
prim-cones $\mapsto$ prim-cones
extend GLOBAL-SEARCH **renamed as**
D $\mapsto$ STREAM(INCREMENT)
R $\mapsto$ SET(CONE)
SD $\mapsto$ CACHE $\times$ STREAM(INCREMENT)
extend TARGET-TRAVERSAL(P)
operations
gs-cones : STREAM(INCREMENT) $\rightarrow$ SET(CONE)
axioms
—*the initial state is the empty cache*
—*together with the initial stream of increments*
$\hat{r}_0(s) = \langle \emptyset, s \rangle$
—*the next state expands the cache in the current state*
—*using some parse of some increment from the input stream*
$\text{Split}(\langle \text{cache}, t \rangle, \langle \text{cache}', u \rangle) \iff$
$u = \text{tl}(t) \wedge \exists F \in \text{parses}(\text{hd}(t)) \cdot \text{cache}' = \text{update}(\text{cache}, \text{cons}(F, \text{nil}))$
—*the denotation of a state descriptor* $\langle \text{cache}, s \rangle$
—*is the set of all cones on the pattern cover* $\mathcal{C}/P$ *which can be*
—*obtained by updating the cache using parses of the stream* s
$\text{Satisfies}(C, \langle \text{cache}, s \rangle) \iff C = \text{update}(\text{cache}, s')(\mathcal{C}/P)$
where $s' = /\,\text{concat} \circ \alpha(\text{parses})(s)$
—*cones on* $\mathcal{C}/P$ *are directly extractable in any state*
$\text{Extract}(\text{cache}(\mathcal{C}/P), \langle \text{cache}, s \rangle)$
—*the effective function computed by the state space*
$\text{gs-cones}(s) = /\cup \circ \{\, z \mid \text{Satisfies}(z, \hat{r}_0(s)) \,\}$
end

Using the search theory above, we get the following implementation of pattern matching.

spec GS-PATTERN-MATCH($\langle \mathcal{C}, J \rangle$ :: FINITARY-NOETHERIAN-PM-SITE) =
extend GS-CONES
extend CONE-DECOMPOSITION-1 **with**
decompose **renamed as** decompose . CONE
operations
gs-occs : $\mathrm{Obj}(\mathcal{C}), \mathrm{Obj}(\mathcal{C}) \to$ SET(OCCURRENCE)
decompose : $\mathrm{Obj}(\mathcal{C}) \to$ SIEVE($\mathcal{C}$)
axioms
gs-occs$(p,t) = \alpha(\mathrm{glue}_P) \circ \text{gs-cones}_{1,P}(s)$
where $P = \mathrm{decompose}(p)$, $s = \mathrm{traverse}_P(t)$
—decomposition yields the finest covering sieve
$\mathrm{decompose}(p) \in J(p) \wedge \forall R \in J(p) \cdot \mathrm{decompose}(p) \subseteq R$
—computation of cones on $\mathcal{C}/Q_j$ is done by "dc-cones"
prim-cones$_{c,P}(F)$ = dc-cones(c, F)
end

Here is the constructor which connects the specification PATTERN-MATCH of pattern matching with the global search implementation above.

$$\text{PATTERN-MATCH} \rightsquigarrow \kappa\text{-GS(GS-PATTERN-MATCH)}$$

where

constructor κ-GS =
derive from GS-PATTERN-MATCH
by { occurrences $\mapsto$ gs-occs }
end

5.3.23.Alt.1 Commutativity in the design space

We could have reached the characterization of pattern matching as search above, GS-PATTERN-MATCH, via a different route. The problem of enumerating compatible families of partial occurrences (see section 5.3.7) can be described as a search problem. The generate-and-test natural transformation formulation produces a search space during the generate phase which is then filtered for feasible solutions. Similarly, the limit formulation generates a search space of products which is then filtered using an equalizer. After applying the theory of cone decomposition (see section 5.3.10) to either search space, we get an algorithm equivalent to the one above.

5.3.24 Search strategies

Two general strategies for searching in a state space are breadth-first search and depth-first search. The choice to be made in each state is which parse of the next increment to choose. A breadth-first strategy explores all parses before moving on to the next increment. A depth-first strategy chooses some parse, and immediately moves on to the next increment. If a depth-first strategy is to enumerate all states satisfying the goal predicate, then it also has to backtrack and explore alternative parses when the input stream is exhausted.

5.3.24.Alt.1 Breadth-first search

A breadth-first search of the state space of cache values is equivalent to the incremental algorithm in FD-PATTERN-MATCH, provided we merge all the states after each ply of the search. This follows because of the following relation between the stream of increments in the two algorithms:

$$\text{traverse}.\,\text{FD-PATTERN-MATCH} = \text{concat} \circ \alpha(\text{parses}) \circ \text{traverse}.\,\text{GS-CONES}.$$

Thus, the sequence of updates in FD-PATTERN-MATCH corresponds to applying the "Split" operator of GS-CONES in all possible ways, and hence to a breadth-first exploration.

5.3.24.Alt.2 Depth-first search

In a depth-first strategy, rather than explore all possible parses, one is chosen for further exploration. Since, the state space in GS-CONES is bounded, both the depth-first and breadth-first strategies explore all the paths in the space, and, therefore, are equivalent.

A depth-first strategy becomes more efficient when we can avoid exploring certain paths. As observed in section 5.3.21, each increment in the stream is capable of generating multiple parses. We can choose only one parse to explore, generating the other parses on demand. This scheme reduces the number of elements in the cache and thus leads to a more efficient algorithm. Since we are interested in reducing the number of elements in a cache, we reformulate the specification GS-CONES as search at a finer grain: the states are cones rather than caches (a cache is a *set* of cones).

5.3.25 Computing cones as search: A backtracking implementation

We reformulate the search algorithm GS-CONES of section 5.3.23 so that the states are cones rather than sets of cones. This involves decomposing a cache into its constituents and recasting the update operation on caches as a search problem.

Before presenting the specification, we need some notation to denote the expansion of cones.

spec CONE-GLUING =
operations
$\_ \otimes \_ : \text{CONE}, \text{CONE} \rightarrow \text{CONE}$ —*partial*
axioms
—*a cone ν on a diagram D is written as ν_D*
—*the restriction of ν_D to a sub-diagram $E \subseteq D$ is written as $\nu|_E$*
Defined($\nu_D \otimes \tau_E$) **if** $\nu|_{D \cap E} = \tau|_{D \cap E}$
$\nu_D \otimes \tau_E = \sigma_{D \cup E}$ **where** $\sigma|_D = \nu_D \wedge \sigma|_E = \tau_E$
end

In the search algorithm of GS-CONES-1 below, a state consists of a cone and a stream of increments. The arcs are parses of increments. Next states are generated by expanding cones.

Paths in this search space are sequences of parses of elements of the input stream. The end-states are those in which the parses along the path can be glued together to produce an occurrence of the pattern. The search space can be viewed as constraint propagation in the extension of occurrence sheaf: the graph of the sheaf represents local constraints (compatibility between partial occurrences), and the goal is to spread these local constraints so as to cover the pattern and produce a full occurrence. This view allows us to derive Waltz filtering [Waltz 75], a relaxation algorithm for interpreting 2-D images by local constraint propagation, using a slight variation of the derivation of the Knuth-Morris-Pratt algorithm (see section 6.3).

spec GS-CONES-1($c :: \mathrm{Obj}(\mathcal{C}), P :: \mathrm{SIEVE}(\mathcal{C})$) =
—*incremental cone computation as search*
—*states are cones, arcs are parses*
include CONE-GLUING
extend GLOBAL-SEARCH **renamed as**
D $\mapsto$ STREAM(INCREMENT)
R $\mapsto$ CONE
SD $\mapsto$ CONE $\times$ STREAM(INCREMENT)
extend TARGET-TRAVERSAL(P)
operations
prim-cones : PARSE $\rightarrow$ CONE
gs-cones-1 : STREAM(INCREMENT) $\rightarrow$ SET(CONE)
axioms
—*the initial state is the cone on the empty diagram, called "0"*
—*and the initial stream of increments*
$\hat{r}_0(s) = \langle 0, s \rangle$
—*a next state expands the cone in the current state*
—*using some parse of some increment from the input stream*
$\mathrm{Split}(\langle \nu_D, t \rangle, \langle \tau_E, u \rangle) \iff$
$u = \mathrm{tl}(t) \wedge \exists F \in \mathrm{parses}(\mathrm{hd}(t)) \cdot \tau_E = \nu_D \otimes \text{prim-cones}(F)$
—*a next state can also be generated*
—*by simply ignoring the current increment*
$\mathrm{Split}(\langle \nu_D, t \rangle, \langle \nu_D, u \rangle) \iff u = \mathrm{tl}(t)$
—*the denotation of a state descriptor* $\langle \nu, s \rangle$
—*is the set of all cones on* $\mathcal{C}/P$ *which can be obtained*
—*by expanding the cone* ν *using increments in the stream* s
—*there is no need for an axiom for* Satisfies *because*
—*this denotation follows from the axioms of* GLOBAL-SEARCH
—*cones on* $\mathcal{C}/P$ *are directly extractable in any state*
$\mathrm{Extract}(\nu_{\mathcal{C}/P}, \langle \nu_{\mathcal{C}/P}, s \rangle)$
—*the effective function computed by the state space*
gs-cones-1$(s) = \{ z \mid \mathrm{Satisfies}(z, \hat{r}_0(s)) \}$
end

We now incorporate the modified search theory in the implementation of pattern matching.

spec GS-PATTERN-MATCH-1($\langle \mathcal{C}, J \rangle$ :: FINITARY-NOETHERIAN-PM-SITE) =
extend GS-CONES-1
extend CONE-DECOMPOSITION-1 **with**
decompose **renamed as** decompose . CONE
operations
gs-occs-1 : Obj($\mathcal{C}$), Obj($\mathcal{C}$) $\rightarrow$ SET(OCCURRENCE)
decompose : Obj($\mathcal{C}$) $\rightarrow$ SIEVE($\mathcal{C}$)
axioms
gs-occs-1$(p, t) = \alpha(\text{glue}_P) \circ \text{gs-cones-1}_{1,P}(s)$
where $P = \text{decompose}(p)$, $s = \text{traverse}_P(t)$
—decomposition yields the finest covering sieve
$\text{decompose}(p) \in J(p) \wedge \forall R \in J(p) \cdot \text{decompose}(p) \subseteq R$
—computation of cones on $\mathcal{C}/Q_j$ is done by "dc-cones"
$\text{prim-cones}_{c,P}(F) = \text{dc-cones}(c, F)$
end

Here is the constructor which connects the specification PATTERN-MATCH of pattern matching with the global search implementation above.

$$\text{PATTERN-MATCH} \rightsquigarrow \kappa\text{-GS-1(GS-PATTERN-MATCH-1)}$$

where

constructor κ-GS-1 =
derive from GS-PATTERN-MATCH-1
by { occurrences $\mapsto$ gs-occs-1 }
end

The global search description of cone computation in GS-CONES-1 can be implemented by a depth-first search strategy with backtracking. This is recorded in the specification BT-CONES below. This implementation is similar to the linear recursive scheme for implementing global search in [Smith 88, Theorem 3.2].

spec BT-CONES($c :: \text{Obj}(\mathcal{C}), P :: \text{SIEVE}(\mathcal{C})$) =
—*backtracking implementation of cone computation using search*
extend GS-CONES-1(c, P)
operations
bt-cones : STREAM(INCREMENT) → SET(CONE)
dfs : SD, SET(CONE), SET(SD) → SET(CONE)
pick-arc : SET(SD) → SD, SET(SD)
pick-bt : SET(SD) → SD, SET(SD)
axioms
—*start search at the initial state*
bt-cones(s) = dfs($\hat{r}_0(s), \emptyset, \emptyset$)
—dfs *takes three arguments: the current state being explored,*
—*a partial set of solutions, and a set of states yet to be explored*
dfs(*cur-state*, *Solutions*, *Active-States*) =
—*process current state: add solutions extractable from current state*
let *New-Solutions* = *Solutions* ∪ { z | Extract(z, *cur-state*) }
—*generate next states*
and *New-States* = { $\hat{s}$ | Split(*cur-state*, $\hat{s}$) } **in**
if *New-States* = ∅ **then**
—*current state cannot be expanded; backtrack to another state*
if *Active-States* = ∅ **then** —*nowhere to backtrack; return solutions*
New-Solutions
else —*pick an unexplored state to backtrack to*
let ⟨*new-state*, *New-Active*⟩ = pick-bt(*Active-States*) **in**
dfs(*new-state*, *New-Solutions*, *New-Active*)
else —*proceed depth-first: pick one of the next states to explore*
let ⟨*new-state*, *New-Active*⟩ = pick-arc(*New-States*) **in**
dfs(*new-state*, *New-Solutions*, *Active-States* ∪ *New-Active*)
—*axioms constraining the choice functions*
pick-arc(S) = ⟨s, T⟩ ⇒ $S = \{s\} \cup T$
pick-bt(S) = ⟨s, T⟩ ⇒ $S = \{s\} \cup T$
end

Here is the constructor which implements the global search specification GS-CONES-1 by backtracking.

$$\text{GS-CONES-1} \rightsquigarrow \kappa\text{-BT(BT-CONES)}$$

where

constructor κ-BT =
derive from BT-CONES
by { gs-cones-1 ↦ bt-cones }
end

5.3.26 Search filters

A search filter removes certain states which do not lead to a solution. Smith considers three kinds of filters, necessary, sufficient, and heuristic, depending on whether the filters are conservative or eager in pruning away states [Smith 88]. We describe below two filters which are necessary, in the sense that the states they prune away are guaranteed not to lead to a solution.

The traversal dead-end filter

In the search space of cones above, the next-state generator is defined such that a state cannot be expanded only if the input stream has been exhausted. We can discard unexpandable states earlier in the search using information from the traversal operation. For example, if we can deduce while traversing the target that a certain part of the target has been completely explored, then partial occurrences in that part of the target cannot be expanded, and can be discarded. In the case of pattern matching in strings, with a left-to-right traversal of the target string, all partial occurrences not touching the current position of traversal can be discarded. This filter is crucial in reducing the complexity of the search. This filter is incorporated in the specification BTF-CONES below.

The invalid partial occurrence filter

If the underlying diagram of a partial occurrence does not correspond to any Π_k, then that partial occurrence can be discarded, because it cannot be expanded into a full occurrence. Formally, we add another filter, "is-partial-occurrence," to the search algorithm in BTF-CONES below.

The full occurrence filter

If the underlying diagram of a partial occurrence is $\mathcal{C}/P$, then we have a full occurrence which cannot be further expanded. We can extract the full occurrence and discard the state. The full occurrence filter is denoted by "is-full-occurrence" in the specification BTF-CONES below.

spec FILTER =

—*this specification just collects the filters for* BTF-CONES

—*it is not expected to be interpreted independently*

operations

filter : SD → BOOL

is-partial-occurrence : SD → BOOL

is-full-occurrence : SD → BOOL

dead-end : SD → BOOL

axioms

—*filter definitions*

filter($\hat{s}$) = dead-end($\hat{s}$) ∨ ¬ is-partial-occurrence($\hat{s}$) ∨ is-full-occurrence($\hat{s}$)

is-partial-occurrence($\langle \nu_D, s \rangle$) **if** $\exists \Pi_k \cdot D = \Pi_k$

is-full-occurrence($\langle \nu_{C/P}, s \rangle$)

—*details of* "dead-end" *are obtained from the traversal mechanism*

end

spec BTF-CONES($c :: \text{Obj}(\mathcal{C}), P :: \text{SIEVE}(\mathcal{C})$) =
—*backtracking implementation of cone computation using search*
—*includes the traversal dead-end, invalid partial occurrence,*
—*and full occurrence filters for eliminating unexpandable cones*
extend GS-CONES-1(c, P)
include FILTER
operations
btf-cones : STREAM(INCREMENT) → SET(CONE)
dfs : SD, SET(CONE), SET(SD) → SET(CONE)
pick-arc : SET(SD) → SD, SET(SD)
pick-bt : SET(SD) → SD, SET(SD)
axioms
—*start search at the initial state*
btf-cones(s) = dfs($\hat{r}_0(s), \emptyset, \emptyset$)
—dfs *takes three arguments: the current state being explored,*
—*a partial set of solutions, and a set of states yet to be explored*
dfs(*cur-state*, *Solutions*, *Active-States*) =
—*process current state: add solutions extractable from current state*
let *New-Solutions* = *Solutions* ∪ { z | Extract(z, *cur-state*) }
—*generate next states*
and *New-States* = { $\hat{s}$ | Split(*cur-state*, $\hat{s}$) } **in**
—*FILTER INCLUDED HERE:* "filter($\hat{s}$)" *means* $\hat{s}$ *can be pruned*
if *New-States* = $\emptyset$ ∨ filter(*cur-state*) **then**
—*current state cannot be expanded; backtrack to another state*
if *Active-States* = $\emptyset$ **then** —*nowhere to backtrack; return solutions* *New-Solutions*
else —*pick an unexplored state to backtrack to*
let ⟨*new-state*, *New-Active*⟩ = pick-bt(*Active-States*) **in**
dfs(*new-state*, *New-Solutions*, *New-Active*)
else —*proceed depth-first: pick one of the next states to explore*
let ⟨*new-state*, *New-Active*⟩ = pick-arc(*New-States*) **in**
dfs(*new-state*, *New-Solutions*, *Active-States* ∪ *New-Active*)
—*axioms constraining the choice functions*
pick-arc(S) = ⟨s, T⟩ ⇒ $S = \{s\} \cup T$
pick-bt(S) = ⟨s, T⟩ ⇒ $S = \{s\} \cup T$
end

By replacing the specification BT-CONES with the specification BTF-CONES (which contains search filters),

$$\text{GS-CONES-1} \rightsquigarrow \kappa\text{-BTF}(\text{BTF-CONES})$$

where

constructor κ-BTF =
derive from BTF-CONES
by { gs-cones-1 $\mapsto$ btf-cones }
end

and composing the constructor implementations using global search (κ-GS-1, section 5.3.25) and backtracking (κ-BTF above), we get the following implementation of pattern matching.[14]

$$\text{PATTERN-MATCH} \rightsquigarrow \kappa\text{-GS-1} \circ \kappa\text{-BTF}(\text{BT-PATTERN-MATCH})$$

where

spec BT-PATTERN-MATCH($\langle \mathcal{C}, J \rangle$ :: FINITARY-NOETHERIAN-PM-SITE) =
extend BTF-CONES
extend CONE-DECOMPOSITION-1 **with**
decompose **renamed as** decompose . CONE
operations
bt-occs : Obj($\mathcal{C}$), Obj($\mathcal{C}$) $\rightarrow$ SET(OCCURRENCE)
decompose : Obj($\mathcal{C}$) $\rightarrow$ SIEVE($\mathcal{C}$)
axioms
bt-occs$(p, t) = \alpha(\text{glue}_P) \circ \text{btf-cones}_{1,P}(s)$
where $P = \text{decompose}(p)$, $s = \text{traverse}_P(t)$
—decomposition yields the finest covering sieve
$\text{decompose}(p) \in J(p) \wedge \forall R \in J(p) \cdot \text{decompose}(p) \subseteq R$
—computation of cones on $\mathcal{C}/Q_j$ is done by "dc-cones"
$\text{prim-cones}_{c,P}(F) = \text{dc-cones}(c, F)$
end

5.3.27 Dependency-directed backtracking: Motivation

The major choice that has to be made in a depth-first backtracking search strategy, corresponding to the function "pick-bt" in the specification BT-CONES above, is the state to which the algorithm backtracks when the current state cannot be expanded by cones generated by any parse of the current increment.

[14] We have taken a slight liberty with the composition; a precise treatment would invoke the compatibility of the constructor κ-BTF (a vertical operation) with the horizontal operation **extend** GS-CONES-1 in the specification GS-PATTERN-MATCH-1.

5.3.27.Alt.1 Chronological backtracking

Chronological backtracking retracts to the point of the most recent unexhausted choice-point and continues exploration there with another choice. This is the simplest choice for backtracking. However, chronological backtracking frequently rediscovers dead-ends and performs some unnecessary recomputation [Mackworth 77]. In the case of pattern matching, the existence of a partial occurrence excludes certain combinations of parses, combinations which would be discovered and eliminated, one-by-one, during chronological backtracking.

5.3.27.Alt.2 The chosen alternative

We can backtrack more intelligently by using information contained in the partial occurrence currently being explored. The general strategy of using information collected along a path to backtrack to another state is called *dependency-directed backtracking.* Usually, the intent is to minimize recomputation of information already collected along the path. As an added benefit, we can avoid exploring certain paths in the space, resulting in an algorithm more efficient than one using a breadth-first strategy.

In the case of pattern matching, the multiple parses corresponding to an increment can be extended to sequences of increments. Thus, if the partial occurrence in the current state is over the diagram Π_k, and is obtained via a collection of increments P_n, all the alternative paths generated by parsing the elements of P_n differently can be computed from the current state. Moreover, most of this computation depends only upon the pattern cover P; thus alternative parses can be precomputed for a given pattern cover.

We can exploit the information contained in the current partial occurrence to backtrack to a state different from that given by chronological backtracking. Keeping with the spirit of expanding partial occurrences as far as possible, we backtrack to the largest alternative partial occurrence which can be derived from the current state.

This scheme of expanding a partial occurrence until it is no longer expandable is a hill-climbing strategy. The backtracking step corresponds to jumping to an adjacent hill when a local maximum is reached.

The precomputation of multiple parses for sets of increments and the partial order imposed on partial occurrences is described in the sections below.

5.3.28 The pattern-pattern-sheaf

As mentioned above, the multiple parses associated with a piece of the target can be extended to bigger pieces. We first show how multiple parses are generated and then show how they can be extended to form a sheaf.

Traversing the target generates occurrence arrows of the form $p_i \rightarrow t$, where p_i is an element in the base of the cover P for the pattern p. As discussed in section 5.3.18, we assume that finding an arrow $p_i \rightarrow t$ also provides, by restriction, all arrows of the form $p_i' \rightarrow t$ where $p_i' \rightarrow p_i$ is an arrow in the sieve P_i corresponding to p_i. Thus, as in the specification TARGET-TRAVERSAL, we assume that traversal generates increments which are functors on the diagrams $\mathcal{C}/P_i$.

We are interested in computing cones over the diagram $\mathcal{C}/P$ of the pattern cover, whereas traversal provides increments over diagrams in the category base(P). In other words, we need increments which are over the diagrams $\mathcal{C}/Q_j$ which comprise the pattern cover. Recall the relationship between the functors (see section 5.3.7.Alt.2)

$$F\colon \text{base}(P)^{\text{op}} \rightarrow \textbf{Set}$$

and

$$F^{\#}\colon \mathcal{C}/P \rightarrow \textbf{Set}$$

given by

$$F^{\#}(p_i \xrightarrow{q_j} p) = F(p_i) \qquad \text{and} \qquad F^{\#}(f) = F(f^{\text{op}}).$$

Traversal provides increments to the functor F whereas we need increments for the functor $F^{\#}$. The transition from F to $F^{\#}$ is in two steps: the computation of sieves Q_j corresponding to a sieve P_i, and the translation of F defined over P_i to $F^{\#}$ defined over Q_j. The first can be precomputed for a given pattern cover, the second is a straightforward application of the definition of $F^{\#}$.

The relationship between the sieves P_i and the sieves Q_j is as follows: there are as many Q_j's corresponding to a P_i as there are arrows $q_j\colon p_i \rightarrow p$ (i.e., the number of times p_i occurs in the pattern p). Each increment over Q_j is said to be a *parse* of the increment on P_i. When extended to bigger pieces, this relation is called the *pattern-pattern-sheaf.*

DEFINITION 5.16: *Pattern-pattern-sheaf.* The underlying category $\Delta_{\text{base}(P)}$ of the pattern-pattern-sheaf consists of all diagrams over base(P) together with inclusion arrows. Covers are as in definition 5.5, section 5.3.10. The functor

$$\Phi: \Delta^{\text{op}}_{\text{base}(P)} \rightarrow \textbf{Set}$$

assigns to each diagram in base(P) a set of parses, i.e., diagrams in $\mathcal{C}/P$. The functor Φ is inductively defined as follows:

1. For a diagram consisting of a single object p_i,

$$\Phi(p_i) = \{ \langle p_i, p_i \xrightarrow{q_j} p \rangle \mid q_j \in \mathcal{C}/P \}.$$

2. For a diagram consisting of a single arrow $f: p_i^1 \rightarrow p_i^2$,

$$\Phi(f) = \{ \langle p_i^2, q_j^2 \rangle \xrightarrow{f^{\text{op}}} \langle p_i^1, q_j^1 \rangle \mid q_j^1 = q_j^2 \circ f \text{ in } \text{base}(P) \}.$$

3. For an arrow of the form $g: p_i^1 \hookrightarrow (p_i^1 \xrightarrow{f} p_i^2)$ between diagrams,

$$\Phi(g): \left(\langle p_i^2, q_j^2 \rangle \xrightarrow{f^{\text{op}}} \langle p_i^1, q_j^1 \rangle \right) \mapsto \langle p_i^1, q_j^1 \rangle.$$

4. For any other diagram δ, let $\delta = \underrightarrow{\text{Colim}}_{x \in X} \delta_x$, where δ_x is a diagram of the forms in items 1 or 2 above. Then,

$$\Phi(\delta) = \underleftarrow{\text{Lim}}_{x \in X} \Phi(\delta_x).$$

5. For any other arrow, Φ is defined via the limit in item 4 above.

The functor Φ evidently forms a sheaf. □

We show, in figure 5.10, an example of the pattern-pattern-sheaf for a tree pattern using the finest cover.

Before proceeding, we define a concept which will be frequently used in the sequel: the underlying diagram of a sub-pattern.

DEFINITION 5.17: *Underlying diagram.* Given a pattern cover P, for any sub-pattern Π_k, the underlying diagram of Π_k, written $\Upsilon(\Pi_k)$, is the diagram P_i such that $\Pi_k \in \Phi(P_i)$, where Φ is the pattern-pattern-sheaf corresponding to P. The assignment of P_i to Π_k is clearly unique and can be extended to arrows. We thus have a partial functor $\Upsilon: \textbf{Set} \rightarrow \Delta^{\text{op}}_{\text{base}(P)}$ which is the left-inverse of Φ. Intuitively, Υ projects a sub-pattern onto its underlying diagram. □

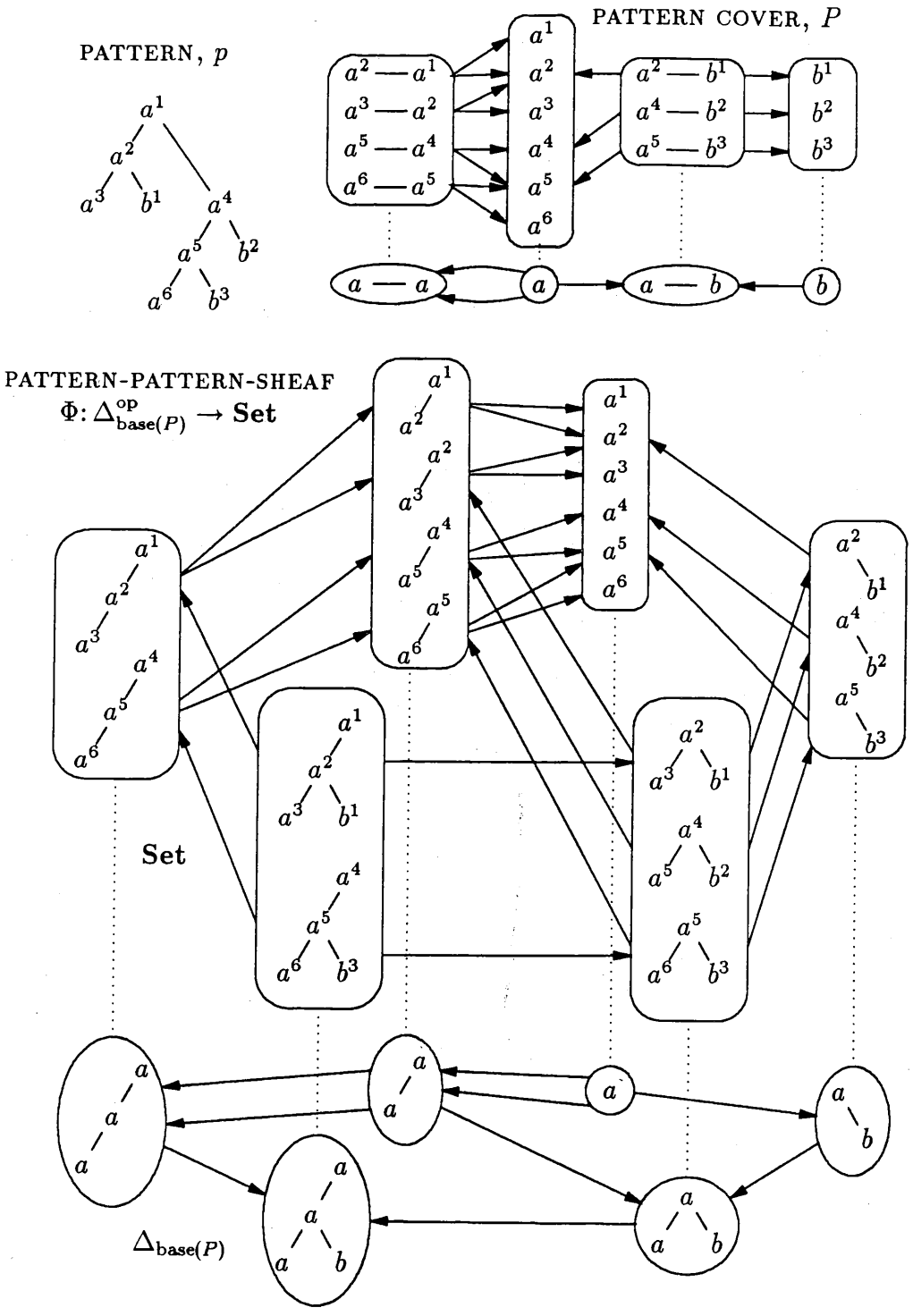


Figure 5.10: The pattern-pattern-sheaf: example with trees

5.3.29 The subsumption relation

To improve the efficiency of the depth-first search of the space of cones (partial occurrences), we explore only one parse at a time, using the pattern-pattern-sheaf to generate the other parses on demand. Alternative parses which can be generated on demand are said to be *subsumed* by the original parse. We only need the subsumption relation between cones on the diagrams Π_k (since cones on other diagrams are filtered out during the search). Moreover, the subsumption relation between parses is uniquely determined by the subsumption relation on sub-patterns. This subsumption relation between sub-patterns is formally defined below.

DEFINITION 5.18: *The subsumption relation.* Given two diagrams Π_k and Π_ℓ in $\mathcal{C}/P$, Π_k subsumes Π_ℓ, written $\Pi_k \succeq \Pi_\ell$, if $\Upsilon(\Pi_\ell) \subseteq \Upsilon(\Pi_k)$ (note the change in direction). □

The subsumption relation is evidently a partial order.

The definition of the subsumption relation comprises two parts: subsumption via alternative parses, and subsumption via inclusion. The first, denoted by $\cong$, is an equivalence relation. It is defined by $\Pi_k \cong \Pi_\ell \iff \Upsilon(\Pi_k) = \Upsilon(\Pi_\ell)$, i.e., two sub-patterns mutually subsume each other if they are alternative parses of the same diagram. The second part, denoted by $\succ$, is strict subsumption induced by strict inclusion of underlying diagrams. It is defined by $\Pi_k \succ \Pi_\ell \iff \Upsilon(\Pi_\ell) \subset \Upsilon(\Pi_k)$.

The assignment of alternative parses to each diagram P_i generates a sheaf, the pattern-pattern-sheaf (section 5.3.28).[15] Hence, to compute the subsumption relation, we can re-apply our derivation to this sheaf: the sheaf condition provides a recursive algorithm which is then converted into an iterative algorithm by traversing the pattern. For the second part of the subsumption relation, that induced by inclusion, it is sufficient to compute the inclusion order on diagrams. This order is synergistically incorporated into the algorithm as the well-founded order required by the divide-and-conquer step.

We omit the details of the recursive sub-derivation and present the final result.[16] Observe that the traversal operation produces occurrence arrows with respect to the pattern, and that the invalid partial occurrence filter (section 5.3.26) has already been incorporated (since the pattern-pattern-sheaf is only computed for diagrams of the form "$\Upsilon(\Pi_k)$").

[15]The two parts of the subsumption relation can be combined into a single sheaf by adding an extra element, $\perp$ (to indicate discarding of that increment), to each set of parses. The sheaf then assigns to each pattern diagram a set of parses, which is then closed under subsumption. We omit this characterization because, later on in the derivation, it becomes necessary to split the subsumption relation into two parts.

[16]The synthesis and optimization of this algorithm can be rigorously carried out as a subderivation. However, we omit the details, since the focus of this dissertation is the main sequence of design decisions which leads to the Knuth-Morris-Pratt algorithm.

spec PATPAT-SHEAF(P :: SIEVE($\mathcal{C}$)) =
—*incremental computation of the pattern-pattern-sheaf*
—*algorithm obtained by a recursive application of the derivation*
—*cf.* FD-CONES, *section 5.3.18*
sorts PP-SHEAF, INCREMENT
—*a pattern-pattern-sheaf associates a set of parses*
—*with each diagram built from pieces of the pattern*
subsorts PP-SHEAF $\prec$ Fun($\Delta^{\mathrm{op}}_{\mathrm{base}(P)}$, **Set**)
—*increments are functors defined over diagrams of prime sieves*
—*the increments are occurrence arrows of the pattern within itself*
subsorts INCREMENT $\prec$ Fun($\mathcal{C}/P_i$, **Set**) **for** $P_i \in$ prime-sieves(base(P))
operations
pp-sheaf : STREAM(INCREMENT) $\rightarrow$ PP-SHEAF
update : PP-SHEAF, STREAM(INCREMENT) $\rightarrow$ PP-SHEAF
axioms
—*start iterative algorithm with empty sheaf*
pp-sheaf(s) = update($\emptyset, s$)
—*incremental update of the pattern-pattern-sheaf*
—*when the increment stream is exhausted, return the current sheaf*
update(pp-sheaf, nil) = pp-sheaf
—*process one element of the stream at a time*
update(pp-sheaf, cons($\mathcal{C}/P_i \xrightarrow{F}$ **Set**, s)) = update(pp-sheaf′, s)
where pp-sheaf′ = pp-sheaf **except**
—*update parses of* $\mathcal{C}/P_i$
pp-sheaf′($\mathcal{C}/P_i$) = pp-sheaf($\mathcal{C}/P_i$) $\cup$ $F(\mathcal{C}/P_i)$ **and**
—*update parses for all diagrams which intersect with* $\mathcal{C}/P_i$
—*using the distributive law; Theorem 5.2*[17]
$\forall \Pi_k \in$ subpatterns(P) $\cdot$ $\mathcal{C}/P_i \subseteq \Upsilon(\Pi_k) \Rightarrow$
pp-sheaf′($\Upsilon(\Pi_k)$) = pp-sheaf($\Upsilon(\Pi_k)$) $\cup$ pp-sheaf($\mathcal{C}/P_i'$) $\times_{\text{pp-sheaf}(\mathcal{C}/P_i' \cap \mathcal{C}/P_i)}$ $F(\mathcal{C}/P_i)$
where $\mathcal{C}/P_i' = \Upsilon(\Pi_k) \sim \mathcal{C}/P_i$
—*the pullback above also updates projections of parses*
end

The axiom for updating the pattern-pattern-sheaf uses quantification over the sub-patterns Π_k. However, the collection of sub-patterns is not known in advance; indeed, one of the purposes of traversing the pattern is to enumerate this collection. We can remove this circularity by building the collection of sub-patterns while building the pattern-pattern-sheaf. This is based on the observation that the value of the pullback pp-sheaf($\mathcal{C}/P_i'$) $\otimes$ F is empty when pp-sheaf($\mathcal{C}/P_i'$) is empty, i.e., the increments to

[17]Strictly speaking, we cannot use this theorem since the elements of the pattern-pattern-sheaf are not cones; they are parses. However, this slight imprecision is justified by the fact that parses are just cones which have already been glued together.

the sheaf built by expanding already existing sub-patterns. Thus, we can build the category $\Delta_{\mathbf{base}(P)}$ while building the pattern-pattern-sheaf.

Let Δ be the category of pattern diagrams (i.e., diagrams of the form "$\Upsilon(\Pi_k)$") found until now. Given an increment $\mathcal{C}/P_i \xrightarrow{F} \mathbf{Set}$, the increment to the category Δ consists of:

New objects: $\mathcal{C}/P_i \cup D$, for all $D \in \mathrm{Obj}(\Delta)$; and

New arrows: $D \hookrightarrow (\mathcal{C}/P_i \cup D)$, for all $D \in \mathrm{Obj}(\Delta)$.

Observe that in incrementally enumerating the category $\Delta_{\mathbf{base}(P)}$, we have computed the inclusion relation on diagrams. However, this inclusion relation need not be completely computed. Our intent is to save on space by representing the collection of active states (i.e., states yet to be explored) in the specification BTF-CONES using generators. This also saves on time by exploring states only on demand. As discussed above, when the current partial occurrence cannot be expanded, we backtrack to the largest partial occurrence that can be derived from the current state using alternative parses. When a unique such partial occurrence does not exist for some Π_k, we have to explore the states corresponding to the least upperbounds of all connected components of the set $\{\, \Pi_\ell \mid \Pi_k \succ \Pi_\ell \,\}$ of sub-patterns strictly subsumed by Π_k. Let us denote this set (of upperbounds) by $\Sigma(\Pi_k)$. Just as the pattern-pattern-sheaf can be computed incrementally, so can the subsumption sets for each sub-pattern. The identity required for the incremental computation is (provided neither diagram is empty):[18]

$$\Sigma(\Pi_k \cup \mathcal{C}/P_i) = \{\Pi_k\} \cup \{\, D \cup \mathcal{C}/P_i \mid D \in \Sigma(\Pi_k) \,\}.$$

Finally, we remove another dependency in the computation of the subsumption relation: the dependency on the pattern cover P. This cover can be built while traversing the pattern. However, as we will see below, this cover is not explicitly required in the pattern matching algorithm; all that is needed is the assumption that traversing an object produces increments over pieces of the finest cover (of that object).

We consolidate all the details discussed above into a computation of the subsumption relation, which includes the computation of the pattern-pattern-sheaf.

[18]This identity can be rigorously derived by applying finite differencing to the recursive definition of the subsumption relation. Again, we omit the details.

spec TRAVERSAL($\langle \mathcal{C}, J \rangle$ :: FINITARY-NOETHERIAN-PM-SITE) =
sorts INCREMENT
subsorts INCREMENT $\prec$ Fun($\mathcal{C}/A_i$, **Set**) **for** $A_i \in$ prime-sieves(base(A))
  where —*A is the finest cover of an object $a \in$ Obj($\mathcal{C}$)*
operations
  traverse : Obj($\mathcal{C}$) $\rightarrow$ STREAM(INCREMENT)
axioms
  —*the pointwise union of the increments generated by traversing*
  —*an object yields the finest cover of that object*
  $/ \cup \circ$ traverse(a) $= A$ —*A is the finest cover of a*
end

spec SUBSUMPTION($p :: \mathrm{Obj}(\mathcal{C}), \langle \mathcal{C}, J \rangle ::$ FINITARY-NOETHERIAN-PM-SITE) =
extend TRAVERSAL($\langle \mathcal{C}, J \rangle$)
sorts PP-SHEAF, IMM-SUBS
—a pattern-pattern-sheaf associates a set of parses
—with each diagram built from pieces of the pattern
—P is the finest cover for the object p
subsorts PP-SHEAF $\prec \mathrm{Fun}(\Delta^{\mathrm{op}}, \mathbf{Set})$ **for** $\Delta \subseteq \Delta_{\mathbf{base}(P)}$
—the immediate subsumption relation $\Sigma(D)$ is the set of
—the least upperbounds of the connected components of
—$\{ E \mid D \succ E \}$, the set of diagrams strictly subsumed by D
subsorts $\Sigma \prec (D \to \mathrm{SET}(D))$ **for** $D \in \Delta_{\mathbf{base}(P)}$
operations
pp-sheaf : $\to$ PP-SHEAF
update : PP-SHEAF, IMM-SUBS, STREAM(INCREMENT) $\to$ PP-SHEAF, IMM-SUBS
axioms
—start iterative algorithm with a sheaf containing only
—the empty diagram and a subsumption relation mapping
—the empty diagram onto the emtpy set
pp-sheaf(s) = update$(\emptyset \to \emptyset, \emptyset \to \emptyset, s)$
—incremental update of the pattern-pattern-sheaf
—when the increment stream is exhausted, return the current sheaf
update(pp-sheaf, Σ, nil) = $\langle$pp-sheaf, $\Sigma\rangle$
—process one element of the stream at a time
update(pp-sheaf, Σ, cons($\mathcal{C}/P_i \xrightarrow{F} \mathbf{Set}, s$)) = update(pp-sheaf$'$, Σ', s)
where pp-sheaf$'$ = pp-sheaf $\wedge$ $\Sigma' = \Sigma$ **except**
—update parses of $\mathcal{C}/P_i$
pp-sheaf$'(\mathcal{C}/P_i)$ = pp-sheaf$(\mathcal{C}/P_i) \cup F(\mathcal{C}/P_i)$ **and**
—populate the sheaf for all new diagrams generated by $\mathcal{C}/P_i$
—using the distributive law; Theorem 5.2[19]
$\forall D \in \mathrm{dom}(\text{pp-sheaf}) \cdot$
pp-sheaf$'(D \cup \mathcal{C}/P_i) = \text{pp-sheaf}(D) \underset{\text{pp-sheaf}(D \cap \mathcal{C}/P_i)}{\times} F(\mathcal{C}/P_i)$
—update the "immediately subsumed" relation
$\Sigma'(\mathcal{C}/P_i) = \emptyset$
$\forall D \in \mathrm{dom}(\text{pp-sheaf}) \cdot$
$D \neq \emptyset \Rightarrow \Sigma'(D \cup \mathcal{C}/P_i) = \{D\} \cup \{ E \cup \mathcal{C}/P_i \mid E \in \Sigma(D) \}$
end

[19]See footnote on page 128.

5.3.30 Dependency-directed backtracking: Algorithm

We now add dependency-directed backtracking to the specification BTF-CONES by replacing the collection of active states by generators. We will need some preliminary definitions.

First, we need an operation to transfer a cone defined on a diagram Π_k to a cone defined on the diagram Π_ℓ, where Π_ℓ is subsumed by Π_k. This operation corresponds to sliding the pattern in the original Knuth-Morris-Pratt algorithm. The definition below is a little complicated; however, the intuition of sliding is straightforward.

DEFINITION 5.19: *Slide.* Let ν be an occurrence (i.e., cone) on a sub-pattern Π_k, and let Π_ℓ be another sub-pattern subsumed by Π_k, i.e., $\Pi_k \succeq \Pi_\ell$. Subsumption implies that the underlying diagrams of the two sub-patterns are related by $\Upsilon(\Pi_\ell) \subseteq \Upsilon(\Pi_k)$, which establishes a correspondence $i\colon \Pi_\ell \mapsto \Pi_k$ between the two sub-patterns. The sliding of the occurrence ν from Π_k to Π_ℓ, written "$\text{slide}_{\Pi_k \succeq \Pi_\ell}(\nu)$," is just the restriction of the cone ν along the arrow i. □

Second, we need to extend the subsumption relation from sub-patterns to partial occurrences, and then to states in the search space of BTF-CONES.

DEFINITION 5.20: *Subsumption of states.* A partial occurrence ν_D subsumes another partial occurrence τ_E, written $\nu_D \succeq \tau_E$, if and only if $D \succeq E$ (subsumption of diagrams, definition 5.18) and $\tau = \text{slide}_{D \succeq E}(\nu)$.

A state[20] $\langle \nu_D, s \rangle$ subsumes another state $\langle \tau_E, t \rangle$, written $\langle \nu_D, s \rangle \succeq \langle \tau_E, t \rangle$, if and only if $\nu_D \succeq \tau_E$ (subsumption of partial occurrences, see above) and $s = t$. □

We now proceed to replace the set of active states in the depth-first search of BTF-CONES by generators. Consider a state $\langle \nu_D, \text{cons}(i, s) \rangle$. If the increment i expands the partial occurrence ν to give τ, then we need not add any new states to the set of active states; the new states which would be added via alternative parses of the increment i are subsumed by $\langle \tau, s \rangle$. However, if the increment i does not expand the partial occurrence ν, then we cannot simply discard it; the state $\langle \nu, s \rangle$ will not subsume any state which incorporates i in its partial occurrence. The appropriate additions to the set of active states is the set of states immediately subsumed by the original state:

$$\{ \langle \tau_E, \text{cons}(i, s) \rangle \mid E \in \Sigma(D), \tau = \text{slide}_{D \succeq E}(\nu) \}.$$

All other states subsumed by the original state can be generated from this set. If the increment i does not expand some τ, then it is tried with another set of subsumed states. This process will ultimately stop because any increment always expands the empty partial occurrence, the bottom element of the subsumption relation.

[20] This refers to the states in the search space of BT-CONES or BTF-CONES. A state is a pair, consisting of a partial occurrence together with the rest of the input stream.

In other words, every increment in the input stream is "accounted" for. At every stage, the collection of active states is capable of generating all states obtainable from any combination of parses of the input stream.

spec DDBT-CONES($c :: \text{Obj}(\mathcal{C}), P :: \text{SIEVE}(\mathcal{C})$) =
—*dependency-directed backtracking implementation of cone computation*
extend GS-CONES-1(c, P)
operations
ddbt-cones : STREAM(INCREMENT) → SET(CONE)
dfs-ddbt : SD, SET(CONE), SET(SD) → SET(CONE)
pick-arc : SET(SD) → SD, SET(SD)
pick-bt : SET(SD) → SD, SET(SD)
filter : SD → BOOL
axioms
—*start search at the initial state: the empty cone and the input stream*
ddbt-cones(s) = dfs-ddbt($\langle 0, s\rangle, \emptyset, \emptyset$)
—dfs-ddbt *takes three arguments: the current state being explored,*
—*a partial set of solutions, and a set of states yet to be explored*
dfs-ddbt(*cur-state*, *Solutions*, *Active-States*) =
let $\langle \nu_D, \text{cons}(i, s)\rangle$ = *cur-state*
—*process current state: add solutions extractable from current state*
and *New-Solutions* = **if** $D = \mathcal{C}/P$ **then** *Solutions* $\cup \{\nu\}$ **else** *Solutions*
—*generate next states*
and *New-States* = $\{\, \langle \tau_E, s\rangle \mid \tau = \nu \otimes \text{prim-cones}(F), F \in \Phi(\text{base}(i)) \,\}$ **in**
if *New-States* = $\emptyset \vee$ filter(*cur-state*) **then**
—*current state cannot be expanded or can be pruned*
—*add subsumed states to the list of active states*
—*discard current state; backtrack to another*
let *New-Active* = *Active-States* $\cup$
$\{\, \langle \tau_E, \text{cons}(i, s)\rangle \mid E \in \Sigma(D), \tau = \text{slide}_{D \succeq E}(\nu) \,\}$ **in**
if *New-Active* = $\emptyset$ **then** —*nowhere to backtrack; return solutions*
New-Solutions
else —*pick an unexplored state to backtrack to*
let $\langle$*new-state*, *New-Active*$\rangle$ = pick-bt(*New-Active*) **in**
dfs-ddbt(*new-state*, *New-Solutions*, *New-Active*)
else —*proceed depth-first: pick one of the next states to explore*
let $\langle$*new-state*, *New-Active*$\rangle$ = pick-arc(*New-States*) **in**
dfs-ddbt(*new-state*, *New-Solutions*, *Active-States* $\cup$ *New-Active*)
—*axioms for* "pick-arc", "pick-bt", *and* "filter" *are as in* BTF-CONES
end

5.3.31 The failure function

In the standard version of the Knuth-Morris-Pratt algorithm [Knuth et al. 77], which works on strings with a left-to-right traversal of the target, only those partial occurrences are considered which touch the right edge of the portion of the target already traversed. Hence, a partial occurrence can be expanded *in only one direction.* Thus, for any partial occurrence which cannot be expanded, i.e., the next character is a mismatch, we can test whether the next immediately subsumed partial occurrence offers a new opportunity for expansion; otherwise, the subsumed partial occurrence can be rejected in favor of a smaller one. Here is the appropriate picture, where the first subsumed occurrence can be rejected since it does not offer a new choice:

target	*a b c a a b c*
matches	✓✓✓✓×
pattern	*a b c a b c*
1st shift	*a b c a b c*
2nd shift	*a b c a b c*

The subsumption relation, after this precomputation has been performed, is called the *failure function.*

In a generalized version of the Knuth-Morris-Pratt algorithm, since partial occurrences can be expanded in more than one direction (consider the site of graphs), the failure function is a multifunction, which computes a subsumed occurrence for *each direction* in which the partial occurrence may be expanded. The directions of expansion depend upon the geometry of the underlying data structures; this part of the geometry is not axiomatized in a Grothendieck topology. Hence, we omit this additional precomputation, relegating it to data-structure-specific optimizations which are done after the generalized algorithm is instantiated.

5.3.32 Assessment

In the last few sections, we have optimized depth-first search in the space of cones by replacing the set of active states with a generator. On scrutiny, this optimization is independent of the search strategy. The optimization represents the entire search space more compactly by exploiting the subsumption relation between states; thus any search strategy which can be used in the original space can also be used in the compressed space, provided non-existent states are always generated when the search strategy requires (demands) them.

5.3.32.Alt.1 Breadth-first search with demand-driven state generation

We now show how the breadth-first search strategy in the space of partial occurrences can be modified to accommodate the representation of states via generators.

spec BFS-CONES$(p, c :: \mathrm{Obj}(\mathcal{C}), \langle \mathcal{C}, J \rangle ::$ FINITARY-NOETHERIAN-PM-SITE$) =$
extend SUBSUMPTION$(p, \langle \mathcal{C}, J \rangle)$, GS-CONES-1$(c, P)$ —$P = \mathrm{decompose}(p)$
operations
bfs-cones : STREAM(INCREMENT) → SET(CONE)
bfs : STREAM(INCREMENT), SET(SD), SET(CONE) → SET(CONE)
expand : SET(SD) → SET(SD)
axioms
—*start search at the initial state:*
—*the empty cone and the input stream*
bfs-cones(s) = bfs$(s, \{\langle 0, s \rangle\}, \emptyset)$
—"bfs" *takes three arguments: the rest of the stream*
—*the set of active states, and a partial set of solutions*
—*the first argument is just a convenience*
—*the algorithm proceeds by processing one increment at a time*
—*all currently active states contain the same stream*
bfs$(s,$ *Active-States*, *Solutions*$)$ =
—*extract full occurrences from active states*
let *New-Solutions* = *Solutions* ∪
$\{ \nu \mid \langle \nu_{C/P}, t \rangle \in$ *Active-States* $\}$ **in**
—*return solution when input stream is exhausted*
if $s = \mathrm{nil}$ **then** *New-Solutions* **else**
—*expand all active states and continue*
bfs$(\mathrm{tl}(s),$ *New-Active*, *New-Solutions*$)$
where *New-Active* = $/ \cup \circ\alpha(\mathrm{expand}_{\mathrm{hd}(s)})($*Active-States*$)$
—*expanding a state*
—*if current state can be filtered, then fail back to subsumed states*
expand$_i(x)$ =
if is-full-occurrence$(x) \vee$ dead-end(x) **then** $\Sigma(x)$
else let *New-States* =
$\{ \langle \nu \otimes G, \mathrm{tl}(s) \rangle \mid x = \langle \nu, s \rangle, G = F^{\#}, F \in \Phi(\mathrm{dom}(i)) \}$ **in**
—*if current state can be expanded, return expansions*
if *New-States* $\neq \emptyset$ **then** *New-States*
—*otherwise, fail back to subsumed states*
else $\Sigma(x)$
end

5.3.33 The generalized Knuth-Morris-Pratt algorithm

The breadth-first search algorithm above, together with the computation of the subsumption relation, comprises the generalized Knuth-Morris-Pratt algorithm. It implements the pattern matching specification PATTERN-MATCH as follows.

$$\text{PATTERN-MATCH} \rightsquigarrow \kappa\text{-KMP}(\text{KMP-PATTERN-MATCH})$$

where

spec KMP-PATTERN-MATCH($\langle \mathcal{C}, J \rangle$ :: FINITARY-NOETHERIAN-PM-SITE) =
extend BFS-CONES($-, -, \langle \mathcal{C}, J \rangle$)
extend CONE-DECOMPOSITION-1 **with**
decompose **renamed as** decompose . CONE
operations
kmp-occs : $\text{Obj}(\mathcal{C}), \text{Obj}(\mathcal{C}) \rightarrow$ SET(OCCURRENCE)
decompose : $\text{Obj}(\mathcal{C}) \rightarrow$ SIEVE($\mathcal{C}$)
axioms
kmp-occs$(p, t) = \alpha(\text{glue}_P) \circ \text{bfs-cones}_{1,p}(s)$
where $P = \text{decompose}(p)$, $s = \text{traverse}(t)$
—decomposition yields the finest covering sieve
$\text{decompose}(p) \in J(p) \wedge \forall R \in J(p) \cdot \text{decompose}(p) \subseteq R$
—computation of cones on $\mathcal{C}/Q_j$ is done by "dc-cones"
$\text{prim-cones}_{c,P}(F) = \text{dc-cones}(c, F)$
end

and the constructor κ-KMP is defined by

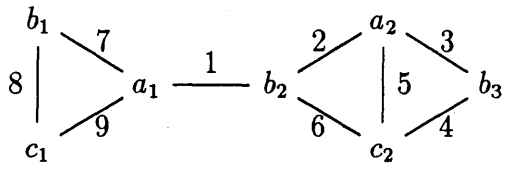
constructor κ-KMP =
derive from KMP-PATTERN-MATCH
by { occurrences $\mapsto$ kmp-occs }
end

The specification KMP-PATTERN-MATCH can be somewhat simplified by removing the dependencies on the pattern cover P, since this cover is implicitly generated by traversal. Moreover, the functions "glue" and "prim-cones" can be specified independently of the pattern cover (but dependent on the underlying site). We omit the details, and use the definition

$\text{kmp-occs}(p, t) = \alpha(\text{glue}) \circ \text{bfs-cones}_{1,p}(s)$
where $s = \text{traverse}(t)$

as the basis of the generic algorithm which will be specialized to strings and graphs in the next chapter.

Traversal order:



Partial occurrence:

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. $a_1b_2,\ c_2a_2b_2c_2,\ a_2b_3c_2,\ b_3c_2a_2,\ c_2a_2b_3$ —*expand* $c_2a_2b_2$
   —*eliminate* $c_2a_2b_2c_2,\ a_2b_3c_2,\ b_3c_2a_2,\ c_2a_2b_3$
   —*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. $a_1b_2,\ a_1b_1c_1a_1$ —*expand* $a_1b_1c_1$
   —*end of algorithm*
   —*traversal dead-end filter removes all partial occurrences*

Figure 5.12: Trace of generalized KMP on graphs (cf. figures 3.2 and 5.9)

Chapter 6
Future Work: Applications of a Sheaf-Theoretic View

The woods are lovely, dark, and deep,
But I have promises to keep,
And miles to go before I sleep,
And miles to go before I sleep.
— Robert Frost, *Stopping by Woods on a Snowy Evening* (1923)

In this chapter, we investigate the effects of relaxing some of the simplifying assumptions we made during the derivation of the Knuth-Morris-Pratt algorithm. Relaxing these assumptions does not debilitate the derivation; instead, we get several related algorithms, comprising the design space around the main derivation. Some of these relations are obvious (e.g., matching with multiple patterns); some, which were only hinted before [Knuth et al. 77, Hoffmann and O'Donnell 82], are rendered explicit (e.g., the connection with LR-parsing and Earley's algorithm for context-free parsing); some others were a complete surprise even to the author (e.g., Waltz filtering for image analysis[1]). The richness of the design space, in spite of a simple set of axioms, is testimony to the power of Grothendieck's machinery.

6.1 More general patterns

The sheaf-theoretic view of pattern matching can handle more general kinds of patterns: patterns with variables, multiple patterns, pattern matching modulo commutativity/associativity, etc. by defining the underlying categories and topologies appropriately.

Suppose we have two patterns, p and q, and want to find all the occurrences of p or q in a target. This can be done by taking the union of the occurrences of p and q. We thus have a logical operation "$p \vee q$" on patterns, defined by

$$\hom(p \vee q, -) = \hom(p, -) \cup \hom(q, -).$$

[1] I thank Dr. Ira Baxter, a fellow graduate student, for pointing out the similarity between the Knuth-Morris-Pratt algorithm and Waltz filtering.

Applying our KMP derivation using the patterns p and q (individually), we get two algorithms for finding occurrences. Occurrences of $p \vee q$ can then be computed by the equation above. However, this involves traversing the target twice. To avoid this, we interleave the two algorithms: the target is traversed only once, but each piece of the target is parsed according to both p and q. We thus get a combined search space of partial occurrences of both p and q as well as a failure function which depends on both patterns. We thus get the Aho-Corasick version of the KMP algorithm for multiple patterns [Aho and Corasick 75].

An alternative way to incorporate disjunction of patterns is to modify the underlying site. Corresponding to any pair of objects, p and q, we add a new object $p \vee q$ together with two new arrows $\pi_1 : p \vee q \to p$ and $\pi_2 : p \vee q \to q$. Corresponding to every arrow $u \to p$, we add a new arrow $u \vee q \to p \vee q$; similarly, an arrow $v \vee p \to p \vee q$ for each arrow $v \to q$. The arrows are then closed under composition. It is clear that $p \vee q$ behaves as a product and satisfies the identity on hom-sets given above.[2] We define a cover of $p \vee q$ to be a product (in the category of sieves) of covers of p and q.

Other operations on patterns, such as $p \wedge q$, can be modeled similarly. The direct characterization, using operations on hom-functors is more convenient for deriving algorithms. We have presented the alternative of modifying the site so as to connect these operations to variables, and context-free parsing.

An untyped variable can be modeled as the pattern $p \vee q \vee r \vee s \cdots$ (infinite disjunction). Thus, a variable is represented by the collection of all its instances (its extension). In a pattern matching algorithm, since a variable matches anything, for every piece of the target t_i, we always have the parse $v \to t_i$. Alternatively, a variable can be added to the underlying site, along with new arrows to every object in the site. Thus, a variable behaves as an initial object, and satisfies the equation

$$\hom(v, -) = \hom(p, -) \cup \hom(q, -) \cup \hom(r, -) \cup \cdots \text{(inifinite union)}.$$

Typed variables can be modeled by only adding arrows to objects in their extension.

6.2 Context-free parsing: Earley's algorithm

Earley's algorithm for context-free parsing [Earley 70] 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 on the compatibility of the increment with the parses.

Earley's algorithm has a dual nature: it can be seen either as a top-down approach (unfolding grammar rules, predicting the next increment, and testing against the

[2]The reason why $p \vee q$, a disjunction, behaves as a product is because of the contravariance of the occurrence sheaf. The object $p \vee q$ is trying to represent the union of occurrences. In the same spirit, $p \wedge q$, representing the occurrence of both p and q, is modeled as a coproduct.

input), or as a bottom-up approach (expanding the current left contexts using the next increment, possibly accompanied by folding of grammar rules). We explain this duality, and also the close connection with the KMP algorithm, using two different topologies.[3]

Bottom-up

The goal of parsing is to assign a parse to a target string. A parse is a derivation of the target string from the start symbol of the grammar. A derivation is a sequence of applications of productions in the grammar. In the bottom-up approach to parsing, we choose a cover of the target string, compute all partial parses of each piece of the target, and glue together compatible partial parses to obtain one or more parse of the entire string. Rather than give detailed definitions, we illustrate these concepts via an example in figure 6.1. The relevant sheaf is the sheaf of partial parses of strings. The resulting topology on parses is close to the topology described in [Eytan 80].

It can be seen that the frontier of a partial parse tree is a sentential form divided into three parts: the middle part is a terminal string, the other two parts are the left and right contexts. A partial parse tree is usually represented using dotted productions. Applying our KMP derivation to the sheaf of partial parses, and using a left-to-right traversal of the target string, we see that the partial parses will always have their left contexts filled out. Thus we need only consider partial parses with right contexts missing: these correspond to the item sets in Earley's algorithm.

Top-down

In a top-down approach, we view the parsing problem as matching with an infinite collection of patterns: the collection of strings which constitute the language generated by the grammar. The problem is not so much to enumerate all the occurrences of these patterns as to determine *which* patterns match the (entire) target. Thus, we can apply the generalization of KMP to multiple patterns. However, this poses a problem because the collection of patterns is infinite. Hence we use the standard technique of representing this collection intensionally.

The grammar provide the requisite intension. Consider a rule of the form:

$$S \rightarrow ABC|CBA.$$

This rule says that the collection of patterns represented by S is union of those represented by ABC and CBA. Thus, we use the generalization of KMP for two patterns. Next, to find occurrences of the patterns represented by ABC, we can use the standard version of KMP (choose a cover of ABC, finds occurrences of each piece,

[3]Compare our approach to the derivation of Earley's algorithm given in [Partsch 84], where the notion of item sets is introduced in an unmotivated embedding of the specification of parsing into a more general specification.

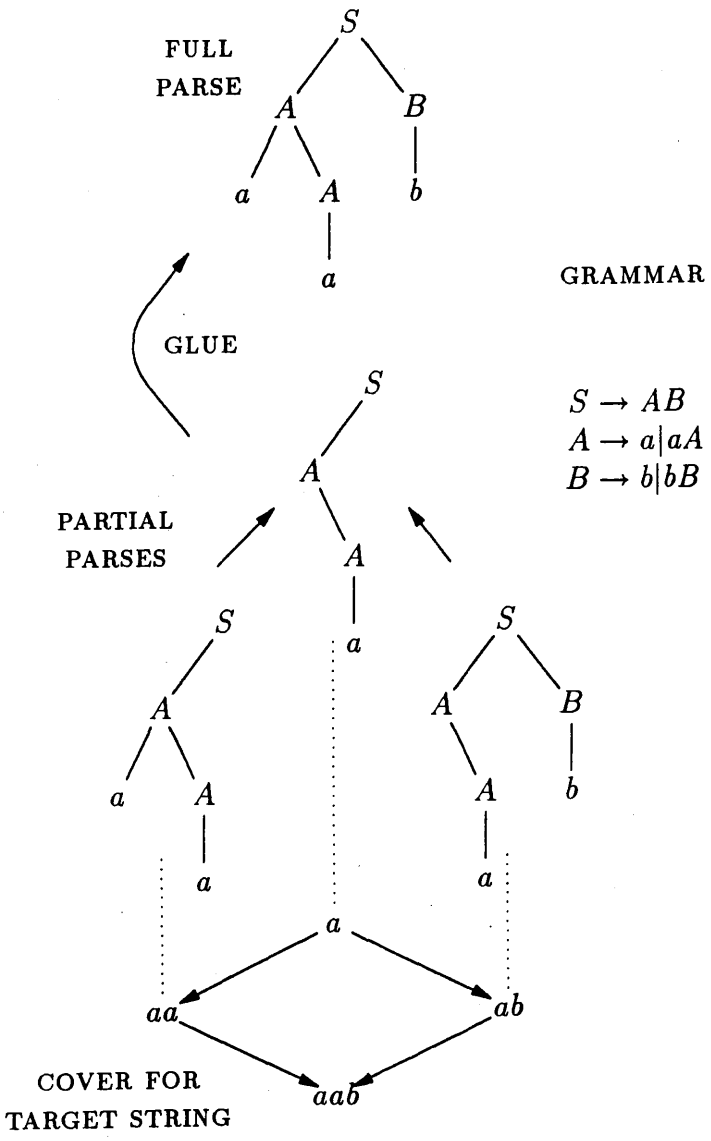


Figure 6.1: A sheaf-theoretic view of context-free parsing

etc.). We continue alternating between logical operations (alternation in a grammar rule) and geometric operations (decomposing a sentential form) until we reach a sentential form which starts with a terminal symbol. This can be tested against the target string. Depending on the result, we either discard some patterns, or proceed to match the next piece of the sentential form.

Implementing the alternation operator (|) by a breadth-first exploration of all alternatives, we see that the item sets of Earley's algorithm represent the collection of all patterns which can potentially match the (entire) target.

6.3 Constraint propagation: Waltz filtering

Relaxation algorithms for constraint propagation can be described by sheaves when the constraints are local. An example is Waltz filtering [Waltz 75], an algorithm which assigns three-dimensional interpretations to two-dimensional line drawings of scenes. The underlying site is that of undirected, connected graphs; 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, L-junctions, T-junctions, forks, etc., are precomputed by using physical properties of three-dimensional space.

The algorithm works by choosing a junction cover (see figure 6.2) 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, then 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 2.36).

6.4 Non-local properties

A sheaf is a useful structure to work with if we are interested in modeling local properties. However, some problems may contain some non-local properties. Consider the property of being a monic arrow in the category of graphs. If we only consider the finest covers for graphs, this is a non-local property, in that a compatible collection of monic arrows on a cover does not yield a monic arrow on the entire graph. This is because the property of being monic requires that no *two* edges in the domain of the arrow be mapped onto the same edge in the codomain. Since the finest cover of a graph consists of individual edges, it cannot detect the incompatibility between maps on *pairs* of edges.

There are two ways to handle such non-local properties in the sheaf characterization of a problem: in the definition of a cover, or in a filter which removes "pseudo-compatible" families. In the example above, we can redefine covers so that two edges

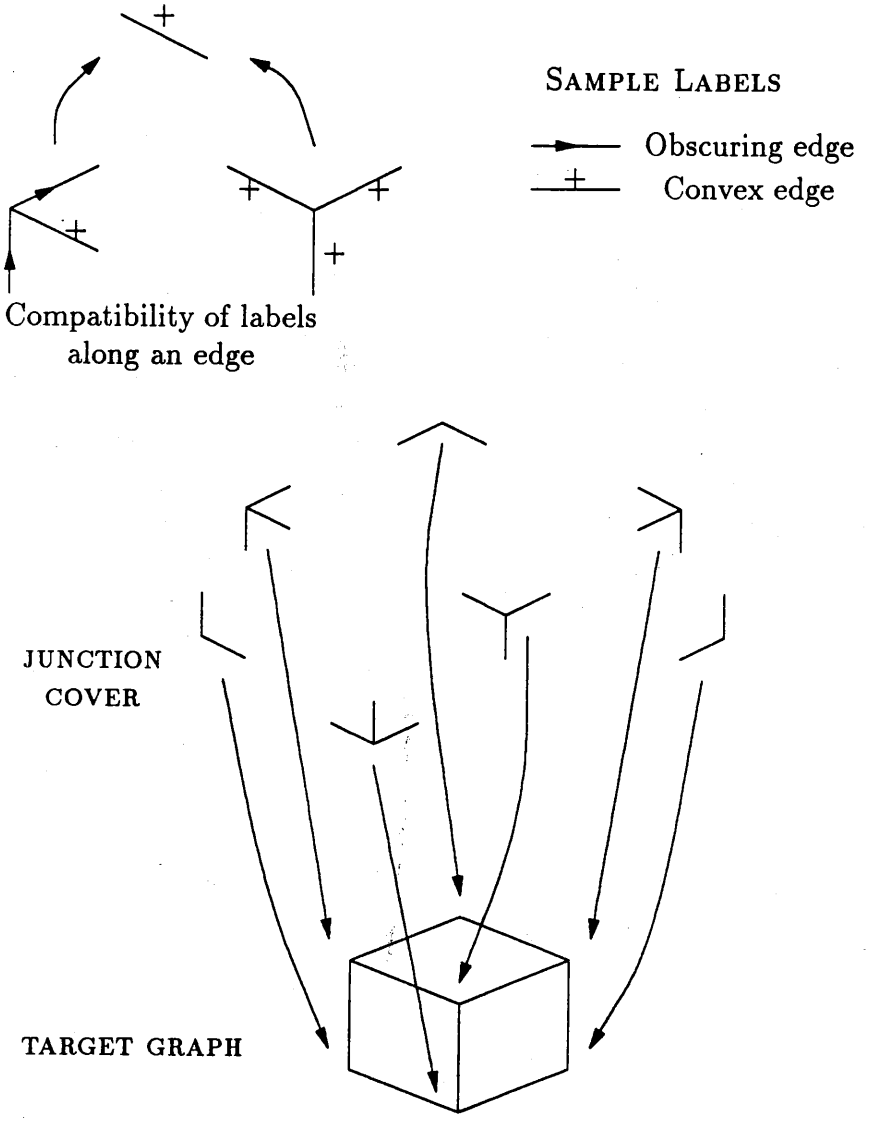


Figure 6.2: A sheaf-theoretic view of Waltz filtering

which can be potentially mapped onto the same edge are never split up in a cover, i.e., the grain of locality of the property of being monic is pairs of edges. We can also work with separated presheaves (resulting in a partial gluing function) and examine compatible families before trying to glue them. Thus, although the occurrence relation defined by monics between graphs does not yield a sheaf, we can work as if it did, and apply the KMP derivation, provided we include a test before gluing compatible families. This is also the appropriate place to handle semantic properties such as those which arise in conditional pattern matching.

6.5 Rewriting

A rewrite rule can be considered as a pair of patterns whose pieces are related by editing relations such as preserve, add, and delete. The geometry of patterns induces a geometry of rewrites. Not only does this view provide a uniform treatment of rewriting (on strings, trees, graphs, hypergraphs, etc.), but it also allows us to decompose rewrites into pieces and glue together these pieces.

Typically a rewrite rule is a pair of patterns, e.g.,

$$x + y \rightarrow x - y.$$

There are several aspects of such a rewrite which are implicit. First, the editing operations—what changes and what remains the same—are implicitly encoded by using the same variable names in both sides. Second, it is implicitly assumed that the part of the target which is outside the binding of the pattern remains unchanged after the rewrite.

To handle the connections of the rest of the target to the rewritten portion (for graph rewriting), Ehrig introduces the notion of gluing points: the target graph is expressed as a pushout of the pattern and a complement graph [Ehrig 78]. This approach can be generalized by defining rewrites at the level of covers. Thus, a rewrite rule would be a pair of covers (for the left and right patterns) together with a description of which parts are preserved, added, and deleted. A rewrite then consists of the following steps:

1. explode the target into pieces, i.e., find a cover which matches that of the left-hand-side pattern;
2. edit the target cover as specified by the rewrite rule; this is a double pushout (as in [Ehrig 78]); and
3. reglue the revised cover to obtain the rewritten target.

By working at the level of covers, we abstract away from the underlying data structures involved, and provide a uniform treatment of rewriting for all sites. This

approach also allows us to decompose rewrites into smaller pieces and investigate relationships among them, e.g., the independence/commutativity of two rewrites. Exploiting the geometry of the data structures is sometimes more efficient [Baxter 90] than using algebraic properties of rewrites (as in [Ehrig 78]).

6.6 Induction and computability

The sheaf condition, which allows us to pass from properties defined on elements of a cover to properties defined on the entire object, is essentially a general form of induction. Given that induction is the only general principle we have for reasoning about infinite collections of entities, the question arises whether computability has a sheaf-theoretic characterization.

Gandy's paper on principles for mechanisms [Gandy 80][4] comes close to a sheaf-theoretic characterization of computability. In investigating what it is "to compute," Gandy goes beyond Turing's machine, and proposes five principles for mechanisms, principles that any mechanism performing a computation should satisfy. These principles assume a state transition model. The important principles are that the next state is obtained by examining a finite amount of information from the current state, and that the next state is built inductively/reductionistically out of pieces of the current state. Gandy observes at the end of his paper that his principles can be formulated using sheaf theory, but he didn't use sheaf theory for expository reasons.

Since sheaf theory is a combination of geometric and algebraic aspects, we ask whether there is a connection between computation and geometry. This connection can be illustrated by an example. Consider a typical algebraic equation such as:

$$(a+b)^2 = a^2 + 2ab + b^2.$$

Let us analyze the meaning of this equation. First, there are a collection of symbols: a, b, 2, $+$, etc. Second, these symbols can be *spatially* arranged in different configurations (as letters on paper, as dots on a screen, as frequency modulation of a wave, etc.). Third, we assign a denotation to each configuration. Fourth, we assert that the denotations assigned to certain pairs of configurations are the same (in this case the two expressions on either side of the equation).

The first two are geometric; they are formal, meaningless manipulations. The last two are algebraic; they allow us to reason about such manipulations.

Formal symbol manipulation is meaningless; to exploit it for some purpose, we should be able to determine the meanings. Computation is geometric manipulation which is meaningful, i.e., which can be inductively reasoned about (finite induction). Given a collection of symbols, legal configurations are inductively defined. The denotation of a configuration is inductively defined in terms of denotations of its pieces (compositional semantics).

[4] See also [Shepherdson 88, Dahlhaus and Makowsky 88].

Different formulations of computability use different geometric entities as their basis: strings (Post systems), trees (lambda-calculus), arrays (Turing machines, also multi-dimensional arrays), etc. The common characteristic is induction and compositional semantics. I conjecture that sheaf theory can be used to precisely state that the induction (sheaf condition, together with a well-founded order on the site) is primary, and the site is just a parameter. This is a much better approach than having several different characterizations of computability and then proving that any two are equivalent.

Chapter 7
Related Work

"The time has come," the Walrus said,
"To talk of many things:
Of shoes—and ships—and sealing-wax—
Of cabbages—and kings—
And why the sea is boiling hot—
And whether pigs have wings."
— Lewis Carroll, *Through the Looking-Glass* (1871)

We divide the work related to this dissertation into two parts: other derivations of the Knuth-Morris-Pratt algorithm, and other uses of sheaf-theoretic methods in computer science.

7.1 Other derivations of KMP

We consider five other derivations of KMP which have appeared in the literature: [Dijkstra 76, Bird et al. 89, van derWoude 89, Morris 90, Partsch and Völker 90]. All these derivations derive the algorithm for strings, represented as arrays or lists, and are difficult to generalize because of their dependence on properties of strings. Their explanation of the failure function is also somewhat inadequate.

To compare these derivations to ours, we list the major design decisions which characterize KMP (these are not mutually exclusive, and the order of presentation below does not reflect logical dependencies):

- The assembly of an occurrence from pieces (divide-and-conquer).
- Searching in the target.
- The invention of the notion of a partial occurrence.
- Incremental building of partial occurrences.
- Exploiting relationships between partial occurrences.
- The invention of the notion of "sliding" the pattern and the failure function.
- Precomputing the failure function by matching the pattern against itself.

7.1.1 Specification of pattern matching as search

The starting specifications for the derivations in [Dijkstra 76, van derWoude 89, Morris 90, Partsch and Völker 90] (array representation) is, for a pattern p and a target t,

enumerate all i such that match(p, t, i), with $1 \leq i \leq |t|$
where
match$(p, t, i) = \forall 1 \leq j \leq |p| \cdot p[j] = t[i + j - 1]$.

The starting specification in [Bird et al. 89] (list representation) is

match $p\ t = \exists u, v \cdot t = u +\!\!+ p +\!\!+ v$ —"++" *denotes concatenation*

where $\_ = \_$, the equality of strings is defined by

$[\,] = [\,]$ —*empty list*
"a" = "a" —*singleton list*
$(x = z \wedge y = w) \Rightarrow x +\!\!+ y = z +\!\!+ w$ —*induction step*

Two features of these characterizations bear highlighting. First, the pattern matching problem is formulated as a search problem: search for those "positions" in the target which satisfy certain properties. Second, the definitions of equality and the match relation implicitly incorporate a divide-and-conquer strategy for constructing matches.

The specifications above immediately yield generate-and-test algorithms. For the list representation, this algorithm is

—*p is a segment (i.e., a substring) of t*
match $p\ t = p \in$ segs t
where
—*segments are obtained by taking all tail segments (i.e., suffixes)*
—*of all initial segments (i.e., prefixes)*
—*/ is the "reduce" operator; $*$ is the "apply-to-all" operator*
segs $= +\!\!+/ \circ$ tails $* \circ$ inits

Observe the bias introduced by the use of a certain set of constructors, e.g., enumerate the indices $0 \ldots |t|$, or enumerate all the tails of a string. A different set of constructors for the data types involved would lead to a (possibly) different order in which the target is searched. Ideally, the search order should be an explicit decision in the derivation. The KMP algorithm does not depend on a left-to-right search order; it works equally well with a right-to-left order, or any mixture (see [Sunday 90] for a search order based on probability of occurrence).

This implicit use of constructors also surfaces when the algorithm is generalized to other data structures. The generalization of the specification

$$\text{match } p\ t = \exists u, v \cdot t = u +\!\!+ p +\!\!+ v$$

to graphs would be

$$\text{match } p\ t = \exists q \cdot t = p \underset{p \cap q}{+} q$$

i.e., the target graph is expressed as a pushout of two graphs, one of which is the pattern. Roughly speaking, the graph q is the "complement" (in t) of the graph p. With this characterization, it is not obvious how to enumerate the collection of q's.

Our characterization of the extension of the occurrence relation as a sheaf is independent of any constructors which are used to define the underlying data structures.

7.1.2 Partial occurrences

The next step is to improve the efficiency of the naive algorithm for matching. In [Dijkstra 76, Partsch and Völker 90], the authors ask the following question, a natural by-product of a derivation style using loop invariants:

> given a mismatch, what is the smallest index where the next occurrence is possible?

The appropriate picture is the following:

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

Implicit in the definition of a mismatch is the notion of a partial occurrence.

In [Bird et al. 89], the notion of a partial occurrence is introduced in an attempt to apply finite differencing to the specification

$$\text{match } p = \vee/ \circ (\text{endswith } p)* \circ \text{inits}$$

To massage the expression into an appropriate form, Bird et al. replace the function "endswith" by a function "overlap": the expression "overlap $p\ x$" returns the longest initial segment of p that ends x. Thus this step introduces partial occurrences. It also converts the recursive algorithm into an iterative one.

The derivations of [van derWoude 89, Morris 90] start with the maximal prefix-suffix problem:

> give a string x,
> for every prefix y of x,
> determine the longest string that is both a prefix and suffix of y.

This problem is a generalization of the pattern matching problem and implicitly contains the notion of a partial occurrence. The relationship with matching is (the definition also applies the function "overlap" above):

> pattern p matches a substring s of target t
> if the longest prefix of p which is also a suffix of s is p itself.

In these derivations, the generalization to the maximal prefix-suffix problem is unmotivated. In our derivation, we consider the extension of the occurrence relation as the pattern varies. This is not a generalization; it is just a description of the essential properties of the problem.

7.1.3 The failure function

The invention of the notion of partial occurrence seems to be the key step en route to the failure function. In [Dijkstra 76, Partsch and Völker 90], the problem of computing slide amounts results in a specification and sub-derivation of essentially the same problem as the maximal prefix-suffix problem. In the other derivations, since pattern matching has already been generalized, we get another instance of the original problem, leading to a recursive algorithm.

In generalizing these derivations to other data structures, the notions of "slide," "prefix," and "suffix" have to be generalized. Observe that a string spreads out in two directions, left and right, leading to the notion of "prefix" and "suffix." Other data structures, e.g., graphs, may spread out in any number of directions. This necessitates the investigation of the geometry of other data structures so as to formalize the notions of "boundary" and "sub-patterns which touch the boundary" (the latter replace prefixes and suffixes). In our derivation, it turns out that the essential notion is not sliding, but that, from a given partial occurrence, other, smaller partial occurrences can be generated. When the larger partial occurrence cannot be expanded, it may be possible to expand the smaller ones, i.e., on a mismatch, we "fail" to an alternative. Our explanation of the failure function in terms of backtracking removes the mystery of the failure function, not by introducing unmotivated generalizations (the maximal prefix-suffix problem) but by reducing it to a standard, and very general, search strategy.

7.1.4 General remarks

We now summarize the major steps in our derivation to highlight the differences with other derivations. We start with an extensional characterization of the pattern matching problem: this removes the bias introduced by using constructors to define the underlying data structures; it also emphasizes the geometric nature of the problem. We then systematically try to convert this extension into an intension. We start with a divide-and-conquer strategy: to find an occurrence of the pattern, find a piece of the target corresponding to each piece of the pattern. This algorithm is made incremental by traversing the target. This step inverts the computation: each piece of the target is "parsed" as a piece of the pattern. It is this notion of parsing that leads to the pattern-pattern-sheaf and the precomputation of the failure function.

The incremental assembly of partial occurrences is then formulated as a search problem: find a full occurrence in a space of partial occurrences (alternatively, find consistent collections of parses of pieces of the target). We use a standard depth-first search strategy with backtracking for searching in this space: expand the current partial occurrence; if expansion is not possible, backtrack to another choice. The pattern-pattern-sheaf defines a partial order on partial occurrences: a partial occurrence is subsumed by another if it can be generated by a different set of parses. Using this subsumption relation, the set of choices for backtracking is replaced by a generator (lazy list): alternatives are generated on demand. The failure function thus corresponds to backtracking to an alternative partial occurrence.

The major difference between our derivation and others is the search space: the space of partial occurrences vs. the space of "positions" in the target, and the consequent explanation of the failure function as backtracking. Once the pattern matching problem is formulated as finding a consistent collection of parses of pieces of the target, the invention of the pattern-pattern-sheaf and backtracking to improve efficiency become inevitable.

The ideal derivation is one in which the final algorithm *inevitably* follows from the specification and given efficiency considerations. We do not claim to have such an ideal derivation; however, our derivation comes closer than others. The derivations of KMP in the literature convincingly demonstrate that the KMP algorithm can be obtained from a formal specification via a sequence of formal transformations. However, the important problem is how to *invent* this sequence of transformations. To this end, alternatives for each transformation have to be explored and the choice of a particular one justified. The derivations in the literature rarely explore alternatives, and sometimes contain leaps (e.g., unmotivated generalizations). In this dissertation, we have attempted to explore alternatives and thus chart the design space surrounding the main derivation path. Other paths in the space lead to interesting algorithms, as shown in Chapter 6.

Finally, we quote from [van derWoude 89] on the resemblance between the preprocessing of the pattern and the main algorithm:

Chapter 8
Summary and Contributions

I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I —
I took the one less traveled by,
And that has made all the difference.
— Robert Frost, *The Road Not Taken* (1916)

8.1 Summary

This dissertation has presented a generic theory of pattern matching and derived a generalized version of the Knuth-Morris-Pratt algorithm. The theory and the derivation are generic in the sense that they apply to any data structure.

A generic theory of pattern matching is developed by emphasizing the geometry of the data structures involved. The geometry is formalized as a Grothendieck topology. The central concept of a Grothendieck topology is that of a cover: the decomposition of an object into a collection of pieces such that the pieces can be glued back together to obtain the original object.

A simple view of matching is adopted, wherein a pattern is said to occur in the target if the pattern is exactly the same as some sub-structure of the target. The extension of the occurrence relation is considered for a fixed target and all possible patterns. This extension forms a sheaf:

1. associated with each pattern p is a set of occurrences of p in the target t;
2. if q is a piece of p, then corresponding to every occurrence of p there is an occurrence of q by restriction;
3. given a cover of p, i..e, a decomposition of p into a set of pieces $\{ p_i \mid i \in I \}$, and given occurrences of each p_i in t, (i.e., partial occurrences of p), such that these occurrences agree with each other whenever two pieces p_i and p_j intersect, the partial occurrences can be glued together to give an occurrence of p.

The derivation of the generalized Knuth-Morris-Pratt algorithm starts with the extension of the occurrence relation and exploits the sheaf condition (item 3 above)

> Several people have noticed the strong resemblance of those parts [preprocessing of the pattern and search in the target], but in the literature we searched in vain for a presentation or derivation (at all) of the algorithm that did justice to that resemblance. ... Since the two parts are almost identical, such a statement is puzzling. ... In our opinion, exploitation of pre- and postfixes simplified the "derivation" of the algorithm ...

Woude achieves the similarity by starting with a generalized problem, the maximal prefix-suffix problem. Note that the preprocessing step of KMP, which computes the slide amount for each piece of the pattern, *is* the maximal prefix-suffix problem. Thus, it is not surprising that the two parts of the algorithm (and their derivations) are similar.

We provide a somewhat deeper explanation of this similarity. First, the two parts of the algorithm are *different:* one is maximal prefix-suffix, the other is matching. What is similar is that both the preprocessing and the matching can be extensionally described by sheaves and the same derivation can be applied to both sheaves: divide-and-conquer, finite differencing, backtracking. The derivations diverge at the end, resulting in the slight difference in the two parts: one argument (pattern against pattern) vs. two arguments (pattern against target).

7.2 Related formal methods

With the increased emphasis on formal methods in computer science, a considerable number of derivations have appeared in the literature. However, most of these derivations tend to list a sequence of transformations from a specification to a program, with little discussion of the rationale behind choosing those transformations. Of the papers which go a little beyond and develop deeper theories, we cite two: [Smith 85] which develops a theory of divide-and-conquer algorithms, and [Partsch 86] which the develops the theory of context-free parsing algorithms.

Category theory is becoming more prevalent in computer science; see, for example the proceedings of the conferences on Category Theory and Computer Science, *Lecture Notes in Computer Science,* Volumes 240, 283, and 389. The use of category theory has been particularly successful in algebraic specification and type theory.

There are only a few uses of sheaf theory in computer science. Concurrent processes are modeled using sheaves in [Monteiro and Pereira 86, Ehrich et al. 91]. Michel Eytan [Eytan 82, Eytan 80] uses Grothendieck topologies on context-free grammars to model similarity of derivations.

Chapter 8
Summary and Contributions

I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I —
I took the one less traveled by,
And that has made all the difference.
— Robert Frost, *The Road Not Taken* (1916)

8.1 Summary

This dissertation has presented a generic theory of pattern matching and derived a generalized version of the Knuth-Morris-Pratt algorithm. The theory and the derivation are generic in the sense that they apply to any data structure.

A generic theory of pattern matching is developed by emphasizing the geometry of the data structures involved. The geometry is formalized as a Grothendieck topology. The central concept of a Grothendieck topology is that of a cover: the decomposition of an object into a collection of pieces such that the pieces can be glued back together to obtain the original object.

A simple view of matching is adopted, wherein a pattern is said to occur in the target if the pattern is exactly the same as some sub-structure of the target. The extension of the occurrence relation is considered for a fixed target and all possible patterns. This extension forms a sheaf:

1. associated with each pattern p is a set of occurrences of p in the target t;
2. if q is a piece of p, then corresponding to every occurrence of p there is an occurrence of q by restriction;
3. given a cover of p, i..e, a decomposition of p into a set of pieces $\{ p_i \mid i \in I \}$, and given occurrences of each p_i in t, (i.e., partial occurrences of p), such that these occurrences agree with each other whenever two pieces p_i and p_j intersect, the partial occurrences can be glued together to give an occurrence of p.

The derivation of the generalized Knuth-Morris-Pratt algorithm starts with the extension of the occurrence relation and exploits the sheaf condition (item 3 above)

to obtain a divide-and-conquer algorithm. An incremental, sequential algorithm is then obtained by applying finite-differencing to the divide-and-conquer algorithm:

- the target is traversed, producing a stream of information about partial occurrences of the pattern;
- a cache of partial occurrences is maintained and updated for each increment from the stream.

This algorithm is quite inefficient because it saves too many partial occurrences in the cache. An immediate optimization is to save only those occurrences which have a possibility of being expanded.

A second optimization is to update the cache lazily. The problem of updating the cache can be recast as a search problem: the state space consists of partial occurrences, and the goal is to find a full occurrence. Lazy updating now corresponds to dependency-directed backtracking. Specifically, the largest partial occurrence is saved (a greedy, or hill-climbing strategy) along the edge of the portion of the target already explored. When it is no longer possible to expand a partial occurrence, the algorithm backtracks to the next largest partial occurrence. These optimizations result in a generalized Knuth-Morris-Pratt algorithm.

Exploration of the design space around the main derivation path, by relaxing some assumptions, and by pursuing alternative paths, provides explanations for related algorithms such as Earley's algorithm for context-free parsing and Waltz filtering for assigning 3-D interpretations to 2-D images. Algorithms for patterns with variables and multiple patterns can be obtained by appropriately instantiating the underlying topology.

8.2 Contributions

The contributions of this dissertation can be summarized as follows:

- A deep domain analysis of pattern matching, emphasizing geometry, and parameterized using category theory.
- A rigorous derivation of a generalized version of the Knuth-Morris-Pratt pattern matching algorithm.
- A derivation style which converts an extension into an intension.
- A theory for a generic, highly reusable component for pattern matching.
- A foundation for studying geometric properties of data structures.

We expand upon each of these in the following sections.

8.2.1 A theory of pattern matching

This dissertation has provided a deep theory of the domain of pattern matching. The domain language primitives are taken from

- category theory, which results in a parameterized theory by abstracting away inessential structure, and
- sheaf theory, which emphasizes the geometry of the data structures involved, specifically, the notion of decomposing a pattern into a collection of pieces (a "cover") and the "gluing" together of partial occurrences.

The surprising variety of applications of this theory illustrates the power of categorical methods to succinctly describe the common features of a class of problems.

8.2.2 A derivation of generalized KMP

We have rigorously derived a generalized version of the Knuth-Morris-Pratt pattern matching algorithm which works for any data structure by exploiting general theorems in category theory and by using general transformations such as divide-and-conquer, finite differencing, and backtracking. The main features of this derivation are

- the expression of the Knuth-Morris-Pratt algorithm as a divide-and-conquer algorithm;
- the explanation (and, hence, generalization) of the "failure function" as backtracking in the space of partial occurrences; and
- the explicit description of the fundamental similarity between KMP, Earley's parsing, and Waltz filtering.

8.2.3 Converting extensions into intensions

The derivation style adopted is this dissertation is that of gradually converting an extensional description of the problem into an intensional description, i.e., an algorithm. The extensional description has the advantages that

- it is impartial to sequential or parallel algorithms, and
- it captures only the essential properties of the problem, thus eliminating accidental features which may creep in due to the use of a particular language or formalism.

It appears that, in converting an extension into an intension, two basic implementation steps are crucial: divide-and-conquer and finite differencing. Divide-and-conquer converts the extension into an inductive computation; this step is essential because induction is the only principle of computation. Finite differencing is essential for any sequential algorithm; an algorithm can only maintain a finite amount of state information and hence has to calculate the output incrementally.

8.2.4 A reusable component for pattern matching

The major result of this dissertation is a theory for a highly reusable component for pattern matching. Such a theory consists of two parts [Arango 88]:

- domain analysis: a language for describing problems in the domain; here, the language of category theory and sheaf theory, as specialized to pattern matching; and
- domain engineering: the recording of design and implementation knowledge; here, the derivation of the generalized KMP and the explanation of related algorithms.

Two features of this theory with respect to reusability bear highlighting:

- Reusability is related to genericity: The variety of applications of our theory arise from using category theory as a language for parameterization. Previous attempts at generalizing KMP have been restricted to modifications pertaining to a single data structure. I firmly believe that such modifications to algorithms can be properly cast as instantiations of a parameterized theory which uses the right primitives and which has the appropriate level of abstractness.
- Design information is important: The final code is just a small part of a reusable component. The explicit recording of design information in our derivation contributed to its reusability by allowing us to modify and reuse the derivation to obtain other algorithms.

8.2.5 The geometry of data structures

Perhaps the most far-reaching contribution of this dissertation is its geometric view of data structures. We seem to routinely, but implicitly, use geometric concepts when dealing with data structures, as indicated, for example, by the phrases "rightmost node in the bottom level of a heap," and "propagate tokens along edges of a graph." Such descriptions are frequently more intuitive than algebraic ones, perhaps because of the increased bandwidth of information transfer. However, little work has been done to formalize and systematically use such concepts in computer science. One

reason for this is that geometry,[1] because of its abstractness and generality, is a hard branch of mathematics; it is even harder to transfer such abstract machinery to computer science, unless there are concrete examples to bring the theory down to ground-level.

I hope that this dissertation, by analyzing a well-known area like pattern matching using basic concepts from sheaf theory, has benefitted both fields: in computer science, by showing that informal usage of geometric concepts can be formalized, and that such formalization has great unifying power; in mathematics, by providing elementary, intuitive, examples of abstract concepts in sheaf theory.

[1] By "geometry," I do not mean Euclidean geometry or co-ordinate geometry; these are specific theories of specific real-world entities. Instead, by "geometry," I mean the spatial, structural, directional, and continuity aspects of any mathematical entity whatsoever; in short, geometry in the abstract, formalized axiomatically as some kind of topology: general topology for studying continuity, or Grothendieck topologies for studying "direction," etc.

Epilogue
On Pure, Applied, and Experimental Computing

We do not know: we can only guess.
And our guesses are guided by the unscientific, the metaphysical ...
faith in laws, in regularities which we can uncover—discover. ...
Once put forward, none of our 'anticipations' are dogmatically upheld.
Our method of research is not to defend them, in order to prove how right we were.
On the contrary, we try to overthrow them.
Using all the weapons of our logical, mathematical, and technical armoury
we try to prove that our anticipations were false—in order to put forward,
in their stead, new unjustified and unjustifiable anticipations ...
— Sir Karl Popper, *The Logic of Scientific Discovery* (1934)

Most derivations of KMP (actually, most program derivations, e.g., in the journal *Science of Computer Programming*) show detailed sequences of low-level transformations which lead to a particular algorithm from a given specification. The only conclusions one can draw from such derivations are something like

if transform T is applied to P, we get Q.

We liken these to observations in physics; in physics, the observations are about nature; in our field, the observations are about computing. Observations in physics typically arise in the context of experiments. Current activity in program derivation can be likened to *experimentation* with computing.

To continue the analogy, an explanation of a set of observed facts is a theory which derives the observations from a small collection of assumptions in a logical way. It follows that we need theories that explain program derivations. Unfortunately, the analogy breaks down here, because our observations can themselves be considered to be theories (observations in physics have an incontrovertible status when compared to theories, e.g., the decomposition of protons by high-energy particles, vs. the quark-theory explanation). The activity of producing theories for explaining observations can be justifiably called *pure* computing.

Given that we have theories explaining other theories, what characteristics should such explanatory theories have? We can apply Occam's razor and mandate that

explanatory theories should be simple, have a small number of assumptions, and explain a large class of observations. There is a subtler property which good theories have: they somehow provide "deeper" explanations.

In this dissertation, we have attempted to provide a deep explanation of pattern matching; not in the sense that a particular algorithm can be derived by some sequence of transformations, but in terms of fundamental properties of patterns (patterns are geometric and they can be glued together). The assumptions we make are minimal: the three axioms for a topology, and strict epimorphic families as covers. A surprising variety of concepts (observations about computing in certain domains) can be explained starting with these assumptions: pattern matching, context-free parsing, rewriting, unification, etc.

Bibliography

[Aho and Corasick 75]

AHO, A. V., AND CORASICK, M. J. Efficient string matching: An aid to bibliographic search. *Commun. ACM 18*, 6 (June 1975), 333–340.

[Arango 88]

ARANGO, G. *Domain Engineering for Software Reuse.* PhD thesis, Dept. of ICS, University of California, Irvine, 1988.

[Astesiano and Wirsing 86]

ASTESIANO, E., AND WIRSING, M. An introduction to ASL. In *IFIP TC2/WG2.1 Working Conference of Program Specification and Transformation* (Bad Tölz, FRG, Apr. 1986), L. G. L. T. Meertens, Ed., North-Holland, pp. 343–365.

[Baker 78]

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

[Barr and Wells 90]

BARR, M., AND WELLS, C. *Category Theory for Computing Science.* Prentice-Hall, New York, 1990.

[Baxter 90]

BAXTER, I. D. *Transformational Maintenance by Reuse of Design Histories.* PhD thesis, Dept. of ICS, University of California, Irvine, 1990. Technical Report 90-36.

[Bird 77]

BIRD, R. Two dimensional pattern matching. *Information Processing Letters 6* (1977), 168–170.

[Bird et al. 89]

BIRD, R. S., GIBBONS, J., AND JONES, G. Formal derivation of a pattern matching algorithm. *Sci. Comput. Programming 12* (1989), 93–104.

[Burghardt 88]

BURGHARDT, J. A tree pattern matching algorithm with reasonable space requirements. In *CAAP'88* (Nancy, France, Mar. 1988), *Lecture Notes in Computer Science,* Vol. 299, Springer-Verlag, pp. 1–15.

Bibliography

[Aho and Corasick 75]

AHO, A. V., AND CORASICK, M. J. Efficient string matching: An aid to bibliographic search. *Commun. ACM 18*, 6 (June 1975), 333–340.

[Arango 88]

ARANGO, G. *Domain Engineering for Software Reuse.* PhD thesis, Dept. of ICS, University of California, Irvine, 1988.

[Astesiano and Wirsing 86]

ASTESIANO, E., AND WIRSING, M. An introduction to ASL. In *IFIP TC2/WG2.1 Working Conference of Program Specification and Transformation* (Bad Tölz, FRG, Apr. 1986), L. G. L. T. Meertens, Ed., North-Holland, pp. 343–365.

[Baker 78]

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

[Barr and Wells 90]

BARR, M., AND WELLS, C. *Category Theory for Computing Science.* Prentice-Hall, New York, 1990.

[Baxter 90]

BAXTER, I. D. *Transformational Maintenance by Reuse of Design Histories.* PhD thesis, Dept. of ICS, University of California, Irvine, 1990. Technical Report 90-36.

[Bird 77]

BIRD, R. Two dimensional pattern matching. *Information Processing Letters 6* (1977), 168–170.

[Bird et al. 89]

BIRD, R. S., GIBBONS, J., AND JONES, G. Formal derivation of a pattern matching algorithm. *Sci. Comput. Programming 12* (1989), 93–104.

[Burghardt 88]

BURGHARDT, J. A tree pattern matching algorithm with reasonable space requirements. In *CAAP'88* (Nancy, France, Mar. 1988), *Lecture Notes in Computer Science,* Vol. 299, Springer-Verlag, pp. 1–15.

[Dahlhaus and Makowsky 88]

DAHLHAUS, E., AND MAKOWSKY, J. A. Gandy's principles for mechanisms as a model of parallel computation. In *The Universal Turing Machine: A Half-Century Survey*, R. Herken, Ed. Oxford University Press, Oxford, 1988, pp. 309–314.

[Demazure 70]

DEMAZURE, M. Topologies et faisceaux. In *Propriétés Générales des Schémas en Groupes, Lecture Notes in Mathematics,* Vol. 151. Springer-Verlag, 1970. Exposé IV of SGA 3 (Séminaire de Géométrie Algébrique du Bois-Marie, 1962/64).

[Dijkstra 76]

DIJKSTRA, E. W. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, NJ, 1976.

[Dijkstra 82]

DIJKSTRA, E. W. Repaying our debts. In *Theoretical Foundations of Programming Methodology*, M. Broy, Ed. D. Reidel Pub. Co., Dordrecht, Holland, 1982.

[Earley 70]

EARLEY, J. An efficient context-free parsing algorithm. *Commun. ACM 13*, 2 (Feb. 1970), 94–102.

[Ehrich et al. 91]

EHRICH, H.-D., GOGUEN, J., AND SERNADAS, A. A categorical theory of objects as observed processes. In *Proceedings, REX/FOOL Workshop on Foundations of Object Oriented Languages.* Springer-Verlag, 1991, p. to appear. Lecture Notes in Computer Science.

[Ehrig 78]

EHRIG, H. Introduction the the algebraic theory of graph grammars. In *Graph Grammars and Their Application to Computer Science and Biology*, V. Claus, H. Ehrig, and G. Rozenberg, Eds., *Lecture Notes in Computer Science,* Vol. 73. Springer-Verlag, New York, 1978, pp. 1–69.

[Ehrig and Mahr 85]

EHRIG, H., AND MAHR, B. *Fundamentals of Algebraic Specification 1: Equational and Initial Semantics, EATCS Monographs on Theoretical Computer Science,* Vol. 6. Springer-Verlag, Berlin, 1985.

[Ehrig and Mahr 90]

EHRIG, H., AND MAHR, B. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints, EATCS Monographs on Theoretical Computer Science,* Vol. 21. Springer-Verlag, Berlin, 1990.

[Eytan 80]

EYTAN, M. Grothendieck topologies on formal grammars. Tech. rep., Univeristé René Descartes, Paris V, 1980.

[Eytan 82]

EYTAN, M. *Sémantique Doctrinale Appliqueé: Méthodes de Logique Catégorique en Informatique, Linguistique, Ensembles Flous.* PhD thesis, Univeristé René Descartes, Paris V, 1982.

[Freeman 87]

FREEMAN, P. A conceptual analysis of the Draco approach to constructing software systems. *IEEE Trans. Softw. Eng. 13* (July 1987), 830–844.

[Freyd 72]

FREYD, P. Aspects of topoi. *Bull. Austral. Math. Soc. 7* (1972), 1–76. Also see Corrigenda, *Bull. Austral. Math. Soc. 7* (1972), 467–480.

[Gandy 80]

GANDY, R. Church's thesis and principles for mechanisms. In *The Kleene Symposium.* North-Holland, 1980, pp. 123–148.

[Goldblatt 84]

GOLDBLATT, R. *Topoi: The Categorial Analysis of Logic.* North-Holland, Amsterdam, 1984.

[Herrlich and Strecker 73]

HERRLICH, H., AND STRECKER, G. E. *Category Theory: An Introduction.* Allyn and Bacon, Boston, 1973.

[Hoffmann and O'Donnell 82]

HOFFMANN, C. M., AND O'DONNELL, M. J. Pattern matching in trees. *J. ACM 29*, 1 (Jan. 1982), 68–95.

[Johnstone 77]

JOHNSTONE, P. T. *Topos Theory.* Academic Press, London, 1977.

[Knuth et al. 77]

KNUTH, D. E., MORRIS, JR., J. H., AND PRATT, V. R. Fast pattern matching in strings. *SIAM J. Comput. 6*, 2 (June 1977), 323–350.

[Kock and Wraith 71]

KOCK, A., AND WRAITH, G. C. Elementary toposes. Lecture Notes Series 30, Matematisk Institut, Aarhus Universitet, Sept. 1971.

[Mac Lane 71]

MAC LANE, S. *Categories for the Working Mathematician.* Springer-Verlag, New York, 1971.

[Mackworth 77]

MACKWORTH, A. K. Consistency in networks of relations. *Artificial Intelligence 8* (1977), 99–118.

[Monteiro and Pereira 86]

MONTEIRO, L. F., AND PEREIRA, F. C. N. A sheaf-theoretic model of concurrency. Tech. Rep. CSLI-86-62, CSLI, Stanford University, Oct. 1986.

[Morris 90]

MORRIS, J. M. Programming by expression refinement: The KMP algorithm. In *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra.* Springer-Verlag, New York, 1990, pp. 327–338.

[Neighbors 84]

NEIGHBORS, J. The Draco Approach to Constructing Software from Components. *IEEE Trans. Softw. Eng. SE-10*, 9 (Sept. 1984), 564–573.

[Paige 81]

PAIGE, R. A. *Formal Differentiation: A Program Synthesis Technique.* UMI Research Press, Ann Arbor, Michigan, 1981. A revision of the author's Ph. D. thesis, New York University, 1979.

[Paige and Koenig 82]

PAIGE, R., AND KOENIG, S. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 402–454.

[Partsch 84]

PARTSCH, H. Structuring transformational developments: A case study based on earley's recognizer. *Sci. Comput. Programming 4* (1984), 17–44.

[Partsch 86]

PARTSCH, H. Transformational program development in a particular domain. *Sci. Comput. Programming 7* (1986), 99–241.

[Partsch 90]

PARTSCH, H. A. *Specification and Transformation of Programs: A Formal Approach to Software Development.* Springer-Verlag, Berlin, 1990.

[Partsch and Völker 90]

PARTSCH, H. A., AND VÖLKER, N. Another case study on reusability of transformational developments: Pattern matching according to knuth, morris, and pratt. Tech. rep., KU Nijmegen, 1990.

[Pierce 88]

PIERCE, B. C. A taste of category theory for computer scientists. Tech. Rep. CMU-CS-88-203, Computer Science Dept, Carnegie Mellon University, Pittsburgh, 1988.

[Rosenfeld 84]

ROSENFELD, A., Ed. *Multiresolution Image Processing and Analysis.* Springer-Verlag, Berlin, 1984.

[Rydeheard and Burstall 88]

RYDEHEARD, D., AND BURSTALL, R. M. *Computational Category Theory.* Prentice-Hall, 1988.

[Sannella and Tarlecki 88a]

SANNELLA, D., AND TARLECKI, A. Specifications in an arbitrary institution. *Inf. and Comput. 76* (1988), 165–210. Preliminary version in Semantics of Data Types, *LNCS 123,* pp. 337–356.

[Sannella and Tarlecki 88b]

SANNELLA, D., AND TARLECKI, A. Towards formal development of programs from algebraic specifications: Implementations revisited. *Acta Inf. 25* (1988), 233–281.

[Schubert 72]

SCHUBERT, H. *Categories.* Springer-Verlag, Berlin, 1972.

[Seebach et al. 70]

SEEBACH, JR., J. A., SEEBACH, L. A., AND STEEN, L. A. What is a sheaf? *American Mathematical Monthly 77,* 7 (Aug. 1970), 681–703.

[SGA4]

ARTIN, M., GROTHENDIECK, A., AND VERDIER, J. L. *Théorie des Topos et Cohomologie Etale des Schémas, Lecture Notes in Mathematics,* Vol. 269. Springer-Verlag, 1972. SGA4, Séminaire de Géométrie Algébrique du Bois-Marie, 1963–1964.

[Shepherdson 88]

SHEPHERDSON, J. C. Mechanisms for computing over arbitrary structures. In *The Universal Turing Machine: A Half-Century Survey,* R. Herken, Ed. Oxford University Press, Oxford, 1988, pp. 581–601.

[Smith 85]

SMITH, D. R. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence 27* (1985), 43–96.

[Smith 88]

SMITH, D. R. The structure and design of global search algorithms. Tech. Rep. KES.U.87.12, Kestrel Institute, Palo Alto, California, July 1988. To appear in *Acta Informatica.*

[Srinivas 90]

SRINIVAS, Y. V. Algebraic specification: Syntax, semantics, structure. Tech. Rep. 90-15, Dept. of ICS, University of California, Irvine, June 1990.

[Sunday 90]

SUNDAY, D. M. A very fast substring search algorithm. *Commun. ACM 33,* 8 (Aug. 1990), 132–142.

[Tennison 75]

TENNISON, B. R. *Sheaf Theory.* Cambridge University Press, 1975.

[van derWoude 89]

VAN DER WOUDE, J. Playing with patterns, searching for strings. *Sci. Comput. Programming 12* (1989), 177–190.

[Verdier 72]

VERDIER, J. L. Topologies et faisceaux. In *Théorie des Topos et Cohomologie Etale des Schémas, Lecture Notes in Mathematics,* Vol. 269. Springer-Verlag, 1972. Exposé II of SGA 4 (Séminaire de Géométrie Algébrique du Bois-Marie, 1963–1964).

[Waltz 75]

WALTZ, D. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision.* Mc-Graw Hill, New York, 1975, pp. 19–91.

[Wirsing 86]

WIRSING, M. Structured algebraic specifications: A kernel language. *Theoretical Comput. Sci. 42* (1986), 123–249. A slight revision of his Habilitationsschrift, Technische Universität München, 1983.

[Wirsing 90]

WIRSING, M. Algebraic specification. In *Formal Models and Semantics,* J. van Leeuwen, Ed., *Handbook of Theoretical Computer Science,* Vol. B. MIT Press/Elsevier, 1990, pp. 675–788.