Mustererkennung, parametrisierte Algorithmen universelles Berechnen via Kategorientheorie ein neues Entwurfsparadigma

Michael W. Bertol
Universität Stuttgart, Institut für Informatik
Breitwiesenstraße 20/22, D-70565 Stuttgart
e-mail:Michael.Bertol@informatik.uni-stuttgart.de
HTTP://www.informatik/ifi/ti/personen/Bertol/bertol.html

19. August 1996, currently under review

Zusammenfassung

In dieser Arbeit wird unter Verwendung von kategoriellen Methoden eine Problemklasse, das universelle Mustererkennen, betrachtet. Dazu wird die notwendige Theorie entwickelt und ein Framework, also ein Algorithmenskelett vorgestellt. Bei der Realisierung hat sich herausgestellt, daß dieses Skelett in kanonischer Weise zu Algorithmeninstanzen für verteilte Systeme, d.h. auf einem Netz mit sich lokal synchronisierenden Berechnungseinheiten (und lokalem Speicher) führt.

Es zeigte sich auch, daß viele Algorithmen Instanzen dieser Designmethode sind, eine Indikation dafür, daß die kategorielle Methodologie hervorragend zur Spezifikation und Realisierung von Informatikproblemen geeignet ist.

Das Verfahren wird für eine sequentielle Maschine optimiert. Eine prototypische Implementierung in einer konkreten Programmiersprache mit einem hinreichend allgemeinem Typmechanismus schließt sich an.

Viele bekannte Probleme, u.a. das Parsen von kontextfreien Grammatiken oder das Erkennen einer Szene lassen sich auf diese Weise parametrisiert beschreiben und verifizieren. Die verwendeten „topologischen Methoden“ sind sogar ausreichend universell, um beispielsweise Interpreter für gängige Programmiersprachen zu realisieren.

Spezifikation, Design und Implementierung von einer großen Klasse von Realisierungsproblemen lassen sich mit dieser Methode elegant konzeptionell lösen. Dazu gehören insbesondere Probleme mit einer inhäerenten topologischen Beschreibung.

Keywords: Modellierung, verteilte Algorithmen, kategorielle Algorithmen, parametrisierte Algorithmen, parametrisierte Spezifikation, Typisierung, funktionale Programmierung.

Inhaltsverzeichnis

1 Motivation

Eines der zentralen Probleme der Informatik ist die Programmierung digitaler Computer. Das korrekte Instruieren eines Rechners ist eine nicht-triviale Problematik. Einziges Ausdrucksmittel war bisher die Formulierung einer Lösung als Algorithmus. John von Neuman, einer der Pioniere, faßte dies in die Worte: „The actual code for a problem is that sequence of coded symbols, that has to be placed in the Selectron memory in order to cause the machine to perform the desired and planned sequence of operations, which amount to solving the problem in question.“

Im Laufe der Zeit wurden die zu lösenden Probleme umfangreicher und komplizierter, und das ursprüngliche Vertrauen in die Korrektheit einer Software wich mit wachsender Programmgröße und Programmkomplexität. Diesen Effekten entgegen zu wirken, wurden Verifikations- und Validierungsmethoden sowie modulare Software-Entwurfsmethoden eingeführt.

Der gesteigerte organisatorische Aufwand bei den Entwicklungsarbeiten, wird immer mehr von Computern unterstützt oder übernommen, genauer von Werkzeugen wie Compilern, Bibliotheken, höheren Programmiersprachen etc. Am Ende dieser Entwicklung stehen die nicht-prozeduralen Programmiersprachen und die Frameworks, sowie einige Heuristiken wie neuronalen Netze oder evolutionären Algorithmen.[1]

Beispielsweise benötigt man für den Entwurf eines klassifizierenden neuronalen Netzes lediglich viele Testdaten, mit denen ein Computer das vom Entwickler vorgeschlagene Netz trainiert. Die Programmierung eines Verfahrens weicht einer Netzbeschreibung, die entsprechend durch Trainingsdaten initialisiert wird.

Hier wird ein Ansatz vorgestellt, in dem durch (topologische) Spezifikation Probleme (ähnlich dem Constraint Programming) gelöst werden. Dazu wird ein universeller (konstruktiver) Mechanismus bereitgestellt und die Problemstellungen als Erkennungsprobleme interpretiert.

Diese Beschreibung erscheint abschreckend abstrakt und ungewöhnlich. Lauschen wir einem Gespräch zwischen zwei Informatikern, Edward Nigma und August Thor.

E.Nigma: Was ist eigentlich Spezifikation?

Au.Thor: Unter Spezifikation verstehe ich eine Sprache, in der Probleme und Lösungen beschreibbar sind.

E.Nigma: Aha, aber warum noch eine Sprache?

Au.Thor: Nun, es gibt viele Probleme und einige Probleme lassen sich in manchen Sprachen gut und in anderen eher umständlich beschreiben. Leider entbehren außerdem sehr viele Sprachen einer rigorosen (mathematisch-logischen) Grundlage.

---

[1] Leider werden diese Werkzeuge meiner Meinung nach viel zu wenig eingesetzt.

E.Nigma: Was heißt in der obigen Erklärung „parametrisierter Algorithmus“?

Au.Thor: Nun, ein Algorithmus ist eine Vorschrift, also ein Ablaufplan, der beschreibt, wie ich zu einer Lösung kommen kann. Parametrisiert bedeutet hier, daß es sich eigentlich um viele Algorithmen handelt — zu jeder etwaigen Problemstellung einer.

E.Nigma: So, warum dann parametrisieren und nicht für jedes Problem einen Algorithmus? Das ist doch viel übersichtlicher, und außerdem kann man dann auch noch spezielle Eigenheiten des Problems ausnutzen, um das Programm effizienter zu gestalten. Daß die abstrakte Programmierung mit Prolog oder Smalltalk nicht schnell ist, weiß ja jeder.

Au.Thor: Nun laß mich erst mal gegen diese Vorwürfe verteidigen bevor ich den Anfang vergesse. Warum nicht viele Algorithmen für viele Probleme schreiben liegt auf der Hand. Viele Probleme sind sich sehr ähnlich, und es wäre nicht sehr produktiv, das Rad mehrmals zu erfinden. Das siehst du sicher ein. Moderne Programmierumgebungen stellen sogenannte Frameworks bereit, die die Entwicklungszeiten von Softwareprojekten stark verkürzen.

E.Nigma: Ja, aber was ist mit der Laufzeit und den Ressourcen.

Au.Thor: Das ist die Kunst beim abstrahieren bzw. parametrisieren. Enthält das parametrisierte Problem noch genügend viel strukturelle Information um ein effizientes Verfahren zu konstruieren, so hat man richtig parametrisiert. Oft ist aber nicht die falsche Parametrisierung Schuld an schlechter Laufzeit, sondern schlechte Instanziierung. Du kennst vielleicht die Fibonaccizahlen.

E.Nigma: Klar, die haben wir im ersten Semester ausrechnen müssen. Dafür gab's eine Rekursive Vorschrift

$$fib(0) = 0, fib(1) = 1, fib(n+1) = fib(n) + fib(n-1) \quad (n \in \mathbb{N})$$

Au.Thor: Wenn du jetzt eine Spezifikationssprache benutzt, die nur rekursive Konstrukte enthält. Kannst du damit einen effizienten Algorithmus formulieren.

E.Nigma: Nein, meine rekursive Formulierung besitzt, wie du sicher auch weißt, exponentielle Laufzeit.

Au.Thor: Deine rekursive Spezifikation besitzt exponentielles Laufzeitverhalten in n, das ist richtig, aber es geht doch schneller.

E.Nigma: Ja iterativ.

Au.Thor: Schau dir mal diese Spezifikation an

$$F(0) = (0,1), F(n+1) = (\pi_2(F(n)), \pi_1(F(n))\pi_2(F(n))) \quad (n \in \mathbb{N})$$
$$f(n) = \pi_1(F(n) \quad (n \in \mathbb{N})$$

(π_i ist die i-te Projektion.) Berechnet man bei der Evaluation $F(n)$ nur einmal, so ist der Algorithmus linear, trotz dem er rekursiv formuliert ist. Ich kenne sogar eine rekursive Formulierung, die nur logarithmisch in n ist.

$$f(n) = f(k)f(n-k) + f(k-1)f(n-k-1) \quad (n, k \in \ )$$

E.Nigma: Toll, das heißt ich muß mich nur in die Sprache eindenken und hoffen, daß der Designer ein eingängiges Konzept hatte.

Au.Thor: Genau. Hat man sich an die Formulierungen gewöhnt, so erscheinen sie ganz einfach und natürlich. Leider ist aller Anfang schwer.

....

Nun hat aber gerade das Mustererkennungsproblem eine natürliche Semantik und eine eingängige Formulierung. Versucht man ein Objekt in einem Raum zu „finden“, beispielsweise eine Orange auf einem Tisch, so ist es relativ unnatürlich, dieses Problem durch eine im herkömmlichen Sinne algorithmisch prozedurale Formulierung zu lösen: scanne den Tisch zeilenweise; wenn mehrere orangene Zeilenabschnitte vorkommen versuche sie als Orange zu interpretieren. Eigentlich reicht es aus, den Raum (im Falle unserer Orange ist das der Tisch auf dem sie sich evtl. befindet) und das Pattern (orange und kugelförmig von entsprechender Dichte — nun eben eine Orange) zu kennen. Ein zweites Beispiel ist das Puzzeln. Dabei wird ein Bild, das gelöste Puzzle, gesucht. Die Topologie liegt in der Beschreibung der Ränder und der Teilbilder der einzelnen Teile. In einem gewissen Sinn ist die Beschreibung solcher Erkennungsproblem gleichzeitig eine Lösung.

Das selbe Vorgehen kann verwendet werden beim Suchen von Mustern in einem Text, beim Suchen eines passenden Modells zu einer logischen Formel, beim Erkennen und Interpretieren eines (Radar-)Bildes, beim rekonstruieren eines Zustandes von einem verteilten System oder beim Parsen eines Textes u.v.m.

Das korrespondierende universelle Verfahren läßt sich unter Zuhilfenahme einiger mathematischer Sprechweisen und abstrakter Programmierung elegant formulieren.

1.1 Überblick

Im ersten Teil der Arbeit werden einige mathematische[2] Sprechweisen vorgestellt. Die Spezifikation wird wesentlich auf den Ideen von Alexander Grothendiek basieren, die diesen Terminus benötigen. Nichtsdestotrotz sind die Ideen intuitiv und eingängig. Mir erscheint dieser Terminus eine geeignete Sprache zur Spezifikation. Da ich kategorielle Sprechweisen nicht voraussetzen will, so daß die Arbeit allgemein verständlich und selbsterklärend bleibt, habe ich eine Einführung in die

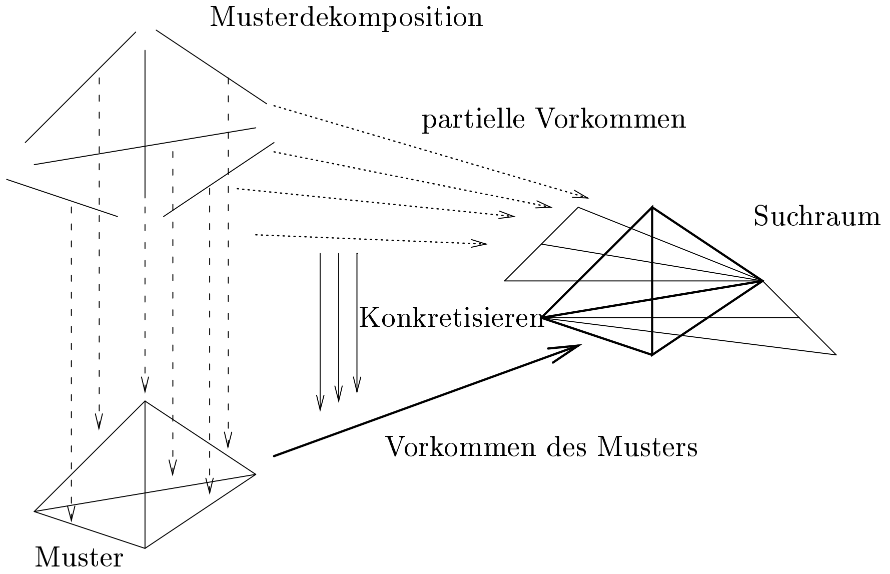
[2] Der mathematische Schnick-Schnack kann von einem eher praktisch-technisch interessierten Leser auch übergangen werden. Um diese Methode anzuwenden ist ein intuitives Verständnis in vielen Fällen ausreichend.

Theorie der Kategorien mit aufgenommen. Sie ist mit einigen konkreten Beispielen, die die Universalität dokumentieren, illustriert.

In dieser Terminologie wird ein Algorithmus — eigentlich eher eine Methodologie[3] — entwickelt, der schrittweise unter Anwendung von allgemeinen Konstruktionsparadigmen in eine extentionale Form transformiert wird. Einige Implementierungsauszüge in einer konkreten Programmiersprache illustrieren die praktische Verwendbarkeit.

2 Die Geometrie des Mustererkennens

Das Mustererkennungsproblem besteht darin, das Auftreten oder Vorkommen eines Musters in einem Suchraum zu erkunden. Die Muster werden gewöhnlich durch Konstanten, sogenannte Patterns, Exemplare[4] oder allgemeiner durch Prädikate beschrieben.



Ein Auftreten eines Musters ist ein Stück des Suchraums bzw. des Objekts, welches nach dem Muster abgesucht werden soll und gleichzeitig mit dem Muster korrespondiert. Wird ein Muster durch eine Konstante beschrieben, so muß das gefundene Stück identisch zu dem Muster sein. Handelt es sich um ein exemplarisches Muster, so muß der spezifizierte Teil des abgesuchten Objekts auf das Exemplar passen. Wird das Muster durch ein Prädikat beschrieben, so muß das gefundene Stück ein Modell sein.[5]

[3]Wegen der Universalität dieses Konzeptes ist die Methode durchaus auch in anderen Bereichen bspw. Softwareengineering oder Hardwaredesign anwendbar.

[4]Unter Exemplaren verstehen wir nicht vollständig spezifizierte Muster.

[5]Man beachte die große Varität der Problemstellungen und deren Gemeinsamkeiten.

Wir untersuchen, wie sich dieses Problem in einem abstrakten Kontext behandeln läßt. Partielle Vorkommen eines partiellen Musters in Teilen eines abstrakten Objekts lassen sich unter gewissen Annahmen über den Zusammenhang und die Lokalität der Objekte und der Zerlegung oder Überdeckung zu einem Vorkommen des gesamten Musters zusammenfassen. Diesen Vorgang nennen wir im weiteren Konkretisieren. Diese Eigenschaft ermöglicht es das Problem, ein Muster zu finden, von einer intensionalen in eine extentionale Form, einen Algorithmus, zu transformieren.

Als ein motivierendes Beispiel betrachten wir eine der zentralen Strukturen, die Graphen. Ein Auftreten eines Musters — modelliert durch einen Morphismus vom Muster zum abzusuchenden Objekt $p \longrightarrow t$, wird durch Zusammenfassen vieler kleinerer verträglicher — einen Begriff, den es noch genau zu fassen gilt — partieller Auftreten $p_i \longrightarrow t_j$ von Musternteilen p_i in Suchraumteilen t_j konstruiert.

Wir formulieren die Geometrie des Problems mit einer Grothendiek Topologie. Für endliche Strukturen, wie wir sie hier betrachten wollen, haben sich diese Topologien als gut geeignet herausgestellt, insbesondere für die angestrebte funktionale Formulierung. Eine gewöhnliche Topologie ist im mathematischen Sinne ein Mengensystem über einer Trägermenge, das unter endlicher Schnittbildung und beliebiger Vereinigung abgeschlossen ist. Grothendiek Topologien sind kategorielle Verallgemeinerungen dieses Begriffes. Das durch die Inklusion partiell geordnete Mengensystem wird durch eine Kategorie ersetzt.[6]

[6]Eine mathematisch abschreckende Formulierung. Sie sollte aber nicht am weiterlesen hindern.

3 Kategorien und universelle Konstruktionen

In diesem Kapitel stellen wir einige Sprechweisen aus der modernen Mathematik bereit, die sich innerhalb der Mathematik bereits bewährt haben. notwendig. U.a. wird eine kurze aber einfache Einführung in die wichtigsten Konzepte der Kategorientheorie gegeben.

Eine *Kategorie* $\mathcal{C}$ besteht aus *Objekten* $A, B, C, \ldots, X, Y, \ldots$, *Morphismen* oder auch *Pfeilen* $f, g, \ldots$ und vier Operationen. Zwei dieser Operationen dom und cod ordnen jedem Morphismus f in $\mathcal{C}$ seinen *Domain* $\mathsf{dom}(f)$ und seinen *Codomain* $\mathsf{cod}(f)$ zu, welche Objekte aus $\mathcal{C}$ sind. Wir notieren $f : C \longrightarrow D$, wenn D der Codomain von f und C der Domain von f ist. Die Operation id assoziiert zu jedem Objekt C einen Morphismus $\mathsf{id}_C : C \longrightarrow C$, den *Identitätsmorphismus* und die vierte Operation ist eine *Komposition*, die einem Morphismenpaar $(f : A \longrightarrow B, g : B \longrightarrow C)$ mit $\mathsf{cod}(f) = \mathsf{dom}(g) = B$ einen Morphismus $g \circ f : A \longrightarrow C$ zuordnet. Es gelten die folgenden Axiome

(i) $\mathsf{dom}(\mathsf{id}_C) = C = \mathsf{cod}(\mathsf{id}_C)$

(ii) $\mathsf{dom}(g \circ f) = \mathsf{dom}(f)$ und $\mathsf{cod}(g \circ f) = \mathsf{cod}(g)$

(iii) $\mathsf{id}_D \circ f = f = f \circ \mathsf{id}_C$

(iv) $(f \circ g) \circ h = f \circ (g \circ h)$,

wobei wir in (i) - (iv) annehmen, daß die jeweiligen Kompositionen definiert sind.

Kategorien sind also nur eine Sammlung von Punkten mit ausgezeichneten Schlingenpfeilen, von denen einige durch Pfeile verbunden sind, so daß für zwei zusammengesetzte Pfeile ein entsprechender vorhanden ist. Wer mit der Theorie der endlichen Automaten vertraut ist, kann sich unter einer Kategorie einen Automaten, der an jedem Zustand eine formale λ-Schlinge besitzt, vorstellen. Die Zustände sind die Objekte, und die Pfeile werden durch die Wörter dargestellt. Die Verkettung ist die Konkatenation von Pfaden.

In einer Kategorie $\mathcal{C}$ nennen wir einen Morphismus[7] $f : C \longrightarrow D$ *Isomorphismus*, wenn ein Morphismus $g : D \longrightarrow C$ mit $f \circ g = \mathsf{id}_D$ und $g \circ f = \mathsf{id}_C$ existiert. Wir sagen dann: C ist isomorph zu D und notieren $C \cong D$. Ein Morphismus $f : C \longrightarrow D$ wird *Epimorphismus* genannt, wenn für jedes Objekt E und Morphismen $g, h : D \longrightarrow E$ in $\mathcal{C}$ die Gleichheit $g = h$ aus $g \circ f = h \circ f$ folgt. Dual dazu nennen wir $f : C \longrightarrow D$ einen *Monomorphismus*, wenn für jedes Objekt B und Morphismen $g, h : B \longrightarrow C$ in $\mathcal{C}$ die Gleichheit $g = h$ aus $f \circ g = f \circ h$ folgt. Zwei Morphismen $f : A \longrightarrow D$ und $g : B \longrightarrow D$ nennen wir *äquivalent*, wenn ein Isomorphismus $h : A \longrightarrow B$ mit $g \circ h = f$ existiert. Ein *Subobjekt* ist eine Äquivalenzklasse von Monomorphismen nach D. Die Klasse der Subobjekte von D, $\mathsf{Sub}_{\mathcal{C}}(D)$, ist via

$$[f] \leq [g] \quad \Longleftrightarrow \quad \exists h : A \longrightarrow B \text{ mit } f = gh$$

[7]Leser, die aus unerfindlichen Gründen eine Abneigung gegen das Wort Morphismus haben, können es (ohne Informationsverlust) durch das Wort Pfeil ersetzen.

partiell geordnet. Für zwei Objekte C und D bezeichnen wir die Klasse der Morphismen $C \longrightarrow D$ als *Homklasse*, $\mathsf{Hom}_{\mathcal{C}}(C, D)$. An dieser Stelle sei angemerkt, daß wir nicht streng mengentheoretisch fundiert vorgehen. Um uns von Antonymen zu befreien, nehmen wir die Existenz eines festen Universums $\mathcal{U}$ an. Mengenwertige Elemente aus $\mathcal{U}$ nennen wir dann „kleine Mengen" und eine Klasse von Elementen aus $\mathcal{U}$, die nicht in $\mathcal{U}$ liegt, nennen wir „große Menge". Wir nennen eine Kategorie $\mathcal{C}$ lokal klein, wenn alle Klassen $\mathsf{Hom}_{\mathcal{C}}(C, D)$ kleine Mengen sind. Die Kategorie ist klein, wenn die Morphismen und die Objekte auch kleine Mengen sind. Für eine Kategorie $\mathcal{C}$ ist $\mathcal{C}^{\mathsf{op}}$, die *duale Kategorie*, die aus denselben Objekten und Morphismen besteht, aber die Domains und die Codomains vertauscht wurden.

Für eine Kategorie $\mathcal{C}$ und ein Objekt C aus $\mathcal{C}$ ist die *Komma-Kategorie* $\mathcal{C}/C$ die Kategorie mit den Morphismen aus $\mathcal{C}$ als Objekten und den Morphismen f die mit Domain $g : D \longrightarrow C$ und Codomain $h : E \longrightarrow C$, die das Diagramm

$$\begin{tikzcd} D \arrow[rr, "f"] \arrow[dr, "g"'] & & E \arrow[dl, "h"] \\ & C & \end{tikzcd}$$

(in $\mathcal{C}$) kommutieren lassen. Die Komposition ist durch Aneinanderkleben von kommutierenden Dreiecken definiert.

3.1 Funktoren — Morphismen zwischen Kategorien

Seien $\mathcal{C}$ und $\mathcal{D}$ zwei Kategorien. Ein *Funktor* von $\mathcal{C}$ nach $\mathcal{D}$ ist eine Operation F, die jedem Objekt C aus $\mathcal{C}$ ein Objekt $F(C)$ aus $\mathcal{D}$ und jedem Morphismus f aus $\mathcal{C}$ einen Morphismus $F(f)$ aus $\mathcal{D}$ so zuordnet, daß die Objekte verträglich miteinander sind, d.h.

(i) $F(\mathsf{dom}(f)) = \mathsf{dom}(F(f))$

(ii) $F(\mathsf{cod}(f)) = \mathsf{cod}(F(f))$

(iii) $F(\mathsf{id}_C) = \mathsf{id}_{F(C)}$

(iv) $F(g \circ f) = F(g) \circ F(f)$,

Für eine Kategorie $\mathcal{C}$ existiert der *Identitätsfunktor* $\mathsf{id}_{\mathcal{C}} : \mathcal{C} \longrightarrow \mathcal{C}$ und eine Komposition $G \circ F : \mathcal{C} \longrightarrow \mathcal{E}$ zweier Funktoren $F : \mathcal{C} \longrightarrow \mathcal{D}$ und $G : \mathcal{D} \longrightarrow \mathcal{E}$. Sind F und G zwei Funktoren von einer Kategorie $\mathcal{C}$ in eine Kategorie $\mathcal{D}$, dann ist eine *natürliche Transformation* α von F nach G, in Zeichen $\alpha : F \longrightarrow G$, eine Operation, die jedem Objekt einen Morphismus $\alpha_C : F(C) \longrightarrow G(C)$ in $\mathcal{D}$ zuordnet, und das so, daß für alle Morphismen $f : C' \longrightarrow C$ in $\mathcal{C}$ das Diagramm

$$\begin{tikzcd} F(C') \arrow[r, "\alpha_{C'}"] \arrow[d, "F(f)"'] & G(C') \arrow[d, "G(f)"] \\ F(C) \arrow[r, "\alpha_C"] & G(C) \end{tikzcd}$$

kommutiert. Wir nennen den Morphismus α_C auch die C-Komponente von α. Wenn alle Komponenten von α Isomorphismen sind, so nennt man α einen *natürlichen Isomophismus*. Sind $\alpha : F \longrightarrow G$ und $\beta : G \longrightarrow H$ zwei natürliche Transformationen zwischen Funktoren $F, G, H : \mathcal{C} \longrightarrow \mathcal{D}$, dann definieren wir die Komposition (von natürlichen Transformationen) $\beta \circ \alpha$ durch $(\beta \circ \alpha)_C = \beta_{G(C)} \circ \alpha_C$. Für zwei Kategorien $\mathcal{C}$ und $\mathcal{D}$ nennen wir die so entstandene Kategorie $\mathcal{D}^{\mathcal{C}}$ mit Funktoren als Objekte und natürliche Transformationen als Morphismen, eine *Funktor-Kategorie*.

Für zwei Kategorien $\mathcal{C}$ und $\mathcal{D}$ nennen wir einen Funktor $F : \mathcal{C}^{\mathsf{op}} \longrightarrow \mathcal{D}$ kontravariant (von $\mathcal{C}$ nach $\mathcal{D}$). Beispielsweise ist $C' \longrightarrow \mathsf{Hom}_{\mathcal{C}}(C', C)$ für ein (festes) Objekt C ein kontravarianter Funktor, der kontravariante *Homfunktor*, während $C \longrightarrow \mathsf{Hom}_{\mathcal{C}}(C', C)$ für ein (festes) Objekt der covariante Homfunktor ist.

Wir nennen einen Funktor $F : \mathcal{C} \longrightarrow \mathcal{D}$ surjektiv (bzw. injektiv), wenn für alle Objekte C und C' aus $\mathcal{C}$ die Abbildung

$$\begin{array}{rcl} \mathsf{Hom}_{\mathcal{C}}(C', C) & \longrightarrow & \mathsf{Hom}_{\mathcal{D}}(F(C'), F(C)) \\ f & \longmapsto & F(f) \end{array}$$

surjektiv (bzw. injektiv) ist. Ein Funktor $\mathcal{C} \longrightarrow \mathcal{D}$, der surjektiv und injektiv ist, so daß jedes Objekt aus $\mathcal{D}$ isomorph zu einem Objekt im Bild von F ist, nennt man *kategorielle Äquivalenz*.

3.2 Produkte und andere Konstruktionen

Ein Objekt X mit zwei Morphismen $\pi_1 : X \longrightarrow A$ und $\pi_2 : X \longrightarrow B$ ist das *Produkt* von A und B, wenn für alle Objekte Y und Morphismen $f : Y \longrightarrow A$ und $g : Y \longrightarrow B$ ein eineindeutiger Morphismus $\langle f, g\rangle : Y \longrightarrow X$ existiert so, daß das Diagramm

$$\begin{tikzcd} & Y \arrow[dl, "f"'] \arrow[d, dotted, "\langle f,g\rangle"] \arrow[dr, "g"] & \\ A & X \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \end{tikzcd}$$

kommutiert. Für X notieren wir auch $A \times B$. Iteriert man diese Konstruktion, erhält man (endliche) Produkte $\prod_{i \in I} A_i$

$$\begin{tikzcd} Y \arrow[d, dotted] \arrow[dr, "f_i"] & \\ \prod_{i \in I} A_i \arrow[r, "\pi_i"] & A_i \end{tikzcd}$$

In einer Kategorie nennen wir ein Objekt mit der Eigenschaft, daß von jedem Objekt aus nur ein eineindeutiger Morphismus auf dieses Objekt zeigt, ein terminales Objekt und bezeichnen es mit $\mathbf{1}$.

Eine Konstruktion, die einen zentralen Platz einnehmen wird, sind die *gefaserten Produkte*. In einer Kategorie $\mathcal{C}$ ist ein Morphismenpaar $p : P \longrightarrow B$ und $q : P \longrightarrow C$ mit dem gemeinsamen Domain P das gefaserte Produkt von dem Morphismenpaar $f : B \longrightarrow A$ und $g : C \longrightarrow A$, wenn das Diagramm

$$\begin{tikzcd} P \arrow[r, "q"] \arrow[d, "p"'] & C \arrow[d, "g"] \\ B \arrow[r, "f"] & A \end{tikzcd}$$

kommutativ ist, und wenn für jedes Objekt X in $\mathcal{C}$ und für alle Morphismen $\beta : X \longrightarrow B$ und $\gamma : X \longrightarrow C$ mit $f \circ \beta = g \circ \gamma$ ein eineindeutiger Morphismus $\delta : X \longrightarrow P$ existiert, so daß $p \circ \delta = \beta$ und $q \circ \delta = \gamma$, wie in folgendem Diagramm.

$$\begin{tikzcd} X \arrow[dr, dotted, "\delta"] \arrow[drr, "\gamma"] \arrow[ddr, "\beta"'] & & \\ & P \arrow[r, "q"] \arrow[d, "p"'] & C \arrow[d, "g"] \\ & B \arrow[r, "f"] & A \end{tikzcd}$$

Mit f und g ist das Produkt P mit den Projektionen p und q bis auf Isomorphie festgelegt und wir notieren $P = B \times_A C$.

Für zwei Morphismen $f, g : A \longrightarrow B$ einer Kategorie $\mathcal{C}$ nennen wir einen Morphismus $e : E \longrightarrow A$ einen *Equalizer*, wenn $f \circ e = g \circ e$ gilt und wenn für alle Morphismen $u : X \longrightarrow A$ in $\mathcal{C}$ mit $f \circ u = g \circ u$ ein eineindeutiger Morphismus $v : X \longrightarrow E$ existiert so, daß $e \circ v = u$ wie im folgendem Diagramm.

$$\begin{tikzcd} E \arrow[r, "e"] & A \arrow[r, shift left] \arrow[r, shift right] & B \\ X \arrow[u, dotted, "v"] \arrow[ur, "u"'] & & \end{tikzcd}$$

Zu jeder vorgestellten Konstruktion existiert eine korrespondierende duale: Coprodukt für Produkt, initiale für terminale Objekte, gefaserte Coprodukte für gefaserte Produkte e.t.c.

3.3 Adjungierte — eineindeutige Beziehungen

Ein weiterer zentraler Begriff ist die *Adjungation.* Betrachten wir zwei Kategorien $\mathcal{C}$ und $\mathcal{D}$ und zwei Funktoren

$$F : \mathcal{D} \longrightarrow \mathcal{C} \qquad G : \mathcal{C} \longrightarrow \mathcal{D}.$$

G ist die *rechts-Adjungierte* zu F (bzw. F ist die *links-Adjungierte* zu G), wenn für jedes Objekt C in $\mathcal{C}$ und D in $\mathcal{D}$ eine natürliche Bijektion zwischen den Morphismen

$$\frac{f : D \longrightarrow G(C)}{g : F(D) \longrightarrow C}$$

besteht. Diese Beziehung soll im folgenden Sinne natürlich sein. Für alle Morphismen $\alpha : C \longrightarrow C'$ in $\mathcal{C}$ und $\zeta : D' \longrightarrow D$ in $\mathcal{D}$ korrespondieren die Morphismen

$$\frac{D' \xrightarrow{\zeta} D \xrightarrow{f} G(C) \xrightarrow{G(\alpha)} G(C')}{F(D') \xrightarrow{F(\zeta)} F(D) \xrightarrow{g} C \xrightarrow{\alpha} C'}.$$

Wir notieren diese bijektive Korrespondenz mit

$$\Theta : \mathsf{Hom}_{\mathcal{D}}(D, G(C)) \quad \xrightarrow{\cong} \quad \mathsf{Hom}_{\mathcal{C}}(F(D), C), \tag{1}$$

und es gilt

$$\Theta(G(\alpha) \circ f \circ \zeta) \quad = \quad \alpha \circ \Theta(f) \circ F(\zeta).$$

Zu jeder Adjungation existiert die *Einheit* und die *Coeinheit.* Sei $C = F(D)$, mit Θ wie in (1) definiert, so erhalten wir eine eineindeutig beschriebene Abbildung

$$\eta = \eta_D : D \longrightarrow G \circ F(D),$$

so daß $\Theta(\eta_D) = \mathsf{id}_{F(D)}$. Die Abbildung η_D nennen wir die *Einheit* der Adjungation (in $\mathcal{D}$). Für $\zeta = \mathsf{id}_C$, $C = F(D)$, $f = \eta$ und $\alpha = h$ entspricht die Komposition $\alpha \circ h \circ F(\zeta)$ dem Morphismus $h : F(D) \longrightarrow C$ und die Komposition $G(\alpha) \circ f \circ \zeta$ der Verknüpfung

$$D \xrightarrow{\eta} G \circ F(D) \xrightarrow{G(h)} G(C).$$

D.h. f bestimmt in eineindeutiger Weise einen Morphismus h, so daß das Diagramm

$$
\begin{tikzcd}
D \arrow[r, "\eta"] \arrow[dr, "f"'] & GF(D) \arrow[d, dotted, "G(h)"] & & F(D) \arrow[d, dotted, "h"] \\
& G(C) & & C
\end{tikzcd}
$$

kommutiert. Das impliziert bei gegebenem Funktor G, daß das Objekt $F(D)$ eineindeutig (bis auf Isomorphie) bestimmt ist. Die Natürlichkeit in (1) impliziert, daß der Einheitsmorphismus $\eta_X : X \longrightarrow G \circ F(X)$ für Objekte X aus $\mathcal{D}$ eine Komponente einer natürlichen Transformation vom Identitätsfunktor aus $\mathcal{D}$ nach $G \circ F : \mathcal{D} \longrightarrow \mathcal{D}$ mit $\Theta(\eta_D \circ \zeta) = F(\zeta) = \Theta((G \circ F)(\zeta) \circ \eta_{D'})$ ist.

Dual zur Einheit einer Adjungation existiert die *Coeinheit.* In der Korrespondenz (1) wählen wir $D = G(C)$ und $f = \mathsf{id}_{G(C)}$. Damit ergibt sich der Morphismus h zu

$$\epsilon = \epsilon_A : F \circ G(C) \longrightarrow C.$$

Das definiert eine natürliche Transformation von $F \circ G$ auf den Identitätsfunktor aus $\mathcal{C}$. Die Coeinheit besitzt die universelle Eigenschaft, daß für jeden Morphismus $h : F(D) \longrightarrow C$ ein einziger Morphismus existiert f, der das Diagramm

$$
\begin{tikzcd}
D \arrow[d, dotted, "f"'] & F(D) \arrow[d, dotted, "F(f)"'] \arrow[dr, "h"] & \\
G(C) & FG(C) \arrow[r, "\epsilon"] & C
\end{tikzcd}
$$

kommutativ macht. Die rechts-Adjungierte G (falls existent) eines Funktors F ist bis auf Isomorphie eineindeutig.

In der Funktor-Kategorie erhalten wir die kommutativen Dreiecke

$$
\begin{tikzcd}
F \arrow[d, dotted, "F(\eta)"'] \arrow[dr, "\mathsf{id}_F"] & & G \arrow[r, "\eta(G)"] \arrow[dr, "\mathsf{id}_G"'] & GFG \arrow[d, "G(\epsilon)"] \\
FGF \arrow[r, "\epsilon(F)"] & F & & G.
\end{tikzcd}
$$

Als Beispiel betrachten wir die Produkt-Kategorie $\mathcal{C} \times \mathcal{C}$ ($\mathcal{C} \times \mathcal{C}$ kann auch als $\mathcal{C}^{\mathbf{2}}$ interpretiert werden, wobei $\mathbf{2}$ die diskrete Kategorie[8] mit zwei Objekten ist). Existiert das Produkt $A \times B$ für beliebige Objekte A und B in $\mathcal{C}$, so existiert auch

[8]Diskrete Kategorien sind Kategorien in denen es außer den Identitäten keine weiteren Morphismen gibt.

der Funktor $\times : \mathcal{C} \times \mathcal{C} \longrightarrow \mathcal{C}$ mit dem Diagonalfunktor $\mathcal{C} \longrightarrow \mathcal{C} \times \mathcal{C}$, $C \mapsto C \times C$ als rechts-Adjungierten. Wenn alle Produkte in $\mathcal{C}$ existieren, so existiert auch der Funktor $A \times \_ : \mathcal{C} \longrightarrow \mathcal{C}$. Besitzt dieser Funktor eine rechts-Adjungierte, notiert man sie mit

$$(-)^A : \mathcal{C} \longrightarrow \mathcal{C},$$

und wir sagen A ist ein Exponent in $\mathcal{C}$. Daß $(-)^A$ eine rechts-Adjungierte ist bedeutet, daß eine bijektive Korrespondenz

$$\frac{C \longrightarrow B^A}{A \times C \longrightarrow B}$$

existiert, die natürlich in B und C ist. Die Coeinheit dieser Adjungation ist die Abbildung

$$\epsilon : A \times B^A \longrightarrow B,$$

mit der Eigenschaft, daß für alle Morphismen $f : C \longrightarrow B^A$ ein (einziger) Morphismus $h : A \times C \longrightarrow B$ existiert. Im Diagramm

$$\begin{tikzcd} A \times B^A \arrow[r, "\epsilon"] & B \\ A \times C. \arrow[u, "\mathsf{id}_A \times f"] \arrow[ur, dotted, "h"'] & \end{tikzcd}$$

Man nennt die Coeinheit ϵ in der Informatik häufig auch Evaluation, da diese Konstruktion der β-Konversion im λ-Kalkül entspricht.

Eine Kategorie ist *kartesisch abgeschlossen*, wenn alle endlichen Produkte (oder äquivalent ein terminales Objekt und alle binären Produkte) existieren, und wenn alle Objekte Exponenten in $\mathcal{C}$ sind (d.h. für alle Objekte A und B existieren die Objekte B^A).

3.4 Limite — Diagramme, Kegel und Universalität

Die letzte kategorielle Konstruktion, die wir hier betrachten, sind *Limite*. Sei $\mathcal{C}$ eine Kategorie. Für eine kleine Kategorie J, die *Index-Kategorie*, betrachten wir die Funktor-Kategorie $\mathcal{C}^J$. Ein Objekt in $\mathcal{C}^J$ nennen wir ein *Diagramm* (vom Typ J).

Analog zum Diagonalfunktor legt jedes Objekt C in $\mathcal{C}$ ein Diagonal-Diagramm $\Delta_J(C)$ fest, der alle $j \in J$ auf C schickt.

Eine natürliche Transformation π vom Diagramm $\Delta_J(C)$ in ein Diagramm A aus $\mathcal{C}^J$ besteht aus den Indizes $j \in J$ so, daß jedes Dreieck

$$\begin{tikzcd} & C \arrow[dl, "f_j"'] \arrow[dr, "f_k"] & & & j \arrow[r, "u"] & k \\ A(j) \arrow[rr, "A(u)"] & & A(k) & & & \end{tikzcd}$$

für alle Morphismen $u \in \mathcal{C}$ kommutiert. Diese natürliche Transformationen nennt man *Kegel* $f : C \longrightarrow A$ (über dem Diagramm A mit Spitze C).

Ein Kegel $\pi : L \longrightarrow A$ ist *universell*, wenn es für alle Kegel $f : C \longrightarrow A$ einen eindeutigen (vermittelnden) Morphismus $g : C \longrightarrow L$ in $\mathcal{C}$ mit $\pi_j \circ g = f_j$ für alle $j \in J$ gibt. Im Diagramm

$$\begin{tikzcd} C \arrow[rr, dotted, "g"] \arrow[dr, "f_j"] \arrow[ddr, "f_k"'] & & L \arrow[dl, "\pi_j"'] \arrow[ddl, "\pi_k"] \\ & A(j) \arrow[d, "A(u)"] & \\ & A(k). & \end{tikzcd}$$

Dieser universelle Kegel $\pi : L \longrightarrow A$ (oder nur seine Spitze L) nennen wir den *Limes* $\mathsf{lim}_J A$ des Diagrammes A. Besitzt jedes Diagramm A in $\mathcal{C}^J$ einen Limes, dann hat der Diagonal-Funktor Δ_J eine rechts-Adjungierte

$$\mathsf{lim}_J : \mathcal{C}^J \longrightarrow \mathcal{C}.$$

Die Coeinheit dieser Adjungation ist die natürliche Transformation

$$\pi : \Delta_J(L) = \Delta_J(\mathsf{lim}_J A) \longrightarrow A.$$

Die duale Konstruktion heißt *Colimes*. Ein *Cokegel* mit Spitze C über einem Diagramm $A : J \longrightarrow C$ ist ein Morphismus $A \longrightarrow \Delta_J(C)$ in der Funktor-Kategorie $\mathcal{C}^J$. Der universelle Cokegel, falls existent, nennen wir den Colimes über dem Diagramm A. Seine Spitze notieren wir mit $\mathsf{co\text{-}lim}_J A$. Existiert der Colimes von einem Diagramm vom Typ J in $\mathcal{C}$, dann ist der Funktor

$$\mathsf{co\text{-}lim}_J : C^J \longrightarrow \mathcal{C}$$

die links-Adjungierte zum Diagonalfunktor $\Delta_J : \mathcal{C} \longrightarrow \mathcal{C}^J$.

Durch einen Funktor $G : \mathcal{C} \longrightarrow \mathcal{D}$ wird ein Diagramm-Funktor $G^J : \mathcal{C}^J \longrightarrow \mathcal{D}^J$ induziert. Existieren Limite vom Typ J in $\mathcal{C}$ und $\mathcal{D}$, so erhält man das folgende kommutative Diagramm

$$\begin{tikzcd}
\mathcal{C}^J \arrow[r, "\mathsf{co\text{-}lim}_J"] \arrow[d, "G^J"'] & \mathcal{C} \arrow[d, "G"] \\
\mathcal{D}^J \arrow[r, "\mathsf{lim}_J"] & \mathcal{D}.
\end{tikzcd}$$

Die universelle Eigenschaft (von Limite) impliziert eine kanonische natürliche Transformation

$$\alpha_J : G \circ \mathsf{co\text{-}lim}_J \longrightarrow \mathsf{lim}_J \circ G^J.$$

Man sagt, der Funktor G erhält Limite, wenn α_J ein natürlicher Isomorphismus ist. So erhält G die Limite, wenn G eine links-Adjungierte besitzt. Dual erhalten Funktoren G Colimite, wenn sie eine rechts-Adjungierte besitzen.

In der Funktor-Kategorie $\mathcal{C}^{\mathcal{D}}$ können Limite punktweise berechnet werden. Sei J eine kleine Index-Kategorie und die Limite vom Typ J existieren in $\mathcal{C}$. Dann existieren die Limite diese Typs auch in $\mathcal{C}^{\mathcal{D}}$ und der Evaluations-Funktor $(\_)_D : \mathcal{C}^{\mathcal{D}} \longrightarrow \mathcal{C}$ für ein Objekt D aus $\mathcal{D}$ erhält die Limite.

In anderen Worten, existieren die Limite in $\mathcal{C}$ für ein ein Diagramm $J \longrightarrow \mathcal{C}$, so erhält man in der Kategorie $\mathcal{C}^{\mathcal{D}}$ für ein Diagramm $A : J \longrightarrow \mathcal{C}^{\mathcal{D}}$ vom Typ J den Limes, indem man die Diagramme $A_D : J \longrightarrow \mathcal{C}$ für Objekte D in $\mathcal{D}$ betrachtet, wobei A_D via

$$A_D(j) \quad = \quad (A(j))(D)$$

definiert ist. Besitzt jedes dieser Diagramme A_D einen Limes $L_D = \mathsf{lim}_J A_D$ in $\mathcal{C}$, so lassen sich diese Limite zu einem Funktor $L : \mathcal{D} \longrightarrow \mathcal{C}$ kombinieren, und dieser Funktor ist der Limes des Diagrammes A. In Zeichen

$$(\mathsf{lim}_J A)(D) \quad = \quad \mathsf{lim}_J A_D$$

für alle Objekte D in $\mathcal{D}$, wobei der linke Limes in $\mathcal{C}^{\mathcal{D}}$ gebildet wird und der rechte in $\mathcal{C}$.

Entsprechend Duales gilt auch für Colimite und führt zur Identität

$$(\mathsf{co\text{-}lim}_J A)(D) \quad = \quad \mathsf{co\text{-}lim}_J A_D.$$

3.5 Spezielle Funktor-Kategorien, die Topoi

Viele Konstruktionen hängen nicht nicht nur von den mathematischen Objekten, sondern auch wesentlich von den Beziehungen zwischen ihnen, den Morphismen, ab. Ein häufiger (in der Natur) vorkommendes Konstruktionsset kann in einer entsprechenden Kategorie mit gewissen Eigenschaften formalisiert werden.

Ein *Topos* ist eine Kategorie, in der die oben vorgestellten fundamentalen Konstruktionen wie Produkte, gefaserte Produkte, Exponenten, u.v.m. für alle Objekte existieren. Mit diesen Konstruktionen etablieren sich viele andere interessante Eigenschaften. Sehr viele konkrete Kategorien sind Topoi (sing.: Topos) (darunter auch der Pate für alle Kategorien, die Mengen mit den Funktionen).

Im folgenden betrachten wir die Funktor-Kategorie $\mathbf{Set}^{\mathcal{C}^{\mathsf{op}}}$ der kontravarianten Funktoren von einer kleinen Kategorie $\mathcal{C}$ nach $\mathbf{Set}$, die Kategorie, in der die Objekte Mengen und die Morphismen Abbildungen sind. $\mathbf{Set}^{\mathcal{C}^{\mathsf{op}}}$ ist die Funktor-Kategorie mit den Objekten $F : \mathcal{C}^{\mathsf{op}} \longrightarrow \mathbf{Set}$ und den natürlichen Transformationen als Morphismen. Jedes Objekt ist also ein mengenwertiger kontravarianter Funktor auf $\mathcal{C}$. Diese Funktoren nennen wir Prägarben und die Kategorie $\hat{\mathcal{C}} = \mathbf{Set}^{\mathcal{C}^{\mathsf{op}}}$ bezeichnen wir als die Kategorie der Prägarben (über $\mathcal{C}$).

Ist P eine Prägarbe über $\mathcal{C}$ und $x \in P(C)$, dann nennen wir den Wert $(P(f))(x)$ von einem Morphismus $f : D \longrightarrow C$ in $\mathcal{C}$ die Restriktion von x (in f) und notieren

$$(P(f))(x) \quad = \quad x \cdot f.$$

Wir schreiben f nach x um den kontravarianten Charakter von P hervorzuheben. Für ein Kompositum $f \circ g$ gilt $x \cdot (f \circ g) = (x \cdot f) \cdot g$.

Zu jedem Objekt C in $\mathcal{C}$ kann man in kanonischer Weise eine Prägarbe $\mathsf{y}(C)$ über $\mathcal{C}$ mit

$$(\mathsf{y}(C))(D) \quad = \quad \mathsf{hom}_{\mathcal{C}}(D, C),$$

für ein Objekt D in $\mathcal{C}$, und

$$\begin{aligned} (\mathsf{y}(C))(f) : \mathsf{hom}_{\mathcal{C}}(D, C) &\longrightarrow \mathsf{hom}_{\mathcal{C}}(D', C) \\ u &\longmapsto u \circ f, \end{aligned}$$

für einen Morphismus $f : D \longrightarrow D'$ in $\mathcal{C}$, konstruieren. Wir schreiben kurz $\mathsf{y}(C) = \mathsf{hom}(-, C)$. Prägarben, die diese Struktur besitzen, nennt man *repräsentierbar*.

Wenn $f : C \longrightarrow C'$ ein Morphismus in $\mathcal{C}$ ist, dann existiert eine natürliche Transformation $\mathsf{y}(C) \longrightarrow \mathsf{y}(C')$ (via Komposition mit f). Damit ist y ein Funktor

$$\mathsf{y} : \mathcal{C} \longrightarrow \mathbf{Set}^{\mathcal{C}^{\mathsf{op}}} \qquad X \longrightarrow \mathsf{hom}_{\mathcal{C}}(-, C).$$

Man nenn y die *Yoneda-Einbettung.* Die Yoneda-Einbettung ist surjektiv und treu. Das folgt aus dem Yoneda-Lemma, das eine bijektive Korrespondenz zwischen natürlichen Transformationen $\mathsf{y}(C) \longrightarrow P$ und den Elementen aus $P(C)$ postuliert:

$$\begin{aligned} \Theta : \mathsf{hom}_{\hat{\mathcal{C}}}(\mathsf{y}(C), P) &\xrightarrow{\cong} P(C) \\ \alpha &\longrightarrow \alpha_C(\mathsf{id}_C), \end{aligned}$$

für $\alpha : \mathsf{y}(C) \longrightarrow P$.

In der Kategorie $\mathbf{Set}/J$ sind die Objekte die Mengen über der Menge J, d.h., nach Definition von Komma-Kategorien, sind die Objekte Funktionen $g : X \longrightarrow J, h : Y \longrightarrow J$, und die Morphismen sind die Abbildungen $f : X \longrightarrow Y$, die die Dreiecke

$$\begin{tikzcd} X \arrow[rr, "f"] \arrow[dr, "g"'] & & Y \arrow[dl, "h"] \\ & J & \end{tikzcd}$$

kommutativ machen. Diese Komma Kategorie ist immer auch isomorph zu einer Funktor-Kategorie. Jedes Objekt $h : X \longrightarrow J$ legt eine J-indizierte Familie $\{H_j \mid j \in J\}$ von Mengen

$$H_j \quad = \quad h^{-1}(j) \quad = \quad \{x \in X \mid h(x) = j\}$$

fest. Jeder Morphismus $f : g \longrightarrow h$ bestimmt eine J-indizierte Familie

$$F \quad = \quad \{f_j : G_j \longrightarrow H_j \mid j \in J\},$$

mit $G_j = g^{-1}(j)$. Betrachten wir die Menge J als diskrete Kategorie (die Kategorie mit den Objekten $j \in J$ und den Idenditätsmorphismen id_j), dann ist jede J-indizierte Funktionen-Familie F eine natürliche Transformation $F : G \longrightarrow H$. D.h.

$$h \longmapsto \{H_j\} \qquad f \longmapsto \{f_j\}$$

bildet einen Funktor $L : \mathbf{Set}/J \longrightarrow \mathbf{Set}^{\mathcal{C}^{\mathsf{op}}}$.

Umgekehrt existiert für jeden Funktor $H : J \longrightarrow \mathbf{Set}$ eine Menge über J, d.h. eine Funktion $h : X \longrightarrow J$, wobei X das Coprodukt (disjunkte Vereinigung in $\mathbf{Set}$) $X = \coprod_{j \in J} H_j$ der Mengen H_j, $j \in J$ ist, und die Abbildungen h jedes H_j auf seinen Index abbildet.

Diese Konstruktion führt zu zwei Funktoren L und M

$$\begin{tikzcd} \mathbf{Set}/J \arrow[r, shift left] & \mathbf{Set}^J \arrow[l, shift left] \end{tikzcd}$$

wobei LM und ML natürlich und isomorph zu den jeweiligen Identitätsfunktoren sind. Diese Konstruktion stellt eine Verbindung zwischen Komma-Kategorien und Funktor-Kategorien von $\mathbf{Set}$ dar. Diese Beziehung ist aber lediglich eine Äquivalenz und kein Isomorphismus, da der Funktor ML nicht der Identitäts-Funktor selbst ist.

Es existieren auch gefaserte Produkte in der Kategorie $\mathbf{Set}$. Das gefaserte Produkt $P = X \times_B Y$ von

$$\begin{tikzcd} P \arrow[r, dotted, "f'"] \arrow[d, dotted, "g'"'] & Y \arrow[d, "g"] \\ X \arrow[r, "f"] & B \end{tikzcd}$$

ist die Menge der geordneten Paare (x, y) von Elementen $x \in X$ und $y \in Y$ mit $f(x) = g(y)$. Speziell, wenn Y eine Teilmenge von B und $g : Y \longrightarrow B$ die Inklusion $Y \hookrightarrow B$ ist, dann wird P das inverse Bild $f^{-1}(Y)$ (bis auf Isomorphie). Sind f und g beides Inklusionen, so ist P die Schnittmenge $f^{-1}(B) \cap g^{-1}(B)$.

Da alle (endlichen) gefaserten Produkte in **Set** existieren, existieren sie auch in den Funktor-Kategorien $\mathbf{Set}^{\mathcal{C}^{\mathsf{op}}}$ und können dort punktweise bestimmt werden. Sind X, Y und B Funktoren und f und g natürliche Transformationen wie im obigen Diagramm, dann ist $P : \mathcal{C}^{\mathsf{op}} \longrightarrow \mathbf{Set}$ der Funktor, der jedes C aus $\mathcal{C}$ auf das gefaserte Produkt von $X(C) \longrightarrow B(C)$ und $Y(C) \longrightarrow X(C)$ schickt. In Notation

$$(X \times_B Y)(C) \xrightarrow{\cong} X(C) \times_{B(C)} Y(C).$$

Die einelementigen Mengen in **Set** sind die terminalen Objekte. Damit sind in $\mathbf{Set}^{\mathcal{C}^{\mathsf{op}}}$ die Funktoren, die jedes Element auf (die gleiche) einelementige Menge schicken, die terminalen Objekte.

Gefaserte Produkte und terminale Objekte sind Limite. Ein gefasertes Produkt ist der Limes des Funktors

$$(\bullet \to \bullet \leftarrow \bullet) \longrightarrow \mathcal{C},$$

während terminale Objekte Limite von der (formal) objektlosen Kategorie in die Kategorie $\mathcal{C}$ sind.

Umgekehrt besitzt eine Kategorie $\mathcal{C}$ alle (endlichen) Limite, wenn sie alle gefaserten Produkte und terminale Objekte enthält. Denn dann besitzt die Kategorie $\mathcal{C}$ (binäre) Produkte $X \times Y$ (die isomorph zum gefaserten Produkt $X \times_1 Y$ sind). Also existieren alle endlichen Produkte. Desweiteren existieren Equalizer für alle Morphismen $f, g : X \longrightarrow Y$, denn diese lassen sich auch auf ein gefasertes Produkt reduzieren. Wir erhalten den Morphismus e via

$$\begin{tikzcd}
E \arrow[r, "fe = ge"] \arrow[d, "e"'] & Y \arrow[d, "\Delta"] \\
X \arrow[r, "\langle f, g \rangle"] & Y \times Y
\end{tikzcd}$$

wobei Δ der Diagonalmorphismus $\langle \mathsf{id}_Y, \mathsf{id}_Y \rangle$ und $\langle f, g \rangle$ der Morphismus aus dem Produktdiagramm von f und g ist. Mit den Equalizern und den Produkten lassen sich die Limite von jedem Diagramm $D : J \longrightarrow \mathcal{C}$ vom Typ J konstruieren. Sei $X = \prod_{(f:i \to j) \in J} D(j)$ das Produkt, das über alle Morphismen in J variiert. Für jeden Morphismus $f : i \longrightarrow j$ existieren zwei mögliche Wege von $\prod_{i \in J} D(i)$, dem Produkt, das über die Objekte in J variiert, nach $D(j)$, $j \in J$, zu gelangen: entweder über den Projektionsmorphismus $\pi_j : \prod_{i \in J} D(i) \longrightarrow D(j)$ oder via $D(f) \circ \pi(f)$, dem entsprechenden Projektionsmorphismus aus dem Produkt $\prod_{(f:i \to j) \in J} D(j)$. Wir erhalten zwei Familien von Morphismen von $\prod_{i \in J} P(i)$ nach $D(j)$, j in J. Da $\prod_{(f:i \to j) \in J} D(j)$ ein Produkt(objekt) ist, existiert für jede Familie ein Morphismus von $\prod_{i \in J} D(i)$ nach $\prod_{(f:i \to j) \in J} D(j)$, wie in folgendem Diagramm.

$$\begin{tikzcd}
 & D(j) \\
\prod_{i\in J} D(i) \arrow[ur, "\pi_j"] \arrow[r, shift left] \arrow[r, shift right] \arrow[d, "\pi_i"'] & \prod_{(f:i\to j)\in J} D(j) \arrow[u, "\pi_f"'] \arrow[d, "\pi_f"] \\
D(i) \arrow[r, "D(f)"] & D(j)
\end{tikzcd}$$

Die Verfeinerung von $\prod_{i\in J} D(i)$, die wir benötigen, ist ein Morphismus $L \longrightarrow \prod_{i\in J} D(i)$, der einen Morphismus $\pi_f \circ p$, der direkt nach $D(j)$ führt, mit einem Morphismus $\pi_f \circ q = D(f) \circ \pi_i$, der über einen Diagrammorphismus führt, identifiziert. Das erreichen wir durch den Equalizer

$$e : L \longrightarrow \prod_{i\in J} D(i)$$

von p und q wie im Diagramm

$$\begin{tikzcd}
L \arrow[r, "e"] & \prod_{i\in J} D(i) \arrow[r, shift left] \arrow[r, shift right] \arrow[d, "\pi_i"'] \arrow[dr, "\pi_j"] & \prod_{(f:i\to j)\in J} D(j) \arrow[d, "\pi_f"] \\
 & D(i) \arrow[r, "D(f)"] & D(j)
\end{tikzcd}$$

Wir erhalten den universellen Kegel

$$\begin{tikzcd}
X \arrow[rr, "u"] \arrow[dr] \arrow[ddr, "g_j"'] & & L \arrow[dl, "e"'] \arrow[ddl, "f_j"] \\
 & \prod_{i\in J} D(i) \arrow[d, "\pi_j"'] & \\
 & D(j) &
\end{tikzcd}$$

mit L als Limes.

3.6 Die Yoneda-Einbettung

Bei der Konkretisierung werden wir die Einbettung in die Kategorie der Mengen mit den Funktionen als Pfeilen verwenden. Der Beweis zu den zitierten Aussagen wurde erstmals von dem japanischen Mathematiker Yoneda geführt. Als einer der wohlbekannte Sätze der Theorie der Kategorien findet man ihn in nahezu jedem Lehrbuch über Kategorientheorie. Deshalb verzichten wir auf die Beweisführung. Man beachte aber die Konsequenzen, die sich aus diesem Lemma ergeben, insbesondere im Hinblick auf die Realisierbarkeit von Algorithmen.

Im weiteren sei $\mathcal{C}$ eine kleine Kategorie. Für jedes Objekt a der Kategorie $\mathcal{C}$ ist der mit a assoziierte Hom-Funktor $\mathsf{hom}_{\mathcal{C}}(-,a) : \mathcal{C}^{\mathsf{op}} \longrightarrow \mathbf{Set}$ wie folgt definiert:

- für ein Objekt b in $\mathcal{C}$ gilt

$$\mathsf{hom}(-,a)(b) \quad = \quad \mathsf{Hom}_{\mathcal{C}}(b,a) \quad = \quad \{f \mid b \xrightarrow{f} a \in \mathcal{C}\}$$

- und für jeden Morphismus $f : b \longrightarrow c$ in $\mathcal{C}$ gilt

$$\begin{array}{rcccc} \mathsf{hom}(-,a)(f) & = & \mathsf{Hom}_{\mathcal{C}}(c,a) & \longrightarrow & \mathsf{Hom}_{\mathcal{C}}(b,a) \\ & & g & \longmapsto & g \circ f \end{array}$$

Eine natürliche Transformation η, von einem Funktor $F : \mathcal{C} \longrightarrow \mathcal{D}$ in einen Funktor $G : \mathcal{C} \longrightarrow \mathcal{D}$, ist eine, über Objekte der Kategorie $\mathcal{C}$ indizierte, Familie $(\eta_a)_{a \in \mathcal{C}}$ von Morphismen der Kategorie $\mathcal{D}$, so daß die Diagramme

$$\begin{tikzcd} F(a) \arrow[r, "\eta_a"] \arrow[d, "F(f)"'] & G(a) \arrow[d, "G(f)"] \\ F(b) \arrow[r, "\eta_b"] & G(b) \end{tikzcd}$$

für alle Morphismen $f : a \longrightarrow b$ kommutieren.

Ein Funktor $K : \mathcal{C} \longrightarrow \mathbf{Set}$ heißt repräsentierbar, wenn ein Objekt r in $\mathcal{C}$ und ein natürlicher Isomorphismus $\Phi : K \longrightarrow \mathsf{hom}_{\mathcal{C}}(r, \_)$ existiert. Dual dazu heißt $K : \mathcal{C}^{\mathsf{op}} \longrightarrow \mathbf{Set}$ repräsentierbar, wenn ein Objekt r in $\mathcal{C}$ und ein natürlicher Isomorphismus $\Phi : K \longrightarrow \mathsf{hom}_{\mathcal{C}}(\_, r)$. Im weiteren erwähnen wir die dualen Aussagen nicht mehr explizit.

Für einen Funktor $K : \mathcal{C} \longrightarrow \mathbf{Set}$ ist jede natürliche Transformation

$$\varphi : \ \mathsf{hom}_{\mathcal{C}}(r, \_) \quad \longrightarrow \quad K$$

bereits vollständig durch die Bilder von id_r bestimmt. Für einen Morphismus $f \in \mathsf{Hom}_C(r,d)$ gilt $\varphi_d(f) = \varphi_d(f \circ \mathsf{id}_r) = K(f) \circ \varphi_r(r)$ wie man dem kommutierenden Diagramm

$$
\begin{tikzcd}
\mathsf{Hom}_{\mathcal{C}}(r,r) \arrow[r, "\varphi_r"] \arrow[d, "f \circ \_"'] & K(r) \arrow[d, "K(f)"] & r \arrow[d, "f"] \\
\mathsf{Hom}_{\mathcal{C}}(r,d) \arrow[r, "\varphi_d"] & K(d) & d
\end{tikzcd}
$$

entnimmt.

Wir werden eine von Yoneda entdeckte Einbettung in die Kategorie der Mengen **Set** an mehreren Stellen verwenden.

Lemma 1 (Yoneda) *Sei $K : \mathcal{C} \longrightarrow$ **Set** ein Funktor. Dann ist die Abbildung*

$$
\begin{array}{rccc}
\Psi_{r,K}: & \mathsf{Nat}(\mathsf{hom}_{\mathcal{C}}(r,\_),K) & \longrightarrow & K(r) \\
 & \varphi : \mathsf{hom}_c(r,\_) \longrightarrow K & \longmapsto & \varphi_r(\mathsf{id}_r)
\end{array}
$$

ein natürlicher Isomorphismus in r und K. Das bedeutet die beiden Diagramme

$$
\begin{tikzcd}
\mathsf{Nat}(\mathsf{hom}_{\mathcal{C}}(r,\_),K) \arrow[r, "\Psi_{r,K}"] \arrow[d, "\mu \circ \_"'] & K(r) \arrow[d, "K(f)"] & K \arrow[d, "\mu"] \\
\mathsf{Nat}(\mathsf{hom}_{\mathcal{C}}(r,\_),H \arrow[r, "\Psi_{r,H}"] & H(r) & H
\end{tikzcd}
$$

$$
\begin{tikzcd}
\mathsf{Nat}(\mathsf{hom}_{\mathcal{C}}(r,\_),K) \arrow[r, "\Psi_{r,K}"] \arrow[d, "\_ \circ f"'] & K(r) \arrow[d, "K(f)"] & r \arrow[d, "f"] \\
\mathsf{Nat}(\mathsf{hom}_{\mathcal{C}}(d,\_),K \arrow[r, "\Psi_{d,K}"] & K(d) & d
\end{tikzcd}
$$

kommutieren.

Theorem 1 *Sei y die Abbildung*

$$
\begin{array}{rccc}
\mathsf{y}: & r & \longmapsto & \mathsf{hom}_{\mathcal{C}}(r,\_) \\
 & d \xrightarrow{g} r & \longmapsto & \_ \circ g : \mathsf{hom}_{\mathcal{C}}(r,\_) \longrightarrow \mathsf{hom}_{\mathcal{C}}(d,\_),
\end{array}
$$

wobei r ein Objekt aus $\mathcal{C}$ ist und $\_ \circ g : \mathsf{hom}_{\mathcal{C}}(r,\_) \longrightarrow \mathsf{hom}_{\mathcal{C}}(d,\_)$ die kanonische natürliche Transformation ist. Dann bettet y die Kategorie $\mathcal{C}^{\mathsf{op}}$ treu in die Funktorkategorie $\textbf{Set}^{\mathcal{C}}$ ein.

Dual dazu ist y die Abbildung, die jedem Objekt r aus $\mathcal{C}$ auf den Hom-Funktor $\mathsf{hom}_{\mathcal{C}}(\_,r)$ und jeden Morphismus $d \xrightarrow{g} r$ auf die natürliche Transformation $g \circ \_ : \mathsf{hom}_{\mathcal{C}}(\_,d) \longrightarrow \mathsf{hom}_{\mathcal{C}}(\_,r)$ schickt. Dann ist y ein treuer kontravarianter Funktor, der die Kategorie $\mathcal{C}$ in die Funktorkategorie $\textbf{Set}^{\mathcal{C}^{\mathsf{op}}}$ abbildet.

4 Applikation der Theorie

Die bis jetzt vorgestellten Konzepte eignen sich zur konstruktiven Beschreibung eines Frameworks. Dazu müssen lediglich die abstrakten mathematischen Objekte in eine konkrete Programmiersprache übersetzt werden.

Im folgenden ist ein offenes ML-Modul (evaluierbarer bzw. compilierbarer Sourcecode) beschrieben, das Kategorien und einige der im vorhergehenden Kapitel beschriebenen Konstruktionen definiert. Insbesondere ist das Limes-Theorem vollständig realisiert.

```
(*******************************************************************
********************************************************************
********
******** Concrete Categories, base definitions
******** Language SML/NY
******** Author M. Bertol (c)
********
******** Release: Tue Aug 13 11:35:40 MET DST 1996
********
********************************************************************
********************************************************************)


(*** datatype for a category ***)

datatype('o, 'a)Cat = cat of ('a->'o) * ('a->'o) * ('o->'a) * ('a*'a->'a);      cat

fun dom( cat(d, _,_,_)) = d;                                                     dom
fun cod( cat(_, c, _,_)) = c;                                                    cod
fun id( cat(_,_, i,_)) = i;                                                      id
fun compose( cat(_,_,_, c)) = c;                                                 compose


(*** universal construction: initial object ***)

type ('o,'a)InitialObj = 'o * ('o -> 'a);                                        type

fun init_obj( (ob, _): ('o,'a)InitialObj) = ob;                                  init_obj
fun init_univ( (_, ar): ('o,'a)InitialObj) = ar;                                 init_univ
fun iso_initial((a,univ_a), (a',univ_a')) = (univ_a(a'),univ_a'(a));


(*** datatype for functors ***)

datatype ('oA, 'aA, 'oB, 'aB)Functor =
                    cat_functor of ('oA, 'aA)Cat *                               cat_functor
                               ('oA -> 'oB) * ('aA -> 'aB) *
                               ('oB, 'aB)Cat;

fun domain(cat_functor(D,_,_,_)) = D;                                            domain
```

```
fun range(cat_functor(_,_,_,R)) = R;                                         range
fun apply_obj(cat_functor(_,fobj,_,_)) = fobj;                               apply_obj
fun apply_arr(cat_functor(_,_,farr,_)) = farr;                               apply_arr


(*** duality operations ***)

fun dual_Cat(cat(s,t,i,c)) = cat(t, s, i, fn (g,f) => c(f,g));               dual_Cat

fun dual_Fun(cat_functor(A,fo,fa,B)) =                                       dual_Fun
                  cat_functor(dual_Cat(A), fo, fa, dual_Cat(B));



(*** universal construction: coproduct ***)

type ('o, 'a)Coproduct_CoCone = ('o * 'a * 'a) * ('o * 'a *'a -> 'a);        type

type ('o, 'a)CoProduct = 'o * 'o -> ('o, 'a)Coproduct_CoCone;                type



(*** universal construction: coequalizer ***)

type ('o, 'a)CoEqualizer = 'a * 'a -> ('o * 'a) * ('o * 'a -> 'a);           type



(*** universal construction: pushout ***)

type ('o, 'a)Pushout = 'a * 'a -> ('a * 'a) * ('a * 'a -> 'a);               type



(*** datatype for diagram graphs ***)

exception undefined_Edge;
exception undefined_Vertex;

datatype ('v,'e)multiGraph = graph of ('v list * 'e list *                   graph
                             ('e -> 'v) * ('e -> 'v));

val nil_graph = graph([], [],                                                nil˙graph
                 fn _ => raise undefined_Edge,
                 fn _ => raise undefined_Edge);

fun is_nil_graph(graph([],_,_,_)) = true                                     is˙nil˙graph
  | is_nil_graph(graph(_,_,_,_)) = false;



(*** datatype for diagrams ***)

exception undefined;
```

```
datatype ('n, 'e, 'o, 'a)Diagram =
          diagram of ('n, 'e)multiGraph * ('n -> 'o) * ('e -> 'a);

fun is_nil_diagram(diagram(g,_,_)) = is_nil_graph(g);
val nil_diagram = diagram(nil_graph,
                  fn x => raise undefined,
                  fn y => raise undefined);
fun obj_at_node(diagram(_, objs, _)) = objs;
fun arr_at_edge(diagram(_, _, arrs)) = arrs;

fun nodes(diagram(graph(vertices,_,_,_),_,_)) = vertices;

fun remove_vertex(graph(vertices,edges,source,drain),v) =
  let fun remove(x,[]) = []
      | remove (x,y::l) = if x = y then l
                          else y::remove(x,l);
      fun remove_incident(v,[]) = []
      | remove_incident(v,e::l) = if (drain(e)=v orelse source(e)=v)
                                  then remove_incident(v,l)
                                else remove_incident(v,l)
  in graph(remove(v,vertices),remove_incident(v,edges),source,drain)
  end;

fun reduce_diagram(diagram(g, objs, arrs), n) =
  diagram(remove_vertex(g,n), objs, arrs);


(*** datatype for cocones ***)

datatype ('n, 'e, 'o, 'a)CoCone =
  cocone of 'o * ('n, 'e, 'o, 'a)Diagram * ('n -> 'a);
datatype ('n, 'e, 'o, 'a)CoCone_Arrow =
  cocone_arrow of ('n, 'e, 'o, 'a)CoCone * 'a * ('n, 'e, 'o, 'a)CoCone;

fun co_apex(cocone( ob, _, _)) = ob;
fun sides(cocone(_,_, fu)) = fu;
fun base(cocone(_, di, _)) = di;

fun co_apex_arrow(cocone_arrow(_, ar, _)) = ar;


(*** datatype for limits ***)

type ('n, 'e, 'o, 'a)Colimiting_CoCone =
  ('n, 'e, 'o, 'a)CoCone *
  (('n, 'e, 'o, 'a)CoCone -> ('n, 'e, 'o, 'a)CoCone_Arrow);

type ('n, 'e, 'o, 'a)CoLimit =
  ('n, 'e, 'o, 'a)Diagram -> ('n, 'e, 'o, 'a)Colimiting_CoCone;


(*** datatype for a complete category ***)
```

complete˙cat

```
datatype ('n, 'e, 'o, 'a)CoComplete_Cat =
   cocomplete_cat of ('o, 'a)Cat * ('n, 'e, 'o, 'a)CoLimit;


(*** datatype for a cocontinous functor ***)
```

cocontinous˙functor

```
datatype ('n, 'e, 'oA, 'aA, 'oB, 'aB)CoContinous_Functor =
   cocontinous_functor of ('oA, 'aA, 'oB, 'aB)Functor *
   (('n, 'e, 'oA, 'aA)Colimiting_CoCone -> ('n, 'e, 'oB, 'aB)Colimiting_CoCone);


(*** functions for determining colimits ***)
```

finite˙coproduct

```
fun finite_coproduct(C,initial,b_coprod) =
   fn d => if is_nil_diagram(d)
            then let val (i, i_univ) = initial;
                   val i_cocone = cocone(i,
                                  nil_diagram,
                                  fn x => raise undefined)
                 in (i_cocone,
                    fn c1 => cocone_arrow(i_cocone, i_univ(co_apex(c1)), c1))
                 end
         else let val n::_ = nodes(d) ;
                 val (c,univc) = finite_coproduct(C,initial,b_coprod)
                                     (reduce_diagram(d,n));
                 val ((b,f,g), univcp) = b_coprod(co_apex(c),obj_at_node(d)(n));
                 val result_cocone = cocone(b,d,
                                   fn m => if m=n then g
                                           else compose(C)(f,sides(c)(m)));
                 val universal = fn c1 => let val u = co_apex_arrow(univc(c1))
                                     val v = univcp(co_apex(c1),
                                                    u,
                                                    sides(c1)(n))
                                  in cocone_arrow(result_cocone, v, c1)
                                  end
             in (result_cocone, universal)
             end;
```

add˙edge

```
fun add_edge (C,coeq) =
   fn ((c,univ), e) => let val diagram(g, fo, fa) = base(c);
                       val graph(vertices,edges,s,t) = g;
                       val ((b,h),ce_univ) = coeq(sides(c)(s(e)),
                                           compose(C)(sides(c)(t(e)),
                                                      fa(e)));
                       val result_graph = graph(vertices,e::edges,s,t);
                       val result_diagram = diagram(result_graph, fo, fa);
                       val result_cocone = cocone(b,
                                           result_diagram,
                                           fn n => compose(C)
                                                   (h,sides(c)(n)));
                       val universal =
                           fn c1 => let val u = co_apex_arrow(univ(c1))
                                        val v = ce_univ(co_apex(c1),u)
```

```
                                in cocone_arrow(result_cocone, v, c1)
                                end
                 in (result_cocone, universal)
                 end;


(*** categorie with initials, coproducts, and coequalizers ***)

datatype ('o, 'a)IO_CP_CE_Cat = io_cp_ce_cat of ('o, 'a)Cat *                 io_cp_ce_cat
                                          ('o, 'a)InitialObj *
                                          ('o, 'a)CoProduct *
                                          ('o, 'a)CoEqualizer;


(*** function for computing the limit objects ***)

fun finite_colimit(cC as io_cp_ce_cat(C, init, b_coprod, coeq)) =             finite_colimit
   fn (d as diagram(graph(nodes,edges,s,t), fo, fa)) =>
   if edges=[] then finite_coproduct(C, init, b_coprod)(d)
   else let val e::edges' = edges
          val d1 = diagram(graph(nodes,edges',s,t), fo, fa)
       in add_edge(C,coeq)((finite_colimit cC d1), e)
       end;


(*** datatypes for the dual constructions ***)


(*** datatype for the product ***)

type ('o, 'a)Product = 'o * 'o -> ('o * 'a * 'a) * ('o * 'a * 'a -> 'a);     type



(*** datatype for a cone ***)

datatype ('n, 'e, 'o, 'a)Cone =                                              cone
   cone of 'o * ('n, 'e, 'o, 'a)Diagram * ('n -> 'a);

datatype ('n, 'e, 'o, 'a)Cone_Arrow =                                        cone_arrow
   cone_arrow of ('n, 'e, 'o, 'a)Cone * 'a * ('n, 'e, 'o, 'a)Cone;



(*** datatype for a limit ***)

type ('n, 'e, 'o, 'a)limiting_Cone =                                         type
   ('n, 'e, 'o, 'a)Cone * (('n, 'e, 'o, 'a)Cone -> ('n, 'e, 'o, 'a)Cone_Arrow);

type ('n, 'e, 'o, 'a)Limit = ('n, 'e, 'o, 'a)Diagram -> ('n, 'e, 'o, 'a)limiting_Cone;   type


(*** datatype for a complete category, realized via dualization ***)
```

```
datatype ('n, 'e, 'o, 'a)Complete_Cat =                                    complete_cat
    complete_cat of ('o, 'a)Cat * ('n, 'e, 'o, 'a)Limit;

fun dual_graph(graph(N,E,s,t)) = graph(N,E,t,s);
fun dual_diagram(diagram(g,nm,em)) = diagram(dual_graph(g),nm,em);

fun dual_cone(cocone(a,D,f),univ) =
    let val result_cone = cone(a,dual_diagram(D),f);
        val universal = fn (c1 as cone(a1,D1,f1)) =>
                              let val c2 = cocone(a1,dual_diagram(D1),f1)
                              in cone_arrow(c1,
                                            co_apex_arrow(univ c2),
                                            result_cone)
                              end
    in (result_cone,universal)
    end


fun dual_colimit(colim) = fn D => dual_cone(colim(dual_diagram(D)));      dual_colimit

(*** end of definition ***)
```

Instanziierbare Typen sind am Rand kommentiert. Eine denkbare instanz ist beispielsweise die Kategorie der Mengen. Eine Implementierung könnte beispielsweise folgendermaßen aussehen

```
(*********************************************************************
*********************************************************************
********
******** Concrete Categories, definition of Set
******** Language SML/NY
******** Author M. Bertol (c)
********
******** Release: Tue Aug 13 11:35:40 MET DST 1996
********
*********************************************************************
*********************************************************************)

use"cat.sml";

use"simpleSet.sml";


exception nonCompatiblePair;

datatype ('a)Set_Arrow = set_arrow of ('a Set) * ('a -> 'a) * ('a Set);   set_arrow
```

```
fun set_s(set_arrow(a,_,_)) = a;                                            set˙s

fun set_t(set_arrow(_,_,a)) = a;                                            set˙t

fun set_ident(a) = set_arrow(a, fn x => x, a);

fun set_comp(set_arrow(c,g,d),set_arrow(a,f,b)) =                           set˙comp
                 if set_eq(c,b) then set_arrow(a,fn x => g(f(x)), d)        set˙eq
                 else raise nonCompatiblePair;


(*** instantiation of finite set ***)

val FinSet = cat(set_s, set_t, set_ident, set_comp);


(*** datatype for coloring elelments ***)

datatype 'a Tag = just of 'a | pink of ('a Tag) | blue of ('a Tag) |        Tag
             pair of ('a Tag) * ('a Tag) | ttrue;


(*** instantiation of initial object ***)

exception undefined;

val set_initial = (emptyset, fn x => set_arrow(emptyset,fn y => raise undefined,x));


(*** instantiation of coproduct ***)

fun set_coproduct(a,b) = let val sumab = union(mapset(pink)(a), mapset(blue)(b))   set˙coproduct
                         val univ = fn (c,
                                    set_arrow(_, f, _),
                                    set_arrow(_, g, _)) =>
                            let val fg = fn pink x => f(x)
                                       | blue x => g(x)
                                       | just x => raise undefined
                            in set_arrow(sumab,fg,c)
                            end
                     in ((sumab,
                         set_arrow(a,pink,sumab),
                         set_arrow(b,blue,sumab)),
                        univ)
                     end;

(*** instantiation of coequalizer ***)

fun set_coequalizer(set_arrow(s,f,b), set_arrow(_,g,_)) =                   set˙coequalizer
   let val cat(_,_, id,comp) = FinSet
   in if is_empty(s) then ((b,id(b)), fn (_,j) => j)
     else if cardinality(s) = 1 then let val (y,_) = singleton_split(s)
                               in if f(y) = g(y) then ((b,id(b)), fn (_,j) => j)
                                 else let val b' = remove(g(y),b)
```

```
                                                fun ff(z) = if z=g(y) then f(y)
                                                            else z
                                            in((b', set_arrow(b,ff,b')),
                                                fn (d', set_arrow(_,j,_))
                                                           => set_arrow(b',j,d'))
                                            end
                                    end
            else let val (a,a') = split(s)
                     val ((_,q),univ) = set_coequalizer(set_arrow(a,f,b),
                                                         set_arrow(a,g,b))
                     val ((d,r),univ') = set_coequalizer(comp(q, set_arrow(a',f,b)),
                                                          comp(q, set_arrow(a',g,b)))
                 in ((d,comp(r,q)), fn (d',j) => univ'(d',univ(d',j)))
                 end
    end


(*** instantiation of io_cp_ce_cat ***)

val iocpceFinSet = io_cp_ce_cat(FinSet,set_initial,set_coproduct,set_coequalizer);

val cocomplete_FinSet = cocomplete_cat(FinSet, finite_colimit(iocpceFinSet));


(*** instantiation of terminal object ***)
```

set˙terminal

```
val set_terminal = let val t = singleton(ttrue)
                   in (t,fn a => set_arrow(a,fn x => ttrue,t))
                   end


(*** instantiation of product ***)
```

set˙times

```
fun set_times(a,b) = let val a_list = set_to_list(a)
                         val b_list = set_to_list(b)
                         fun s_mult(s,[],l) = s
                           | s_mult(s,a::l',l) = let fun e_mult(r,x,[]) = r
                                                       | e_mult(r,x,y::l) =
                                                             e_mult(pair(x,y)::r,x,l)
                                                 in s_mult(e_mult(s,a,l),l',l)
                                                 end
                     in list_to_set(s_mult([],a_list,b_list))
                     end;
```

set˙product

```
fun set_product(a,b) = let val a_cross_b = set_times(a,b)
                           val proj_a = set_arrow(a_cross_b, fn pair(y,z) => y, a)
                           val proj_b = set_arrow(a_cross_b, fn pair(y,z) => z, b)
                           val univ = fn (p,
                                          set_arrow(_,f1,_),
                                          set_arrow(_,f2,_)) =>
                                                set_arrow(p,
                                                          fn xy => pair(f1 xy, f2 xy),
                                                          a_cross_b)
```

```
                 in ((a_cross_b,proj_a,proj_b),univ)
                 end


(*** instantiation of equalizer ***)

fun set_equalizer(f,g) =                                          set˙equalizer
   let val a = dom(FinSet)(f)
      fun filter(r,f1,g1,emptyset) = r
        | filter(r,f1,g1,s) = let val (x,s') = singleton_split(s)
                              in if f1(x)=g1(x) then filter(union(singleton(x),r),
                                                         f1,g1,s')
                                 else filter(r,f1,g1,s')
                              end
      val set_arrow(_,f_fn,_) = f
      val set_arrow(_,g_fn,_) = g
      val e = filter(emptyset,f_fn,g_fn,a)
   in ((e,set_arrow(e,fn x => x, a)),
      (fn (e1,h1) => set_arrow(e1,fn y => h1 y,e)))
   end


(*** instantiation of complete_cat ***)

val completeFinSet =                                              completeFinSet
   complete_cat(FinSet,
            dual_colimit(finite_colimit(io_cp_ce_cat(dual_Cat(FinSet),
                                                     set_terminal,
                                                     set_product,
                                                     set_equalizer))))
```

Der realisierende Mengentyp ist abstrahiert. Die einfachste Realisierung könnte so aussehen

```
(*********************************************************************
******** definition of very simple abstype for Sets
******** Language SML/NY
******** Author M. Bertol (c)
*********************************************************************)


abstype 'a Set = set of 'a list
with
   exception emptySet;

   val emptyset = set([]);                                        emptyset
```

```
    fun is_empty(set([])) = true                                                is_empty
      | is_empty(_) = false

    fun singleton(x) = set([x])                                                 singleton

    fun union(set(s),set(t)) = let fun append([],l) = l                         union
                                     | append(x::l,l') = x::append(l,l')
                               in set(append(s,t))
                               end

    fun member(x,set(l)) = let fun list_member(_,[]) = false                    member
                                 | list_member(x,y::l) = if (x=y) then true
                                                         else list_member(x,l)
                           in list_member(x,l)
                           end

    fun remove(x,set(l)) = let fun list_remove(x,[]) = []                       remove
                                 | list_remove(x,y::l) = if (x=y) then list_remove(x,l)
                                                         else y::list_remove(x,l)
                           in set(list_remove(x,l))
                           end
    fun insert(x,set(l)) = set(x::l)                                            insert

    fun singleton_split(set([])) = raise emptySet                               singleton_split
      | singleton_split(set(x::s)) = (x, remove(x,set(s)))

    fun split(set([])) =  raise emptySet                                        split
      | split(set(a::l)) = (set([a]),set(l))

    fun is_subset(set([]),_) = true                                             is_subset
      | is_subset(set(x::s), t) = member(x,t) andalso is_subset(set(s),t)

    fun set_eq(s,t) = is_subset(s,t) andalso is_subset(t,s)                     set_eq

    fun cardinality(set([])) = 0                                                cardinality
      | cardinality(set(a::s)) = 1 + cardinality(remove(a,set(s)))

    fun set_to_list(set(l)) = l                                                 set_to_list

    fun list_to_set(l) = set(l)                                                 list_to_set

    fun mapset(f)(set(l)) = let fun maplist(f)([]) = []                         mapset
                                  | maplist(f)(h::t) = (f(h))::(maplist(f)(t))  maplist
                            in set(maplist(f)(l))
                            end
end;
```

Die Implementierung durch Listen ist strait forward. Da sehr viele Kategorien durch die Kategorie der Mengen repräsentierbar sind lohnt es sich hier effiziente Algorithmen zu verwenden. Ein Verfahren mit logarithmischer Komplexität (in Anzahle der Elemente) könnte auf den folgenden Konstruktoren basieren. Der abstrakte Datentyp ist aus Platzgründen nicht vollständig dargestellt. Der Funktor realisiert jedoch die drei wichtigsten Zugriffsoperationen: insert, delete und lookup.

```
(*****************************************************************
******************************************************************
********
******** TTTtree, a efficient insert/delete data structure
******** Language SML/NY
******** Author M. Bertol (c)
********
******** Release: Thu Aug 15 14:40:26 MET DST 1996
********
******************************************************************
******************************************************************)




signature ITEM =
  sig
    type key                                        key
    val eq : key * key -> bool                      leq
    val leq : key * key -> bool                     eq
  end;

signature SETTREE =
   sig
    type tree                                       tree
    type key                                        key
    val empty : tree                                empty
    val insert : key * tree -> tree                 insert
    val delete : key * tree -> tree                 delete
    val lookup : key * tree -> bool                 lookup
  end;


functor TTTree( Item: ITEM ): SETTREE=              TTTree

struct

  type key = Item.key

  fun leq(p: key, q: key) : bool = Item.leq(p,q);
  fun eq(p: key, q: key) : bool = Item.eq(p,q);

  infix leq;
  infix eq;

  datatype tree = empty
   | tree2 of tree * key * tree * key
   | tree3 of tree * key * tree * key * tree * key
   | leaf of key

  exception  maxEmptyTreeError;
```

```
fun max(empty) = raise maxEmptyTreeError
  | max(leaf(x)) = x
  | max(tree2(_, _, _, x)) = x
  | max(tree3(_, _, _, _, _, x)) = x;


exception   lookupEmptyTreeError;


fun lookup(a: key, empty) = raise lookupEmptyTreeError                     lookup
  | lookup(a: key, leaf(b)) =  a eq b
  | lookup(a: key, tree2(s,x,t,y)) =
        if (x eq a) orelse (y eq a) then true
        else if a leq x
                then lookup(a, s)
             else if a leq y then lookup(a, t)
                  else false
  | lookup(a: key, tree3(r,x,s,y,t,z)) =
        if  (x eq a) orelse (y eq a) orelse (z eq a) then true
        else if a leq x
                then lookup(a, r)
             else if a leq y then lookup(a, s)
                  else if a leq z then lookup(a, t)
                       else false;

datatype insert_status = grown | proceed;


exception  MisteriousException;

exception   insertEmptyTreeError;

exception   deleteEmptyTreeError;


fun insertA(a: key, empty) = (grown, leaf(a))
  | insertA(a: key, leaf(b)) =
         if a eq b then (proceed, leaf(b))
         else if a leq b
                 then (grown, tree2(leaf(a),a , leaf(b), b))
              else (grown, tree2(leaf(b),b , leaf(a), a))
  | insertA(a: key, tree2(s,x,t,y)) =
         if (x eq a) orelse (y eq a) then (proceed, tree2(s,x,t,y))
         else if a leq x
                 then let val (stat, s') = insertA(a, s)
                      in if stat = grown
                             then case s' of
                                 tree2(s1,x1,t1,y1)
                                 =>  (proceed, tree3(s1,x1,t1,y1,t,y))
                                | tree3(_,_,_,_,_,_)
                                 => raise MisteriousException
                                | empty
                                 => raise insertEmptyTreeError
```

```
                              | leaf(_)
                                => raise MisteriousException
                          else (proceed, tree2(s',max(s'),t,y))
                        end
                 else let val (stat, t') = insertA(a, t)
                      in if stat = grown
                             then case t' of
                                 tree2(s1,x1,t1,y1)
                                 =>  (proceed,  tree3(s,x,s1,x1,t1,y1))
                               | tree3(_,_,_,_,_,_)
                                 => raise MisteriousException
                               | empty
                                 => raise  insertEmptyTreeError
                               | leaf(_)
                                 => raise MisteriousException
                          else (proceed, tree2(s,x,t',max(t')))
                      end
  | insertA(a: key, tree3(r,x,s,y,t,z)) =
          if  (x eq a) orelse
              (y eq a) orelse
              (z eq a) then (proceed, tree3(r,x,s,y,t,z))
          else if a leq x then let val (stat, r') = insertA(a, r)
                                   val m = max(r')
                               in if stat = grown
                                      then (grown, tree2(r',m,tree2(s,y,t,z),z))
                                  else (proceed, tree3(r',m,s,y,t,z))
                               end
               else if a leq y
                        then let val (stat, s') = insertA(a, s)
                             in if stat = grown
                                    then case s' of
                                        tree2(s1,x1,t1,y1)
                                        =>  (grown, tree2(tree2(r,x,s1,x1),x1,
                                                         tree2(t1,y1,t,z),z))
                                      | tree3(_,_,_,_,_,_)
                                        => raise MisteriousException
                                      | empty
                                        => raise  insertEmptyTreeError
                                      | leaf(_)
                                        => raise MisteriousException
                                 else (proceed, tree3(r,x,s',max(s'),t,z))
                             end
                    else let val (stat, t') = insertA(a, t)
                             val m = max(t')
                         in if stat = grown
                                then (grown, tree2(tree2(r,x,s,y),y,t',m))
                            else (proceed, tree3(r,x,s,y,t',m))
                         end;


fun insert(a: key, t) = let val (_,t') = insertA(a, t) in t' end;
```

insert

```
datatype delete_status = propagateDelete | ContainedAndDeleted | notContained;
```

```
fun deleteA(a: key, empty) = (notContained, empty)
  | deleteA(a: key, leaf(b)) = if a eq b
                                  then (propagateDelete, empty)
                               else (notContained, leaf(b))
  | deleteA(a: key, tree2(s,x,t,y)) =
        if a leq x then let val (stat, s') = deleteA(a, s)
                        in if stat=propagateDelete
                              then case t of
                                  tree3(r1,x1,s1,y1,t1,z1)
                                  => (ContainedAndDeleted,
                                      tree2(tree2(s',max(s'),r1,x1),x1,
                                            tree2(s1,y1,t1,z1),z1))
                                | tree2(s1,y1,t1,z1)
                                  => (propagateDelete,
                                      tree3(s',max(s'),s1,y1,t1,z1))
                                | leaf(x)
                                  => (propagateDelete, leaf(x))
                                | empty
                                  => raise deleteEmptyTreeError
                           else (stat, tree2(s',max(s'),t,y))
                        end
        else if a leq y then
            let val (stat, t') = deleteA(a, t)
            in if stat=propagateDelete
                  then case s of
                      tree3(r1,x1,s1,y1,t1,z1)
                      => (ContainedAndDeleted,
                          tree2(tree2(r1,x1,s1,y1),y1,
                                tree2(t1,z1,t',max(t')),max(t')))
                    | tree2(s1,y1,t1,z1)
                      => (propagateDelete, tree3(s1,y1,t1,z1,t',max(t')))
                    | leaf(x)
                      => (propagateDelete, leaf(x))
                    | empty
                      => raise deleteEmptyTreeError
               else (stat, tree2(s,x,t',max(t')))
            end
            else (notContained,  tree2(s,x,t,y))
  |  deleteA(a: key, tree3(r,x,s,y,t,z)) =
        if a leq x then let val (stat, r') = deleteA(a, r)
                        in if stat=propagateDelete
                              then case s of
                                  tree3(r1,x1,s1,y1,t1,z1)
                                  => (ContainedAndDeleted,
                                      tree3(tree2(r',max(r'),r1,x1),x1,
                                            tree2(s1,y1,t1,z1),z1,
                                            t,z))
                                | tree2(s1,x1,t1,y1)
                                  => (ContainedAndDeleted,
                                      tree2(tree3(r',max(r'),s1,x1,t1,y1),y1,
                                            t,z))
                                | leaf(x1)
```

```
                              => (ContainedAndDeleted,
                                  tree2(s,y,t,z))
                            | empty
                              => raise deleteEmptyTreeError
                        else (stat, tree3(r',max(r'),s,y,t,z))
                      end
        else if a leq y
             then let val (stat, s') = deleteA(a, s)
                  in if stat=propagateDelete
                     then case r of
                        tree3(r1,x1,s1,y1,t1,z1)
                        => (ContainedAndDeleted,
                            tree3(tree2(r1,x1,s1,y1),y1,
                                  tree2(t1,z1,s',max(s')),max(s'),
                                  t,z))
                      | tree2(s1,x1,t1,y1)
                        => (ContainedAndDeleted,
                            tree2(tree3(s1,x1,t1,y1,s',max(s')),max(s'),
                                    t,z))
                      | leaf(x1)
                        => (ContainedAndDeleted,
                            tree2(r,x,t,z))
                      | empty
                        => raise deleteEmptyTreeError
                    else (stat, tree3(r,x,s',max(s'),t,z))
                  end
             else if a leq z
                  then let val (stat, t') = deleteA(a, t)
                       in if stat=propagateDelete
                          then case s of
                             tree3(r1,x1,s1,y1,t1,z1)
                             => (ContainedAndDeleted,
                                 tree3(r,x,
                                       tree2(r1,x1,s1,y1),y1,
                                       tree2(t1,z1,t',max(t')),max(t')))
                           | tree2(s1,x1,t1,y1)
                             => (ContainedAndDeleted,
                                 tree2(r,x,
                                       tree3(s1,x1,
                                             t1,y1,
                                             t',max(t')),max(t')))
                           | leaf(x1)
                             => (ContainedAndDeleted,
                                 tree2(r,x,s,y))
                           | empty
                             => raise deleteEmptyTreeError
                       else (stat, tree3(r,x,s,y,t',max(t')))
                     end
                  else (notContained, tree3(r,x,s,y,t,z));

    fun delete(a,t) = let val (_,t') = deleteA(a,t) in t' end;             delete

end (* TTTree *)                                                             end
```

```
structure IntegerItem =                                     IntegerItem
    struct
        type key = int;                                     key
        fun leq (i:key, j:key) =  i<=j;                     leq
        fun eq(i:key, j:key) =  i=j;                        eq
    end;


structure IntegerTTTree = TTTree(IntegerItem);              IntegerTTTree
```

Mit diesem Basisvorrat lassen sich unter Verwendung von speziellen kategoriellen Konstruktionen wie beispielsweise die schon angesprochenen Kommakategorie-Konstruktion oder die Funktorkategorie-Konstruktion viele neue und interessante Kategorien aufbauen. Zuvor geben wir jedoch eine Implementierung dieser beiden Konstruktionen inklusive eines Limitlifting an.

```
(********************************************************************
*********************************************************************
********
******** Special Categories, base definitions
******** Language SML/NY
******** Author M. Bertol (c)
********
******** Release: Tue Aug 13 11:35:40 MET DST 1996
********
*********************************************************************
*********************************************************************)


use"cat.sml";


(*** datatypes for comma categories ***)

datatype ('oA,'aA,'oC,'aC,'oB,'aB)Comma_Arrow =              datatype
                  comma_arrow of ('oA * 'aC * 'oB) *          comma_arrow
                                 ('aA * 'aB) *
                                 ('oA * 'aC * 'oB);

fun comma_cat(L,R) = let val A = domain(L);                  comma_cat
                         val B = domain(R)
                     in cat(fn comma_arrow(Y,_,_) => Y,
                            fn comma_arrow(_,_,Y) => Y,
                            fn (a,b,c) => comma_arrow((a,b,c),
                                              (id(A)(a),id(B)(c)),
                                              (a,b,c)),
```

```
                fn (comma_arrow(_,(f,g),Z),
                    comma_arrow(Y,(h,j),_)) =>
                        comma_arrow(Y,
                                (compose(A)(f,h),compose(B)(g,j)),
                                Z))
        end;

(*** retract operations for comma categories ***)

fun left_obj(l,_,_) = l;                                          left_obj

fun left_arr(comma_arrow((Aobj, Carr, Bobj),                      left_arr
                        (Aarr, Barr),
                        (Aobj', Carr', Bobj'))) = Aarr;
fun left_Diag(diagram(g,obj,arr)) = diagram(g,                    left˙Diag
                                    fn ver => left_obj(obj(ver)),
                                    fn edg => left_arr(arr(edg)) );
fun left_CoCone(cocone(c,d,marr)) = cocone(left_obj(c),           left˙CoCone
                                    left_Diag(d),
                                    fn ver => left_arr(marr(ver)));

fun right_obj(_,_,r) = r;                                         right˙obj
fun right_arr(comma_arrow((Aobj, Carr, Bobj),                     right˙arr
                        (Aarr, Barr),
                        (Aobj', Carr', Bobj'))) = Barr;
fun right_Diag(diagram(g,obj,arr)) = diagram(g,                   right_Diag
                                    fn ver => right_obj(obj(ver)),
                                    fn edg => right_arr(arr(edg)) );
fun right_CoCone(cocone(c,d,marr)) = cocone(right_obj(c),         right_CoCone
                                    right_Diag(d),                right_Diag
                                    fn ver => right_arr(marr(ver)));


(*** necessary extensions of the functor datatype ***)


fun apply_Fun_Diag(Fct,d) = let val diagram(g,obj,arr) = d;       apply_Fun_Diag
                                val obj' = fn ver => (apply_obj(Fct))(obj(ver));
                                val arr' = fn edg => (apply_arr(Fct))(arr(edg))
                            in diagram(g, obj',arr')
                            end;


fun apply_Fun_CoCone(Fct, cocone(c,d,marr)) = cocone(apply_obj(Fct)(c),    apply_Fun_CoCone
                                              apply_Fun_Diag(Fct,d),
                                              fn ver =>
                                                  apply_arr(Fct)(marr(ver)));

fun apply_Fun_CoConeArrow(Fct, cocone_arrow(Acone,arr,Bcone)) =   apply_Fun_CoConeArro
        cocone_arrow(apply_Fun_CoCone(Fct,Acone),
                     apply_arr(Fct)(arr),
                     apply_Fun_CoCone(Fct,Bcone));
```

```
(*** functor have to preserve colimits ***)

fun preserve(cocontinous_functor(Fct,translation)) = translation;          preserve
fun real(cocontinous_functor(Fct,translation)) = Fct;                      real


(*** lifting of the colimits into the comma category ***)


fun lift_colimit(colimA,contL,R,colimB) =                                  lift_colimit
  fn D => let val (c_A,univ_A) = colimA(left_Diag(D));
              val (c_B,univ_B) = colimB(right_Diag(D));
              val (Lc,Luniv) = preserve(contL)(c_A,univ_A);
              val Rc = apply_Fun_CoCone(R,c_B);
              val c1 = cocone(co_apex(Rc),
                              base(Lc),
                              fn n => let val (_,f_n,_) = obj_at_node(D)(n)
                                      in compose(range(real(contL)))
                                           (sides(Rc)(n),f_n)
                                      end);
              val f = co_apex_arrow(Luniv c1);
              val colim_obj = (co_apex(c_A),f,co_apex(c_B))
              val result_cocone = cocone(colim_obj,
                                         D,
                                         fn n =>
                                           comma_arrow(obj_at_node(D)(n),
                                                       (sides(c_A)(n),sides(c_B)(n)),
                                                       colim_obj));
              val universal = fn c =>
                                let val uA = co_apex_arrow(univ_A(left_CoCone(c)))
                                    val uB = co_apex_arrow(univ_B(right_CoCone(c)))
                                in cocone_arrow(result_cocone,
                                                comma_arrow(colim_obj,
                                                            (uA,uB),
                                                            co_apex(c)),
                                                c)
                                end
          in (result_cocone, universal)
          end


(*** cocomplete comma categories ***)

fun cocomplete_comma_cat(cA,cB) (contL,R) =                                cocomplete_comma_ca
          let val cocomplete_cat(_,colimA) = cA
              val cocomplete_cat(_,colimB) = cB
          in cocomplete_cat(comma_cat(real(contL),R),
                            lift_colimit(colimA,contL,R,colimB))
          end



(*** datatypes for functor categories ***)
```

```
datatype ('oA,'aA,'oB,'aB)Nat_transform = nat_transform of ('oA,'aA,'oB,'aB)Functor *
                                                       ('oA -> 'aB) *                     nat_transform
                                                       ('oA, 'aA, 'oB, 'aB)Functor

fun id(A,cat(_,_,i,_)) F = nat_transform(F, fn a => i(apply_obj(F)(a)),F);

fun dotcomp(A,cat(_,_,_,comp)) =                                                          dotcomp
          fn (nat_transform(_,beta,H),
              nat_transform(F,alpha,_)) =>
                    nat_transform(F,fn a => comp(beta(a),alpha(a)),H);

fun Fun_comp(cat_functor(_,Fobj,Farr,cB),cat_functor(cA,Gobj,Garr,_)) =                   Fun_comp
          cat_functor(cA,
                    fn obj => Fobj(Gobj(obj)),
                    fn arr => Farr(Garr(arr)),
                    cB);

fun ringcomp(cat(_,_,_,comp)) =                                                           ringcomp
                    fn (nat_transform(G,beta,G'),
                        nat_transform(F,alpha,F')) =>
                            nat_transform(Fun_comp(G,F),
                                      fn a => comp(beta(apply_obj(F')(a)),
                                                   apply_obj(G)(alpha(a))),
                                      Fun_comp(G',F'));


(*** definition of functor categories ***)

fun cat_of_functors(A,B) = cat(fn nat_transform(s,_,_) => s,                             cat_of_functors
                               fn nat_transform(_,_,t) => t,
                               id(A,B),
                               dotcomp(A,B));
```

Beispielsweise lassen sich diese Konstruktoren zur Definition von Relationen und Graphen verwenden. Man erhält gleichzeitig die Methoden für Joins, Schnitte, Vereinigungen, Subobjekte etc. via der entsprechenden Limite. So kann beispielsweise eine verteilte Parallele Datenbank unter Verwendung eines effizienten Basistyps für die Kategorie **Set** in ein paar Zeilen definiert werden. Die Korrektheit der Implementierung folgt unmittelbar aus den strukturellen Eigenschaften, die man bei der Definition verwendet hat. Der Typmechanismus von ML trägt dazu bei, das falsche Instanzen fast nicht definiert werden können. Die Paralellisierung kann einem verteilten ML-System wie CML überlassen werden.

```
(*********************************************************************
**********************************************************************
********
******** Concrete Problem Solutions
******** Language SML/NY
******** Author M. Bertol (c)
```

```
********
******** Release: Tue Aug 13 11:35:40 MET DST 1996
********
************************************************************************
************************************************************************)



(*** computing a push out diagram in set ***)

datatype n˙labels = x˙a | x˙b | x˙c;
datatype e˙labels = x˙f | x˙g;


val G = let val N = [x˙a,x˙b,x˙c]
            val E = [x˙f,x˙g]
            fun src x˙f = x˙a
              | src x˙g = x˙a
            fun tgt x˙f = x˙b
              | tgt x˙g = x˙c
        in graph(N,E,src,tgt)
        end;

val a˙set = union(singleton(just "A"),
                  union(singleton(just "B"),
                        union(singleton(just "C"),
                              union(singleton(just "D"),emptyset))))

val b_set = union(singleton(just "B"),
                  union(singleton(just "E"),
                        union(singleton(just "F"),
                              union(singleton(just "G"),emptyset))))

val c_set = union(singleton(just "C"),
                  union(singleton(just "D"),
                        union(singleton(just "E"),
                              union(singleton(just "F"),
                                    union(singleton(just "G"),emptyset)))))


fun f_fn(just"A") = just "C"
  | f_fn(just"B") = just "C"
  | f_fn(just"C") = just "D"
  | f_fn(just"D") = just "E"

fun g_fn(just"A") = just "B"
  | g_fn(just"B") = just "E"
  | g_fn(just"C") = just "B"
  | g_fn(just"D") = just "F"

fun nodes_to_sets (x_a) = a_set
  | nodes_to_sets (x_b) = b_set
  | nodes_to_sets (x_c) = c_set
```

```
fun edges_to_arrows(x_f) = set_arrow(a_set,f_fn,b_set)
  | edges_to_arrows(x_g) = set_arrow(a_set,g_fn,c_set)

val po_diagram = diagram(G,nodes_to_sets,edges_to_arrows)

let val cocomplete_cat(_,set_colimit) = cocomplete_FinSet in set_colimit(po_diagram) end


(*** definiing the category of (not only set) relations ***)

(*** only composition should be redefined ---- rest is inherited ***)
```

relation_compose

```
fun relation_compose(cC)((f',g'),(f,g)) = let val complete_cat(C,_) =cC
                                              val (p,p'),_ = pullback(cC)(g,f')
                                          in (compose(C)(f,p),compose(C)(g',p'))
                                          end;


(*** definiing the category of finite graphs ***)

val cross_product =
        cat_functor(FinSet,
                fn a => set_times(a,a),
                fn set_arrow(a,f,b) => set_arrow(set_times(a,a),
                                        fn pair(x,y) =>
                                              pair(f(x),f(y)),
                                        set_times(b,b)),
                FinSet)


fun I(A) = cat_functor(A, fn x=> x, fn x => x, A);
val graphCat = comma_cat(I(FinSet),cross_product);
```

cocomplete_FinGraph

```
val cocomplete_FinGraph = let val C = cocomplete_FinSet
                          in  cocomplete_comma_cat(C,C)(cocontinous_I(FinSet),
                                                        cross_product)
                          end;
```

5 Die Geometrie des Erkennens

Wir formulieren die Geometrie des Problems mit einer Grothendiek Topologie. Eine gewöhnliche Topologie ist im mathematischen Sinne ein Mengensystem über einer Trägermenge, das unter endlicher Schnittbildung und beliebiger Vereinigung abgeschlossen ist. Grothendiek Topologien sind kategorielle Verallgemeinerungen dieses Begriffs. Das durch die Inklusion partiell geordnete Mengensystem wird durch eine Kategorie ersetzt.

5.1 Garben

In der Theorie der Bündel oder Garben werden globale Konsequenzen, die sich aus lokalen Eigenschaften ableiten, untersucht. Der Begriff lokal wird durch eine Topologie beschrieben. Eine Abbildung nennen wir eine Garbe, wenn verträgliche Abbildungen, die lokal definiert sind, sich eindeutig fortsetzen lassen. Das bedeutet, daß das Bild eines Objektes aus den Bildern einer durch die Topologie erzeugten Zerlegung (Überdeckung) rekonstruiert werden kann. Die Vorstellung dahinter sind einzelne Funktionspfeile einer Abbildung von einer Zerlegung die wie Halme in einer Garbe zu einer (globalen) Abbildung zusammengebündelt werden.

Neben der Abbildung von Objekten in Mengen ordnet eine Garbe auch noch jedem Pfeil der Kategorie eine Injektion (in die andere Richtung) zu. Wir interpretieren die Objekte als Zusicherungen und ihre Bilder als Modelle, d.h. erfüllende Instanziierungen. So erklärt sich auch die Kontravarianz der Garben. Modelle, die einer stärkeren Eigenschaft genügen, erfüllen auch eine schwächere. Das induziert eine Inklusion in die andere Richtung.

$$\begin{tikzcd} a \arrow[r, "\mathsf{sh}"] \arrow[dd] & \mathsf{sh}(a) \\ {} \arrow[r, dotted, "\mathsf{sh}"] & {} \\ b \arrow[r, "\mathsf{sh}"] & \mathsf{sh}(b) \arrow[uu] \end{tikzcd}$$

Der Übergang von lokalen Eigenschaften zu globalen Konsequenzen geschieht via bündeln der Funktor-Pfeile einer kompatiblen (d.h. lokal zusammenpassenden) Familie von Elementen einer Überdeckung. Eine Überdeckung eines Objektes kann als eine Dekomposition in einfachere Objekte angesehen werden. Eine Garbe ordnet jedem Element dieser Überdeckung eine Menge zu. Eine Auswahl von Elementen dieser Menge bildet eine kompatible Familie, wenn die Wahl mit der Restriktion im Falle einer Überlappung verträglich ist. Induziert eine solche (lokale) Wahl eindeutig eine Auswahl auf dem Mengen-Bild des überdeckten Objekt (d.h. die einzelnen Abbildungspfeile lassen sich so bündeln, daß das überdeckte Objekt als Domäne auftritt), so sprechen wir von einer Garbe (Garben-Eigenschaft).

6 Grothendiek Topologien

Motiviert durch die geometrische Sichtweise des Mustererkennungsproblems, werden wir einen kategoriellen Begriff der verallgemeinerten Nachbarschaft betrachten. Sei $\mathcal{C}$ eine kleine Kategorie und $\mathbf{Set}^{\mathcal{C}^{\mathsf{op}}}$ die entsprechende Funktorkategorie. Wir notieren die Yoneda Einbettung $\mathsf{y}(C) = \mathsf{Hom}_{\mathcal{C}}(-, C)$ mit

$$\mathsf{y} : \mathcal{C} \longrightarrow \mathbf{Set}^{\mathcal{C}^{\mathsf{op}}}.$$

Ein Sieb S (über dem Objekt C) ist ein Subobjekt $S \subseteq \mathsf{y}(C)$ in $\mathbf{Set}^{\mathcal{C}^{\mathsf{op}}}$. Alternativ kann S als eine Familie von Morphismen $(f_i)_{i \in I}$ in $\mathcal{C}$ mit co-Domain $\mathsf{cod}(f_i) = C$ für alle i in I beschrieben werden, daß

$$f \text{ in } S \quad \Longrightarrow \quad f \circ g \text{ in } S$$

erfüllt ist, wenn immer die Komposition definiert ist. Mit anderen Worten ist S ein Ideal. Wenn S ein Sieb (über C) ist und $h : D \longrightarrow C$ ein beliebiger Morphismus der Kategorie $\mathcal{C}$ mit co-Domain $\mathsf{cod}(h) = C$ ist, dann ist

$$h^{\leftarrow}(S) \quad = \quad \{g \mid \mathsf{cod}(g) = D, h \circ g \in S\} \tag{2}$$

auch ein Sieb (über D).

Definition 1 (Grothendiek Topologie) *Eine Grothendiek Topologie auf einer Kategorie $\mathcal{C}$ ist eine Funktion $\mathcal{J}$ die jedem Objekt C der Kategorie $\mathcal{C}$ eine Menge von Sieben so zuordnet, daß*

(i) der maximale Sieb $t_C = \{f \mid \mathsf{cod}(f) = C\}$ in $\mathcal{J}(C)$ liegt,

(ii) wenn S in $\mathcal{J}(C)$ liegt auch $h^{\leftarrow}(S)$ in $\mathcal{J}(D)$ für alle Morphismen $h : D \longrightarrow C$ (Stabilität unter Basiswechseln) und

(iii) wenn S in $\mathcal{J}(C)$ liegt und R ein Sieb über C ist mit $h^{\leftarrow}(R)$ in $\mathcal{J}(D)$ für alle $h : D \longrightarrow C$ in S, dann ist auch R in $\mathcal{J}(C)$ (Transitivitätsaxiom).

Daraus folgt unmittelbar aus $S \in \mathcal{J}(C)$, daß auch jeder größere Sieb $R \supseteq S$ (über C) ein Element von $\mathcal{J}(C)$ ist. Betrachte $h : D \longrightarrow C$ mit $h \in S$. Dann ist $\mathsf{id}_D \in h^{\leftarrow}(S)$, und somit $h^{\leftarrow}(S)$ der maximale Sieb über D mit $h \in S$. Da $h^{\leftarrow}(S) \subseteq h^{\leftarrow}(R)$ muß $h^{\leftarrow}(R)$ auch maximal (über D) sein und somit $h^{\leftarrow}(R) \in \mathcal{J}(D)$. Das gilt für alle $h \in S$ und wegen des Transitivitätsaxioms liegt R in $\mathcal{J}(C)$.

Mit dieser Beobachtung folgt aus dem Transitivitätsaxiom

(*iii'*) Ist S in $\mathcal{J}(C)$ und existiert für jedes $f : D_f \longrightarrow C$ in S ein Sieb $R_f \in \mathcal{J}(D_f)$, dann ist die Menge aller (zusammengesetzten) Morphismen $f \circ g$ mit $f \in S$ und $g \in R_f$ ein Sieb in $\mathcal{J}(C)$.

Ein Gebiet ist ein Paar $\langle \mathcal{C}, \mathcal{J} \rangle$ das aus einer kleinen Kategorie $\mathcal{C}$ und einer Grothendiek Topologie $\mathcal{J}$ (auf $\mathcal{C}$) besteht. Wir nennen $S \in \mathcal{J}(C)$ einen überdeckenden Sieb oder S ist eine ($\mathcal{J}$-)Überdeckung von C oder einfach S überdeckt C. Wir benützen den gleichen Begriff auch für Morphismen. Ein Sieb S (über C) überdeckt einen Morphismus $f : D \longrightarrow C$ falls $f^{\leftarrow}(S)$ das Objekt D überdeckt. In dieser Terminologie formulieren sich die Grothendiek Axiome wie folgt:

(ia) Wenn S ein Sieb über C ist mit $f \in S$, dann überdeckt S auch f.

(iia) Wenn S einen einen Morphismus $f : D \longrightarrow C$ überdeckt, dann überdeckt S auch den Morphismus $f \circ g$, für jeden Morphismus $g : E \longrightarrow D$ (Stabilität).

($iiia$) Wenn S einen einen Morphismus $f : D \longrightarrow C$ überdeckt und R ein Sieb über C ist, der alle Morphismen aus S überdeckt, dann überdeckt R auch f.

Die ursprünglichen Bedingungen (i)–(iii) aus der Definition folgen unmittelbar aus den Morphismenbedingungen (ia)–($iiia$) durch Anwendung auf die Identitäten id_C. Umgekehrt leiten sich auch die Bedingungen (ia)–($iiia$) von den ursprünglichen ab. Beispielsweise folgt die dritte Bedingung. Angenommen S überdeckt einen Morphismus $f : D \longrightarrow C$ und R überdeckt alle Morphismen aus S. D.h. $f^{\leftarrow}(S) \in \mathcal{J}(D)$ und $h^{\leftarrow}(R) \in \mathcal{J}(\mathsf{dom}(h))$ für alle $h \in \mathcal{J}(S)$. Das bedeutet für jeden Morphismus $g : E \longrightarrow D$ in $f^{\leftarrow}(S)$ existiert ein Morphismus $fg \in S$. Deshalb gilt $g^{\leftarrow}(R) = (fg)^{\leftarrow}(S) \in \mathcal{J}(E)$. Da das für alle Morphismen g gilt impliziert das Axiom (iii) $f^{\leftarrow}(R) \in \mathcal{J}(D)$, mit anderen Worten R überdeckt f.

Aus den Axiomen folgt, daß zwei beliebige Überdeckungen eine gemeinsame Verfeinerung besitzen.

(iv) Wenn $R, S \in \mathcal{J}(C)$, dann ist auch $R \cap S \in \mathcal{J}(C)$,

oder in Morphismenform

(iva) wenn R und S beide den Morphismus $g : D \longrightarrow C$ überdecken, dann wird er auch von $R \cap S$ überdeckt.

Wenn $f : D \longrightarrow C$ ein Element von R ist, dann gilt wegen (ii) $f^{\leftarrow}(S \cap R) = f^{\leftarrow}(S) \in \mathcal{J}(D)$ und deshalb wegen (iii) $R \cap S \in \mathcal{J}(C)$.

Ein topologischer Raum mit der üblichen Definition von Überdeckung ist ein Beispiel für ein Gebiet. Die partiell geordnete Menge $\mathcal{O}(\mathcal{X})$ von offenen Teilmengen von X kann als Kategorie betrachtet werden, wobei zwischen zwei Objekten ein Morphismus $U \longrightarrow V$ genau dann existiert, falls $U \subseteq V$. Damit ist ein Sieb (über U) eine Familie S von offenen Teilmengen von U mit $V' \subseteq V \in S$ impliziert $V' \in S$. Dann überdeckt der Sieb S den Morphismus $W \subseteq U$ dann und nur dann, wenn W in der Vereinigung der offenen Mengen in S enthalten ist.

Im Falle eines herkömmlichen topologischen Raumes beschreibt man gewöhnlicher Weise eine offene Überdeckung von U durch eine Familie $\{U_i \mid i \in I\}$ von offenen Teilmengen von U mit Vereinigung $\bigcup_{i \in I} U_i = U$. Eine Familie ist nicht notwendiger Weise ein Sieb, aber sie erzeugt einen, nämlich alle Teilmengen $V \subseteq U_i$ für ein U_i aus $\{U_i \mid i \in I\}$.

In dieser Art erzeugte überdeckende Siebe können im allgemeineren Kontext von Kategorien mit gefaserten Produkten (Pullbacks) definiert werden. Dazu führen wir den Begriff Basis für Grothendiek Topologien ein.

Definition 2 *Eine Basis für eine Grothendiektopologie über einer Kategorie $\mathcal{C}$ mit gefaserten Produkten ist eine Funktion $\mathcal{K}$ die jedem Objekt C eine Menge $\mathcal{K}(C)$ von Familien von Morphismen mit co-Domain C so zuordnet, daß*

(i′) *wenn $f : C' \longrightarrow C$ ein Isomorphismus ist, liegt auch $(f : C' \longrightarrow C)$ in $\mathcal{K}(C)$,*

(ii′) *wenn $(f_i : C_i \longrightarrow C)_{i \in I} \in \mathcal{K}(C)$, dann liegt für alle Morphismen $g : D \longrightarrow C$ die Familie der gefaserten Produkte $(\pi_2 : C_i \times_C D \longrightarrow D)_{i \in I}$ in $\mathcal{K}(D)$,*

(iii′) *wenn $(f_i : C_i \longrightarrow C)_{i \in I} \in \mathcal{K}(C)$ und für alle $i \in I$ eine Familie $(g_{ij} : D_{ij} \longrightarrow C_i)_{i \in I, j \in I_i}$ in $\mathcal{K}(C)$ existiert, dann liegt die Familie der Kompositionen $(f_i \circ g_{ij} : D_{ij} \longrightarrow C)_{i \in I, j \in I_i}$ auch in $\mathcal{K}(C)$.*

Bedingung (ii') nennen wir wieder Stabilitätsaxiom und (iii') Transitivitätsaxiom. Das Paar $\langle \mathcal{C}, \mathcal{K} \rangle$ wird auch Gebiet genannt und die Elemente aus $\mathcal{K}(C)$ bezeichnen wir als (elementare) Überdeckungen oder überdeckende Familien (des Gebiets).

Man beachte, wegen (i') muß eine Topologie $\mathcal{J}$ nicht notwendiger Weise eine Basis sein. Aber wenn $\mathcal{K}$ eine Basis (über $\mathcal{C}$) ist, dann generiert $\mathcal{K}$ eine Topologie $\mathcal{J}$ via

$$S \in \mathcal{J}(C) \iff \exists R \in \mathcal{K}(C) : R \subseteq S. \tag{3}$$

D.h. ein Sieb ist eine $\mathcal{J}$-Überdeckung, falls er eine $\mathcal{K}$-Überdeckung enthält. Es ist einfach zu überprüfen, daß die Grothendiek Axiome erfüllt sind; beispielsweise gilt das Stabilitätsaxiom. Sei $S \in \mathcal{J}(C)$ und $g : D \longrightarrow C$ ein beliebiger Morphismus. Wähle $R \subseteq S$ mit $R \in \mathcal{K}(C)$ wie in (3) und sei $T \in \mathcal{K}(D)$ die $\mathcal{K}$-Überdeckung von D in (ii'), die man über die gefaserten Produkte erhält. Genauer besteht T aus allen Morphismen h, die das Diagramm

$$\begin{tikzcd} D \times_C C' \arrow[r] \arrow[d, "h"'] & C' \arrow[d, "f"] \\ D \arrow[r, "g"] & C \end{tikzcd}$$

für ein f in R kommutieren lassen. Dann gilt $T \subseteq g^{\leftarrow}(S)$ und somit $g^{\leftarrow}(S) \in \mathcal{J}(S)$.

Es existiert eine maximale Basis $\mathcal{K}$, die eine Topologie $\mathcal{J}$ auf $\mathcal{C}$ erzeugt. Diese Basis wird durch

$$R \in \mathcal{K}(C) \iff (R) \in \mathcal{J}(C) \tag{4}$$

beschrieben, wobei

$$(R) \quad = \quad \{f \circ g \mid f \in R, \mathsf{dom}(f) = \mathsf{cod}(g)\} \tag{5}$$

der Sieb ist, der durch die Familie R erzeugt wird. Aussage (iv) über Schnitte von Überdeckungen ist gilt nicht in dieser Form für beliebige Basen. Angenommen die Familie $(f_i : C_i \longrightarrow C)_{i \in I}$ verfeinere die Familie $(g_i : C_i \longrightarrow C)_{i \in I'}$, d.h. jeder Morphismus f_i wird von einem Morphismus g_j faktorisiert.

(iv') Für zwei Überdeckungen $R, P \in \mathcal{K}(C)$, existiert eine gemeinsame Verfeinerung in $\mathcal{K}(C)$.

Die Bedingung (iv') folgt aus (iv). Wenn $\mathcal{J}$ eine Grothendiektopologie auf $\mathcal{C}$ ist, die durch die Basis $\mathcal{K}$ erzeugt wird, dann ist $(R), (P) \in \mathcal{J}(C)$ und nach (iv) auch $(R) \cap (P) \in \mathcal{J}(C)$. Nach (3) existiert damit ein $T \in \mathcal{K}(C)$ mit $T \subseteq (R) \cap (P)$. Aber $T \subseteq (R)$ bedeutet das T den Sieb R verfeinert, und ebenso verfeinert T auch P, womit (iv') bewiesen ist.

6.1 Mathematische Beispiele

Um mit den eingeführten Griffe vertraut zu werden führen wir einige (leider noch immer sehr abstrakte) Beispiele an.

6.1.1 Triviale Topologie

Wenn $\mathcal{C}$ eine beliebige Kategorie ist, dann nennen wir die triviale Topologie auf $\mathcal{C}$ die, die als einzige Siebe (über C) die maximalen Siebe t_C enthält. Diese Topologie ist die kleinste (gröbste) unter den Topologien auf $\mathcal{C}$.

6.1.2 Topologie der offenen Überdeckungen

Sei $\mathcal{T}$ eine kleine Kategorie topologischer Räume, die unter endlichen Limite und unter offenen Subräumen abgeschlossen ist. Beispielsweise wenn $\mathcal{T}$ die Kategorie aller separablen Räume ist. Wir definieren eine Basis $\mathcal{K}$ auf $\mathcal{T}$ durch $(f_i : Y_i \longrightarrow X)_{i \in I} \in \mathcal{K}(X)$ wenn für alle i der Raum Y_i ein offener Subraum ist mit der entsprechenden Einbettung f_i und $\bigcup_{i \in I} Y_i = X$ gibt. Die Intention dieser Konstruktion ist es alle offenen Gebiete $\mathcal{O}(\mathcal{X})$ von den topologischen Räumen X zu einem großen Raum zusammenzufassen.

6.1.3 Die Coprodukttopologie

Eine größere und damit feinere Topologie über einer Kategorie $\mathcal{T}$ wird durch die Basis $\mathcal{K}'$ erzeugt, die durch $(f_i : Y_i \longrightarrow X)_{i \in I} \in \mathcal{K}'(X)$, falls $f : \coprod Y_i \longrightarrow X$ eine offene Surjektion ist, definiert ist. Dabei ist $\coprod Y_i$ die disjunkte Summe (Coprodukt) der Räume Y_i und f die Abbildung, die durch die einzelnen f_i induziert ist. Zum Beweis, daß $\mathcal{K}'$ eine Basis ist beachte man nur, daß im gefaserten Produkt von topologischen Räumen

$$\begin{tikzcd}
Z \times_X Y \arrow[r] \arrow[d, "q"'] & C' \arrow[d, "p"] \\
D \arrow[r, "f"] & X
\end{tikzcd}$$

q eine offene Surjektion ist wenn p eine ist.

6.1.4 Vollständige Heyting Algebren

Ein interessantes Beispiel sind die vollständigen Heyting Algebren (**cHa**). Mit **cHa**s lassen sich beispielsweise viele Phänomene, die bei nebenläufigen Vorgängen auftreten einfach und prägnant modellieren, was zu einer logikbasierten Semantik führt.

Eine Heyting Algebra ist ein distributiver Verband A mit einem Implikationsoperator $\Rightarrow$, der dem Gesetz

$$a \leq (b \Rightarrow c) \quad \Longleftrightarrow \quad a \wedge b \leq c \tag{6}$$

für alle $a, b, c \in A$ genügt. Eine Heyting Algebra ist vollständig wenn der Verband $(A, \leq)$ vollständig ist, d.h. alle Suprema und Infima existieren. Wenn A eine **cHa** ist und $\{a_i \mid i \in I\}$ und b Elemente aus A sind, dann gilt das allgemeine (unendliche) Distributivgesetz

$$\bigvee_{i \in I} (b \wedge a_i) \quad = \quad b \wedge \bigvee_{i \in I} a_i. \tag{7}$$

Das folgt aus (6). Für jedes $c \in A$ gilt

$$\begin{aligned}
\bigvee_{i \in I} (b \wedge a_i) \leq c \quad &\Longleftrightarrow \quad \forall i \in I : \; b \wedge a_i \leq c \\
&\Longleftrightarrow \quad \forall i \in I : \; a_i \leq (b \Rightarrow c) \\
&\Longleftrightarrow \quad b \wedge \bigvee_{i \in I} a_i \leq c.
\end{aligned}$$

Umgekehrt kann man zeigen, daß ein Verband A mit beliebigen (allen) Suprema, der Gleichung (7) genügt, bereits die Struktur einer Heyting Algebra besitzt, wobei die Implikation $\Rightarrow$ durch

$$b \Rightarrow c \quad = \quad \bigvee \{a \mid a \wedge b \leq c\} \tag{8}$$

definiert ist. Sei A eine **cHa**. Wir können A als eine Kategorie auffassen. (Der Morphismus $a \longrightarrow b$ mit Domain a und co-Domain b existiert genau denn, wenn $a \leq b$.) Diese Kategorie kann mit einer Basis $\mathcal{K}$ einer Grothendiektopologie ausgestattet werden, die durch

$$(a_i)_{i \in I} \in \mathcal{K}(c) \quad \Longleftrightarrow \quad \bigvee_{i \in I} a_i = c \tag{9}$$

definiert wird. (Auf der linken Seite identifizieren wir ein Element $a_i \leq c$ mit dem Morphismus $a_i \longrightarrow c$.) Die Gleichung (7) ist dann das Analogon zum Stabilitätsaxiom (ii').

Ein Sieb S über einem Element c aus A ist eine Teilmenge S von Elementen $b \leq c$ so, daß $a \leq b \in S$ auch $a \in S$ impliziert. In der Topologie $\mathcal{J}$ mit Basis $\mathcal{K}$ wie in (9) definiert überdeckt ein Sieb über c das Element c, falls $c = \bigvee S$. Deswegen wird diese Topologie häufig auch Supenumtolpologie genannt. Ist beispielsweise A die Algebra aller offenen Teilmengen eines topologischen Raumes (im gewöhnlichen Sinne), so ist diese Topologie die oben eingeführte Topologie der offenen Teilmengen. In diesem Sinne verallgemeinern die Betrachtungen von vollständigen Heyting Algebren die Betrachtungen gewöhnlicher Topologien.

6.1.5 Die dichte Topologie

Ein weiteres Beispiel ist die dichte Topologie. Wenn P eine partiell geordnete Menge $(P, \leq)$ mit $p \in P$ ist, dann sagt man von einer Teilmenge D eines Primideals $\{q \in P \mid q \leq p\}$ das sie dicht ist, falls für alle $r \leq p$ ein $q \leq r$ existiert mit $q \in D$. Wie oben kann P als eine Kategorie interpretiert werden und die dichten Teilmengen als Siebe bilden eine Topologie auf P mit

$$\mathcal{J}(p) \quad = \quad \{D \mid \forall q \in D : \ q \leq p,\, D \text{ ist ein dichter Sieb unter } p\}.$$

Eigentlich sollten die Überdeckungen $D \in \mathcal{J}(p)$ aus Morphismen $q \longrightarrow p$ bestehen. Wir identifizieren auch hier die Morphismen $q \longrightarrow p$ mit den Elementen q, $q \leq p$.

Tatsächlich kann die dichte Topologie $\mathcal{J}$ auf einer beliebigen (kleinen) Kategorie $\mathcal{C}$ definiert werden. Für ein Sieb S gilt dann

$$S \in \mathcal{J}(C) \iff \forall f : D \longrightarrow C\ \exists g : E \longrightarrow D : fg \in S. \tag{10}$$

6.1.6 Die atomare Topologie

Das letzte Beispiel ist die atomare Topologie. Sei $\mathcal{C}$ eine Kategorie und $\mathcal{J}$ werde durch

$$S \in \mathcal{J}(C) \iff S \neq \emptyset \tag{11}$$

definiert. Damit $\mathcal{J}$ das Stabilitätsaxiom erfüllt verlangen wir weiter, daß für zwei Morphismen $f : D \longrightarrow C$ und $g : E \longrightarrow C$ mit gemeinsamem co-Domain C das kommutierende Diagramm

$$\begin{tikzcd} \bullet \arrow[r, dotted] \arrow[d, dotted] & D \arrow[d, "f"] \\ E \arrow[r, "g"] & C \end{tikzcd}$$

eindeutig vervollständigt werden kann.

7 Garben

Sei $\mathcal{C}$ eine kleine Kategorie und $\mathcal{J}$ eine Grothendiek Topologie auf $\mathcal{C}$. Eine Prä-Garbe ist ein Funktor

$$P : \mathcal{C}^{\mathsf{op}} \longrightarrow \mathbf{Set}\,.$$

Ist P eine Prä-Garbe und S ein ein Objekt C aus $\mathcal{C}$ überdeckender Sieb. Eine Matching-Familie für S von Elementen aus P ist eine Funktion, die jedem Morphismus $f : D \longrightarrow C$ aus S ein Element $x_f \in P(D)$ zuordnet, so daß

$$x_f \cdot g \quad = \quad x_{fg} \quad \forall\, g : E \longrightarrow D \text{ in } \mathcal{C}. \tag{12}$$

Dabei ist fg wieder ein Morphismus aus S, da S ein Sieb ist. $x_f \cdot g$ steht hier abkürzend für $P(g)(x_f)$. Ein Amalgam (Vermengung) einer Matching-Familie ist ein Element $x \in P(C)$ mit

$$x \cdot f \quad = \quad x_f \quad \forall\, f \in S \tag{13}$$

Wir nennen P eine Garbe (für $\mathcal{J}$) wenn jede Matching-Familie einer Überdeckung eines beliebigen Objektes aus $\mathcal{C}$ ein eineindeutiges Amalgam (Amalgamation) besitzt.

Ein Sieb S über C ist ein Sub-Funktor der Yoneda-Einbettung

$$\mathsf{y}C = \mathsf{Hom}(-, C),$$

eine Matching-Familie

$$f \longmapsto x_f, f \in D$$

ist nichts anderes als eine natürliche Transformation $S \longrightarrow P$. Eine Prä-Garbe ist eine Garbe, wenn sich jede natürliche Transformation $\tau : S \longrightarrow P$ von einem Sieb S, der C überdeckt, eineindeutig auf $\mathsf{y}C$ wie in folgendem Diagramm

$$\begin{tikzcd}
S \arrow[r, "\tau"] \arrow[d] & P \\
\mathsf{y}C \arrow[ur, dotted] &
\end{tikzcd}$$

fortsetzten läßt. Mit anderen Worten ist P eine Garbe, wenn für jeden C überdeckenden Sieb S die Inklusion $S \longrightarrow \mathsf{y}C$ einen Isomorphismus

$$\mathsf{Hom}(S, P) \quad \cong \quad \mathsf{Hom}(\mathsf{y}C, P)$$

induziert.

Das bedeutet für jedes Objekt C in $\mathcal{C}$ und jeden überdeckenden Sieb $S \in \mathcal{J}(C)$ kommutiert das Diagramm

$$P(C) \xleftarrow{e} \prod_{f \in S} P(\mathsf{dom}\, f) \underset{p}{\overset{a}{\rightrightarrows}} \prod_{\substack{f \in S \\ \mathsf{dom}(f) = \mathsf{cod}(g)}} P(\mathsf{dom}(g)) \tag{14}$$

so, daß e ein Equalizer auf **Set** ist. Dabei ist e die Abbildung

$$e : x \quad \longmapsto \quad (x \cdot f)_{f \in S} \;=\; (P(f)(x))_{f \in S}.$$

Das zweite Produkt wird über alle verknüpfbaren Morphismenpaare f, g mit $f \in S$ (also auch $fg \in S$, die Abbildung p rührt von der Komposition in $\mathcal{C}$ her und a ist von der Aktion von P auf $\mathcal{C}$ induziert). D.h. wenn $(x_f)_{f \in S}$ aus $\prod_{f \in S} P(\mathsf{dom}\, f)$ stammt, dann ist

$$p(x)_{f,g} \;=\; x_{fg}, \qquad \text{und} \qquad a(x)_{f,g} \;=\; x_f \cdot g. \tag{15}$$

Wenn (14) ein Equalizer für eine Überdeckung S ist, also jede Familie in der Überdeckung eine (eineindeutige) Amalgamation besitzt, sagen wir P erfüllt die Garben-Bedingung bzgl. S.

Sei $\mathcal{K}$ eine Basis für eine Topologie $\mathcal{J}$ auf einer Kategorie $\mathcal{C}$ mit gefaserten Produkten. Garben in $\mathcal{J}$ können nun mit dieser Basis wie folgt beschrieben werden. Wenn

$$R \quad = \quad (f_i : C_i \longrightarrow C)_{i \in I} \quad \in \quad \mathcal{K}(C)$$

eine $\mathcal{K}$-Überdeckung von einem Objekt C in $\mathcal{C}$ ist, dann ist eine Familie mit den Elementen $x_i \in P(C_i)$, $(i \in I)$, ein Matching für R, wenn

$$\forall i, j \in I \quad x_i \cdot \pi^1_{ij} \;=\; x_j \cdot \pi^2_{ij} \tag{16}$$

wobei π^1 und π^2 die Projektionen aus dem gefaserten Produkt

$$\begin{tikzcd} C_i \times_C C_j \arrow[r, "\pi^2_{ij}"] \arrow[d, "\pi^1_{ij}"'] & C_j \arrow[d, "f_j"] \\ C_i \arrow[r, "f_i"] & C \end{tikzcd}$$

sind. Ein Amalgam für $\{x_i \mid i \in I\}$ ist ein $x \in P(C)$ mit der Eigenschaft $x \cdot f_i = x_i$ für alle $i \in I$.

Proposition 1 *Sei P eine Prä-Garbe auf $\mathcal{C}$. P ist dann eine Garbe und nur dann, wenn für alle Basisüberdeckungen $(f_i : C_i \longrightarrow C)_{i \in I} \in \mathcal{K}(C)$ jede Matching-Familie $\{x_i \mid i \in I\}$ eine eineindeutige Amalgamation hat.*

Dieser Satz kann auch durch ein entsprechendes Equalizer-Diagramm ausgedrückt werden. Eine Prä-Garbe auf $\mathcal{C}$ ist eine Garbe auf der Topologie $\mathcal{J}$ mit Basis $\mathcal{K}$, falls für alle Überdeckungen $(f_i : C_i \longrightarrow C)_{i\in I} \in \mathcal{K}(C)$ das Diagramm

$$P(C) \xleftarrow{e} \prod_{i\in I} P(C_i) \overset{p_1}{\underset{p_2}{\rightrightarrows}} P(C_i \times_C C_j) \tag{17}$$

ein Equalizer ist. Hierbei gilt $e(x) = \{x \cdot f_i \mid i \in I\}$, sowie $p_1(\{x_i\})_{i,j} = x_i \cdot \pi^1_{ij}$ und $p_2(\{x_i\})_{i,j} = x_j \cdot \pi^2_{ij}$ (π^1_{ij}, π^2_{ij} wie oben).
Beweis: Sei P eine Garbe auf $\mathcal{J}$ und sei $R = (f_i : C_i \longrightarrow C)_{i\in I} \in \mathcal{K}(C)$ eine Überdeckung aus der Basis $\mathcal{K}$. Sei $\{x_i \mid i \in I\}$ eine Matching-Familie für diese Überdeckung. Betrachten wir den von R erzeugten Sieb

$$S \;=\; (R) \;=\; \{g : D \longrightarrow C \mid g = D \xrightarrow{k} C_i \xrightarrow{f_i} C, i \in I\}.$$

Wir definieren durch $\{y_g \mid g \in S\}$ eine Matching-Familie mit

$$y_g \;=\; x_i \cdot h,$$

$g = f_i \circ h$. Die Wahl ist unabhängig von i und h, denn wenn $f_i \circ h = g = f_j \circ k$, dann existiert eine Abbildung ℓ (universelle Eigenschaft des gefaserten Produkts) mit $\pi^1_{ij} \circ \ell = h$ und $\pi^2_{ij} \circ l = k$,

$$x_i \cdot h \;=\; (x_i \cdot \pi^1_{ij}) \cdot \ell \;=\; (x_j \cdot \pi^2_{ij}) \cdot \ell \;=\; x_j \cdot k.$$

(Die mittlere Identität gilt, da $\{x_i \mid i \in I\}$ ein Matching für R ist.) Damit existiert (genau) ein Element $y \in P(C)$ mit $y \cdot g = y_g$ (für alle $g \in (R)$), denn P ist eine Garbe (in $\mathcal{J}$). Insbesondere gilt $y \cdot f_i = y_{f_i} = x_i$, womit y ein Amalgam für $\{x_i \mid i \in I\}$ ist. Das Amalgam von $\{x_i \mid i \in I\}$ ist eineindeutig, denn wenn ein zweites y' mit $x_i = y' \cdot f_i$ existiert, dann gilt für jedes $g \in S = (R)$, $g = f_i \circ h$, $y' \cdot g = (y' \cdot f_i) \cdot h = y_g$, womit y' auch ein Amalgam für die Überdeckung (R) ist, also $y = y'$.

Um die andere Implikation zu zeigen nehmen wir an, daß $S \in \mathcal{J}(C)$ eine Überdeckung von C mit einer Basisüberdeckung $R \subseteq S$ in $\mathcal{K}(C)$ ist. Weiter sei $\{y_g \mid g \in S\}$ eine Matching-Familie für S. Dann ist die Sub-Familie $\{y_f \mid f \in R\}$ erst recht ein Matching, womit ein eineindeutiges Amalgam $y \in P(C)$ mit $y \cdot f = y_f$ existiert. Um zu zeigen, daß $y \cdot g = y_g$ für alle $g \in S$ gilt betrachtet man die gefaserten Produkte

$$\begin{tikzcd}
C' \times_C D \arrow[r, "\rho_{f,g}"] \arrow[d, "\pi_{f,g}"'] & C' \arrow[d, "f"] \\
D \arrow[r, "g"] & C.
\end{tikzcd}$$

Es gilt $(\pi_{f,g})_{f\in R} \in \mathcal{K}(D)$ wegen des Stabilitätsaxioms und weiter gilt für jedes $f \in S$

$$\begin{array}{rcll}
(y \cdot g) \cdot \pi_{f,g} & = & (y \cdot f) \cdot \rho_{f,g} & \text{(Diagramm, Kontravarianz)} \\
& = & y_f \cdot \rho_{f,g} & (f \in R) \\
& = & y_{f \circ \rho_{f,g}} & (\{y_g \mid g \in R\} \text{ ist Matching}) \\
& = & y_{g \circ \pi_{f,g}} & \text{(Diagramm)} \\
& = & y_g \cdot \pi_{f,g}. &
\end{array}$$

Wenn $f'' : C'' \longrightarrow C$ in R liegt existieren die entsprechenden gefaserten Produkte mit den Projektionen $\pi_{f,g}$ und $\pi_{f'',g}$ so, daß $y \cdot g \cdot \pi_{f,g}$ und $y \cdot g \cdot \pi_{f'',g}$ übereinstimmen. Da $R' = (\pi_{f,g})_{f\in R}$ eine Überdeckung in $\mathcal{K}$ und die Familie $\{y \cdot g \cdot \pi_{f,g} \mid f \in R\}$ ein Matching von R' mit (eineindeutigem) Amalgam ist, gilt $y \cdot g = y_g$. ∎

7.1 Mathematischen Beispiele (forts.)

Es wurden bereits einige Topologien angeführt. Hier betrachten wir einige Beispiele für Garben.

7.1.1 Die Topologie der offenen Überdeckungen

Sei $\mathcal{T}$ ein (topologisches) Gebiet, ausgestattet mit der Topologie der offenen Überdeckungen. Dann ist für jeden topologischen Raum Y die repräsentierbare Prä-Garbe

$$\mathsf{y}(Y) \;=\; \mathcal{T}(-, Y) : \mathcal{T}^{\mathsf{op}} \longrightarrow \mathbf{Set}$$

eine Garbe, da für alle offenen Überdeckungen $\{U_i \mid i \in I\}$ eines Raumes $T \in \mathcal{T}$ die Funktionsfamilien $\{\alpha_i : U_i \longrightarrow Y\}$, die auf den Schnitten $U_i \cap U_j$, $i, j \in I$, übereinstimmen eineindeutig, d.h. nur auf eine Weise zu einer Funktion $\alpha : T \subseteq \bigcup_{i\in I} U_i \longrightarrow Y$ zusammengesetzt werden können.

Wenn $\mathcal{J}$ eine Basis einer Topologie auf einer Kategorie $\mathcal{C}$ ist, so daß alle repräsentierbaren Prä-Garben auf $\mathcal{C}$ Garben sind, dann nennt man die Topologie $\mathcal{J}$ subkanonisch. Die kanonische Topologie ist die größte subkanonische Topologie auf $\mathcal{C}$.

Die durch die offenen Surjektionen gegebene Topologie $\mathcal{T}$ ist subkanonisch. Wenn $(f_i : Y_i \longrightarrow X)_{i\in I}$ eine Basisüberdeckung ist, daß die durch die Familie induzierte Abbildung $f : \coprod Y_i \longrightarrow X$ auch eine offene Surjektion ist, dann ist eine Matching-Familie stetiger Funktionen $g_i : Y_i \longrightarrow Z$ die Funktion $g : X \longrightarrow Z$ mit $g \circ f_i = g_i$, $i \in I$, ein Amalgam. g ist stetig, denn für eine offene Teilmenge $U \subseteq Z$ gilt $g^{-1}(U) = ff^{-1}g^{-1}(U)$ da f surjektiv ist, und da f offen ist reicht es zu zeigen, daß $f^{-1}g^{-1}$ offen (auf $\coprod Y_i$) ist. Aber für alle Summanden $Y_j \subseteq \coprod Y_i$ gilt $f^{-1}g^{-1}(U) \cap Y_j = f_j^{-1}g^{-1}(U) = g_j^{-1}(U)$, und das ist eine offene Menge, da g_j stetig war. Da $f^{-1}g^{-1}(U) \cap Y_j$ für jedes $j \in I$ offen ist, folgt daß auch $f^{-1}g^{-1}(U)$ offen ist.

7.1.2 Die atomare Topologie

Auf der atomaren Topologie können Garben wie folgt beschrieben werden.

Lemma 2 *Eine Prä-Garbe P ist dann und nur dann auf der atomaren Topologie über $\mathcal{C}$ eine Garbe, wenn für alle Morphismen $f : D \longrightarrow C$ und jedes $y \in P(D)$ aus $y \cdot g = y \cdot h$ für alle Diagramme*

$$E \overset{h}{\underset{g}{\rightrightarrows}} D \xrightarrow{f} C$$

$(fg = fh)$ ein eindeutiges $x \in P(C)$ mit $y = x \cdot f$ existiert.

Beweis: Angenommen $y \in P(D)$ besitzt die Eigenschaft $y \cdot g = y \cdot h$ für alle g und h mit $fg = fh$. Sei $S = (f) = \{t \mid t = f \circ k\}$ der durch f erzeugte Sieb und $\{x_t\}_{t \in S}$ eine Matchingfamilie via $x_t = y \cdot k$ mit $t = f \circ k$. Diese Definition ist unabhängig von der Wahl von k. Da P eine Garbe ist existiert $x \in P(C)$ mit $x \cdot t = x_t$, $t \in S$. Insbesondere gilt $x \cdot f = y$. Jedes andere x' mit $x' \cdot f = y$ würde auch $x' \cdot t = x_t$ für alle $t \in S$ genügen, womit x eineindeutig mit $x \cdot f = y$ eineindeutig ist.

Für die andere Implikation sei S ein überdeckender Sieb in $\mathcal{C}$ und $\{x_f \mid f \in S\}$ eine zugehörige Matchingfamilie von S. Sei $f_0 \in S$. Dann impliziert $f_0 g = f_0 h$ natürlich $x_{f_0} \cdot g = x_{f_0} \cdot h$ und wir finden ein eineindeutiges Element $x \in P(C)$ mit $x \cdot f_0 = x_{f_0}$. Es bleibt noch $x_f = x \cdot f$ zu zeigen. Sei f in S mit

$$\begin{array}{ccc} D & \xrightarrow{v} & B \\ {\scriptstyle u}\big\downarrow & & \big\downarrow{\scriptstyle f_0} \\ A & \xrightarrow{f} & C. \end{array}$$

Sei $y = x_{f_0} \cdot v = x \cdot f_0 v \in P(D)$. Wenn $g, h : E \longrightarrow D$ zwei Morphismen mit $ug = uh$ sind, dann ist $f_0 vg = g_0 vh$, damit auch auch $yg = yh$. Also existiert ein eineindeutiges $z \in P(A)$ mit $z \cdot u = y$. Aber x_f für $f \in S$ ist eine Matchingfamilie und damit erfüllt sowohl x_f als auch $x \cdot f$ die Gleichung und somit gilt $x_f = x \cdot f$. ■

7.2 Limite von Garben

In diesem kleinen Abschnitt beweisen wir das inverse Limite von Garben wieder Garben sind. Ein Prä-Garben-Diagramm (in der Kategorie $\mathbf{Set}^{\mathcal{C}^{\mathsf{op}}}$) ist die Abbildung

$$I \longrightarrow \mathbf{Set}^{\mathcal{C}^{\mathsf{op}}}, \qquad i \longmapsto P_i.$$

Der inverse Limes in $\mathbf{Set}^{\mathcal{C}^{\mathsf{op}}}$ wird punktweise berechnet, d.h.

$$(\text{co-lim}_{i \in I} P_i)(C) = \text{co-lim}_{i \in I} P_i(C). \tag{18}$$

Rechts wird der Limes in $\mathbf{Set}$ gebildet.

Proposition 2 *Sei $\langle \mathcal{C}, \mathcal{J}\rangle$ ein Gebiet und sei $I \longrightarrow \textbf{Set}^{\mathcal{C}^{op}}$ ein Diagramm aus Prä-Garben P_i. Wenn alle P_i Garben sind, dann ist auch $\mathsf{co\text{-}lim}_{i\in I} P_i$ eine Garbe.*

Beweis: Sei $P = \mathsf{co\text{-}lim}_{i\in I} P_i$ der inverse Limes in der Kategorie $\textbf{Set}^{\mathcal{C}^{\mathsf{op}}}$ der Prä-Garben, also $P(C) = \mathsf{co\text{-}lim}_{i\in I} P_i(C)$. Ist S eine Überdeckung eines Objektes C, dann existiert nach Annahme der Equalizer

$$P_i(C) \longleftarrow \prod_{f\in S} P_i(\mathsf{dom}(f)) \rightrightarrows \prod_{f,g} P_i(\mathsf{dom}(g)) \tag{19}$$

wie in (14). Da sich (inverse) Limite vertauschen lassen ergibt der inverse Limes dieser Equalizerdiagramme wieder ein Equaliserdiagramm der Form (14)

$$P(C) \longleftarrow \prod_{f\in S} P(\mathsf{dom}(f)) \rightrightarrows \prod P(\mathsf{dom}(g)), \tag{20}$$

womit P eine Garbe ist. ∎

7.3 Der Garben-Funktor

Sei $\mathcal{C}$ eine kleine Kategorie und $\mathcal{J}$ eine Grothendiek Topologie auf $\mathcal{C}$. Mit $\mathsf{sh}(\mathcal{C}, \mathcal{J})$ bezeichnen wir die (vollständige) Subkategorie von $\textbf{Set}^{\mathcal{C}^{\mathsf{op}}}$, die aus $\mathcal{J}$-Garben besteht. Ziel dieses Abschnitts wird es sein das folgende fundamentale Resultat zu beweisen.

Theorem 2 *Der Inklusionsfunktor $i : \mathsf{sh}(\mathcal{C}, \mathcal{J}) \longrightarrow \textbf{Set}^{\mathcal{C}^{op}}$ besitzt eine Linksadjungierte*

$$\mathsf{adj} : \textbf{Set}^{\mathcal{C}^{op}} \longrightarrow \mathsf{sh}(\mathcal{C}, \mathcal{J}),$$

die man den (assoziierten) Garben-Funktor nennt. Dieser Funktor vertauscht mit endlichen Limite.

Sei P eine Prä-Garbe über $\mathcal{C}$, d.h. ein Funktor $\mathcal{C}^{\mathsf{op}} \longrightarrow \textbf{Set}$. Die Linksadjungierte adj wird in zwei Stufen konstruiert: zuerst konstruieren wir eine (separierte) Prä-Garbe P^+, die nicht notwendiger weise eine Garbe sein muß. Wir nennen eine Prä-Garbe separiert, wenn eine Matching-Familie höchstens ein Amalgam besitzt. Mit anderen Worten ist P dann separiert, wenn

$$\forall x, y \in P(C) \ \ \forall S \in \mathcal{J}(C) \ \ \forall f \in S \ : \ x \cdot f \ = \ y \cdot f \quad \Longrightarrow \quad x \ = \ y.$$

Also erfüllen separable Prä-Garben die Eineindeutigkeit von Amalgamen aber nicht notwendigerweise deren Existenz. Wir werden für jede Prä-Garbe zeigen, daß P^+ separabel ist und daß P^+ eine Garbe ist, wenn P separabel war. Man erhält adj durch zweimaliges Anwenden von $^+$

$$\mathsf{adj}(P) \ = \ (P^+)^+.$$

Um die Definition von P^+ zu verstehen beachte man, wenn P eine Garbe ist, daß dann für jedes Überdeckung eines Objekts C in $\mathcal{C}$, die Matching-Familie $\{x_f \mid f \in R\}$ durch ein eindeutiges Element aus $P(C)$ repräsentiert werden kann. Mehr noch, wenn P eine Garbe ist und wenn $S \in \mathcal{J}(C)$ eine Verfeinerung von R ist, d.h. $S \subseteq R$, dann wird die Subfamilie $\{x_f \mid f \in S\}$, die eine Matching-Familie für S ist, durch das selbe Element repräsentiert. Sei P eine beliebige Prä-Garbe. Wir definieren P^+ durch

$$P^+ \quad = \quad \textsf{co-lim}_{R \in \mathcal{J}(C)} M(R, P), \tag{21}$$

wobei $M(R, P)$ die Menge der Matching-Familien für die Überdeckungen R von C bezeichnet und der Limes über alle Überdeckungen, geordnet nach umgekehrter Inklusion, gebildet wird. Mit anderen Worten ist ein Element von $P^+(C)$ eine Äquivalenzklasse von Matching-Familien

$$x \;=\; \{x_f \mid f : D \longrightarrow C \in R\},\; x_f \in P(D), \qquad \text{und} \qquad x_f \cdot k \;=\; x_{fk}, \tag{22}$$

für alle $k : E \longrightarrow D$. Zwei Familien $x = \{x_f \mid f \in R\}$ und $y = \{y_f \mid f \in S\}$ sind äquivalent, wenn eine gemeinsame Verfeinerung $T \subseteq R \cap S$,$T \in \mathcal{J}(C)$, mit $x_f = y_f$ für alle $f \in T$ existiert.

Das so definierte P^+ besitzt dann die Struktur einer Prä-Garbe. Die Restriktionsabbildung $P^+(C) \longrightarrow P^+(C')$ für einen Morphismus $h : C' \longrightarrow C$ in $\mathcal{C}$ wird durch

$$\{x_f \mid f \in R\} \cdot h \quad = \quad \{x_{hf'} \mid f' \in h^{\leftarrow}(R)\} \tag{23}$$

festgelegt. (Man beachte, daß diese Abbildung wohldefiniert ist.) Diese Konstruktion $P \longmapsto P^+$ ist ein Funktor von P, da jede Abbildung $\Phi : P \longrightarrow Q$ von Prä-Garben (also natürliche Transformationen) auf evidente Art und Weise eine Abbildung $\Phi^+ : P^+ \longrightarrow Q^+$ induziert. Mehr noch, es existiert eine kanonische Abbildung

$$\eta : P \quad \longrightarrow \quad P^+$$

die alle $x \in P(C)$ auf die Äquivalenzklasse der Matching-Familie schickt

$$\eta_C(x) \quad = \quad \{x \cdot f \mid f \in t_C\}, \tag{24}$$

wobei t_C der maximale Sieb ist.

Als Illustration betrachten wir einige Instanzen dieser Konstruktion (über einem topologischen Raum X (Topologie der offenen Überdeckungen). Sei B die Prä-Garbe B so, daß $B(U)$ die Menge aller beschränkten und stetigen Funktionen $f : U \longrightarrow R$ ist. Im allgemeinen ist B keine Garbe, da X offene Mengen besitzen kann, die nicht kompakt sind. Es könnte eine unendliche Überdeckung von U existieren, die (kombiniert) zu einer unbeschränkten stetigen Funktion führen könnte. Betrachten wir aber die Prä-Garbe C in der $C(U)$ die Menge aller stetigen Funktionen $U \longrightarrow R$ ist, so stellen wir fest, daß es sich tatsächlich um eine

Garbe handelt, und es existiert eine Abbildung $B^+(U) \longrightarrow C(U)$, da jede Matching-Familie beschränkter und stetiger Funktionen $f_n : U_n \longrightarrow R$ auf einer beschränkten Überdeckung U_i (von U) sich zu einer eindeutig bestimmten stetigen Funktion $f : U \longrightarrow R$ kombinieren läßt. Äquivalente Matchings liefern natürlich die selbe Funktion. Umgekehrt läßt sich jede stetige Funktion $f : U \longrightarrow R$ durch eine Matching-Familie $f_n : U_i \longrightarrow R$ beschreiben. Beispielsweise ist $U_n = \{x \mid |f(x)| < n\}$ mit $f_n = f_{|U_n}$ eine solche Beschreibung. Also überführt die Konstruktion die Prä-Garbe B in ihre assoziierte Garbe C.

Jedoch wird eine einmalige Anwendung von $^+$ nicht immer zum Ziel führen. Betrachten wir beispielsweise die Menge S (mit mehr als einem Element) und die zugehörige konstante Prä-Garbe P auf dem Raum X, die durch $P(U) = S$ auf der offenen Menge U definiert ist (mit allen Restriktionen und der Identität auf S). Dann ist $P^+(U) = S$ wenn $U \neq \emptyset$, während $P^+(\emptyset) = 1$ (für die leere Überdeckung gibt es genau eine Familie, die leere). Deshalb ist P^+ keine Garbe, denn sind U_1 und U_2 zwei disjunkte offene Mengen auf X, dann muß $s_1 \in P^+(U_1)$ und $s_2 \in P^+(U_2)$ die eineindeutige Restriktion auf $P^+(\emptyset)$ aus dem Element in $P^+(\emptyset)$ bestehen. Also ist danach $\{s_1, s_2\}$ eine Matching-Familie für die Überdeckung (U_1, U_2) von $U_1 \cup U_2$. Jedoch sind s_1 und s_2 verschieden und somit existiert kein Amalgam. Betrachtet man jedoch $P^{++}(U)$ so existieren Amalgame. Ein Element aus $P^{++}(U)$ ist eine (Äquivalenzklasse von Mengen von Elementen $s_i \in P^+(U_i)$, die matchen ($s_i = s_j$), falls sie sich überlappen, d.h. $U_i \cap U_j \neq \emptyset$. Also passen diese Elemente $s_i \in S$ zusammen und ergeben eine Funktion $s : U \longrightarrow S$ mit der Eigenschaft, daß jeder Punkt von U eine offene Überdeckung (beispielsweise ein U_i) besitzt, auf dem die Funktion konstant ist. Mit anderen Worten ist $P^{++}(U)$ die Menge der lokal konstanten Funktionen $S : U \longrightarrow S$.

Allgemein erzeugt die $^+$-Konstruktion eindeutige Amalgame, wenn die Matchingfamilien von Elementen die aus P stammen.

Lemma 3 *(i) Eine Prä-Garbe P ist separiert dann und nur dann, wenn $\eta : P \longrightarrow P^+$ ein Monomorphismus ist.*

(ii) Eine Prä-Garbe ist genau dann eine Garbe, wenn $\eta : P \longrightarrow P^+$ ein Isomorphismus ist.

Beweis: Zum Beweis von (i) sei $x, y \in P(C)$ für ein Objekt C in $\mathcal{C}$. Dann bedeutet $\eta(x) = \eta(y)$ genau $x \cdot f = y \cdot f$ für alle f in einem überdeckenden Sieb S. Die Existenz eines Amalgams in (ii) ist ebenfalls offensichtlich. ■

Lemma 4 *Ist F eine Garbe und P eine Prä-Garbe, dann faktorisiert jede Abbildung $\Phi : P \longrightarrow F$ eineindeutig als $\Phi = \tilde{\Phi} \circ \eta$ mit $\eta : P \longrightarrow P^+$.*

$$\begin{tikzcd}
P \arrow[r, "\eta"] \arrow[dr, "\Phi"'] & P^+ \arrow[d, dotted, "\tilde{\Phi}"] \\
 & D.
\end{tikzcd}$$

Beweis: Ein Element aus $P^+(C)$ wird durch die Matching-Familie $\{x_f \mid f \in R\}$ von P für eine Überdeckung von C (aus $\mathcal{C}$) repräsentiert. Für alle Morphismen $h : D \longrightarrow C$ in R ist nach Definition in (24) $\eta_D(x_h) = \{x_h \cdot k \mid k \in t_D\}$. Es gilt aber auch $\{x_f \mid f \in R\} \cdot h = \{x_{hf'} \mid f' \in h^{\leftarrow}(R)\}$. $h^{\leftarrow}(R)$ ist aber der maximale Sieb t_D wegen $h \in R$. Da $\{x_f \mid f \in R\}$ eine Matching-Familie ist, erhält man die Gleichung

$$\eta_D(x_h) \quad = \quad \{x_f \mid f \in R\} \cdot h, \quad h : D \longrightarrow C \ \in \ R.$$

Deshalb nimmt die Abbildung $\tilde{\Phi}$ (in dem Diagramm) (falls sie existiert) den eineindeutigen Wert $y \in F(C)$ mit

$$y \cdot h \ = \ \tilde{\Phi}\{x_f \mid f \in R\} \ = \ \tilde{\Phi}(\{x_f \mid f \in R\} \cdot h) \ = \ \tilde{\Phi}(\eta_D(x_h)) \ = \ \Phi(x_h)$$

für alle $h \in R$ auf der Familie $\{x_f \mid f \in R\}$ an. Aber dieses (eineindeutige) $y \in F(C)$ existiert, da F eine Garbe und $\{\Phi(x_h) \mid h \in R\}$ eine Matching-Familie ist. ■

Lemma 5 *Für alle Prä-Garben P ist P^+ eine separierte Prägarbe.*

Beweis: Betrachten wir zwei Elemente $x, y \in P^+(C)$ mit $x \cdot h = y \cdot h$ für alle h in einer Überdeckung Q von C. Wir zeigen $y = x$. Seien x und y die Familien $x = \{x_f \mid f \in R\}$ und $y = \{y_g \mid g \in S\}$ für Überdeckungen R und S in $\mathcal{J}(C)$. Die Gleichung $x \cdot h = y \cdot h$ für $h : D \longrightarrow C$ in Q bedeutet, daß es eine Überdeckung $T_h \subseteq h^{\leftarrow}(R) \cap h^{\leftarrow}(S)$ von D mit $x_{ht} = y_{ht}$ für alle $t \in T_h$ gibt. Mit dem Transitivitätsaxiom ist auch die Familie

$$T = \quad \{ht \mid h \in Q, t \in T_h\}$$

eine Überdeckung von C (mit $T \subseteq R \cap S$). Also sind $x = y$ Elemente aus $P^+(C)$. ■

Lemma 6 *Ist P eine separierte Garbe, dann ist P^+ eine Garbe.*

Beweis: Um zu zeigen, daß P^+ eine Garbe ist müssen wir für jede Matching-Familie (aus Elementen in P^+) ein Amalgam finden. Eine Matching-Familie wird durch eine Überdeckung $R \in \mathcal{J}(C)$ und die Elemente x_f, $f \in R$, beschrieben ($x_f \in P^+(D)$), falls $f : D \longrightarrow C$. Jedes x_f ist durch eine Äquivalenzklasse von Matching-Familien aus P

$$x_f \quad = \quad \{x_{f,g} \mid g \in S_f\}, \quad g : E \longrightarrow D, \quad x_{f,g} \in P(E) \tag{25}$$

für eine Überdeckung $S_f \in \mathcal{J}(D)$ beschrieben. Das x_f zu $f : D \longrightarrow C$ korrespondiert bedeutet, daß für einen (beliebigen) Morphismus $h : D' \longrightarrow D$ die Beziehung $x_f \cdot h = x_{fh}$ für Elemente aus $P^+(D')$ erfüllt ist. Nach Definition von $- \cdot h$ in (23) ist das gleichbedeutend mit der Äquivalenz

$$\{x_{f,hg'} \mid g' \in h^{\leftarrow}(S_f)\} \quad \cong \quad \{x_{fh,g} \mid g \in S_{fh}\},$$

und das wiederum bedeutet, daß eine Überdeckung $T_{f,h} \subseteq g^{\leftarrow}(S_f) \cap S_{fh}$ von D' mit

$$x_{f,hg''} \;=\; x_{f\circ h,g''} \quad \forall g'' \in T_{f,h} \tag{26}$$

existiert. Sei Q der Sieb $\{f \circ g \mid f \in R, g \in S_f\}$. Da sowohl R als such S_f Überdeckungen sind, ist auch (nach dem Transitivitätsaxiom) Q eine Überdeckung von C. Wir definieren eine Familie $y \in P^+(C)$ für diese Überdeckung durch

$$x_{f\circ g} \;=\; x_{f,g} \tag{27}$$

Wir zeigen, daß diese Definition unabhängig von der Wahl der Faktorisierung von $f \circ g$ ist. Sei $fg = f'g'$ für Morphismen $f, f' \in R$ und $g \in S_f$, $g' \in S_{f'}$. Dann gilt für ein $k \in T_{f,g} \cap T_{f',g'}$

$$\begin{aligned}
x_{f,g} \cdot k &= x_{f,gk} && \text{da } x_f \text{ eine Matching-Familie ist} \\
&= x_{f\circ g,k} && \text{nach (26)} \\
&= x_{f'\circ g',k} && \text{nach Annahme} \\
&= x_{f',g'k} && \text{nach (26)} \\
&= x_{f',g'} \cdot k && \text{da } x_{f'} \text{ eine Matching-Familie ist.}
\end{aligned}$$

Da P separiert ist und da $T_{f,g} \cap T_{f',g'}$ eine Überdeckung von D' ist gilt $x_{f,g} = x_{f',g'}$. Somit ist y in (27) wohldefiniert. Da die y_f Matching-Familien waren ist auch $y = \{x_h \mid h \in Q\}$ eine Matching-Familie und somit ein Element von $P^+(C)$.

Weiter können wir zeigen, daß das Element y ein Amalgam für die Matching-Familie $\{x_f \mid f \in R\}$ ist. Für jedes $f : D \to C$ in R zeigen wir, daß

$$y \cdot f \;=\; \{y_{f\circ h} \mid h \in f^{\leftarrow}Q\} \quad \text{und} \quad x_f \;=\; \{x_{f,g} \mid g \in S_f\}$$

das selbe Element in $P^+(D)$ repräsentiert. Da $S_f \subseteq f^{\leftarrow}(Q)$ ist das (nach Definition von Q) für alle $g \in S_f$ gilt nach (27) $y_{f\circ g} = x_{f,g}$. Das beweist die Existenz eines Amalgams. Da P^+ separiert ist, (siehe voriges Lemma), ist das Amalgam eineindeutig. Damit ist P^+ eine Garbe. ∎

Aus den beiden Lemmata folgt unmittelbar das Theorem. Man bracht nur den assoziierten Garben-Funktor adj via $a(P) = (P^+)^+$ zu definieren. Das definiert einen Funktor auf den Prä-Garben P und mit den beiden Lemmata erhält man durch Anwendung von adj eine Garbe. Die in (24) definierte Abbildung führt zu einer zusammengesetzten Abbildung

$$P \xrightarrow{\eta(P)} P^+ \xrightarrow{\eta(P^+)} P^{++} \tag{28}$$

von P zur Garbe P^{++}. Diese Komposition ist universell unter den Abbildungen von P. Deshalb ist adj die Linksadjungierte und zu den Inklusionen von Garben in Prä-Garben und die Komposition (28) ist die Einheit der Adjungation.

Man beachte ist P bereits separiert, dann benötigt man nur einen Schritt in (28), und wenn P bereits eine Garbe ist, dann ist die Einheit ein Isomorphismus. Mit anderen Worten

Korollar 1 *Die zusammengesetzte Abbildung* adj *nach der Inklusion*

$$\mathit{adj} \circ i : \mathit{sh}(\mathcal{C}, \mathcal{J}) \longrightarrow \mathit{sh}(\mathcal{C}, \mathcal{J})$$

ist natürlich isomorph zum Identitäts-Funktor.

Es bleibt zu zeigen, daß der assoziierte Garben-Funktor adj endliche Limite erhält. Es reicht zu zeigen, daß die $^+$-Konstruktion $P \longrightarrow P^+$ endliche Limite erhält. Als erstes beobachten wir, daß für ein Objekt C in $\mathcal{C}$ mit einem überdeckenden Sieb $P(R) \in \mathcal{J}(C)$ der Funktor $P \longrightarrow M_C(R, P)$ von den Prä-Garben nach **Set** alle Limite erhält. D.h. für alle Funktoren $P : I \longrightarrow \mathbf{Set}^{\mathcal{C}^{\mathsf{op}}}$ von einer (Index-)Kategorie I in die Prä-Garben gilt

$$M_C(R, \mathsf{co\text{-}lim} P_i) \quad = \quad \mathsf{co\text{-}lim} M_C(R, P_i) \tag{29}$$

Tatsächlich existiert für jede Prä-Garbe P ein natürlicher Isomorphismus

$$M_C(R, P) \quad = \quad \mathsf{Hom}_{\mathbf{Set}^{\mathcal{C}^{\mathsf{op}}}}(R, P),$$

und der Funktor $\mathsf{Hom}(R, -)$ erhält Limite.

Das Bild $P^+(C)$ ist in (21) durch ein co-Limes von M über alle durch (umgekehrte) Inklusion geordnete Überdeckungen definiert. Diese partielle Ordnung ist Filtern. (Je zwei Überdeckungen besitzen eine gemeinsamer Verfeinerung, ihren Durchschnitt.) Gefilterte co-Limite vertauschen mit endlichen Limite. Also erhält $P \longrightarrow P^+$ endliche Limite.

8 Gebiete, Garben und Mustererkennung

Den Begriff Gebiet benützen wir im Sinne eines Datenraumes, da der Ausdruck Bereich (Domain) schon anderweitig in der Semantik belegt ist. Auch spiegelt die Bezeichnung Gebiet eine gewissen geometrischen Charakter wieder. Siebe, im weiteren auch elementare Überdeckungen genannt, beschreiben einen (lokalen) Zusammenhang.

8.1 Beispiele aus der Informatik

Wir werden eine überdeckende elementare Umgebung mit einer Familie erzeugender Pfeile beschreiben. Diese Familien nennen wir Überdeckungsfamilien. In analoger Weise verfahren wir bei der Beschreibung der Topologie, indem wir eine Menge von Überdeckungen beschreiben. Die erzeugte Topologie soll die kleinste (bzgl. Inklusion) sein, die die Menge der Überdeckungen enthält und den drei Axiomen genügt (Basis).

Beispiel 1 (Mengen) Eine Überdeckung einer Menge S in der Kategorie **Set** ist eine Familie von Teilmengen $(S_i)_{i\in I}$ von S (bzw. die Familie der Injektionen $(S_i \xrightarrow{i} S)_{i\in I}$), deren Vereinigung S ist ($\bigcup_{i\in I} S_i = S$). ∎

Beispiel 2 (Zusammenhängende Graphen) Unter einem Graphen verstehen wir ein Paar von Mengen (V, E), Knoten mit V und Kanten mit $E \subseteq V \times V$ bezeichnet sind. Ein Pfad ist eine Knotensequenz $(a_1, a_2, \ldots, a_n) \in V^*$, so daß $(a_i, a_{i+1}) \in E$ für alle $1 \leq i < n$ gilt. Er ist zusammenhängend, falls zwischen je zwei Knoten ein Pfad existiert. Ein Graphmorphismus $f : G = (V_G, E_G) \longrightarrow H = (V_H, E_H)$ ist ein Paar von Abbildungen

$$f_V : V_G \longrightarrow V_H, \quad f_E : E_G \longrightarrow E_H,$$

die mit den Kantenrelationen verträglich sind, d.h.

$$\forall (a, b) \in E_G : \qquad f_E(a, b) = (f_V(a), f_V(b))$$

Ein Untergraph soll ein Graph sein, dessen Knoten- und Kantenmenge Teilmengen des Ausgangsgraphen sind. Die zusammenhängenden Graphen mit ihren Morphismen formen die Kategorie **ConGraph**.

Eine Überdeckung eines Graphen definieren wir durch die Familie aller zusammenhängenden Untergraphen $(i : G_i \longrightarrow G)_{i\in I}$, so daß

$$\bigcup_{i\in I} V_{G_i} = V_G \quad \mathit{und} \quad \bigcup_{i\in I} E_{G_i} = E_G$$

gilt.

Damit sind die Kategorien der zusammenhängenden ungerichteten Graphen **UndirConGraph**, die der azyklischen zusammenhängenden ungerichteten Graphen **AcycUndirConGraph**, auch **Tree** genannt, mit einer Topologie versehen.
∎

Beispiel 3 (Wörter) Unter einem (abstrakten) Wort verstehen wir ein Paar $(S, <_S)$, bestehend aus einer Menge S und einer linearen Ordnung $<_S \subseteq S \times S$. D.h., $<_S$ ist total, irreflexiv, transitiv und antisymmetrisch. Eine Teilmenge $R \subseteq S$ nennen wir nebeneinanderstehend, wenn

$$\forall a, b \in R\, \forall x \in S : \; a <_S x <_X \; b \implies x \in R.$$

Ein Morphismus ist eine, die Ordnung erhaltende, Abbildung zwischen zwei Wörtern mit nebeneinanderstehenden Bildern. Wörter und Morphismen formen eine Kategorie **String**. Ein String $(T, <_T)$ ist ein Teilstring von $(S, <_S)$, falls $T \subseteq S$ eine nebeneinanderstehende Menge in S ist. Als Überdeckung eines Strings definieren wir die Menge aller Subwörter. ■

Beispiel 4 (Markierte Strukturen) Graphen können markiert werden. Eine Markierung einer Struktur S ist eine Funktion

$$\Lambda : U(S) \longrightarrow L,$$

die der Struktur zugrunde liegenden Menge $U(S)$ Markierungen aus der Menge L zuordnet. Ein Morphismus zwischen markierten Strukturen ist ein gewöhnlicher Morphismus zwischen den nicht markierten Strukturen, der mit der Markierungsfunktion verträglich ist, d.h.

$$\forall a \xrightarrow{f} b\, \forall s \in U(a) : \; \Lambda_a(s) = \Lambda_b(f(s)).$$

Diese Definition führt zu Kategorien mit markierten Objekten. Die Definition einer Topologie überträgt sich auf den markierten Fall. ■

9 Garben und Umgebungen

Elementare Umgebungen waren als Pfeilmengen, die unter rechts-Komposition abgeschlossen waren, definiert. Damit besitzt eine elementare Umgebung J die Eigenschaften eines Funktors $J : \mathcal{C} \longmapsto \mathcal{C}^{\rightarrow}$ in die entsprechende Pfeilmengen-Kategorie, der Bild-Kategorie des Hom-Funktors. Wenn wir uns auf Pfeile in elementare Umgebungen beschränken, so ist der elementare Umgebungs ein Sub-Funktor des Hom-Funktors.

Definition 3 (elementare Umgebungen in Funktorrepräsentation) *Eine elementare Umgebung $R = (a_i \xrightarrow{f_i} a)_{i \in I}$ von einem Objekt a aus $\mathcal{C}$ wird als Sub-Funktor des Hom-Funktors $\mathsf{hom}_{\mathcal{C}}(-, a)$ betrachtet.*

$$\begin{array}{rcl} b & \longmapsto & \{f \mid f \in R \;\; \mathsf{dom}(f) = b\} \\ c \xrightarrow{g} b & \longmapsto & g^{\leftarrow} : f \longmapsto f \circ g \end{array}$$

Definition 4 (Prä-Garben) *Eine Prä-Garbe auf einer Kategorie $\mathcal{C}$ ist ein kontravarianter Funktor von $\mathcal{C}$ in die Kategorie der Mengen **Set**.*

$$\mathsf{psh} : \mathcal{C}^{\mathsf{op}} \longrightarrow \boldsymbol{Set}$$

Eine Garbe ist eine Prä-Garbe, die es erlaubt aus lokaler Information auf globalen Eigenschaften zu schließen.

Definition 5 (Garben) *Eine Prä-Garbe auf einem Gebiet $\langle \mathcal{C}, \mathcal{J} \rangle$ ist eine Garbe, wenn für jedes Objekt a in $\mathcal{C}$ und jede überdeckende elementare Umgebung $R \in J(a)$ jeder Morphismus $R \longrightarrow F$ in $\boldsymbol{Set}^{\mathcal{C}^{\mathsf{op}}}$ eine eindeutige Erweiterung auf einen Morphismus $\mathsf{hom}_{\mathcal{C}}(-, a) \longrightarrow F$ besitzt.*

In der Definition der Garben werden elementare Umgebungen implizit als Funktoren betrachtet. Ein Morphismus $\tau : R \longrightarrow F$ in der Funktor-Kategorie $\mathbf{Set}^{\mathcal{C}^{\mathsf{op}}}$ ist eine natürliche Transformation.

Mit dieser Darstellung läßt sich das Konzept einer *kompatiblen Familie* einer Überdeckung modellieren. Für einen Sieb $R = (f_i : a_i \longrightarrow a)_{i \in I}$ ist eine kompatible Familie von Elementen aus F eine Menge von Elementen $(b_i \in F(a_i))_{i \in I}$; ein Element für jeden Pfeil im Sieb. Kompatibilität bedeutet dabei, daß jeder Pfeil $a_i \xrightarrow{u} a_j$ aus $\mathcal{C}$, für den das Diagramm

$$\begin{tikzcd}
a_i \arrow[rr, "u"] \arrow[dr, "f_i"'] & & a_j \arrow[dl, "f_j"] \\
& a &
\end{tikzcd}$$

kommutiert, die Funktion $F(u)$ s_j auf s_i schickt.

$$\begin{tikzcd}
& a_i \arrow[dl] \arrow[dd, "u"] & F(a_i) = s_i \\
a & & \\
& a_j \arrow[ul] & F(a_j) = s_j \arrow[uu, "F(u)"']
\end{tikzcd}$$

Morphismen der Form $\mathsf{hom}_{\mathcal{C}}(-, a) \longrightarrow F$ korrespondieren in eineindeutiger Weise zu den Elementen der Menge $F(a)$. Das bedeutet: die Garben-Bedingung sichert eine Bijektion zwischen kompatiblen Familien und Elementen aus $F(a)$, unabhängig von der Überdeckung von a, zu.

10 Beispiele — Färbbarkeit und Funktionen

Beispiel 1 (Graphfärbbarkeit) Betrachten wir das Gebiet der ungerichteten zusammenhängenden Graphen $\langle \mathbf{UndirConGraph}, \mathcal{J} \rangle$. Die Sub-Kategorie, die nur Inklusionspfeile enthält, bezeichnen wir mit $\mathbf{UndirConGraph}^{\hookrightarrow}$. Wir betrachten das *NP*-vollständige Problem eine Färbung eines vorgegebenen Graphen mit k Farben zu ermitteln. Wir definieren dazu den kontravarianten Funktor

$$C : \mathbf{UndirConGraph}^{\hookrightarrow} \longrightarrow \mathbf{Set}\,,$$

wobei $C(G)$, die Menge aller k-Färbungen von G und $C(f)$, die Restriktionen der Färbungen von H auf G mit $G \xrightarrow{f} H$ ist. (Wenn ein Graph eine k-Färbung besitzt, dann besitzt auch jeder seiner Untergraphen diese Färbung.)

Dieser Funktor C ist eine Garbe, da für eine Überdeckung $(G_i \xrightarrow{f_i} G)_{i\in I} \approx (G_i)_{i\in I}$ und eine Familie von möglichen Färbungen $(c_i \in C(G_i))_{i\in I}$, die auf den Schnitten $G_i \cap G_j$ übereinstimmen, eine Färbung auf dem ganzen Graphen induziert wird. ∎

Da der Kategorienbegriff durch Mengen und Funktionen motiviert wird, betrachten wir noch Garben auf der Kategorie **Set** .

Beispiel 2 (Funktionen) Sei D eine Menge. Die Potenzmenge $\mathcal{P}(D)$ formt mit den Injektionen als Pfeilen in natürlicher Weise eine Kategorie. Wir erhalten ein Gebiet durch Definieren einer Umgebung einer Teilmenge von D. Beispielsweise können wir eine Umgebung einer Menge $X \subseteq D$ durch eine Familie $(X_i)_{i\in I} = (X_i \xrightarrow{X_i} X)_{i\in I}$ mit $\bigcup_{i\in I} X_i = X$ definieren. Sei R^D die Menge der Funktionen, die D nach R abbilden. Wir definieren einen kontravarianten Funktor $F : \mathcal{P}(D)^{\mathsf{op}} \longrightarrow \mathbf{Set}$ durch

- $F(X) = R^X$ für alle $X \subseteq D$ und
- $F(i : X \hookrightarrow Y) : g \longmapsto g_{|X} = g \circ i$ für die Injektionen i.

Dieser Funktor ist eine Garbe, da eine Funktion f durch Wertepaare $(x, f(x))$ vollständig beschrieben ist. Wenn die Familie $(X_i)_{i\in I}$ die Menge X überdeckt und $(f_i : X_i \longrightarrow R)_{i\in I}$ eine zu dieser Familie korrespondierende Familie von Funktionen ist, mit $f_{i|X_i\cap X_j} = f_{j|X_i\cap X_j}$ $(i,j \in I)$, dann existiert eine eindeutig bestimmte Funktion, so daß $f(x) = f_i(x)$ für alle $x \in X_i$ gilt. ∎

11 Spezifikation des Erkennungsproblems

Wir modellieren Vorkommen eines Musters p in einem Suchbereich t als Pfeil $p \longrightarrow t$ in einem Gebiet. Dazu treffen wir ein paar vereinfachende plausible Annahmen, die keine wesentlichen Einschränkungen für unsere Anwendungen darstellen.

- Endlichkeit: alle Objekte im Gebiet sind endlich.

- Diskretion: jedes Objekt besitzt eine bezüglich Inklusion $\subseteq$ feinste Überdeckung, die endlich ist, d.h. kein Objekt besitzt eine beliebig feine Zerlegung.
- Modell: jeder Pfeil im Gebiet modelliert ein Vorkommen.
- Vollständigkeit: alle Überdeckungen sind strikt epimorphe Familien.

Die letzte Bedingung bedarf einer Erklärung. Eine epimorphe Familie von Pfeilen ist eine Verallgemeinerung von surjektiven Funktionsfamilien. Das sind Funktionsfamilien $(f_i : A_i \longrightarrow A)_{i\in I}$ die $\bigcup_{i\in I} f(A_i) = A$ genügen, d.h. jedes der Elemente im Bildbereich wird getroffen.

Definition 6 (Epimorphismus) *Ein Morphismus $a \xrightarrow{f} b$ in einer Kategorie $\mathcal{C}$ ist ein Epimorphismus, wenn alle Pfeile $b \xrightarrow{g} c$ und $b \xrightarrow{h} c$ die $g \circ f = h \circ f$ genügen bereits identisch sind, d.h. $g = h$.*

Wir definieren jetzt Familien, die die gesamte Information über die co-Domains enthalten. Das bedeutet, daß Morphismen die auf Elementen einer Überdeckung eines Objektes definiert sind, bereits eindeutig auf dem durch die Überdeckung induzierten Gebiet vollständig festgelegt sind.

Definition 7 (strikt epimorphe Familie) *Sei $F = (a_i \xrightarrow{f_i} a)_{i\in I}$ eine Familie von Morphismen. Wir nennen $G = (a_i \xrightarrow{g_i} a)_{i\in I}$ mit F kompatibel (oder verträglich), wenn für jedes Objekt c und alle Indizes $i, j \in I$ und allen Morphismen $u : c \longrightarrow a_i$, $v : c \longrightarrow a_j$) mit $f_i \circ u = f_j \circ v$ aus $g_i \circ u = g_j \circ v$ folgt. D.h., für alle u, v und c zieht die Kommutation des ersten Diagrammes die des zweiten nach sich.*

$$\begin{array}{ccc} c & \xrightarrow{u} & a_i \\ \downarrow{\scriptstyle v} & & \downarrow{\scriptstyle f_i} \\ a_j & \xrightarrow{f_j} & a \end{array} \quad \Longrightarrow \quad \begin{array}{ccc} c & \xrightarrow{u} & a_i \\ \downarrow{\scriptstyle v} & & \downarrow{\scriptstyle g_i} \\ a_j & \xrightarrow{g_j} & a \end{array}$$

Eine Familie F ist strikt epimorph, wenn für jede mit F kompatible Familie G ein eindeutig bestimmter Pfeil $a \xrightarrow{h} b$ existiert, der G faktorisiert, d.h.

$$\forall i \in I : \; g_i = h \circ f_i.$$

Das Zuordnen eines faktorisierenden Morphismuses h zu einer Familie G kann als eine Bündel-Operation betrachtet werden. Die Pfeile von F werden quasi zu Bündeln (in G) zusammengefaßt. Intuitiv ist eine Überdeckung $(f_i : a_i \longrightarrow a)_{i\in I}$ eine Dekomposition des Objektes a, und der Pfeil h macht diese Zerlegung zum Teil wieder rückgängig — h konzentriert die Domains von g_i.

Diese Eigenschaft ähnelt der eines inversen Limes, ist aber nicht ganz so stark. Bei einem inversen Limes fordert man die Universalität für jedes Diagramm (in der Kategorie), wohingegen wir hier nur die Universalität für die durch die Familie erzeugte elementare Umgebung verlangen.

$$\begin{tikzcd}
 & a_i \arrow[dr, "f_i"] \arrow[drr, "g_i"] & & \\
c \arrow[ur, "u"] \arrow[dr, "v"'] & & a \arrow[r, dotted, "h"] & a \\
 & a_j \arrow[ur, "f_j"'] \arrow[urr, "g_j"'] & &
\end{tikzcd}$$

Beispiel (Einführende Beispiele, Forts.) Bei allen einführenden Beispielen waren die Überdeckungen strikt epimorphe Familien außer bei dem Gebiet $\langle \mathbf{String}, \mathcal{J} \rangle$. Wir definieren den Überdeckungsbegriff für Strings neu, so daß die überdeckenden Familien strikt epimorph werden.

Sei $(s, <_s)$ ein String und $x, y \in s$ zwei benachbarte Buchstaben. Dann definieren wir eine Überdeckung eines Strings $s = a_1 < a_2 < \ldots < a_n$ als eine Familie $(s_i \longrightarrow s)_{i \in I}$, so daß je für zwei benachbarte Buchstaben a_i, a_{i+1} ein Teilstring $s_i = s'_i < a_i < a_{i+1} < s''_i$ in der Familie vorkommt. ∎

11.1 Die Datenstruktur

Mit den bis jetzt eingeführten Begriffen sind wir in der Lage, das abstrahierte Mustererkennungsproblem zu formulieren. Wir beschreiben das Problem durch

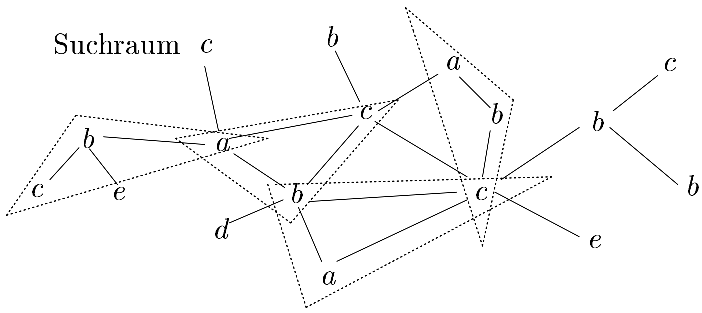
- eine Datenstruktur, ein Gebiet $\langle \mathcal{C}, \mathcal{J} \rangle$, das den oben angeführten Forderungen genügt
- und eine Vorkommen-Relation zwischen Objekten p, t der zugrunde liegenden Kategorie $\mathcal{C}$, die wir durch Pfeile $p \longrightarrow t$ modellieren.

Kategoriell formuliert bedeutet das, daß wir die Menge $\mathsf{hom}_{\mathcal{C}}(p, t)$ bestimmen wollen.

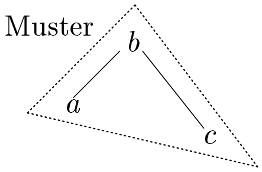
Wir werden die Vorkommen-Relation als einen Garben-Funktor $\mathsf{hom}_{\mathcal{C}}(-, t)$ darstellen. Aus unserer Annahme über die strikte Epimorphie überdeckender Familien folgt, daß die Hom-Funktoren Garben sind. Die Garben-Eigenschaft erlaubt es uns dann, Vorkommen durch Dekomposition der Objekte zu erkennen.

Durch unsere Einschränkungen, werden die Hom-Funktoren automatisch Garben, und das Mustererkennungsproblem ist bereits durch die Topologie der Datenstruktur vollständig beschrieben.

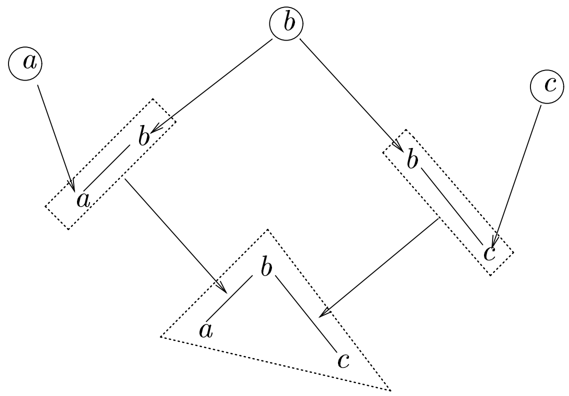
Beispiel (Graphen, Forts.) In diesem Beispiel betrachten wir an Knoten markierte, ungerichtete (endliche), zusammenhängende Graphen aus der Kategorie **LabUndirConGraph** mit der früher schon auf solchen Graphen definierten Topologie.



In dem Bild wird ein Graph gezeigt, in dem ein Muster



gesucht werden soll. Dazu wird eine Überdeckung des Musters konstruiert



und hernach die Mengen $\mathsf{hom}_{\mathcal{C}}(p_i, t)$ berechnet (partielle Vorkommen von Musterteilen) und dann zu vollständigen Vorkommen $p \longrightarrow t$ gebündelt. Es verbleibt lediglich noch, kompatible Familien von partiellen Vorkommen zu identifizieren. ■

Die Lösung des Erkennungsproblems besteht also im Berechnen von $\mathsf{hom}_{\mathcal{C}}(p, t)$. Wir konstruieren die Vorkommen via Garben der Form $\mathsf{hom}_{\mathcal{C}}(-, t)$. Die Tatsache, daß bereits die Hom-Funktoren Garben sind, erleichtert uns die Konstruktion eines Algorithmus. Das Problem konnte so einfach beschrieben werden, weil eine entsprechende Topologie auf der Datenstruktur vorausgesetzt war. Die Garben-Eigenschaft erlaubt es, Vorkommen durch Auffinden von partiellen Vorkommen zu rekonstruieren, indem via eines faktorisierenden Morphismuses h (siehe strikt epimorphe Familien) die Vorkommen zu Bündeln zusammengefaßt werden können.

Ein konstruktiver Beweis für die strikte Epimorphie einer Überdeckung führt zu einer konkretisierenden Operation h.

12 Die extentionalen Darstellung

Wir transformieren die Idee, das Auftreten von Mustern mit Hilfe von Garben zu rekonstruieren, durch Anwenden einiger Konstruktionsparadigmen für den Entwurf von Algorithmen, in eine extentionale Darstellung. Die folgenden vier Paradigmen werden dabei angewandt.

- *Divide and Conquer:* Erforschen der einzelnen „Halme“ der Garben und Zusammenfassen der gleichen zu einem Vorkommen
- *Finite Differencing:* Speichern partieller Vorkommen sowie Verwenden und Erweiterndieses Wissens während des Traversierens des Suchobjekts
- *Backtracking:* Durch Verwerfen von nicht vervollständigbarer partieller Vorkommen
- *Partial Evaluation:* Durch Vorwegberechnen konstanter Information wie beispielsweise die der Musterzerlegung

12.1 Einige Notationen

Wir gehen im weiteren davon aus, daß das Gebiet $\langle \mathcal{C}, \mathcal{J} \rangle$, ohne daß es explizit erwähnt wird, den getroffenen Annahmen genügt. Den Hom-Funktor $\hom_{\mathcal{C}}(-, t)$ kürzen wir mit h^t ab, die Kategorie der Prä-Garben über der Kategorie $\mathcal{C}$ bezeichnen wir mit $\mathsf{PreShv}(\mathcal{C})$ und die Kategorie der Garben mit $\mathsf{Shv}(\mathcal{C})$. Wenn wir von einer Überdeckung als Familie von Pfeilen $(a_i \longrightarrow a)_{i \in I}$ sprechen, so soll der Index über die Pfeile variieren. Wenn es zwei Pfeile $p \rightrightarrows q$ zwischen zwei Objekten gibt, so notieren wir dies durch $p_i \longrightarrow q$, $p_j \longrightarrow q$ mit $i \neq j$ aber $p_i = p_j$.

12.2 Dekomposition des Musters

Wir nutzen die Garben-Eigenschaft aus, um Vorkommen via $\hom_{\mathcal{C}}(-, t)$ durch Problemreduktion zu berechnen, und gehen wie folgt vor:

(i) *Dekomposition:* Wählen einer Überdeckung $(p_i \longrightarrow p)$ des Musters.

(ii) *Löse die Teilprobleme:* Finden partielle Vorkommen $p_i \longrightarrow t$, d.h. Vorkommen von Elementen aus der Überdeckung des Musters

(iii) *Konkretisieren:* Verträgliche partielle Vorkommen werden zusammengefügt

(iv) *Rekursionsabbruch:* Atomare, nicht zerlegbare Teile p_i des Musters p werden im Suchraum t identifiziert

Das Transitivitätsaxiom vermeidet Rekursion. Wir wählen die feinste Dekomposition als Überdeckung. Das Konkretisieren im dritten Schritt ist die konstruktive Version der Bijektion, die durch die Garben-Eigenschaft zugesichert wird. Dieser Isomorphismus notiert sich formal durch

$$\mathsf{Nat}(P, h^t) \quad = \quad h^t(p) \quad = \quad \hom_{\mathcal{C}}(p, t).$$

P bezeichnet die (gewählte) Überdeckung des Musters. Da wir Auftreten als Pfeile mit einer geeigneten Topologie modelliert haben, so daß Garben eineindeutig auf Hom-Funktoren erweiterbar sind, erhalten wir die Konkretisierung durch Definition einer strikt epimorphen Familie. Der eineindeutige und existierende Konkretisierungspfeil in dieser Definition wird lediglich durch die Datenstruktur (Gebiet) bestimmt.

12.3 Dekomposition des Suchraumes

Das Problem wurde darauf reduziert, Teile des Musters zu suchen. Das verbleibende Problem, diese Stücke zu finden, lösen wir durch Dekomposition des Suchraum-Objektes. Dabei müssen wir jedoch beachten, daß die Zerlegung der Objekte miteinander verträglich ist. Es besteht die Möglichkeit, daß Vorkommen von Teilen aus der Musterüberdeckung mehrere Teile der Dekomposition (nicht vollständig) überlappt, und deshalb könnten partielle Vorkommen übersehen werden. Formal zerlegen wir den Garben-Funktor $\hom_{\mathcal{C}}(-, t)$ (Problemreduktion), da der Funktor

$$\mathsf{epi}_{\langle \mathcal{C}, \mathcal{J} \rangle} : \mathcal{C} \xrightarrow{\mathsf{y}} \mathsf{PreShv}(\mathcal{C}) \xrightarrow{\mathsf{sh}} \mathsf{Shv}\langle \mathcal{C}, \mathcal{J} \rangle$$

Überdeckungen auf epimorphe Familien schickt. y ist die Yoneda Einbettung und sh assoziiert eine Topologie (inklusive Garben-Eigenschaft). Eine Dekomposition des Suchraum-Objektes induziert eine Dekomposition der korrespondierenden Garben(-Funktoren). Diesen Sachverhalt verwenden wir für die algorithmische Darstellung:

(i) *Dekomposition:* Wählen einer verträglichen Überdeckung $(t_i \longrightarrow t)_{i \in I}$ des abzusuchenden Objektes

(ii) *Lösen der Sub-Probleme:* Konstruktion der Garben-Funktoren $\hom_{\mathcal{C}}(-, t_j)$ für jeden Teil t_j.

(iii) *Konkretisierung:* Kombinieren der Garben-Funktoren zu einem Garbenfunktor $\hom_{\mathcal{C}}(-, t)$.

(iv) *Rekursionsabbruch* bei atomaren, nicht zerlegbaren Teilen t_i via Identifikation (auf $\mathcal{C}$).

Die Konkretisierung des Garben-Funktors kann als ein inverser Limes

$$\hom_{\mathcal{C}}(-, t) \quad = \quad \mathsf{co\text{-}lim}\ \hom_{\mathcal{C}}(-, t_i)$$

in der Kategorie der Garben $\mathsf{Shv}\langle\mathcal{C},\mathcal{J}\rangle$ interpretiert werden. Da der Funktor sh eine Linksadjungierte (zum Vergiß-Funktor u) ist, d.h. für jeden Pfeil $f : a \longrightarrow u(b)$ existiert ein eineindeutig bestimmter Pfeil $\tilde{f} : sh(a) \longrightarrow b$, so daß folgendes Diagramm kommutiert,

$$\begin{tikzcd}
a \arrow[r, "\mathsf{sh} \circ u"] \arrow[dr, "f"'] & u(\mathsf{sh}(a)) \arrow[d, dotted, "u(f^{\#})"] \\
& b
\end{tikzcd}$$

können wir den obigen Limes als

$$\hom_{\mathcal{C}}(-,t) \quad = \quad \mathsf{sh} \circ \mathsf{co\text{-}lim}\ \hom_{\mathcal{C}}(-,t_i)$$

schreiben. Der Limes kann nun in der Kategorie der Prä-Garben $\mathsf{PreShv}(\mathcal{C})$ (punktweise) berechnet werden.

12.4 Konstruktion von Vorkommen

Kombiniert man die beiden Dekompositionen, die des Musters und die des Suchraumes, so muß nur noch aus den vielen partiellen Vorkommen in Teilen des Suchraum-Objektes $\hom_{\mathcal{C}}(p_i,t_j)$ der Wert $\hom_{\mathcal{C}}(p,t)$ rekonstruiert werden. Dazu berechnen wir die Mengen $\hom_{\mathcal{C}}(p_i,t)$, die Vorkommen von partiellen Mustern und vervollständigen diese dann via der Garben-Eigenschaft zu Vorkommen $\hom_{\mathcal{C}}(p,t)$. Die Vorgehensweise läßt sich wieder in einzelne Schritte unterteilen.

(i) *Dekomposition:* Wähle die feinste Überdeckung $(p_i \longrightarrow p)_{i\in I}$ des Musters p und eine Überdeckung $(t_j \longrightarrow t)_{j\in J}$ des abzusuchenden Objektes, die aus Monomorphismen besteht.

(ii) *Teilen in kleinere Probleme:* Erkunde die partiellen Vorkommen $p_i \longrightarrow t_j$ in den Teilen des abzusuchenden Objektes.

(iii) *Zusammenfügen der Lösungen:*

   (a) Die Co-Domäne wird via eines inversen Limes in der Kategorie der Prä-Garben aus den partiell berechneten Funktoren $\hom_{\mathcal{C}}(-,t_j)$ ermittelt. Einen partiell berechneten Funktor $\hom_{\mathcal{C}}(-,t)$ erhält man durch Berechnen der Pfeile aus $\hom_{\mathcal{C}}(p_i,t)$, durch Komposition der partiellen Vorkommen in Teilen $p_i \longrightarrow t_j$ und den korrespondierenden Überdeckungspfeilen $t_j \longrightarrow t$.

   (b) Die Domäne wird vervollständigt, indem via der Garben-Eigenschaft die verträglichen partiellen Vorkommen zusammengefaßt werden.

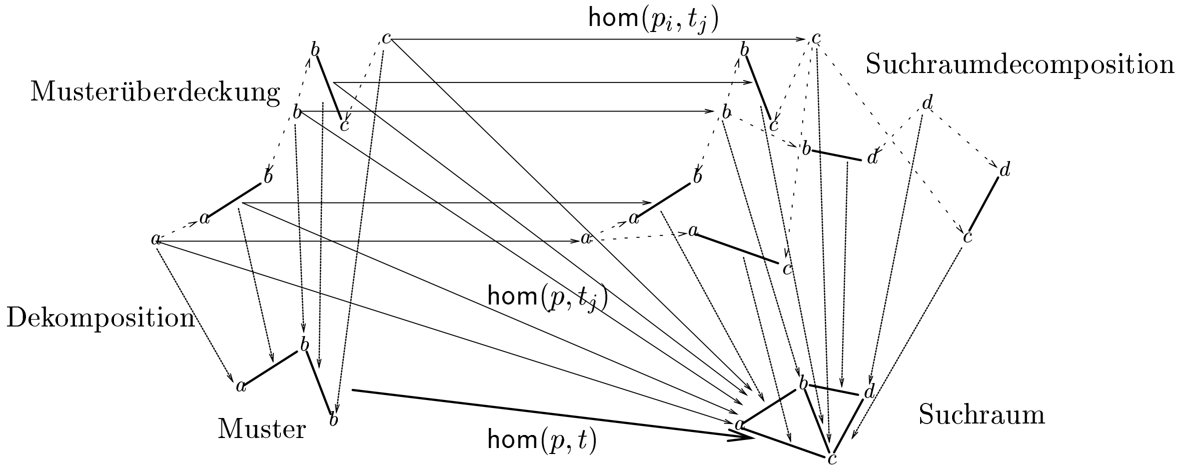


Abbildung 1: Konkretisieren von partiellen Vorkommen

Damit kennt man die Menge $\hom_{\mathcal{C}}(p,t)$, die alle Vorkommen des Musters modelliert.

(iv) Der *Rekursionsabbruch* besteht in der Identifikation von atomaren Vorkommen in Teilen des Suchraums ($p_i \longrightarrow t_j$).

Diese Strategie, Muster zu erkennen, ist die Formalisierung der in der Einleitung vorgestellten intuitiven geometrischen Vorgehensweise.

12.5 Aufzählen kompatibler Familien

In der oben beschriebenen Problemreduktion ist der Rekursionsabbruch, der gewöhnlicherweise darin besteht, elementare Vorkommen $p_i \longrightarrow t_j$ — in der kategoriellen Perspektive bedeutet das Identitätspfeile — zu erkennen, trivial. Der kompliziertere Teil besteht in dem Synthetisieren zu in einem gewissen Sinne größeren Vorkommen. Es gibt mehrere Möglichkeiten kompatible Familien zu berechnen. Garben sind natürlichen Transformationen. Das legt das kanonische Verfahren, systematisches Ausprobierenden oder erschöpfende Suche nahe. Wir verfolgen eine andere Variante und nutzen die enge Verwandschaft zu den inversen Limite, die sich in folgender Gleichung ausdrückt.

$$\mathsf{Nat}(P, h^t) \quad == \quad \mathsf{lim}_{\langle p_i \longrightarrow p \rangle \in \mathcal{C} \setminus P} \quad h^t(p_i),$$

P bezeichnet wieder die Überdeckung des Musters und $\mathcal{C} \setminus P$ die entsprechende Komma-Kategorie. Dieser Limes kann via Produkte und Equalizer berechnet werden.

Wir orientieren uns an einem schnellen, bereits existierenden Verfahren Muster zu erkennen, dem Algorithmus von Knuth, Morris und Pratt. Aus der bestehenden Topologie konstruieren wir eine Topologie, auf den Überdeckungen. Das ermöglicht es, die Musterüberdeckung in Stücke zu zerteilen und aus ihnen kompatible Familien zu konstruieren.

Kompatible Familien $\mathsf{Nat}(P, h^t)$ formen einen Funktor in P:

$$\begin{array}{rccc} \mathsf{Nat}(-, h^t): & \mathsf{PreShv}(\mathcal{C})^{\mathsf{op}} & \longrightarrow & \mathbf{Set} \\ & F & \longmapsto & \mathsf{Nat}(F, h^t) \\ & \tau \big\downarrow & & \big\uparrow {\_ \circ \tau} \\ & G & \longmapsto & \mathsf{Nat}(F, h^t) \end{array}$$

Im Falle einer Überdeckung durch das Muster selbst ist τ einfach ein Inklusionspfeil von einem Teil der Musterüberdeckung und $\_ \circ \tau$ ist die Restriktion einer Kompatiblen Familie auf dieses Stück.

Sei das Muster via einem Co-Limes

$$P \quad = \quad \mathsf{co\text{-}lim}_i P_i$$

dekompositioniert. Da der Funktor $\mathsf{Nat}(-, h^t)$ co-Limes mit Limes verträglich konvertiert, eine Eigenschaft aller kontravarianten Hom-Funktoren, gilt

$$\mathsf{Nat}(P, h^t) \quad = \quad \lim \mathsf{Nat}(P_x, h^t).$$

Wenn wir nun diese Topologie über den Überdeckungen geeignet wählen, ist $\mathsf{Nat}(-, h^t)$ ein Garben-Funktor.

12.6 Dekomposition von Überdeckungen

Wir definieren hier eine Topologie in der $\mathsf{Nat}(-, h^t)$ ein Garben-Funktor wird.

Definition 8 (Topologie auf elementaren Überdeckungen) *Seien*

$$F, G : \mathcal{C}^{op} \longrightarrow \boldsymbol{Set}$$

zwei Funktoren nach ***Set***. *Eine Inklusion* $F \dot{\longrightarrow} G$ *ist eine natürliche Transformation, wenn jede der Komponenten ein Inklusionspfeil ist, d.h., für jedes Objekt* c *in* $\mathcal{C}$ *gilt* $F(c) \subseteq G(c)$.

Betrachten wir die Kategorie der Elementaren Überdeckungen in $\mathcal{C}$ *mit der Sub-Funktor Pfeilen. Dann verstehen wir unter einer Überdeckung für* F *eine Familie* $(F_x \dot{\longrightarrow} F)$ *von Sub-Funktoren so, daß für alle Objekte* c *in* $\mathcal{C}$

$$\bigcup_x F(x) \quad = \quad F$$

gilt.

Die Zusammenfassung aller Sub-Funktoren einer elementaren Überdeckung S (oder Sub-Überdeckung S) formt eine partiell geordnete Menge. Die Vereinigung zweier Sub-Überdeckungen, notiert durch $X \cup Y$, ist die elementweise Vereinigung der Funktoren oder äquivalent die Vereinigung aller Pfeile. Der Schnitt wird analog punktweise definiert.

Definition 9 (Primpfeile) *Sei S eine elementare Überdeckung in der Kategorie $\mathcal{C}$. Wir bezeichnen einen Pfeil aus S als prim (in S) wenn er nicht durch andere Pfeile faktorisiert werden kann. Formal heißt das, daß es keinen Pfeil h in S und keinen Pfeil g in $\mathcal{C}$ gibt, die verknüpft $f (= h \circ g)$ ergeben.*

Analog ist ein Sub-Überdeckung $R \subseteq S$ prim in S, wenn es durch einen Primpfeil f aus S erzeugt wird. D.h. $R = \{f \circ g \mid g \text{ aus } \mathcal{C}\}$.

Die Intention die wir mit Primpfeilen verfolgen ist ein Lifting von Überdeckungen aus einem Gebiet auf die zugehörige Funktor-Kategorie. Sei beispielsweise $\langle \mathcal{C}, J \rangle$ ein Gebiet, p ein Muster und P seine feinste Überdeckung. Sei $(p_i \longrightarrow p)$ die Familie der Primpfeile aus P. Dann sind die durch die Primpfeile p_i generierten Primüberdeckungen P_i eine Überdeckung der elementaren Überdeckung P innerhalb der Funktor-Kategorie. Tatsächlich wird das die feinste Überdeckung sein.

Die folgende Definition eines Komplements einer elementaren Überdeckung wird hilfreich bei der Zerteilung von Überdeckungen sein.

Definition 10 (Komplement einer elementaren Überdeckung) *Sei S eine R subsumierende elementare Überdeckung ($R \subseteq S$), dann verstehen wir unter dem relativen Komplement von R (bzgl. S) die elementare Überdeckung, die durch $S \setminus R$ erzeugt wird. Wir bezeichnen es mit $S - R$. Wenn S eindeutig aus dem Kontext hervorgeht, dann schreiben wir auch nur $-R$.*

Man beachte, das der Schnitt zweier komplementärer elementaren Überdeckungen nicht notwendig leer sein muß. Komplementäre elementare Überdeckungen ergeben folgendes cokartesisches Diagramm

$$\begin{tikzcd} -R \cap R \arrow[r] \arrow[d] & R \arrow[d] \\ -R \arrow[r] & S \end{tikzcd}$$

Das Ziel ist das Liften von Überdeckungen in die Funktor-Kategorie um Dekompositionen der Musterüberdeckung zu erhalten. Mit folgender Definition gelangen wir zu Musterteilen.

Definition 11 (Sub-Muster) *Sei p das Muster und P eine feinste Überdeckung. Dann ist ein Sub-Muster eine Vereinigung von Primüberdeckungen aus P.*

Man beachte, da Sub-Muster über die Funktor-Kategorie definiert wurden, muß nicht notwendigerweise ein Objekt in dem Gebiet existieren, daß mit dem Sub-Muster korrespondiert.

12.7 Konstruktion kompatibler Familien

Wenn wir die oben definierte Topologie auf den Überdeckungen verwenden, können wir kompatible Familien $\mathsf{Nat}(P, h^t)$ durch einen Limes berechnen.

$$P \;=\; \mathsf{co\text{-}lim} P_x \quad\Longrightarrow\quad \mathsf{Nat}(P, h^t) \;=\; \lim \mathsf{Nat}(P_x, h^t)$$

Wenn die Dekomposition P in nur zwei Stücke zerlegt, läßt sich der Limes über ein gefasertes Produkt reduzieren. Die oben definierte Topologie garantiert die Existenz einer solchen zweielementigen Zerlegung und gefaserte Produkte können in **Set** leicht berechnet werden. Das gefaserte Produkt von $A \xrightarrow{f} C$ und $B \xrightarrow{g} C$ ist durch

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

mit den zwei entsprechenden Projektionen in A bzw. B beschrieben.

12.8 Binäre Dekomposition

In jedem Schritt zerteilen wir die Musterüberdeckung in zwei Teile, so daß wir einen binären Zerlegungsbaum erhalten. Die legt einen „teile und beherrsche"-Ansatz nahe:

(i) Dekomposition: Zerlege die Musterüberdeckung P in zwei Teile P_x und P_y, so daß ihr Schnitt P_{xy} zu folgendem gefasertem co-Produkt führt:

$$\begin{tikzcd} P_{xy} \arrow[r] \arrow[d] & P_y \arrow[d] \\ P_x \arrow[r] & P. \end{tikzcd}$$

(ii) Rekursion: Berechne kompatible Familien auf den Teilen $(\mathsf{Nat}(P_x, h^t))$ und die zu den Inklusionen $P_{xy} \longrightarrow P_x$, $P_{xy} \longrightarrow P_y$ korrespondierenden Restriktionen.

(iii) Synthese: Die kompatiblen Familien werden via dem folgenden kommutierenden Diagramm zusammengefügt.

$$\begin{tikzcd} \mathsf{Nat}(P_{xy}, h^t) & \mathsf{Nat}(P_y, h^t) \arrow[l] \\ \mathsf{Nat}(P_x, h^t) \arrow[u] & \mathsf{Nat}(P, h^t). \arrow[l] \arrow[u] \end{tikzcd}$$

(iv) Rekursionsabbruch: Für atomare, d.h. nicht weiter zerlegbare Teile der Überdeckung, kategoriell durch Primüberdeckungen P_i beschrieben, erhält man kompatible Familien aus $\hom_{\mathcal{C}}(p_i, t)$ und deren Restriktionen. Diese Werte können aus partiellen Vorkommen $\hom_{\mathcal{C}}(p_i, t_j)$ durch Zerlegen des abzusuchenden Objektes berechnet werden.

Da die Musterüberdeckung endlich ist und damit auch der Zerlegungsbaum, terminiert dieser rekursive Algorithmus. Wir haben einen nicht-inkrementellen Algorithmus der Top-Down verfährt erhalten. Diese Klasse von Algorithmen eignet sich besonders für die Implementierung auf parallelen Maschinen, da die Unabhängigkeit der Daten ausgenutzt werden kann. Es bieten sich insbesondere SIMD-artige Maschinen oder Datenflußmaschinen an. Der Schlüssel zu dieser Parallelität ist die Lokalität in der Definition der Garben, die sich auf die Lokalität der Operationen dieser parallelen Architekturen vererbt.

13 Die inkrementellen Version

Der binäre Teile und Beherrsche Algorithmus benötigt alle Vorkommen $p_i \longrightarrow t$ vorweg. In diesem Abschnitt wird ein Algorithmus entwickelt, der inkrementell Vorkommen während eines Traversierens durch den Suchraum berechnet. Diese Vorgehensweise orientiert sich an einer Verallgemeinerung des Algorithmus von Knuth, Morris und Pratt. Wir gehen davon aus, das während des Traversierens inkrementell kompatible Familien (unter Verwendung von Primsieben) bestimmt werden.

Betrachten wir dazu den Zusammenhang zwischen Limes und inversem Limes noch einmal:

$$P = \mathsf{co\text{-}lim} P_x \quad \Longrightarrow \quad \mathsf{Nat}(P, h^t) = \lim \mathsf{Nat}(P_x, h^t)$$

und wählen wieder eine binäre Dekomposition (Pushout). Um eine kompatible Familie von P aus kompatiblen Familien von P_x zu berechnen, benötigen wir die kompatiblen Familien des Komplements $-P_x$. Iteriert man diese Vorgehensweise, so benötigt man die Familien aller Primsiebkombinationen, da ein Fortschritt eine kompatible Familie per se von jedem Primsieb aus stattfinden kann. Das bedeutet, daß wir alle kompatiblen Familien auf Sub-Mustern benötigen.

Um diese Familien $\mathsf{Nat}(\Pi, h^t)$ auf allen Sub-Mustern Π mit der Fortschritten auf Primsieben zu aktualisieren, verfolgen wir die Technik Finite Differencing. Zur Vereinfachung der Darstellung führen wir die folgende Notation für Pushouts und Pullbacks ein.

Definition 12 (Pushouts, Pullbacks) *$\_ \oplus \_$ notiert ein Pushout und $\_ \otimes \_$ ein Pullback*[9]*. Das Resultat dieser Operationen ist ein Paar von Pfeilen und ein Objekt.*

[9]gefaserte Produkte und Coprodukte

Die Notation verbirgt das dritte Objekt und die verbleibenden Pfeile in dem kommutierenden Viereck. Diese Objekte gehen eindeutig aus dem Kontext, in dem die Notation verwendet werden soll, hervor. Das dritte Objekt wird immer ein Schnitt und die beiden Pfeile die entsprechenden Inklusionen sein.

Betrachten wir folgende Beziehung zwischen dem Garben-Funktor sh und Vorkommen in Suchraumteilen t_j

$$\hom_{\mathcal{C}}(-,t) = \mathsf{sh} \circ \mathsf{co\text{-}lim} \hom_{\mathcal{C}}(-,t_i)$$

in dem $(t_j \longrightarrow t)$ monomorphe Überdeckung des Suchraumes ist. Angenommen wir hätten schon die Garben für einen Teil t des Suchraumes berechnet und wir erkunden den Raum um ein Stück δt weiter, so daß wir das folgende Pushout erhalten:

$$\begin{tikzcd}
t \cap \delta t \arrow[r] \arrow[d] & t \arrow[d] \\
\delta t \arrow[r] & t \oplus \delta t
\end{tikzcd}$$

Der neu erkundete Teil δt des Suchraumes führt zu Neuentdeckungen von elementaren partiellen Vorkommen der Form $\hom_{\mathcal{C}}(p_i, \delta t)$ von einigen Teilen p_i des Musters. Wendet man obige Gleichung auf den Pushout an, so erhält man

$$\hom_{\mathcal{C}}(-, t \oplus \delta_t) = \mathsf{sh} \circ (\hom_{\mathcal{C}}(-,t) \oplus \hom_{\mathcal{C}}(-,\delta t)).$$

Das Standardresultat über Garben-Funktoren beinhaltet ein inversen Limes über Überdeckungen. Da wir nicht an dem vollständigen Garben-Funktor interessiert sind nützen wir den Isomorphismus , der durch die Garben-Eigenschaft gesichert ist, aus.

$$\hom_{\mathcal{C}}(-, t \oplus \delta_t)(p) \cong \mathsf{Nat}(P, h^t \oplus h^{\delta t})$$

p ist wieder das Muster und P die Musterüberdeckung. Das Problem ist auf die Aktualisierung kompatibler Familien $\mathsf{Nat}(P, h^t)$ (wenn sich h^t ändert) reduziert. $h^t \oplus h^{\delta t}$ ist ein inverser Limes von Funktoren, die punktweise definiert sind. Damit sind Werte von $h^t \oplus h^{\delta t}$ durch mengentheoretische Vereinigung gegeben, und wir können das Coprodukt distributiv berechnen.

Der Garben-Funktor $\mathsf{Nat}(-, h^t)$ kann induktiv berechnet werden:

(i) Induktionsabbruch: Bei Primsieben P_i leitet sich $\mathsf{Nat}(P_i, h^t)$ von den Werten $h^t(p_i)$ und den entsprechenden Restriktionen ab.

(ii) Induktion: Sub-Muster Π, die nicht prim sind, werden durch den Pushout $\Pi_x \oplus \Pi_y$ dekomponiert. Damit gilt

$$\mathsf{Nat}(\Pi, h^t) = \mathsf{Nat}(\Pi_x, h^t) \oplus \mathsf{Nat}(\Pi_y, h^t)$$

Ein Fortschritt $h^{\delta t}$ läßt sich in obiger Definition propagieren, was zum folgender Methode führt:

(i) Induktionsabbruch: Für einen Primsieb P_i leiten wir $\mathsf{Nat}(\Pi_x, h^t \oplus h^t)$ aus den Werten $(h^t \oplus h^{\delta t})(p_i)$ ab.

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

(ii) Induktion: Für Sub-Muster Π, die als Pushout $\pi_x \oplus \Pi_y$ dargestellt werden können, gilt

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

Der gesammte Garben-Funktor $\mathsf{Nat}(-, h^t)$ wir so nach und nach aktualisiert, da man für jedes Musterteil p_i neue elementare Vorkommen und damit neue kompatible Familien für alle Primsiebe P_i erhält. Die binären Dekompositionen sorgen dafür, daß kein partielles Vorkommen übersehen werden kann.

Die Garben-Funktoren breiten sich in zwei Dimensionen aus, die eine sind die Fortschritte, die andere die Musterteile. Durch das Distributivgesetz wird die Tatsache ausgedrückt, das diese beiden Dimensionen im wesentlichen unabhängig (voneinander) sind.

13.1 Binäre Dekomposition

Die Darstellung der Menge $\mathsf{Nat}(\Pi, h^t \oplus h^{\delta t})$ kann durch ein systematisches Aktualisieren (eine Bottom-Up Vorgehensweise) vereinfacht werden. Wir erhalten einen Algorithmus, der einen Fortschritt auf einem Primsieb P_i erzielt, indem die Musterüberdekung in P_i und $-P_i$ zerlegt wird, und die entsprechende kompatible Familie via einem Pullback berechnet wird. Bei jedem Fortschritt müssen auch die eventuell betroffenen partiellen Familien aktualisiert werden.

(i) Initialisierung: Wir beginnen mit leeren partiellen Vorkommen.

(ii) Aktualisierung: Wir berechnen einen Fortschritt via dem Primsieb P_i:

(a) Erweitere $\mathsf{Nat}(P_i, h^t)$ mit
$$\mathsf{Nat}(P_i, h^t \oplus h^t) = \mathsf{Nat}(P_i, h^t) \cup \mathsf{Nat}(P_i, h^{\delta t})$$

(b) Für alle Sub-Muster in Π die durch P_i betroffen sind bestimme
$$\mathsf{Nat}(\Pi, h^t \oplus h^{\delta t}) = \mathsf{Nat}(\Pi, h^t) \cup (\mathsf{Nat}(-P_i, h^t) \otimes \mathsf{Nat}(P_i, h^{\delta t}))$$

Die Illustration zeigt einen solchen Fortschritt auf dem Gebiet der markierten Graphen.

13.2 Verbesserung der inkrementellen Version

In der bis jetzt beschrieben Vorgehensweise werden alle partiellen Vorkommen zum frühest möglichen Zeitpunkt erkannt und gespeichert. Das scheint nicht effizient. Für Strings beispielsweise bedeutet das, das mit einem Muster der Länge $|p|$ und einem Text der Länge $|t|$ alle möglichen und unmöglichen partiellen Vorkommen für jede Stelle im Text berechnet werden. Das führt zu einer lächerlichen Komplexität von $\mathcal{O}(2^{|p|} \cdot |t|)$. Um diesen Algorithmus zu verbessern, müssen die Daten der partiellen Vorkommen auf das notwendige reduziert werden.

Eine Methode die Größe der Daten Einhalt zu gebieten besteht im Entfernen bzw. Verwerfen aller derjenigen partiellen Vorkommen, für die keine Chance besteht, daß sie Erweitert werden könnten. Für Strings beispielsweise heißt daß, das bei einem von links nach rechts Vorgehen, daß höchstens $|p| - 1$ partielle Vorkommen zum Muster erweiterbar wären, was die Komplexität auf die des kanonischen Verfahrens reduziert ($\mathcal{O}(|l| \cdot |p|)$). Diese Optimierung hängt, wie an diesem Beispiel bereits erkennbar, von der Traversierungsmethode bzw. dem unterliegenden Gebiet stark ab und wird deshalb in diesem allgemeinen Rahmen nicht weiter verfolgt.

Ein weitere Möglichkeit das Datenaufkommens zu reduzieren besteht in der näheren Untersuchung der Abhängigkeiten zwischen den partiellen Vorkommen selbst. Wenn zum Beispiel ein Vorkommen von anderen hergeleitet werden kann, dann benötigt man die Teile aus denen die Herleitung besteht (für den Moment) nicht mehr. In der Darstellung des Algorithmus wird das Speichern der partiellen Vorkommen durch ein Neu-Generieren ersetzt.

Die Abhängigkeiten treten dann auf, wenn ein Teil des Musters im Muster Vorkommt. Sei beispielsweise $p_i \longrightarrow t$ ein partielles Vorkommen und existiere ein Pfeil $p_j \longrightarrow p_i$ zwischen zwei Musterteilen, so können wir durch Komposition ein partielles Vorkommen $p_j \longrightarrow p_i \longrightarrow t$ ableiten. Diese Subsumption induziert eine partielle Ordnung auf Vorkommen. Verwenden wir diese Ordnung indem wir die partiellen Vorkommen durch die maximalen Elemente beschreiben, so reduziert sich das Datenaufkommen erheblich. Nicht gespeicherte Vorkommen können jederzeit durch Komposition berechnet werden.

13.3 Erzeugen von atomaren Vorkommen

In diesem Abschnitt untersuchen wir den Rekursionsabbruch, d.h. die induktive Verankerung, genauer. Selbiger besteht im Berechnen der elementaren Vorkommen[10]. Wir gehen davon aus, das eine Prozedur existiert, die den Suchraum durchläuft und ihn en passon in Atome t_j zerlegt (feinste Überdeckung). Bei diesem Durchlaufen kann das gleiche Atom mehrfach vorkommen. In anderen Worten kann ein elementarer Teil mehrmals Vorkommen und damit kann eine atomare Zerlegung als ein Multivorkommen von elementaren Teilen betrachtet werden. Im allgemeinen werden diese verschiedenen gleichen elementaren Vorkommen nicht

[10] kategoriell betrachtet ist das die Bestimmung der Hom-Mengen $\hom_{\mathcal{C}}(p_i, t_j)$

viel miteinender gemeinsam haben und wir können keine weiteren Strukturaussagen machen.

Treffen wir die Annahme, daß für jedes atomares Multi-Vorkommen von Teilen des Musters $(p_x \longrightarrow t_j)_{x \in X}$ in einem Teil t_j des Suchraums auch ein partielles Vorkommen $p_i \longrightarrow t_j$ existiert, so daß alle Vorkommen aus dieser Menge durch Primelemente p_i faktorisiert werden $(p_x \longrightarrow p_i \longrightarrow t_j)$, dann sind Informationen, die nur aus dem Muster resultieren, zur Berechnung verwendbar. Mit dieser Eigenschaft können wir wie bei dem Algorithmus von Knuth, Morris und Pratt vorgehen und partielle Vorkommen des Musters in Teilen des Musters analysieren bzw. die Ergebnisse dieser Analyse wieder verwenden.

13.4 Muster-gegen-Muster Analyse

Das Erzeugen aller multiplen atomaren Vorkommen kann durch Verwenden spezieller multipler partieller Vorkommen vermieden werden. Ist beispielsweise $\mathcal{F}$ eine kompatibel Familie von einem Sub-Muster Π, so können kompatible Familien aus Teilen von $\mathcal{F}$ berechnet werden. Diese Familien nennen wir herleitbar oder subsumiert. Wie die Topologie des Gebietes eine auf den Überdeckungen induziert, so kann auch die Derivationsrelation auf die Topologie der Sub-Muster und partiellen Vorkommen geliftet werden. Diese Relation formt einen Funktor, den Muster-gegen-Muster-Funktor. Er ist eine Sammlung von Garben-Funktoren (einen für jedes Sub-Muster) und ist wie folgt definiert.

Definition 13 (Subsumption) *Seien Π und Π' zwei Sub-Muster, so ist ein Subsumptions-Pfeil $\Pi \longrightarrow \Pi'$ eine natürliche Transformation, die man als eine kompatible Familie von Vorkommen von Elementen aus Π in Π' erhält.*

Ein Sub-Muster Π subsumiert ein Submuster Π', notiert durch $\Pi' \preceq \Pi$, wenn ein Subsumptions-Pfeil $\Pi \longrightarrow \Pi'$ existiert.

Diese Relation ist auf Vorkommen erweiterbar.

Definition 14 (Subsumption, allg.) *Ein durch die Komposition $\Pi' \longrightarrow \Pi \longrightarrow h^t$ beschriebenes partielles Vorkommen wird durch das partielle Vorkommen $\Pi \longrightarrow h^t$ subsumiert.*

Im weiteren bezeichnen wir mit $\mathsf{Sub}(P)$ die Kategorie aller Sub-Funktoren eines überdeckenden Siebes P. Diese Kategorie enthält unter anderem auch alle natürlichen Transformationen.

Definition 15 (Muster-Funktor) *Für ein Sub-Muster Π eine Musterüberdeckung P ist der Funktor $\mathcal{S} : \mathsf{Sub}(P)^{op} \longrightarrow \boldsymbol{Set}$ als Subsumptions-Garbe durch*

$$\begin{array}{rcl} \Pi' & \longmapsto & \{\Pi' \xrightarrow{s} \Pi \mid s \text{ ist Subsumptionspfeil } \} \\ \Pi'' \xrightarrow{f} \Pi' & \longmapsto & \varphi^{\Pi}(\Pi') \xrightarrow{f^*} \varphi^{\Pi}(\Pi''), \text{ mit } f^* = \_ \circ f \end{array}$$

definiert.

Bezeichne J_P die auf $\mathsf{Sub}(P)$ restringierte Topologie der Siebe. Der Muster-Funktor für eine Musterüberdeckung P ist dann der durch

$$\begin{array}{rcl} \Pi & \longmapsto & \varphi^{\Pi} \\ \Pi \xrightarrow{f} \Pi' & \longmapsto & \varphi^{\Pi}(\Pi') \xrightarrow{\Theta(f)} \varphi^{\Pi}(\Pi) \ \textit{(komponentenweise)} \end{array}$$

definierte Funktor $\Theta : \mathsf{Sub}(P) \longrightarrow \mathsf{Shv}\langle \mathsf{Sub}(P), J_P\rangle$.

Dieser Funktor erlaubt es alle subsumierten Vorkommen in einem Vorkommen zu erzeugen, wodurch die angestrebte Reduktion des Datenaufkommens ermöglicht wird.

13.5 Berechnung des Muster-Funktors

Der Muster-Funktor oder besser Muster-gegen-Muster-Funktor beinhaltet alle Information über Vorkommen von Sub-Mustern in einem Muster. Unser ursprüngliches Problem, ein Muster in einem Raum zu finden, wurde in das Problem multiple Muster in multiplen Räumen zu finden transformiert.

Die Struktur dieses Problems legt einen Teile und Beherrsche Ansatz nahe. Jeder Subsumptions-Funktor φ^{Π} kann aus kleineren via der Garben-Eigenschaft rekonstruiert werden — und das via der selben Methode: Domänen-Dekomposition und sukzessives bilden von inversen Limite (co-Domäne Dekomposition und Konkretisierung).

(i) Rekursionsabbruch: Berechne den Subsumptions-Funktor für jeden Primsieb, indem die Werte von $\hom_{\mathcal{C}}(p_j, p_i)$ bestimmt und gebündelt werden.

(ii) Rekursion. Für alle Sub-Muster Π die nicht prim ein Primsieb sind wähle eine binäre Dekomposition $\Pi = \Pi_x \oplus \Pi_y$ (Pushout). Dann ist die Subsumption durch

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

gegeben.

Der Muster-Funktor kann aber auch inkrementell durch Traversieren des Musters berechnet werden. Bei dem Durchlaufen kann auch die Musterüberdeckung generiert werden.

(i) Initialisierung. Beginne mit dem leeren Muster-Funktor.

(ii) Aktualisierung. Sei p_i des Inkrement und P_i der korrespondierende Sieb.

   (a) Subsumptionsgarbe für P_i. Erweitere den Subsumptions-Funktor mit Werten aus $\hom_{\mathcal{C}}(p_j, p_i)$ und bündele die Information bezüglich allen schon bekannten Sub-Mustern Π.

(b) Generiere neue Domänen. Erweitere $\mathsf{Sub}(P)$ mit P_i und Π für jedes bekannte Sub-Muster. Berechne entsprechende Subsumptionen und bündele.

(c) Generiere neue co-Domänen. Übertrage die neu berechneten Sub-Muster in die Domänen-Kategorie des Muster-Funktors. Jedes neue Sub-Muster erfordert die Berechnung der Subsumptionen via einem inversen Limes auf den Garben einer Überdeckung.

In vielen Fällen sind weitere Optimierungen möglich. Beispielsweise könnte man durch rekursives Anwenden dieses inkrementellen Verfahrens den Muster-Funktor eventuell effizienter berechnen — ein Effekt der ua. auch beim Knuth, Morris und Pratt Algorithmus ausgenützt wird.

13.6 Ein Algorithmus mit Subsumption

In den vorgehenden Abschnitten haben wir die Berechnung der Beziehungen zwischen partiellen Vorkommen betrachtet. Wir verwenden jetzt diese Methoden um das Datenaufkommen im inkrementellen Algorithmus während des Erkennungsvorganges zu vermindern. Die Subsumption ist eine partielle Ordnung zwischen Vorkommen, zumindest, wenn man den Quotienten unter gegenseitigem Vorkommen bildet.

Wie schon erwähnt kann dann sämtliches notwendiges Wissen aus maximalen Elementen (Vorkommen) durch Entfernen aller subsumierten Vorkommen bestimmt werden. Dazu erweitern wir den früheren inkrementellen Algorithmus so, daß die entsprechende Information zur Verfügung steht, um Subsumptionen auszunutzen. Die Simulation wird durch das folgende Diagramm veranschaulicht.

$$\begin{tikzcd}
\mathsf{Nat}(-, h^t) \arrow[r] \arrow[d, "update"'] & R(-, h^t) \arrow[d, "??"] \\
\mathsf{Nat}(-, h^t \oplus h^{\delta t}) \arrow[r] & R(-, h^t \oplus h^{\delta t})
\end{tikzcd}$$

Bei der Berechnung der Funktion Update auf der modifizierten Datenhaltung nutzen wir die Tatsache, daß die Erweiterung eines Vorkommens die Subsumption erhält.

$$(\pi \preceq \pi' \ \wedge \ \Phi \preceq \Phi') \implies (\pi \otimes \Phi \preceq \pi' \otimes \Phi'),$$

wobei π, π', Φ, Φ' partielle Vorkommen sind und die Erweiterung auf der rechten Seite existiert.

Bei einem elementarem Vorkommen gehen wir also wie folgt vor: versuche jedes partielle Vorkommen im Speicher (maximal) zu erweitern. Falls eines dieser

Vorkommen nicht erweitert werden kann, versuche rekursiv den unmittelbaren Subsumenten zu expandieren und so fort. Diese Vorgehensweise garantiert, daß alle partiellen Vorkommen, die die frühere Version des Algorithmus erzeugt hat, auch jetzt noch erkannt werden.[11]

13.7 Inkrementelle Version mit Subsumption

Integrieren wir noch die einzelnen Vorgehensweisen zu einem Algorithmenskelett, das die benötigten Informationen aus maximalen Elementen berechnen kann, so erhalten wir einen guten inkrementellen Algorithmus. Die modifizierte Funktion *update* erhält man unmittelbar durch Speichern von einem hinreichend großem Datenaufkommen und Subsumption.

(i) Initialisierung. Beginne mit einem leeren Speicher (keine partiellen Vorkommen).

(ii) Aktualisierung. Sei der Fortschritt t_j. Für jedes partielle Vorkommen π, d.h. für jede kompatible Familie eines Sub-Musters Π, das gespeichert ist, führe die beiden folgenden Schritte durch:

   (a) Expandiere. Für alle elementaren Vorkommen v (via dem Sieb P_i), die durch den Fortschritt entstanden sind teste ob v kompatibel mit π ist. D.h. teste ob die Restriktionen von v und π auf $\Pi \cap P_i$ identisch sind. Falls ja, füge das erweiterte Vorkommen $\pi \otimes v$ zu den partiellen Vorkommen Π zu und fahre mit dem nächsten π fort. Ansonsten setze zurück (Backtracking).

   (b) Ungenützte Fortschritte. Wenn der Fortschritt t_j zu keiner Expansion geführt hat, so speichere alle elementaren Vorkommen, die durch t_j erkannt werden.

Das Ersetzen von echte Vorkommen durch subsumierte kann man sich als eine Bewegung des Musters, oder Teile des Musters, über das abzusuchende Objekt vorstellen.

Dieser Algorithmus verallgemeinert präzise das Vorgehen beim Algorithmus von Knuth, Morris und Pratt auf topologische spezifizierte Datenstrukturen.

14 Instanziierungen

14.1 Das Patternmatching auf String

Die diesem Problem zugrunde liegende Kategorie **labeledString** wurde schon in einem früheren Beispiel definiert. Wir gehen davon aus, daß alle Strings endlich

[11] Um ein größtes Element in einer linearen Ordnung mit (maximalem Element) das einem Prädikat genügt zu finden, ist es ausreichend die Elemente ihrer Größe nach zu betrachten. Die Update-Prozedur verallgemeinert dieses Methode auf partielle Ordnungen mit maximalen Elementen.

sind und alle Pfeile Monomorphismen sind. Wir verwenden die selbe Topologie wie im Beispiel, in dem die Überdeckungen strikt epimorphe Familien sind. Damit sind alle Voraussetzungen für die Instanziierung des Algorithmus erfüllt.

Bei der Anpassung an die inkrementelle Version verwenden wir ein von-links-nach-rechts Lesen auf dem Suchbereich. Selbstverständlich sind auch andere Durchlaufstrategien denkbar, diese führt jedoch zum bekannten Algorithmus von Knuth, Morris und Pratt. Nach dieser Definition von Überdeckungen müssen alle Paare von Stringelementen des Suchbereichs enumeriert werden. Die links-nach-rechts Lesestrategie hat die Eigenschaft, daß wir nur die rechten Nachbarvorkommen, die die rechte Kante der aktuellen Position berühren, speichern müssen. Andere Vorkommen sind nicht potentiell erweiterbar. Diese Menge der partiellen Vorkommen sind alle die Vorkommen, die den Präfixen des Musters entsprechen. Das reduziert die Anzahl der potentiellen Vorkommen eines Musters p der Länge $|p$ von $2^{|p|}$ auf $|p$. Dieses inkrementelle Vorgehen (ohne Finite Differencing) ist der naive $\mathcal{O}(|t||p|)$-Algorithmus.

Als nächstes eliminieren wir alle die Vorkommen, die von anderen subsumiert werden. Für Strings ist die Subsumtionsrelation die Teilstringrelation. Ein Vorkommenpfeil $p \longrightarrow t$ subsumiert einen Pfeil $q \longrightarrow t$, falls q ein Faktor in p ist. Kombiniert mit der vorhergehenden Optimierung kann die Subsumptionrelation auf die Suffixrelation eingeschränkt werden.

Die Vorwegberechnung des Muster-zu-Muster Funktor ist hier einfach. Da wir nur Präfixe des Musters als Vorkommen speichern ordnet die Subsumptionsrelation jedem Präfix das Musters alle die Präfixe zu, die gleichzeitig Suffixe sind, zu. Konkret berechnet sich die Subsumptionsrelation S dann nach folgendem Distributivgesetz:

$$S(p \cdot i) \quad = \quad \{y \cdot i \mid y \in S(x),\ y \cdot i \in \mathrm{Pref}(p)\}.$$

Der Punkt notiert dabei die Konkatenation, p ist das Muster, x ein Teilmuster und i ist das Inkrement. Diese Berechnung läßt sich weiter optimieren indem nur das Vorkommen berechnet wird, das unmittelbar subsumiert.

Diese Methode ist im wesentlichen der Algorithmus von Knuth, Morris und Pratt. Wenn die Subsumptionsrelation berechnet wurde, läßt sich der inkrementelle Algorithmus so optimieren, daß genau diese partiellen Vorkommen entwickelt werden. Es gibt zu jeder Berechnungsphase immer nur ein (erweiterbares) aktives Vorkommen, da alle anderen von diesem Vorkommen subsumiert werden. Kann das aktuelle Vorkommen nicht erweitert werden, wird via Backtracking das nächstkleinere subsumierende Vorkommen betrachtet, und der Algorithmus fährt so fort bis der Suchraum vollständig erforscht ist oder ein Muster gefunden wurde.

15 Ausblick

15.1 Was noch zu tun bleibt

Zuallererst fehlt noch eine vollständige Programmierumgebung für dieses Framework. Diese Umgebung sollte einige wohlbekannte Datenstrukturen samt ihrer Topologien als Bibliothek besitzen. Leider ist das ein sehr ehrgeiziges und arbeitsintensives Unterfangen.

Als nicht weniger wichtiges Vorhaben erachte ich die Klassifikation bereits bekannter Algorithmen in dieses Schema. Die Klassifikation dieser Algorithmen führt sicherlich zu neuen Einsichten, wie die meine Doktorarbeit und die von Thomas Schöbel-Theuer illustriert[12].

Viele Erkennungs- und Klassifikationsalgorithmen sind Instanzen dieses Entwurfsparadigma. So zum Beispiel auch das Walz'sche Filtern. Auch interdisziplinäre Verbindungen wie beispielsweise eine Definition für ein Raum-Zeit Modell (Physik) oder die Verbindungen von Molekülen (Vanderwaals-Oberflächen) lassen sich mit der Methode formulieren und analysieren. Selbst in der Biologie ist Mustererkennung ein wichtiges Prinzip. So sind Genetiker an Garbeneigenschaften von genetischen Codes sicherlich interessiert, und diese könnten mit diesem Werkzeug erforscht werden.

Nicht zuletzt ist der abstrakte Softwareentwurf — das zusammenbauen von einzelnen Komponenten (Engineering) auch durch entsprechende Garben definier und damit auch kontrollierbar.

15.2 Was habe ich getan

Ich habe lediglich unter dem Gesichtspunkt Mustererkennung einige Entwurfsparadigmen aus der Algorithmik (Divide-and-Conquer, etc.) mit einigen Ideen des Softwareingineering (parametrischer Entwurf, etc.) in mathematischer und intuitiver Sprache formuliert. Es stellte sich heraus, daß vielen Methoden dem gleichen Entwurfsprinzip zugrunde liegen, was dazu verwendet werden kann, parametrisch, eine Vielfalt an neuen Algorithmen als Instanzen der allgemeinen Methode zu finden.

16 Danksagung und Schlußwort

Natürlich danke ich allen Lesern, die bis hier ausgeharrt haben und hoffe das einige dieses Gebiet weiter erforschen. Ich hoffe der Fehlerteufel und der Federteufel haben nicht allzuoft zugeschlagen.

To be continued ...

[12]als Übungsaufgabe empfehle ich dem interessierten Leser die Parsing-Garbe und die Subsumptionskonstruktion, die er verwendet, zu definieren