Masterarbeit
im Fach Informatik

Ein graphbasierter Formalismus zur Programmmanipulation

Bernhard Stadler

7. November 2012

Betreuer:
Dr.-Ing. Sven Apel

Fakultät für Informatik und Mathematik
Universität Passau
Innstraße 41, 94032 Passau

Zusammenfassung

In dieser Arbeit wird ein Formalismus zur implementierungs-, sprach- und repräsentationsunabhängigen Beschreibung von Programmmanipulationen wie beispielsweise Merge, Featurekomposition und Interaktionsanalyse entwickelt. Dieser soll insbesondere der anwendungsnahen und wiederverwendbaren Definition von Programmanipulationen dienen.

Programmmanipulationen werden auf repräsentationsunabhängige, deklarative Art mit Mitteln der Kategorientheorie beschrieben. Auf Grundlage einer Analyse des Text-To-Model-Frameworks EMFText werden Graphvarianten aus der algebraischen Graphtransformationstheorie zur Programmrepräsentation ausgewählt. Diese Graphvarianten werden anhand ihrer charakteristischen Merkmale in Graphfeatures aufgespalten. Auf Grundlage von Sketches, einer graphbasierten und kategorientheoretischen Art der algebraischen Spezifikation, wird eine Vorgehensweise für die formale Definition von Graphfeatures entwickelt. Diese Graphfeatures kann man auf flexible Weise zu sprachübergreifenden graphbasierten Programmrepräsentationen kombinieren. Als Anwendungen des Formalismus wurden der Merge und die Featurekomposition beschrieben.

Inhaltsverzeichnis

Notationsverzeichnis

Schreibweisen:

$\forall x$: $P(x)$ und $\exists x$: $P(x)$	Quantoren; erstrecken sich möglichst weit nach rechts.
gdw.	Genau dann wenn; logische Äquivalenz
$R : M \nrightarrow N$	Relation R von M nach N
$f : M \to N$	Funktion f von M nach N
$f : M \to N : m \mapsto t(m)$	Funktion f von M nach N mit $\forall m \in M$: $f(m) = t(m)$
$a \bullet b$	Komposition der Relationen/Funktionen/Pfeile a und b (von links nach rechts zu lesen)
$b \circ a$	Komposition der Relationen/Funktionen/Pfeile a und b (von rechts nach links zu lesen)
R^+	Transitive Hülle von R
R^*	Transitiv-reflexive Hülle von R
$R^{\equiv}$	Äquivalenzhülle von R
B^A	Menge der Funktionen/Graphhomomorphismen/Funktoren/... von A nach B für Mengen/Graphen/Kategorien/... A, B
$\mathbf{C}^0$	Klasse der Knoten/Objekte von Köcher $\mathbf{C}$
$\mathbf{C}^1$	Klasse der Pfeile von Köcher $\mathbf{C}$
$\mathbf{C}[A, B]$	Klasse der Pfeile von A nach B in Köcher $\mathbf{C}$
$comm(D)$	Diagramm $D : I \to \mathbf{C}$ kommutiert

Symbole für Klassen:

$\mathfrak{Set}$: Mengen; $\mathfrak{Graphs}$: Graphen;

Kategorien:

Symbol	**Objekte**	**Pfeile**	**Komposition**
Set	$\mathfrak{Set}$	Funktionen	Funktionskomposition
Graphs	$\mathfrak{Graphs}$	Graphhomomorphismen	Funktionskomposition
Logi	Aussagen	Implikation	Modus ponens
POSet	$\mathfrak{Po}$	Monotone Funktionen	Funktionskomposition

Kapitel 1

Einleitung

1.1. Motivation

In der Softwaretechnologie werden Programmtransformationen wie Merge [Buf95], Featurekomposition [BSR04] oder Refactoring [FB99] und Programmanalysen wie Typprüfung [Pie02] und Interaktionsanalyse [Ape+10] benutzt. Eng verknüpft mit diesem Bereich sind auch Methodologien wie modellgetriebene, aspektorientierte oder featureorientiere Softwareentwicklung und Frameworks wie das Modellierungs-Framework EMF [Ste+08] oder das Compiler-Framework LLVM [LA04]. Da Programmtransformationen und Programmanalysen gemeinsam haben, dass sie auf Programmen in einer Objektsprache arbeiten, wird hier beides als *Programmmanipulation* zusammengefasst. In dieser Arbeit werden die Grundlagen eines Formalismus zur Beschreibung von Programmmanipulationen gelegt. Eine solche Formalisierung kann mehreren Zwecke dienen:

- **Korrektheit von Implementierungen:** Ohne Formalisierung ist es schwierig, sich selbst oder andere von der korrekten Funktionsweise oder von anderen Eigenschaften einer Programmmanipulation bzw. ihrer Implementierung zu überzeugen. Deshalb kann man formale Spezifikationen benutzen, um sicherzustellen, dass eine Programmmanipulation korrekt implementiert ist.

- **Anwendungsnähe/Abstraktion:** Um die korrekte Implementierung einer Programmmanipulation sicherzustellen, kann man sie in einer imperativen oder funktionalen Programmiersprache implementieren, formal spezifizieren und anschließend mit einem Theorembeweiser die Korrektheit der Implementierung beweisen. Dies hat aber zur Folge, dass die Beschreibung der Programmmanipulation von Implementierungsdetails abhängt: Die Beschreibung abstrahiert nicht von der Implementierung der Programmmanipulation, und ebenso wenig von der Objektsprache, also der Sprache der manipulierten Programme. Außerdem ist sie von der gewählten Programmmrepräsentation abhängig. Dies hat zur Folge, dass in der Beschreibung Implementierungsdetails vorhanden sind, die sie komplexer machen und von der Kernidee der Programmmanipulation ablenken. Durch mehr Abstraktion würde die Beschreibung von solchen unwesentlichen Details befreit und somit anwendungsnäher.

- **Wiederverwendbarkeit:** Zudem kann man eine schwach abstrahierende Spezifikation einer Programmmanipulation nicht für andere Implementierungen, Objektsprachen oder Programmrepräsentationen wiederverwenden, da diese mit der Beschreibung verwoben sind. Man muss jede Programmmanipulation für jede Implementierung, Objektsprache und Programmrepräsentation einzeln spezifizieren und implementieren. Eine abstraktere Beschreibung hätte diese Nachteile nicht und könnte besser wiederverwendet werden.

Dass Formalisierung nicht nur der Korrektheit von Implementierungen, sondern auch der Abstraktion und Anwendungsnähe der Beschreibung und der Wiederverwendbarkeit von Beschreibung und Implementierung von Programmmanipulationen dienen kann, wurde bereits in einigen Forschungsansätzen gezeigt, die im Folgenden kurz erläutert werden.

Um eine Programmmanipulation unabhängig von der Implementierung zu beschreiben, kann man sie durch deklarative oder logische Programmierung implementieren. Beispielsweise kann man ein „Move Method"-Refactoring, das Methoden innerhalb eines Java-Programmes verschiebt, mithilfe von Constraint Solving deklarativ implementieren [ST09; SP11]. Die Implementierung des Refactoring besteht dadurch im Wesentlichen aus seiner Beschreibung, nämlich aus einer generierten logischen Formel, die mit einem Constraint-Solver gelöst wird. Der Vorteil hiervon ist, dass die Beschreibung weniger unwesentliche Details enthält als bei einer imperativen Implementierung und sich näher an der Anwendung, also an der jeweiligen Programmmanipulation, bewegt. Außerdem ist der verwendete Constraint-Solver generisch, was im Sinne der Wiederverwendbarkeit liegt. Eine solche deklarative, implementierungsunabhängige Beschreibung dient dieser Arbeit als Vorbild.

Allerdings ist auch in diesem Fall die Programmmanipulation nur für eine einzige Objektsprache beschrieben. Dadurch fließen Eigenheiten der Objektsprache in die Beschreibung ein, die von der Kernidee der Programmmanipulation ablenken und die Beschreibung verkomplizieren [SKP11]. Im Gegensatz dazu ist es beispielsweise für die Featurekomposition bekannt, dass man sie im Wesentlichen sprachunabhängig implementieren kann: FeatureHouse [AKL12] erreicht durch ein sprachübergreifendes formales Programmmodell, dass man neue Objektsprachen im Wesentlichen durch eine Beschreibung ihrer Grammatik integrieren kann. Dadurch ist in FeatureHouse die Featurekomposition an sich implementiert, und man kann diese Implementierung der Featurekomposition für verschiedene Objektsprachen wiederverwenden. Eine solche Unabhängigkeit von der Objektsprache wird auch in dieser Arbeit angestrebt.

Auch FeatureHouse und die oben genannten deklarativen Refactorings bauen auf einem festen formalen Programmmodell auf. In [Rut+09] und [Tae+10] findet sich hingegen eine repräsentationsunabhängige Beschreibung des Merge, also der Zusammenführung zweier unabhängiger Programmänderungen, auf Grundlage der Kategorientheorie [BW12]. Neben der Implementierungs- und Sprachunabhängigkeit wäre Unabhängigkeit von der Programmrepräsentation eine weitere Möglichkeit, Programmmanipulationen abstrakter und besser wiederverwendbar zu machen. Auch wäre es zu Forschungszwecken nützlich, dieselbe Programmmanipulation auf verschiedenen Programmrepräsentationen zu untersuchen. Durch unterschiedliche Repräsentationen können sich nämlich Eigenschaften wie die Genauigkeit oder die Laufzeit einer Programmmanipulation ändern (vgl. etwa [Leß12]).

Zum einen wird deshalb in dieser Arbeit eine Art der repräsentationsunabhängigen Beschreibung von Programmmanipulationen mit Mitteln der Kategorientheorie vorgeschlagen. Zum anderen wird besonderes Gewicht auf eine Vorgehensweise zur modularisierten Definition vom Programmrepräsentationen gelegt. Zur formalen Beschreibung hiervon werden Sketches verwendet, eine graphbasierte und kategorientheoretische Art der algebraischen Spezifikation [BW12]. Hierdurch kann man Programmrepräsentationen nach dem „Lego-Prinzip" zusammensetzen.

1.2. Ziele

Wie im vorherigen Abschnitt festgestellt wurde, kann man durch Formalisierung nicht nur Korrektheit und andere Eigenschaften sicherstellen, sondern auch eine wiederverwendbare und anwendungsnahe Beschreibung von Programmmanipulationen erreichen, und zwar durch Implementierungs-, Sprach- und Repräsentationsunabhängigkeit. Die Ideen der dort genannten Arbeiten werden von dieser Masterarbeit fortgeführt.

Es wird ein allgemeiner Formalismus konzipiert, mit dem man Programmmanipulationen beschreiben und untersuchen kann. Ziel des Formalismus ist es, Programmmanipulationen unabhängig von ihrer Implementierung, der Sprache der manipulierten Progamme und der benutzten Programmrepräsentation zu beschreiben.

Als Grundlage hierfür wird eine Auswahl von Programmmanipulationen untersucht. Dabei wird ein informelles Schema entwickelt, mit dem Programmmanipulationen auf hoher Abstraktionsebene anhand der Merkmale Programmrepräsentation, -eigenschaften, -beziehungen und Operationen charakterisiert werden. Es werden graphbasierte Ansätze untersucht, mit denen sich die statische Semantik[1] von Programmen auf einer konzeptuellen Ebene repräsentieren lässt. Eine Vorgehensweise zur modularisierten formalen Definition von sprachübergreifenden graphbasierten Programmrepräsentationen auf Grundlage von Sketches wird entwickelt. Programmeigenschaften und -Beziehungen werden mit kategorientheoretischen Mitteln formal beschrieben.

Eine Umsetzung des Formalismus in Software wird nicht angestrebt, da sie den Rahmen dieser Arbeit sprengen würde. Aus demselben Grund wird auch nur die Beschreibung, nicht aber die Ausführung der Programmmanipulationen untersucht.

1.3. Aufbau

Der Rest dieser Arbeit gliedert sich wie folgt:

Zunächst werden in Kapitel 2 einige Beispiele von Programmmanipulationen als Kandidaten für die Beschreibung mit dem zu entwickelnden Formalismus untersucht. Dabei wird außerdem ein informelles Schema zur Charakterisierung von Programmmanipulationen auf hoher Abstraktionsebene vorgeschlagen. Die Anwendbarkeit des Schemas wird anhand der vorgestellten Programmmanipulationen demonstiert.

In Kapitel 3 wird der Formalismus konzipiert. Zunächst wird ein bestehender Ansatz (EMFText/Ecore) zur Repräsentation der statischen Semantik von Programmen untersucht. Dieser dient als Richtschnur für die Konzeption des Formalismus. Anschließend werden bestehende graphbasierte Ansätze vorgestellt, die die so gewonnenen Anforderungen erfüllen. Außerdem werden einige Grundbegriffe der Kategorientheorie und ihre Anwendung zur Beschreibung von Programmmanipulationen informell erläutert.

Ausgearbeitet wird der Formalismus in Kapitel 4. Besonderes Gewicht liegt dabei auf der modularen Definition von graphbasierten Programmrepräsentationen durch Sketches. Für einige der in Kapitel 3 vorgestellten Ansätze werden modulare Definitionen vorgeschlagen.

Wie der Formalismus angewendet werden kann, wird anhand von Merge und Featurekomposition in Kapitel 5 illustriert.

[1] Statische Semantik: siehe Einleitung Kapitel 3

Kapitel 6 fasst die Ergebnisse der Arbeit zusammen und abschließend werden mögliche auf der hier gelegten formalen Grundlage aufbauende Arbeiten vorgeschlagen.

Kapitel 2

Programmmanipulationen

Das Hauptziel dieser Arbeit ist die formale Beschreibung von Programmmanipulationen. Deshalb werden in diesem Kapitel einige Programmmanipulationen untersucht, die als Kandidaten für die Beschreibung mit dem Formalismus ausgewählt wurden.

Um einen übergreifenden Formalismus für verschiedene Programmmanipulationen zu schaffen, wurde eine gemeinsame Grundstruktur der verschiedenen Programmmanipulationen gesucht. Es wurden vier Merkmale gefunden, anhand derer sich Programmmanipulationen charakterisieren lassen.

Programmrepräsentation (R) Um Programme algorithmisch manipulieren zu können, ist eine Programmrepräsentation notwendig. Beispielsweise kann man Programme als eine Folge von Textzeilen oder als Baum repräsentieren. Ein Ziel dieser Arbeit ist es, die Beschreibung von Programmmanipulationen repräsentationsunabhängig zu machen, so dass die Programmrepräsentation gegen eine andere austauschbar ist.

Programmeigenschaften (E) und -beziehungen (B) Programmtransformationen erzeugen aus bestehenden Programmen neue Programme, die bestimmte Eigenschaften besitzen oder in bestimmten Beziehungen zu den bestehenden Programmen stehen. Programmanalysen berechnen Eigenschaften oder Beziehungen von bestehenden Programmen. In diesem Sinn liegt jeder Programmmanipulation mindestens eine Programmeigenschaft oder Programmbeziehung zugrunde, wobei nicht beides gleichzeitig nötig ist. Der Unterschied zwischen Eigenschaften und Beziehungen besteht darin, dass eine Eigenschaft sich auf ein einzelnes Programm, eine Beziehung sich hingegen auf mehrere Programme bezieht. Beispielsweise ist die Featurekomposition (siehe Abschnitt 2.3) eine Programmtransformation, die eine bestehende Variante eines Softwareprodukts zu einer neuen Produktvariante verfeinert, die ein zusätzliches Feature enthält. Das heißt, dass der Featurekomposition die Eigenschaft „Produktvariante X enthält Feature F“ und die Beziehung „Produktvariante X verfeinert Produktvariante Y“ zugrundeliegen. Diff ist eine Programmanalyse, die eine Ableitungsbeziehung zwischen zwei Programmen berechnet (siehe Abschnitt 2.1).

(O) Operationen Eine Programmtransformation erzeugt neue Programme mit bestimmten Eigenschaften oder Beziehungen, und eine Programmanalyse berechnet Eigenschaften oder Beziehungen von Programmen. Das eigentliche Erzeugen neuer Programme bzw. Berechnen von Eigenschaften, also die Ausführung einer Programmmanipulation, wird hier als Operation bezeichnet.

Im Folgenden werden die Programmmanipulationen eingeführt und es wird gezeigt, dass sie sich anhand dieser Merkmale charakterisieren lassen.

2.1. Die Werkzeuge `diff` und `patch`

Eine gängige Programmmanipulation ist die Kombination der Unix-Werkzeuge `diff` [MM85] und `patch`. Von `diff` werden die Unterschiede zwischen den zwei Versionen eines Programms berechnet, also eine Beziehung zwischen zwei Programmen. Genauer gesagt berechnet es eine minimale Menge von Änderungen, also von Textzeilen, die gelöscht und eingefügt werden müssen um die zweite Version zu erhalten. Das Ergebnis nennt man Diff- oder Patch-Datei. In der umgekehrten Richtung kann man mit der Programmtransformation `patch` aus einer der beiden Versionen und der Diff-Datei die jeweils andere Version berechnen.

```
class MyClassB {
  final void bar() {
    println("foo");
  }}
```

```
class MyClassB {
  final void bar() {
    println("bar");
  }}
```

```
@@ -3 +3 @@
-     println("foo");
+     println("bar");
```

Abbildung 1: Beispiel zu `diff`: Oben Version 1 (links) und Version 2 (rechts) eines Java-Programms, unten die zugehörige Diff-Datei im "Kontext-Ausgabeformat"

Beispiel: Diff

Im Beispiel aus Abb 1 ist das Ergebnis der Ausführung von `diff` auf zwei Versionen eines (Java-)Programms zu sehen: Die dritte Zeile wird von `println("foo")` in Version 1 zu `println("bar")` in Version 2 geändert.

Einordnung

Tatsächlich lassen sich `diff` und `patch` anhand der vier Merkmale charakterisieren:

(R) Das Programm wird als eine Folge von Textzeilen repräsentiert.

(E) (nicht vorhanden)

(B) Eine Änderungsbeziehung zwischen zwei Programmen, repräsentiert durch eine Diff-Datei.

(O) `diff` berechnet eine Änderungsbeziehung zwischen zwei gegebenen Programmen. `patch` berechnet ein Programm, das mit einem gegebenen Eingabeprogramm in einer gegebenen Änderungsbeziehung steht.

2.2. Merge

Durch einen Merge (engl. three-way merge) werden zwei Änderungen eines Programmes zu einer konsolidierten Änderung zusammengefasst, sofern sich die Änderungen nicht widersprechen. Im einfachsten Fall ist der Merge ähnlich wie `diff` zeilenbasiert. Einen solchen

zeilenbasierten Merge bezeichnet man als lexikalischen Merge. Eine Implementation des lexikalischen Merge liefert das Unix-Tool `diff3`. Für genauere Informationen zu `diff3` sei auf [KKP07] verwiesen. Formale Behandlungen des Merge findet sich in [Lyn10], [Jac09], [Rut+09], [Tae+10] und [Tae+12].

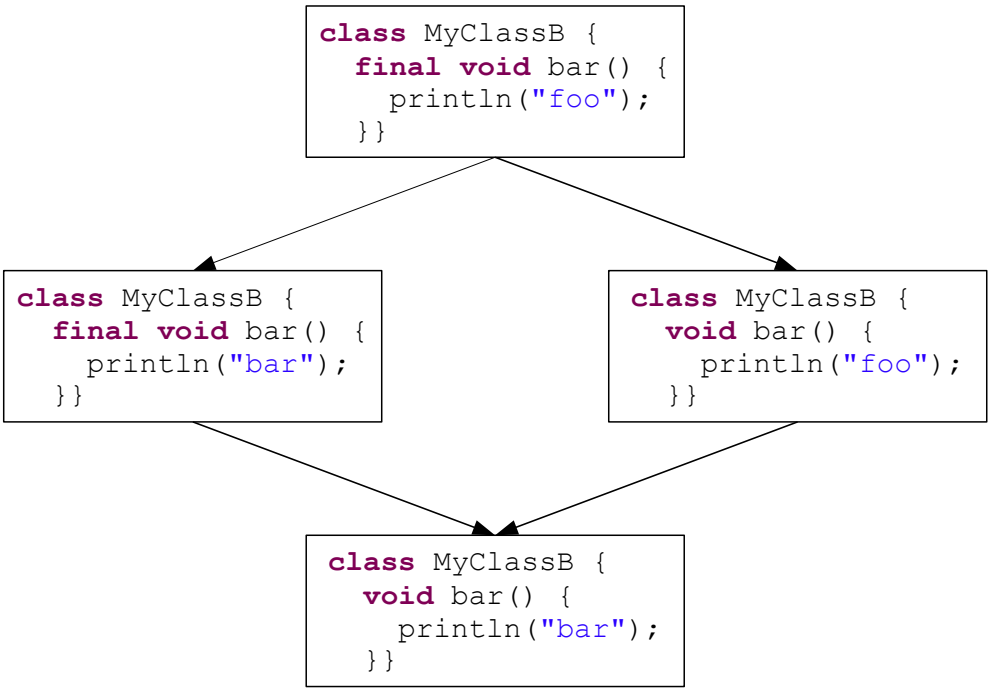


Abbildung 2: Merge

Beispiel: Merge

Abbildung 2 illustriert einen Merge: Gegeben sind die beiden Programmversionen aus dem vorherigen Beispiel und eine weitere abgeänderte Version, die den `final`-Modifier in der zweiten Zeile entfernt. Indem man die Änderungen zeilenweise zusammenführt, ergibt sich eine konsolidierte Version, die tatsächlich beide Änderungen enthält und auch die Absicht hinter den beiden Änderungen wiedergibt.

Einordnung: Lexikalischer Merge

Ebenso wie `diff` und `patch` lässt sich `diff3` anhand der vier Merkmale charakterisieren:

(R) Das Programm wird als eine Folge von Textzeilen repräsentiert.

(E) (nicht vorhanden)

(B) Es werden Änderungsbeziehungen zwischen der Grundversion und den geänderten Versionen bzw. der konsolidierten Version behandelt. Die konsolidierte Version enthält die Änderungen der beiden geänderten Versionen.

(O) `diff3` berechnet aus zwei gegebenen Änderungsbeziehungen eine konsolidierte Änderungsbeziehung bzw. eine konsolidierte Version, die mit der Grundversion in dieser Beziehung steht.

Schwachstellen von `diff3`

Allerdings ist `diff3` nicht immer in der Lage, zwei Änderungen zusammenzuführen bzw. sie korrekt zusammenzuführen:

1. Wenn in den beiden Versionen die Formatierung einer Zeile in unterschiedlicher Weise geändert wurde, ist ein Merge mit `diff3` bereits unmöglich.

2. Wenn beispielsweise beide Änderungen an derselben Stelle eine Methode in eine Java-Klasse eingefügen, kann `diff3` nicht entscheiden, welche von beiden zuerst kommen soll, weshalb es die Änderungen nicht zusammenführen kann.

3. Außerdem kann das Ergebnis eine fehlerhafte Syntax haben oder Typfehler enthalten, auch wenn die Eingabeprogramme syntaktisch korrekt und typkorrekt sind.

4. Noch problematischer ist die Tatsache, dass das Ergebnis manchmal zwar syntaktisch korrekt und typkorrekt ist, aber die Semantik des Programmes sich in unbeabsichtigter Weise ändert.

Beispiel: Semantischer Konflikt beim Merge

Eine Situation, in der sich die Semantik des Programmes in unbeabsichtigter Weise ändert, ist in Abb. 3a illustriert: In der Grundversion gibt es ein Feld `b` und eine Methode `a` ohne Parameter. Änderung 1 fügt eine Zuweisung `b = 4` an das Feld `b` zur Methode `a` hinzu. Änderung 1 führt einen neuen Parameter `b` in `a` ein (und benennt `b` in `c` um, um zu verdeutlichen, dass nicht lediglich das Feld `b` verdeckt wird). Wenn man nun die beiden Änderungen mit `diff3` zusammenführt, erhält man ein syntaktisch korrektes und typkorrektes Programm. Dieses spiegelt jedoch nicht die Intention von Änderung 1 wieder: Die Zuweisung `b = 4` weist nämlich nicht mehr einem Feld `b` einen neuen Wert zu, sondern dem neu hinzugefügten Parameter mit demselben Namen. Um der Intention hinter der Änderung zu entsprechen, müsste die Zuweisung in der konsolidierten Version `c = 4` lauten (siehe Abb. 3b).

Ursache: Lexikalische Operationsebene

Die Ursache der oben genannten Probleme liegt darin, dass `diff3` rein textuell (auf lexikalischer Ebene) operiert. `diff3` nimmt über die Struktur der Dateien lediglich zwei Dinge an: erstens, dass sie aus einer Folge von durch Zeilenumbrüche getrennten Textzeilen bestehen, und zweitens, dass zwei Zeilen genau dann gleichgesetzt werden dürfen, wenn ihr Inhalt gleich ist. Die Syntax und die Semantik der Programme berücksichtigt es nicht.

Strukturierter Merge

Ein Lösungsansatz für einige der Probleme des lexikalischen Merge sind der strukturierte [Wes91; Buf95] bzw. semistrukturierte [Ape+11; Leß12] Merge. Diese operieren nämlich auf der statisch semantischen oder syntaktischen Ebene anstatt auf der lexikalischen Ebene. Die Programmstrukturen werden berücksichtigt und Änderungen von Programmelementen anstatt von Textzeilen zusammengeführt.

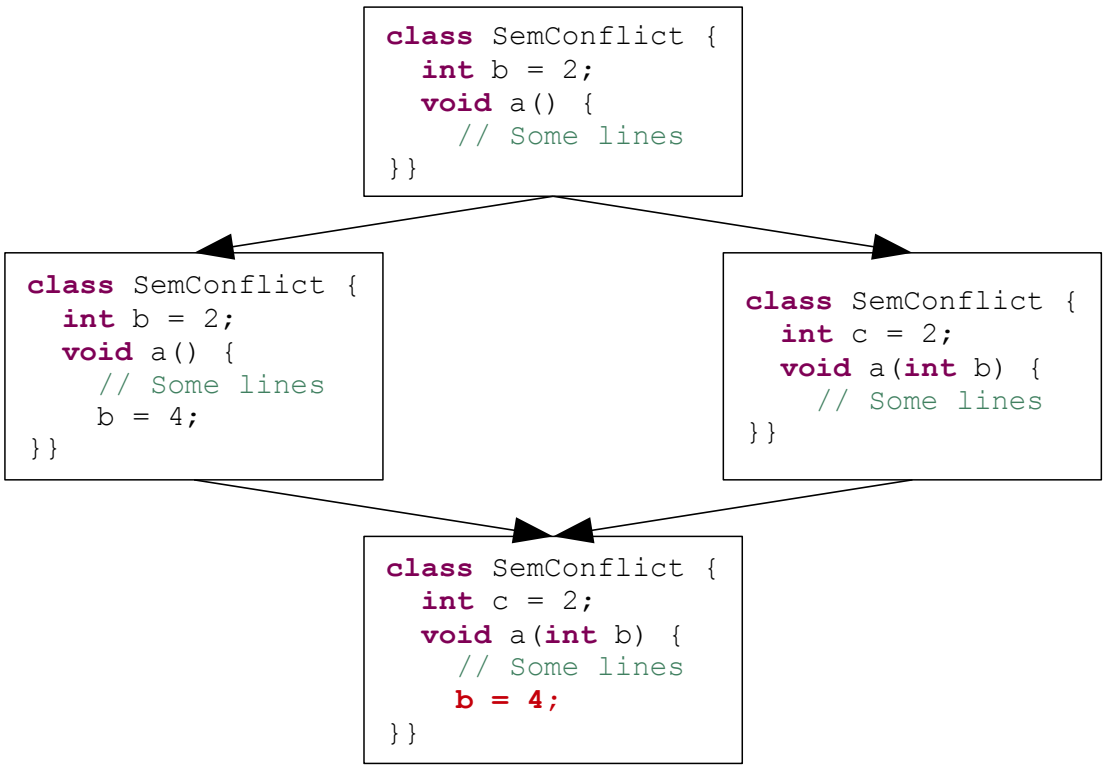
Einordnung: Semistrukturierter Merge

Der strukturierte bzw. semistrukturierte Merge unterscheidet sich von `diff3` im Wesentlichen nur in der Repräsentation. Beispielsweise werden für den strukturierten Merge abstrakte Syntaxbäume verwendet:

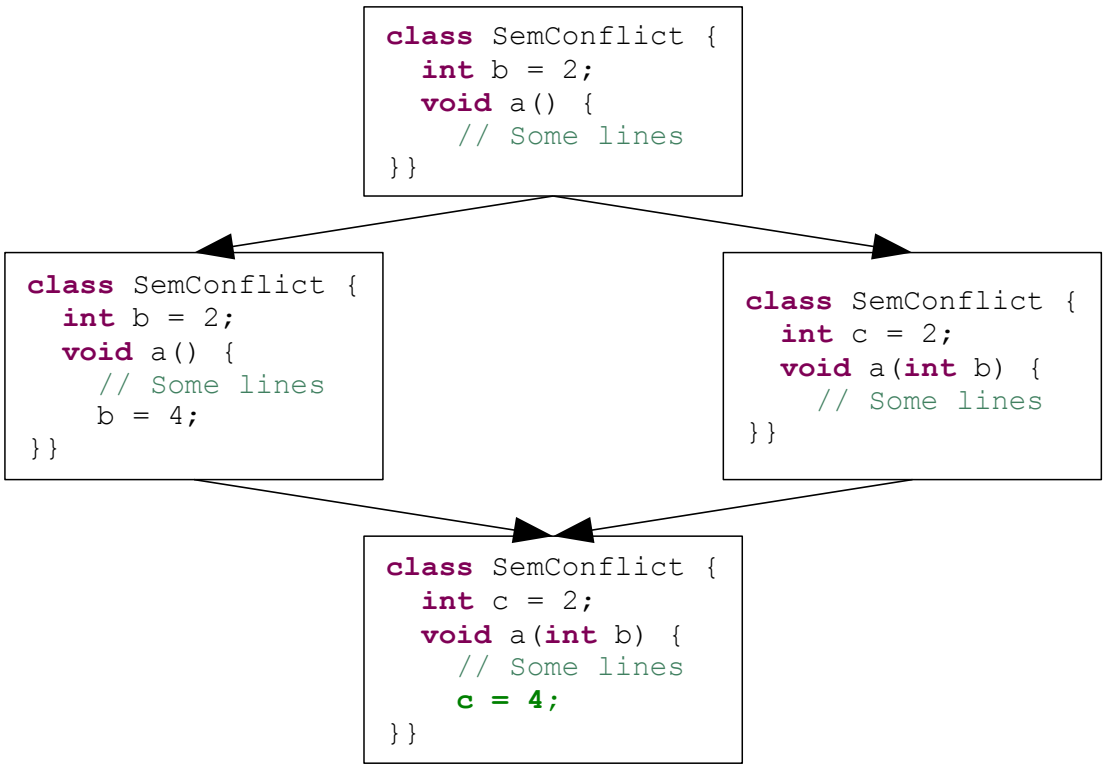
(R) Das Programm wird als ein Baum repräsentiert.

(E) (nicht vorhanden)

(B) Es werden Änderungsbeziehungen zwischen der Grundversion und den geänderten Versionen bzw. der konsolidierten Version behandelt. Die konsolidierte Version enthält die Änderungen der beiden geänderten Versionen.



(a) Tatsächliches Ergebnis



(b) Erwartetes Ergebnis

Abbildung 3: Semantischer Konflikt

(O) Der strukturierte Merge berechnet aus zwei gegebenen Änderungsbeziehungen eine konsolidierte Änderungsbeziehung bzw. eine konsolidierte Version, die mit der Grundversion in dieser Beziehung steht.

An diesem Beispiel zeigt sich, dass es möglich ist, dieselbe Programmmanipulation, in diesem Fall den Merge, auf unterschiedlichen Programmrepräsentationen auszuführen. Die Ergebnisse unterscheiden sich aber je nach gewählter Repräsentation. Ein (semi-)strukturierter Merge erzeugt zufriedenstellendere Ergebnisse, benötigt aber im Gegenzug mehr Laufzeit als ein lexikalischer Merge. Durch unterschiedliche Programmrepräsentationen lässt sich so ein Trade-Off zwischen verschiedenen Eigenschaften des Merge vornehmen.

2.3. Featurekomposition

Featuremodule und Featurekomposition sind wesentliche Grundbegriffe der featureorientierten Programmierung (FOP), einem Programmierparadigma zur Umsetzung von Software-Produktlinien [Pre97]. Eine Produktlinie ist eine Menge von Produkten, die sich aus einer Menge von Features entsprechend den in einem sogenannten Domänenmodell festgelegten Regeln kombinieren lassen. Im Fall von Software-Produktlinien handelt es sich bei den Produkten um Programme. Die Domänenmodelle können als Feature-Diagramme [Kan+90] oder als boolsche Formeln ausgedrückt werden.

Im FOP-Ansatz der schrittweisen Verfeinerung ([BSR04]) ist ein Featuremodul eine Programmänderung, die genau ein Feature des Domänenmodells realisiert und kapselt. Die Operation, die Featuremodule zu Produkten kombiniert, nennt man Featurekomposition. Sie führt also ebenso wie der Merge Programmänderungen zusammen. Der Unterschied ist, dass die Änderungen mit den Mitteln einer Programmiersprache ausgedrückt werden und die Programmstruktur nicht auf eine freie, beliebige Art geändert werden kann.

Superimposition

Superimposition [AL08] ist eine Ansatz zur Featurekomposition, bei der Featuremodule in sogenannten Feature Structure Trees (FSTs) als Bäume repräsentiert werden. FSTs spiegeln den für die Featurekomposition relevanten Teil des abstrakten Syntaxbaums wieder. Bei Java ist dies die Package- und Klassenstruktur, wobei Methodenrümpfe abgetrennt werden. Die Superimposition überlagert zwei FSTs, indem Knoten mit gleichem Namen gleichgesetzt werden. Falls Knoten auf unterster Ebene mit gleichem Namen (bzw. Signatur) vorhanden sind, wird eine sprachabhängige Komposition ausgeführt. Im Fall von Java können dabei nur Methoden überlagert werden, was auch als Methodenverfeinerung bezeichnet wird. Diese wird mittels des neu eingeführten Schlüsselworts `original()` realisiert. Mit `original()` wird von der Methodenverfeinerung aus die ursprüngliche Methode aufgerufen. Als Klassenverfeinerung wird bezeichnet, wenn durch Superimposition Felder oder Methoden zu einer Klasse hinzugefügt oder verfeinert werden.

Beispiel: Featurekomposition durch Superimposition

Im Beispiel aus Abb. 4 sind die Feature Structure Trees für ein Basismodul und ein Modul für gewichtete Kanten aus einer Produktlinie von Graphen dargestellt, die miteinander komponiert werden. Im Basismodul ist unter anderem eine Klasse `Edge` enthalten, die eine Kante in einem Graphen repräsentiert und deren Quellcode in Abb. 5 abgebildet ist. Wie man sieht, definiert die Klasse `Edge` Felder für Quell- und Zielknoten der Kante und eine Methode `print`, die eine textuelle Beschreibung der Kante liefert. Das Featuremodul für Kantengewichte enthält eine Verfeinerung für `Edge`, die ein neues Feld vom Typ `double` für das Gewicht einführt und die Methode `print` so verfeinert, dass sie das Gewicht mit

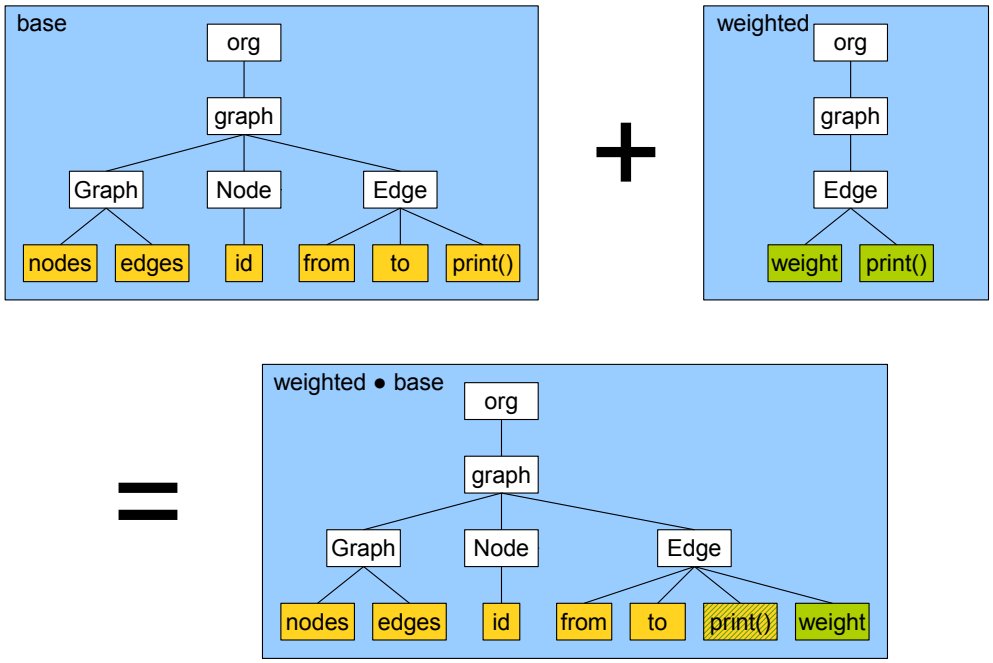


Abbildung 4: Superimposition von Feature Structure Trees

```
class Edge {
  Node from, to;
  String print() {
    return from.id +
      " -> " + to.id;
}}
```

```
refines class Edge {
  double weight;
  refines String print() {
    return original() + " ("
              + weight + ")";
}}
```

```
class Edge {
  Node from, to;
  double weight;
  String print() {
    String v1 = from.id
           + " -> " + to.id;
    String v2 = v1 + " ("
             + weight + ")";
    return v2;
}}
```

Abbildung 5: Featurekomposition von Java-Klassen mittels Superimposition (entlehnt aus [Ape+10])

ausgibt. Das Ergebnis enthält die Felder der Basisklasse und der Verfeinerung, und das Ergebnis der Verfeinerung der Methode `print`, das zu der Methode im unteren Rechteck von Abb. 5 verhaltensäquivalent ist.

Einordnung

Featurekomposition durch Superimposition lässt sich ebenfalls in das Schema der vier Merkmale einordnen:

(R) Programme sind Feature Structure Trees.

(E) Ein Produkt (Programm) implementiert ein Feature oder nicht.

(B) Produkte können in Verfeinerungsbeziehungen zueinander stehen.

(O) Die Featurekomposition eines Produktes mit einem Featuremodul ergibt ein Produkt, das das zugehörige Feature implementiert und das mit dem zugrundeliegenden Produkt in einer Verfeinerungsbeziehung steht.

Diese Einordnung würde abgesehen von der Repräsentation auch auf andere Ansätze der Featurekomposition als die Superimposition zutreffen. Aus dem Beispiel FeatureHouse folgt, dass man durch eine sprachübergreifende Programmmrepräsentation sprachunabhängige Programmtransformationen erreichen kann, ähnlich wie bei Zwischenrepräsentationen für Compiler [CNL79].

2.4. Syntaktische Interaktionsanalyse

Bei der featureorientierten Umsetzung von Softwareproduktlinien kann ein Featuremodul von Definitionen anderer Featuremodule abhängig sein oder mit Definitionen anderer Featuremodule im Konflikt stehen. Hierdurch können einzelne Featurekombinationen zu Kompilierfehlern führen, während sich andere ohne Fehler kompilieren lassen. Die (syntaktische) Interaktionsanalyse soll automatisch erkennen, bei welchen laut dem Domänenmodell (siehe Abschnitt 2.3) *erlaubten* Featurekombinationen solche Fehler auftreten. Hierdurch werden Schwächen im Domänenmodell und in der Implementierung aufgedeckt. Der hier untersuchte Spezialfall sucht nach erlaubten Featurekombinationen, die zu Dangling References führen, also zu Verweisen auf Klassen, Methoden, oder Feldern, die in der jeweiligen Featurekombination nicht definiert sind [Ape+10].

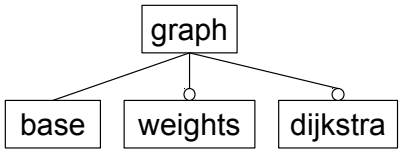


Abbildung 6: Featurediagramm (entlehnt aus [Ape+10])

Hierzu wird zunächst das Domänenmodell als boolsche Formel *DM* formuliert, deren Aussagenvariablen dafür stehen, welche Features aktiviert sind. Die Abhängigkeiten zwischen vorhandenen Features, die sich aus deren Implementierung ergeben, werden in einem sogenannten Implementierungsmodell abgebildet. Dieses wird als boolsche Formel *IM* ausgedrückt. Als Abhängigkeiten müssen in der featureorientierten Erweiterung von Java hierbei Benutzung, Erweiterung und Verfeinerung von Klassen (auch Interfaces und Enums),

Implementierung von Interfaces, Aufruf, Überschreibung und Verfeinerung von Methoden und Feldzugriffe gezählt werden. Mit einem SAT-Solver wird abschließend geprüft, ob die Formel $DM \Rightarrow IM$ immer wahr ist. Wenn dies zutrifft, stellt das Domänenmodell sicher, dass die Abhängigkeiten des Implementierungsmodells beachtet werden.

Beispiel: Interaktionsanalyse

Beispielsweise könnte es in der Graph-Produktlinie aus Abschnitt 2.3 ein weiteres Feature `dijkstra` zur Berechnung kürzester Wege im Graphen geben, das die in `weights` hinzugefügten Kantengewichte als Weglängen benutzt. Laut dem in Abb. 6 als Featurediagramm dargestellten Domänenmodell muss das Feature `base` immer aktiv sein und `weights` und `dijkstra` sind optional. Als boolsche Formel lässt es sich folgendermaßen ausdrücken:

$$DM = graph \wedge (graph \Leftrightarrow base) \wedge (weights \Rightarrow graph) \wedge (dijkstra \Rightarrow graph)$$

In unserem Beispiel werden in Feature `weights` die Klasse `Edge` und die Methode `Edge.print` verfeinert, die in `base` definiert sind. Das in `weights` hinzugefügte Feld `Edge.weight` wird von Feature `dijkstra` ebenso benutzt wie die Klassen des Features `base`. Hieraus ergibt sich als Implementierungsmodell die Formel $IM = (weights \Rightarrow base) \wedge (dijkstra \Rightarrow base) \wedge (dijkstra \Rightarrow weights)$

Die sich ergebende Formel $DM \Rightarrow IM$ ist nicht immer wahr: Für die Belegung $\{graph \mapsto true, base \mapsto true, weights \mapsto false, dijkstra \mapsto true\}$, also wenn man versucht, den Kürzeste-Wege-Algorithmus hinzuzufügen ohne die dafür nötigen Kantengewichte miteinzubeziehen, evaluiert sie zu *false*.

Einordnung

Die syntaktische Interaktionsanalyse ist ein Beispiel einer reinen Analyse, d.h. die Operation berechnet nicht neue Programme (in diesem Fall Produktlinien), sondern Eigenschaften von Programmen. Der hier betrachtete Spezialfall [Ape+10] ist folgendermaßen einzuordnen:

(R) Programme (Produktlinien) sind durch ein Domänenmodell und eine Menge von Abhängigkeiten repräsentiert, die in eine boolsche Formel abstrahiert wird.

(E) Die Implementierung berücksichtigt das Domänenmodell: Jede erlaubte Featureauswahl ist frei von dangling references.

(B) (keine)

(O) Finde dangling references bzw. beweise ihre Abwesenheit (durch Lösen der Formel).

2.5. Refactoring

Refactoring bezeichnet nach Fowler [FB99] Änderungen an der internen Struktur von Software mit dem Ziel, sie leichter verständlich und änderbar zu machen, ohne dabei ihr beobachtbares Verhalten zu ändern. Refactoring wirkt der Verschlechterung der Codequalität entgegen, die sich im Laufe der Weiterentwicklung von Software ergibt.

Deklarative Implementierung von Refactorings

In der Praxis eingesetzte Programmiersprachen sind nach [ST09] zu komplex um korrekte Refactoringwerkzeuge, die alle Spezialfälle abdecken und insbesondere alle inkorrekten Refactorings verweigern, *imperativ* zu implementieren. Deshalb wird vorgeschlagen, Refactorings auf *deklarative* Weise mithilfe von Constraint-Satisfaction-Problemen (CSPs) über endlichen Domänen zu implementieren. Ein CSP besteht aus einer endlichen Menge von

Bedingungen (Constraints) mit Variablen, die mit Werten aus gegebenen endlichen Mengen auf eine solche Weise belegt werden sollen, dass alle Constraints erfüllt sind [Tsa93].

Ein Teil der Constraints wird mithilfe des zu ändernden Programmes aus einer Menge von programmunabhängigen Formelschemata generiert, die einen Teil der statischen Semantik der Programmiersprache ausdrücken. Anstatt imperativ die Schritte des Refactorings zu beschreiben, gibt der Entwickler des Refactoring-Werkzeugs Regeln zur Erstellung von Constraints an. Die Einhaltung der erstellten Constraints stellt sicher, dass sich die Bedeutung des Programmes nicht ändert. Das Ziel des Refactorings wird ebenfalls mittels Constraints definiert. Dadurch ist die Arbeitsebene sehr nahe am Kern des Problems und dadurch laut den Autoren weniger fehleranfällig.

Beispiel: Constraint-basiertes Refactoring

Schon bei einem der primitivsten Refactorings, dem Ändern der Zugreifbarkeit in Java (`public`, `protected`, ...), erweist sich der deklarative Ansatz als nützlich. Auch bei dieser möglicherweise trivial erscheinenden Änderung kann sich nämlich die Bindung einer Methode und somit die Bedeutung des Programms ändern, was verhindert werden muss. Dies verdeutlicht das folgende, aus [ST09] übernommene Beispiel:

```
package a;
public abstract class A {
  protected /*d1*/ abstract void m(String s);
}

package a;
public class B extends A {
  public    /*d2*/ void m(Object o) {...}
  protected /*d3*/ void m(String s) {...}
}

package b;
public class C {
  void n() {
    /*r1*/ (new a.B()).m("abc");
  }
}
```

Listing 2.1: Beispiel: Refactoring; übernommen aus [ST09]

Wenn man in dem Programm aus 2.1 die Zugreifbarkeit der mit `d1` bezeichneten Deklaration von `A.m(String)` zu `public` erhöhen will, muss auch ihre Überschreibung `d3` in `B` auf `public` gesetzt werden, da in Java die Zugreifbarkeit von Methoden nicht reduziert werden darf. Hierdurch würde aber in `C.n()` der Aufruf `r1` nicht mehr an `B.m(Object)` binden, sondern an `B.m(String)`. Da sich durch ein Refactoring die Bindung der Methoden nicht ändern darf, darf dieses Refactoring nicht ausgeführt werden.

Damit stimmt auch das Ergebnis des constraint-basierten Refactorings überein, denn das entsprechende CSP ist unlösbar. Aus Platzgründen können hier nicht alle Regeln und Constraints aufgezählt werden – hierfür sei auf [ST09] verwiesen. Die minimale Teilmenge der generierten Constraints, die zur Unlösbarkeit führen, ergibt sich aus den beiden im Folgenden genannten Regeln. Die Regeln sind von der Form "**Wenn** F, **dann** G" und besagen, dass für jede Kombination von Programmelementen, auf die F zutrifft, eine entsprechende Instanz von G zum CSP hinzugefügt wird.

- **Wenn** eine Methode d' eine Methode d überschreibt oder verdeckt, **dann** muss die

deklarierte Zugreifbarkeit von d' größer oder gleich der deklarierten Zugreifbarkeit von d sein **(Sub-1)**.

- **Wenn** eine in einer Klasse c definierte Methode d' eine Methode d überlädt, an die eine Referenz r bindet, und c die Empfängerklasse von r oder eine Oberklasse davon ist, **dann** darf d' von der r enthaltenden Klasse aus nicht sichtbar sein, da r ansonsten an d' binden müsste **(Ovr)**.

Hieraus ergeben sich unter anderem die beiden folgenden Constraints:

- Die Sichtbarkeit von `d3` muss mindestens der Sichtbarkeit von `d1` entsprechen, da es `d1` überschreibt.
- Die Sichtbarkeit von `d3` muss kleiner als `public` sein, damit `r1` an `d2` bindet.

Ein drittes Constraint besagt, dass `d1` die Sichtbarkeit `public` haben muss, da dies das Ziel des Refactorings ist. Diese Constraints sind zusammen nicht lösbar, da `d3` gleichzeitig `public` und kleiner als `public` sein müsste. Also muss das Refactoring abgelehnt werden.

Wenn `d2` nicht vorhanden wäre, würde die zweite Constraint wegfallen und das System wäre lösbar. Diese Lösung würde besagen, dass sowohl `d1` als auch `d3` auf `public` gesetzt werden müssen. Dies entspricht dem gewünschten Ergebnis.

Einordnung

Dieser Ansatz des constraint-basierten Refactorings lässt sich in die vier Merkmale folgendermaßen einordnen:

(R) Die für das Refactoring relevanten Aspekte von Programmen sind durch Variablen eines CSP repräsentiert. Jede Lösung davon entspricht einem Programm.

(E) (keine)

(B) Alle Lösungen des CSP sind zueinander verhaltensäquivalente Programme. Außerdem entspricht ein Refactoring einer (disziplinierten) Programmänderung.

(O) Durch das Lösen des CSP können verhaltensäquivalente Abänderungen eines Programms berechnet werden.

2.6. Zusammenfassung

In diesem Kapitel wurden gängige Programmanipulationen vorgestellt und es wurde gezeigt, dass sie sich anhand des hier vorgeschlagenen Schemas der vier Merkmale charakterisieren lassen: Zu jeder dieser Programmmanipulationen gehört eine Programmrepräsentation und eine darauf arbeitende Operation, die Programmeigenschaften oder -beziehungen berechnet.

Außerdem wurden bei der Featurekomposition und beim Merge Feststellungen bezüglich Sprach- bzw. Repräsentationsunabhängigkeit gemacht, deren Verallgemeinerung Inhalt dieser Arbeit ist: An der Featurekomposition durch Superimposition hat sich gezeigt, dass man durch eine sprachübergreifende Programmrepräsentation auch sprachunabhängige Programmmanipulationen erreichen kann. Beim Merge wurde festgestellt, dass er – auf hoher Ebene betrachtet – auf unterschiedlichen Programmrepräsentationen operieren kann. Dabei variieren funktionale und nichtfunktionale Eigenschaften des Merge, wie etwa das sich ergebende Programm oder die Laufzeit.

In den nächsten Kapiteln wird untersucht, wie man diese Merkmale auf generische Art umsetzen und dadurch die Definition der Programmanipulationen auf eine höhere Ebene verlagern kann.

Kapitel 3

Graphbasierte Programmmanipulation

Ziel dieser Arbeit ist es, einen implementierungs-, sprach- und repräsentationsunabhängigen Formalismus zur Beschreibung von Programmmanipulationen zu schaffen. In diesem Kapitel wird dieser Formalismus konzipiert. Zu diesem Zweck werden von den vier Merkmale von Programmmanipulationen die Programmrepräsentation, die Programmeigenschaften und die Programmbeziehungen in allgemeiner Form formal umgesetzt. Das vierte Merkmal, die Operationen, werden in dieser Arbeit nicht betrachtet.

Um Implementierungs-, Sprach- und Repräsentationsunabhängigkeit zu erreichen, muss man die Programmmanipulationen auf einer sehr abstrakten Ebene beschreiben. Im Bereich der Modellierung bestehen bereits Ansätze, die dies für den Merge von Modellen erreichen [Tae+10; Rut+09] (Merge siehe Abschnitt 2.2). Die hohe Abstraktionsebene wird in diesen Ansätzen durch die Anwendung der Kategorientheorie erreicht, einem Gebiet der Mathematik mit zahlreichen Anwendungen in der Informatik [BW12].

Im Ansatz von [Tae+10] werden Modelle durch Graphen der algebraischen Graphtransformationstheorie [Ehr+06] dargestellt und Änderungen davon zusammengeführt. Da diese Art von Graphen formal sind, sind sie für den hier definierten Formalismus relevant. Man kann mit solchen Graphen nämlich nicht nur Modelle, sondern auch Programme repräsentieren. Der Merge wird in Abschnitt 5.1 auf dieselbe Art wie in [Tae+10] formalisiert.

Um zu bestimmen, welche Modellierungselemente Graphen für die Programmrepräsentation bieten sollen, ist es hilfreich, EMFText [Hei+09] zu betrachten. Dabei handelt es sich um einen funktionierenden Ansatz zur Repräsentation der statischen Semantik von Programmen mit der Modellierungssprache Ecore [Ste+08]. Die statische Semantik in diesem Sinn ist ein Zwischenschritt zwischen abstrakter Syntax und dynamischer Semantik. Sie spiegelt die Strukturelemente der Programme wider und abstrahiert dabei von im Quelltext der Programme vorhandenen, aber für die dynamische Semantik irrelevanten Informationen. Beispielsweise sind Begriffe wie „Klasse“, „While-Schleife“, „Referenz“, „Generic“ oder „String-Literal“ zwingende Bestandteile der statischen Semantik von Java, während beispielsweise Kommentare, Formatierung oder die Definitionsreihenfolge von Methoden ignoriert werden können. Die in Kapitel 2 untersuchten Programmmanipulationen

Die von EMFText erzeugte Ecore-Programmrepräsentation ist graphähnlich, ist aber nicht formal definiert. Deshalb eignet sie sich zwar nicht unmittelbar zur Definition des Formalismus, kann aber als Richtschnur dienen, um zu entscheiden, welche Modellierungselemente im Formalismus vorgesehen werden sollen.

Neben EMFText/Ecore gibt es auch abstrakte Semantikgraphen (ASGs) [RW91; Rag03; Rag+04] zur Repräsentation der statischen Semantik von Programmen. Allerdings besitzen diese nur eine feste, nicht auswechselbare Programmrepräsentation. Außerdem gibt es im Bereich der Programmanalyse die abstrakte Interpretation [CC77; CC92], die ebenfalls eine

Art der statischen Semantik repräsentiert, die aber nicht mit dem hier verwendeten Begriff übereinstimmt. Deshalb wird hier nur EMFText/Ecore genauer betrachtet.

In diesem Kapitel werden zunächst EMFText/Ecore untersucht und anhand dessen Anforderungen an den Formalismus entwickelt. Anschließend werden Schritt für Schritt Graphvarianten der algebraischen Graphtransformationstheorie erklärt, die diese Anforderungen umsetzen. Schließlich werden einige Grundbegriffe Kategorientheorie eingeführt, die der Beschreibung von Programmmanipulationen auf abstrakter Ebene dienen.

3.1. Anforderungen an eine graphbasierte Programmrepräsentation

EMF (Eclipse Modeling Framework) ist ein Framework zur Modellierung, das auf der UML-ähnlichen visuellen Modellierungssprache Ecore aufbaut [Ste+08]. EMFText [Hei+09] ist ein Ansatz zur Repräsentation von (textuellen) Programmen als Instanzen eines Ecore-Modelles oder in umgekehrter Richtung zur textuellen Repräsentation von Instanzen eines Ecore-Modelles. EMFText bietet die Möglichkeit, für eine Programmiersprache mittels eines Ecore-Modelles ihrer statischen Semantik und einer Beschreibung ihrer Grammatik einen Parser zu erzeugen, der aus Programmdateien Instanzen des Ecore-Modelles erzeugt. Außerdem kann man in umgekehrter Richtung eine Instanz des Ecore-Modells wieder als Programmdatei speichern. Mit JaMoPP [Hei+10] gibt es ein Ecore-Modell und eine EMFText-Grammatik für Java, und somit eine Modellierung der statischen Semantik von Java in Ecore.

Da Ecore visuell ist, ähneln Ecore-Modelle und -Instanzen Graphen, sind aber im Gegensatz zu diesen nicht formal. Damit kommt EMFText dem, was der hier zu entwickelnde Formalismus bezüglich der Programmrepräsentation leisten soll, sehr nahe – nur dass es nicht formalisiert ist. Deshalb wird im Folgenden zunächst ein kurzer Überblick über Ecore gegeben und dann wird anhand von EMFText analysiert, welche Teile von Ecore für den Formalismus relevant sind. Auf diese Weise ergeben sich Anforderungen an den Formalismus, die besagen, welche Modellierungselemente die Programmrepräsentation bieten soll.

Ecore ist eine Modellierungssprache, die auf die praktische Anwendbarkeit im Java-Umfeld ausgerichtet ist. Dadurch enthält es neben den für den Formalismus relevanten Modellierungselemente andere Sprachelemente, die für die Repräsentation der statischen Semantik von Programmen nicht relevant sind und ein formales Modell unnötig komplex machen würden. Hierbei handelt es sich um Sprachelemente zur Generierung von Java-Implementierungen, zur Speicherung von Instanzen in Dateien, zur Modularisierung, zum reflektiven Zugriff auf Modelle und Annotationen. Diese irrelevanten Nicht-Modellierungs-Elemente sollen im Formalismus nicht berücksichtigt werden. Die wichtigsten Modellierungselemente von Ecore, von denen die meisten von EMFText benutzt werden, sind die folgenden:

- *Klassen* –
  Die grundlegendste Abstraktion von Ecore sind Klassen im Sinne der objektorientierten Programmierung, also Objekttypen.

  Mit ihnen kann man die syntaktischen Kategorien einer Programmiersprache, also ihre grundlegenden Strukturierungseinheiten, erfassen.

- *Referenzen –*
  Referenzen erlauben es, Klassen miteinander in Beziehung zu setzen. Mit ihnen kann man verschiedenartige Zusammenhänge zwischen Instanzen von Klassen ausdrücken. Es gibt eine Unterscheidung zwischen Containment- und Nicht-Containment-Referenzen. Containment-Referenzen können dazu dienen, die hierarchische Struktur von Programmen zu repräsentieren. Mit Nicht-Containment-Referenzen kann man beispielsweise eine Referenz auf eine Variable mit deren Definition verknüpfen.

- *Attribute und Datentypen –*
  Klassen können auch Attribute enthalten, die durch sogenannte Datentypen typisiert sind. Datentypen kann man dabei auch selbst definieren. Im Unterschied zu Klassen können von Datentypen keine Referenzen ausgehen und sie besitzen keine Attribute.

  Ein separat berücksichtigter Spezialfall von Datentypen sind Enums, also Aufzählungsdatentypen, als Modellierung von Java-Enums. Ecore definiert einige Datentypen vor - insbesondere Repräsentationen der wichtigsten Java-Datentypen.

  Attribute und Datentypen sind geeignet, um Bezeichner und Literale zu modellieren.

- *Generische Typen –*
  Ab Version 2.3 unterstützt EMF auch generische Typen und Operationen im Sinne von Java-Generics.

  Diese könnten für der Programmrepräsentation verwendet werden, um redundante Definitionen zu vermeiden.

- *Operationen –*
  Operationen repräsentieren beliebige Methoden. Die Parameter und der Rückgabewert können sowohl durch Datentypen als auch durch Klassen typisiert sein. Im Rahmen der Semantik von Ecore ist keine Möglichkeit vorgesehen, die Funktionsweise der Operation festzulegen. Es handelt sich im Wesentlichen um Platzhalter.

  Solange Operationen reine Platzhalter sind, haben sie keinen Wert für die Programmrepräsentation. Andernfalls wäre es beispielsweise denkbar, sie für die Beschreibung von Polymorphie oder zur Ableitung von nicht explizit im Modell repräsentierten Informationen einzusetzen.

- *Multiplizitäten, Ordnung, Eindeutigkeit –*
  Referenzen, Attribute, Parameter und Rückgabewerte von Operationen können prinzipiell auch mehrwertig sein. Man kann jeweils eine Mindest- und eine Höchstzahl an Werten spezifizieren. Außerdem kann man spezifizieren, ob die Menge der Werte geordnet ist und ob derselbe Wert mehrmals auftreten darf.

Die Entscheidung, ob ein bestimmtes Modellierungselement von Ecore im Formalismus einbezogen werden soll, soll sich an folgenden Richtlinien orientieren:

- **OnlyModelling**: *Nur Modellierung –*
  Es sollen nur die Modellierungselemente von Ecore formalisiert werden. Generierung, Serialisierung, Modularisierung und Reflexion sollen also ignoriert werden.

- **IncProgRep**: *Programmrepräsentation –*
  Diejenigen Modellierungselemente von Ecore, die zur Programmrepräsentation notwendig sind, sollen formalisiert werden. Als Richtlinie dient dabei ihre Verwendung in EMFText zur Repräsentation der statischen Semantik.

EMFText ist zunächst dazu gedacht, für EMF-Modelle eine textuelle Syntax zu definieren und Parser und Pretty-Printer (Ausgabewerkzeuge) für diese Syntax zu generieren. Man kann aber auch in umgekehrter Richtung vorgehen und für eine bestehende Programmiersprache ein Ecore-Modell definieren und mithilfe einer passenden Syntaxdefinition einen Parser erzeugen, der Programme einliest und daraus Instanzen des Ecore-Modelles produziert.

Auf diese Art wurde beispielsweise mit JaMoPP [Hei+10; Hei+09] ein Ecore-Modell und eine Syntaxdefinition für Java geschaffen, mitsamt zugehörigem Parser und Pretty-Printer. Im Ecore-Modell von JaMoPP werden die Sprachelemente von Java wie z.B. Klassen, Methoden, die verschiedenen Arten von Statements und Ausdrücken, Typen usw. als Ecore-Klassen repräsentiert. Gemeinsamkeiten zwischen diesen Sprachelementen werden durch abstrakte Oberklassen zusammengefasst, wovon in JaMoPP ausgiebig Gebrauch gemacht wird. Beispielsweise sind lokale Variablen und Felder beides Variablen. Daneben werden abstrakte Klassen an den Stellen benutzt, wo es mehrere Auswahlmöglichkeiten gibt, z.B. können Java-Klassen Feld- und Methodendefinitionen in beliebiger Reihenfolge enthalten, weshalb sie durch die abstrakten Ecore-Klasse „Member“ zusammengefasst werden müssen. Containment-Referenzen drücken den hierarchischen Aufbau des Programmes aus, z.B. die Beziehung zwischen einer Java-Klasse und einer darin definierten Methode. Nicht-Containment-Referenzen werden verwendet, um Verweise z.B. auf Variablen aufzulösen. Namen, Literale und Modifikatoren werden durch Attribute ausgedrückt.

Die Syntax der Programmiersprache und die Zuordnung zum Ecore-Modell wird durch eine Grammatik festgelegt, die als „konkrete Syntax“ bezeichnet wird. Zu diesem Zweck werden die Klassen, Referenzen und Attribute des Ecore-Modells benutzt. Für jede konkrete (d.h. nicht abstrakte) Klasse des Ecore-Modells wird in der konkreten Syntax eine Produktionsregel definiert. Dabei müssen sämtliche Attribute und Referenzen in der Produktionsregel vorkommen, und zwar in Übereinstimmung mit ihrer Multiplizität im Ecore-Modell. Obwohl man sie im Ecore-Modell verwenden kann, werden von EMFText aus der konkreten Syntax heraus keine Generics, Operationen oder Exceptions generiert.

Aufgrund dieser Feststellungen wurden die für die Programmrepräsentation relevanten Modellierungselemente von Ecore nach den oben genannten Richtlinien ausgewählt. Die wichtigsten ausgewählten Modellierungselemente sind:

- Klassen
- Containment- und Nicht-Containment-Referenzen
- Datentypen und Attribute
- Ordnung
- Multiplizitäten
- Eindeutigkeit

Eine detaillierte Auflistung findet sich in Anhang C.

3.2. Graphen zur Programmrepräsentation

Im vorherigen Abschnitt wurde eine Menge von Modellierungselementen ausgewählt, die für Repräsentation der statischen Semantik von Programmen relevant sind und die der Formalismus deshalb unterstützen sollte. Abb. 7 illustriert die Konzeption des Formalismus und die Umsetzung der vier Merkmale von Programmmanipulationen: Die Programmmanipulationen werden graphbasiert und mithilfe der Kategorientheorie umgesetzt: Die Programmrepräsentation erfolgt durch auswechselbare Graphvarianten. Programmeigenschaften und -beziehungen werden mit den kategorientheoretischen Konstruktionen ausgedrückt. Diese werden unabhängig von der graphbasierten Programmrepräsentation beschrieben, können aber darauf angewendet werden. Die Operationen entsprechen der Berechnung der kategorientheoretischen Konstruktionen.

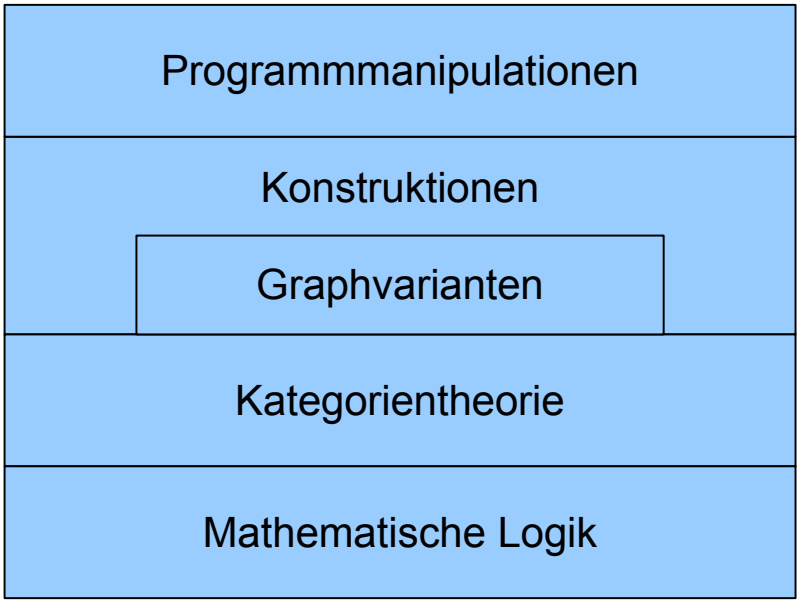


Abbildung 7: Schichtenaufbau des Formalismus

Graphfeatures

In der algebraischen Graphtransformationstheorie gibt es zahlreiche verschiedene Graphvarianten mit unterschiedlichen Modellierungselementen. Üblicherweise wird dabei jede Graphvariante von Grund auf neu definiert, obwohl es zwischen den Graphvarianten zahlreiche Gemeinsamkeiten gibt. Dies führt zum Einen zu Redundanzen und zum Anderen zu sehr komplizierten Definitionen. Um die Definitionen besser handhabbar zu machen, wird hier versucht, die einzelnen Modellierungselemente der Graphvarianten als modulare **Graphfeatures** aufzufassen und möglichst unabhängig voneinander zu beschreiben. Diese Graphfeatures kann man zu komplexen Graphvarianten kombinieren.

Aufbau

In diesem Abschnitt werden zunächst Graphfeatures zur Programmrepräsentation vorgestellt, die die in Abschitt 3.1 ausgewählten Modellierungselemente von Ecore umsetzen. Außerdem wird ein Überblick über einige Grundideen der Kategorientheorie gegeben, um die Grundidee der kategorientheoretischen Beschreibung von Programmmanipulationen zu erläutern. Die formale Darstellung ist in Kapitel 4 ausgelagert.

In diesem Abschnitt werden die folgenden Graphfeatures informell dargestellt und auf ihre Relevanz für die Programmrepräsentation hin untersucht:

- Die untypisierten Graphen (also gerichteten Graphen mit Mehrfachpfeilen) als Grundvariante;
- Typisierung zur Unterscheidung verschiedener Knoten- und Pfeilarten, ähnlich Ecore-Klassen und -Referenzen;
- Attributierung zur Repräsentation von Namen und Literalen, ähnlich Ecore-Attributen;
- Containment zur Repräsentation der hierarchischen Struktur von Programmen, ähnlich Ecore-Containment-Referenzen;
- Ordnung zur Repräsentation der Anordnung von Programmelementen, ähnlich zu geordneten Ecore-Referenzen;
- Vererbung zur Abstraktion und Vermeidung von Redundanz in Typgraphen, ähnlich zur Vererbung in Ecore;
- Constraints zur Sicherstellung von Konsistenzeigenschaften, die in Ecore durch Multiplizitäten und Eindeutigkeitsbedingungen formuliert werden.

3.2.1. Untypisierte Graphen

Pfeile: Eigenständig, mehrfach, gerichtet

Die in der algebraischen Graphtransformationstheorie benutzten Graphen unterscheiden sich von den in der Graphentheorie üblichen. Pfeile (Kanten) haben eine eigenständige Identität wie die Knoten, definieren sich also nicht nur über die Knoten, die sie verbinden. Daraus ergibt sich auch, dass zwischen zwei Knoten mehrere Pfeile erlaubt sind. Außerdem sind die Pfeile immer gerichtet, d.h. jeder Pfeil hat einen Anfangsknoten und einen Endknoten. In solchen Graphen können Objekte eines Anwendungsbereiches als Knoten und die Beziehungen zwischen den Objekten als Pfeile dargestellt werden.

```
class MyClassB {
  final void bar() {
    println("foo");
  }}
```

Abbildung 8: Einfaches Programmbeispiel: Java-Klasse mit Methode

Beispiel: Untypisierter Graph

Bereits mit dieser Grundvariante kann man Programmstrukturen modellieren, etwa die Struktur des Codebeispiels aus Abb. 8. In dem Graph in Abb. 9 wird beispielsweise die Klasse `MyClassB` als Knoten cMyClassB repräsentiert, die Methode `bar` als Knoten mBar und die Tatsache, dass `bar` ein Member von `MyClassB` ist, durch einen Pfeil von cMyClassB nach mBar, und so weiter.

Die formale Beschreibung der untypisierten Graphen findet sich in Unterabschnitt 4.3.1.

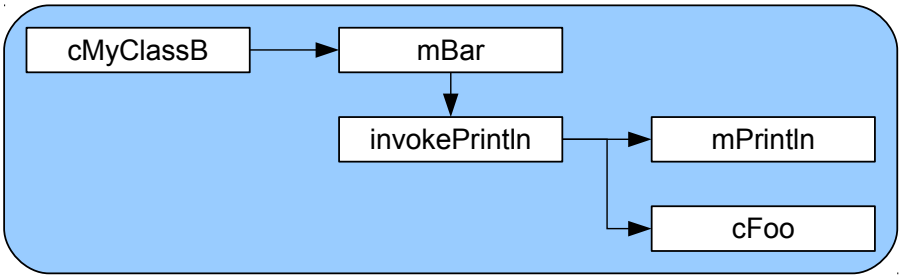


Abbildung 9: Graph als Programmmodell zu Abb. 8

Bewertung: Fehlende Informationen

Für spezielle Programmmanipulationen mag solch eine Darstellung schon ausreichend sein, aber der Formalismus soll die statische Semantik vollständig repräsentieren, was hier nicht der Fall ist: mBar repräsentiert zwar die Methode `bar`, aber die Tatsache, dass es sich dabei um eine Methode handelt, ist nicht repräsentiert. Noch besser ist es am Aufruf `println("foo")` zu erkennen: Das Aufrufstatement an sich ist als Knoten invokePrintln repräsentiert, die aufgerufene Methode als mPrintln und der Parameter `"foo"` als mFoo. mPrintln und mFoo sind auf ununterscheidbare Art durch einen Pfeil mit invokePrintln verbunden. Die Information, dass mPrintln die Rolle der aufgerufenen Methode und mFoo die Rolle eines Parameters spielt, ist nicht vorhanden, obwohl sie ganz entscheidend für die statische Semantik ist. Es fehlen Modellierungselemente, die den Klassen und Referenzen von Ecore entsprechen.

Prinzipiell könnte man versuchen, diese Rolle durch künstliche Knoten und Pfeile in den Graphen hineinzucodieren. Abgesehen davon, dass sie den Graphen unnötig aufblähen und die eigentliche Bedeutung verschleiern würde, würde durch eine derartige Codierung die Struktur der Software verloren gehen. Die Strukturerhaltung ist nun aber ein wichtiger Grund, überhaupt Graphen für Softwaremodellierung einzusetzen, so dass ein solcher Ansatz nicht zweckdienlich wäre.

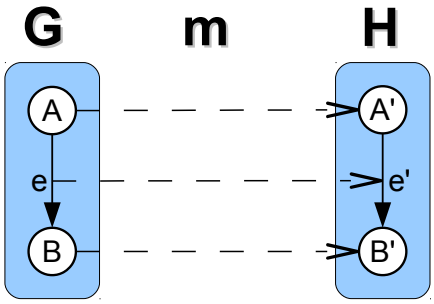


Abbildung 10: Abbildungsregel für Graphhomomorphismen

Informationen zuordnen: Homomorphismen

Um die fehlenden Informationen unmittelbar zu repräsentieren, ist deshalb ein Mechanismus notwendig, der sie den Knoten und Pfeilen zuzuordnet. Graphhomomorphismen (auch: Graphmorphismen) können diese Aufgabe leisten, soweit sich die zuzuordnenden Informationen mit Graphen ausdrücken lassen – daraus ergibt sich der Begriff des typisierten

Graphen (siehe Unterabschnitt 3.2.2).

Bei einem Graphhomomorphismus handelt es sich um eine Abbildung zwischen zwei Graphen. Jeder Pfeil wird auf einen Pfeil und jeder Knoten auf einen Knoten abgebildet. Dabei muss der Quellknoten jedes Pfeiles auf den Quellknoten seines Bildpfeiles und der Zielknoten jedes Pfeiles auf den Zielknoten seines Bildpfeiles abgebildet werden, wie Abb. 10 verdeutlicht.

Eine formale Definition findet sich in Unterabschnitt 4.3.1.

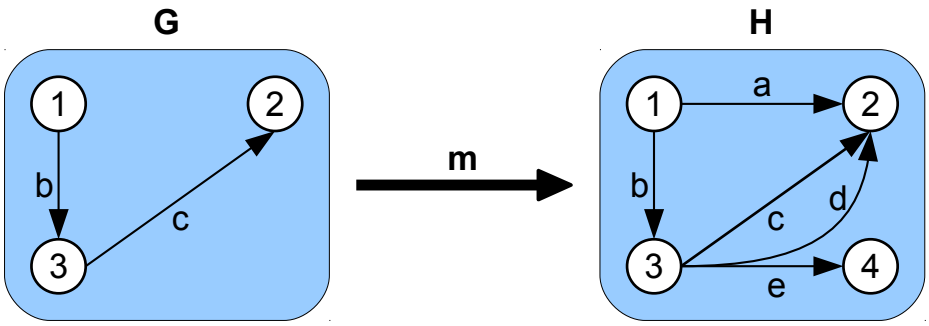


Abbildung 11: Graphhomomorphismus (1)

Beispiele: Graphhomomorphismen

Abbildung 11 illustriert einen Graphhomomorphismus m von einem Graphen G in einen Graphen H. Um den Graphhomomorphismus übersichtlich abzubilden, sind die Knoten und Pfeile von H mit Zahlen bzw. Kleinbuchstaben markiert. Die Knoten und Pfeile von G sind jeweils mit den Namen ihrer Bilder in H markiert, d.h. beispielsweise wird der rechte untere Knoten von G auf den Knoten 2 von H abgebildet.

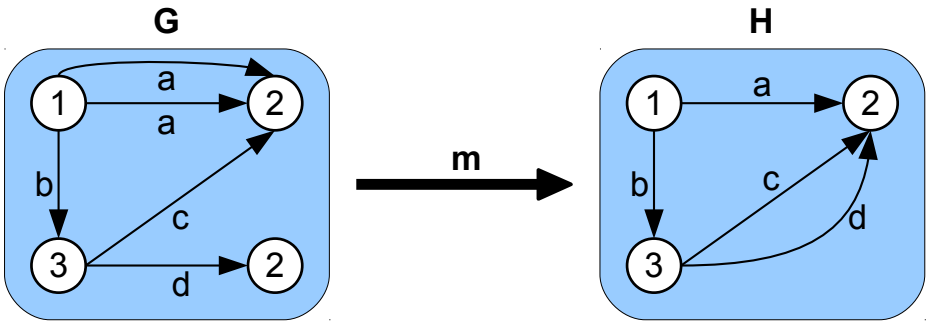


Abbildung 12: Graphhomomorphismus (2)

Injektivität

Bei Graphhomomorphismus (1) wird keinen zwei Knoten derselbe Bildknoten und keinen zwei Pfeilen derselbe Bildpfeil zugeordnet, d.h. er ist injektiv, ein Monomorphismus. Dies muss nicht immer der Fall sein, wie Abb. 12 beweist. Dort werden nämlich sowohl Knoten als auch Pfeile miteinander verschmolzen (zu Knoten 2 bzw. Pfeil a). Allerdings

Surjektivität

hat Graphhomomorphismus (2) eine andere Eigenschaft, nämlich dass jeder Knoten und jeder Pfeil in H auch ein Urbild in G hat, d.h. er ist surjektiv, ein Epimorphismus.

Eine weitere wichtige Eigenschaft mancher Homomorphismen ist Isomorphie. Diese besagt, dass die zwei Graphen strukturgleich sind, also dass sie gleich viele Knoten und Pfeile haben und die Pfeile die Knoten in gleicher Weise verbinden. Der Isomorphismus ordnet dabei jedem Knoten und jedem Pfeil seine Entsprechung im anderen Graphen zu. Isomorphie ist deshalb sehr bedeutend, weil es Graphen gibt, die dasselbe ausdrücken, aber trotzdem nicht gleich sind, weil die Mengen der Knoten und Pfeile unterschiedlich gewählt wurden. *Isomorphie*

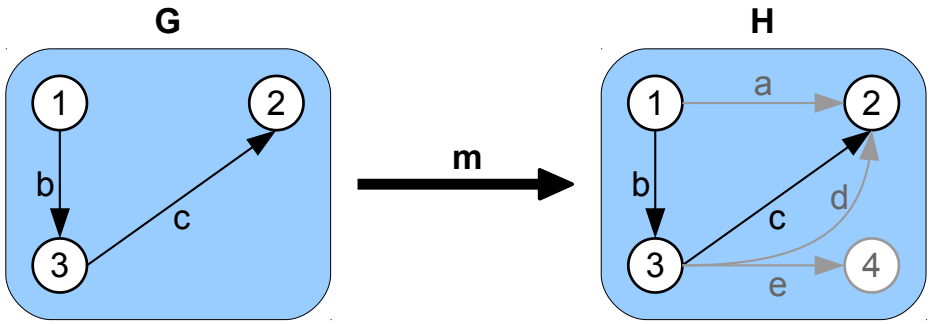


Abbildung 13: Graphhomomorphismus (1) als Vorkommen von G in H

Homomorphismen können unterschiedliche Rollen spielen und man kann sie auf verschiedene Arten interpretieren. Zwei dieser Betrachtungsweisen sind bereits bekannt: *Interpretation*

- **Zuordnung:** Wie bereits beschrieben ordnet ein Graphhomomorphismus m von G nach H jedem Knoten und jedem Pfeil von G eine Information zu. Dies ermöglicht es etwa, zwischen verschiedenen Knoten- und Pfeiltypen zu unterscheiden.

- **Vorkommen:** Wenn man in Abb. 11 die Knoten und Pfeile in H ausblendet, die nicht im Bild von m enthalten sind, ist zu erkennen, dass es sich bei einem Graphhomomorphismus um eine Art Vorkommen von G in H handelt (siehe Abb. 13). Jeder Knoten und jeder Pfeil von G hat eine Entsprechung in H - im Fall eines injektiven Graphhomomorphismus sogar eine eindeutige.

- **Schranke:** Weiterhin bedeutet die Existenz eines Graphhomomorphismus m von G nach H, dass G eine untere Schranke für H und H eine obere Schranke für G ist: In H muss alles vorkommen, was in G ist, und in G kann nur vorkommen was auch eine Entsprechung in H hat.

- **Hinzufügen und Verschmelzen:** Außerdem kann man mit Graphhomomorphismen das Hinzufügen und Verschmelzen von Knoten und Pfeilen modellieren: In H sind diejenigen Knoten und Pfeile von H neu hinzugefügt, die kein Bild eines Knotens bzw. eines Pfeiles von G sind, also außerhalb des Vorkommens von G in H liegen. Diejenigen Knoten und Pfeile von H, die Bild mehrerer Knoten bzw. Pfeile sind, kann man als Verschmelzung dieser Knoten bzw. Pfeile betrachten.

- **Entfernen und Vervielfältigen:** Wenn man das Hinzufügen und Verschmelzen in umgekehrter Richtung betrachtet, ergibt sich daraus das Entfernen und Vervielfältigen von Knoten und Pfeilen: In G sind diejenigen Knoten und Pfeile von H entfernt,

die kein Bild eines Knotens bzw. eines Pfeiles von G sind, also außerhalb des Vorkommens von G in H liegen. Diejenigen Knoten und Pfeile von H, die Bild mehrerer Knoten bzw. Pfeile von G sind, werden zu diesen Knoten bzw. Pfeile vervielfältigt.

3.2.2. Typisierung

In Unterabschnitt 3.2.1 wurde festgestellt, dass Graphen in der Grundvariante nicht ausreichen, um die statische Semantik von Programmen zu modellieren. Dies zeigte sich darin, dass im Graph keine Informationen darüber repräsentiert waren, welche Knoten welche Arten von Sprachkonstrukten repräsentieren und welche Pfeile welche Rollen spielen. In den typisierten Graphen, einer formalen Graphvariante der algebraischen Graphtransformationstheorie erfüllen Knoten- bzw. Pfeiltypen diese Aufgaben.

Um Knoten- und Pfeiltypen zu definieren, ist keine neue Sprache nötig. Stattdessen können sie einfach mittels eines untypisierten Graphen definiert werden, den man dann als Typgraphen bezeichnet. Ein Typgraph ist vergleichbar mit einem oder Ecore-Modell (siehe Abschnitt 3.1), das nur Klassen und Referenzen (binäre Assoziationen) enthält. Die Knoten des Typgraphen werden Knotentypen genannt und sie entsprechen etwa den Ecore-Klassen, die Pfeile werden als Pfeiltypen bezeichnet, und sie spielen eine ähnliche Rolle wie Referenzen. Ein typisierter Graph besteht aus einem Instanzgraphen, einem Typgraphen und einem Typisierungshomomorphismus vom Instanzgraphen in den Typgraphen, der jedem Knoten einen Knotentyp und jedem Pfeil genau einen Pfeiltyp zuordnet. Eine formale Beschreibung findet sich in Unterabschnitt 4.3.2.

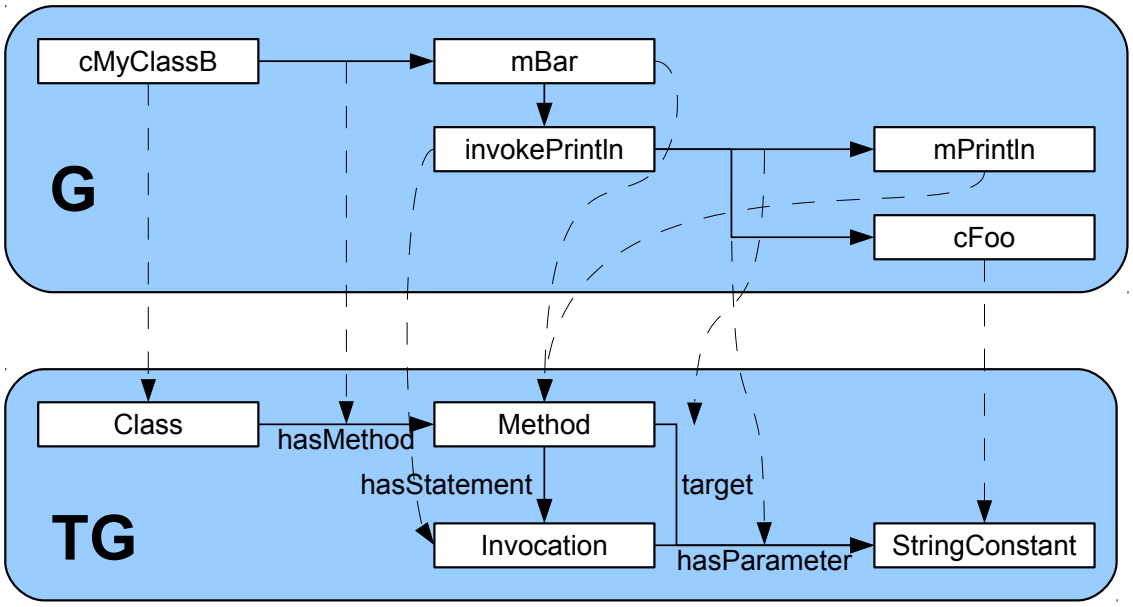


Abbildung 14: Beispiel mit typisiertem Graph (explizit)

Beispiel: Typisierter Graph

Abb. 14 zeigt eine Möglichkeit, das Beispiel aus Abb. 8 mit typisierten Graphen zu modellieren. Da diese explizite Art, einen typisierten Graphen darzustellen, nicht besonders gut lesbar ist, wird im Folgenden eine UML-ähnliche Notation wie in Abb. 15 verwendet: Knoten werden mit „Name : Knotentyp“ und Pfeile mit ihrem Pfeiletyp bezeichnet.

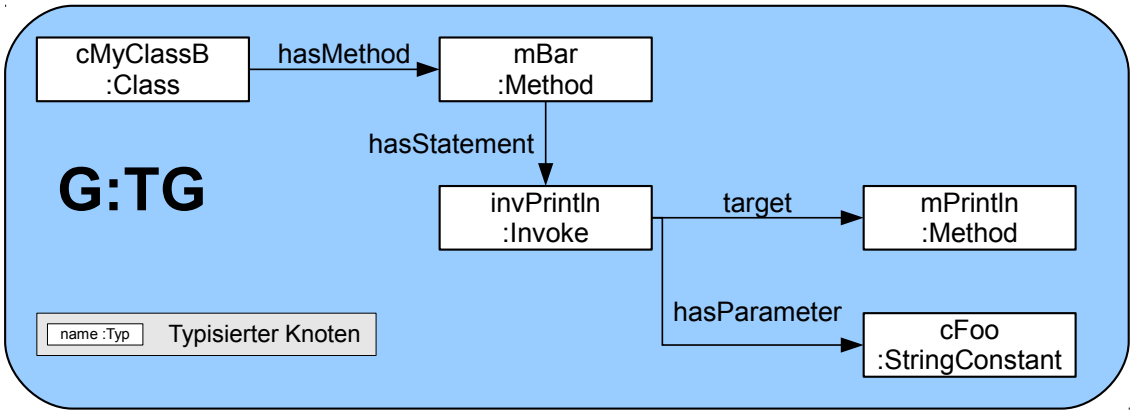


Abbildung 15: Beispiel mit typisiertem Graph

Konzeptuelle Semantik

Im Beispiel ist auch bereits zu erkennen, dass die statische Semantik bei typisierten Graphen berücksichtigt wird. Es handelt sich nicht um rein synaktische Informationen, wie an dem Pfeil target zu sehen ist: Es wird bereits repräsentiert, dass sich der Aufruf auf eine Methode namens println bezieht, also dass der Aufruf aufgelöst wird. In einem einfachen abstrakten Syntaxbaum würde im Gegensatz dazu zum Knoten invPrintln ein Attribut für den Namen der aufgerufenen Methode hinzugefügt.

Homomorphismen typisierter Graphen

Für alle Grapharten, die in diesem Kapitel definiert werden, gibt es einen Homomorphismusbegriff. Im Fall von typisierten Graphen beschränkt sich dieser Begriff auf Homomorphismen zwischen zwei Graphen, die über dem selben Typgraphen typisiert sind. Ein Homomorphismus typisierter Graphen ist ein Homomorphismus von Graphen, bei dem das Bild jedes Knoten und jedes Pfeiles denselben Typ hat wie das Urbild. Das heißt beispielweise, dass ein Knoten vom Typ Method immer auf einen Knoten vom Typ Method abgebildet werden muss.

Bewertung

Typisierte Graphen sind ein Mittel zur Repräsenation eines Teiles der statischen Semantik von Programmen. Wenn man sie wie in diesem Unterabschnitt vorgestellt verwendet, modellieren die zugehörigen Typgraphen die Sprachelemente der jeweiligen Programmiersprache und ihre Beziehungen. Deshalb kann man typisierte Graphen ähnlich wie Feature Structure Trees (siehe Abschnitt 2.3) als sprachübergreifendes Programmmodell betrachten. Als solches eignen sie sich, um das Ziel einer sprachunabhängigen Repräsentation von Programmen zu erreichen.

Mit Typgraphen kann man festlegen, welche Arten von Knoten und Pfeilen in seinen Instanzen unterschieden werden, also darin enthalten sein *dürfen*. Durch die Einbeziehung von Typisierung wird ein sehr grundlegender Teil von Ecore formalisiert, denn Knoten- und Pfeiltypen in Typgraphen bilden eine Analogie zu Klassen und Nicht-Containment-Referenzen in Ecore-Metamodellen.

Zu beachten ist, dass in typisierten Graphen immer eine beliebige Anzahl von Pfeilen eines bestimmten Typs von einem Knoten ausgehen darf. Dies ein bedeutender Unterschied zu Referenzen in Java und in Ecore: In Java kann eine Referenz immer einen oder gar keinen (`null`) Wert haben. In Ecore kann eine Referenz auch mehrere Werte haben, wobei man im Modell die erlaubte Anzahl von Werten durch Multiplizitäten nach oben oder unten beschränken kann. Zwar kann man TG aufgrund des Typisierungshomomorphismus type

im Sinne von Unterabschnitt 3.2.1 als eine obere Schranke für über TG typisierte Graphen interpretieren; dies bedeutet aber nur, dass es keine Knoten und Pfeile in G geben kann, die keinem in TG vorhandenen Typ angehören. Es kann dennoch eine beliebige Anzahl von Knoten und Pfeilen jedes einzelnen Typs geben.

Es gibt in typisierten Graphen noch einige Schwachstellen, die im Folgenden erläutert werden.

Ungültige Graphen

Zunächst kann man typisierte Graphen konstruieren, die keinem korrekten Programm entsprechen: Man kann weder verlangen, dass für Knoten eines gewissen Typs bestimmte Pfeile vorhanden sein *müssen*, noch kann man die Anzahl der Pfeile, die von einem Knoten ausgehen, nach oben oder nach unten beschränken. Auch kann man mit Typgraphen beispielsweise nicht verbieten, dass Variablen außerhalb ihres Gültigkeitsbereichs referenziert werden. Hiergegen helfen Constraints, die in Unterabschnitt 3.2.7 behandelt werden.

Keine Werte

Eine gravierendere Schwachstelle ist in Abb. 14 zu erkennen: Der übergebene Parameter cFoo stellt ein String-Literal dar, aber die Information, dass dieses den Inhalt `"foo"` hat, ist nicht im Graphen repräsentiert. Deshalb kann man den Graph nicht unterscheiden von dem Graph eines Programmes, das `"bar"` anstatt `"foo"` ausgibt – die Graphen sind isomorph. Auch für die im Beispiel gar repräsentierten Klassen-, Methoden- und Variablennamen gilt dasselbe. Solange man sich nicht auf künstliche Codierungen einlässt, kann mit einem endlichen Typgraphen lediglich eine feste, endliche Menge von Schlüsselwörtern repräsentiert werden. Für Literale gibt es jedoch unendlich viele mögliche Werte. Hierdurch lässt sich von der Repräsentation eines Programmes seine dynamische Semantik nicht mehr ableiten, weshalb Typisierung allein für die Repräsentation der statischen Semantik von Programmen nicht ausreicht. Dieses Problem löst die in Unterabschnitt 3.2.3 beschriebene Attributierung.

3.2.3. Attributierung

Attribute

Die im vorherigen Unterabschnitt angesprochene Schwachstelle typisierter Graphen, dass Literale nicht ohne weiteres ausgedrückt werden können, kann man durch Attributierung lösen [Lar+07]. Diese erlaubt es, Knoten und Pfeilen Werte wie etwa Zahlen oder Strings zuzuordnen – sie spielt also eine ähnliche Rolle wie Attribute in Ecore. Die Werte sind dabei durch Datentypen typisiert und man kann mit ihnen rechnen, also z.B. Zahlen addieren oder Strings zusammenfügen.

Datentypen

Die Datentypen und Rechenoperationen werden entweder durch algebraische Spezifikationen [Ehr+01; EM85] oder durch Sketches [BW12] definiert. Eine algebraische Spezifikationen bzw. ein Sketch definiert eine Menge von Datentypen (in algebraischen Signaturen Sorten genannt), Konstanten- und Funktions- bzw. Operationssymbolen. Aus diesen Symbolen kann man dann Terme zusammensetzen. Für eine genauere Beschreibung sei auf die Literatur verwiesen. Sketches werden in dieser Arbeit verwendet, um Graphfeatures zu definieren und werden deshalb in Abschnitt 4.2 erläutert.

Beispiel: Attributierte typisierte Graphen

Wenn man Attributierung mit Typisierung verbindet, erhält man attributierte typisierte Graphen. Damit lässt sich das am Anfang dieses Abschnitts genannte Problem lösen, wie in Abb. 16 zu sehen ist: Die verwendete Algebra enthält eine Sorte für Strings, mit der sich dann das Stringliteral `"foo"` repräsentieren lässt.

Bewertung

Attributierte typisierte Graphen sind zur Repräsentation der statischen Semantik von Programmen hinreichend, weil sich mit ihnen Abstract Semantic Graphs (ASGs) [RW91;

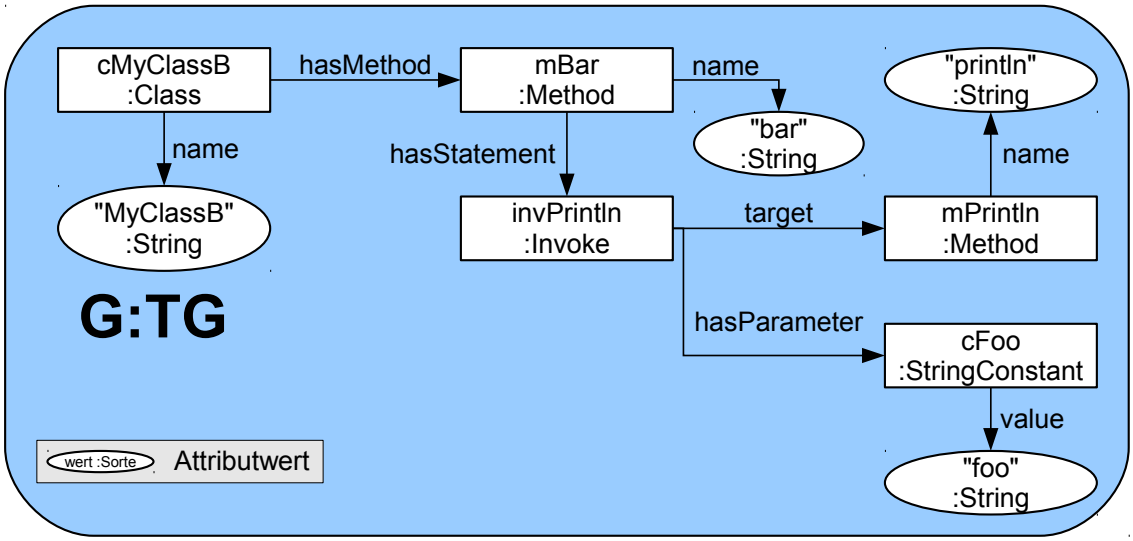


Abbildung 16: Beispiel mit attributiertem typisiertem Graph

Rag03; Rag+04] darstellen lassen. Bei diesen handelt es sich nämlich um eine graphbasierte Repräsentation der statischen Semantik von Programmen. Zu ASGs gelangt man, indem man abstrakte Syntaxbäume (ASTs) [Aho+06] mit zusätzlichen Pfeilen, sogenannten Nicht-Baumpfeilen (engl. *non-tree-edges*) anreichert, die die Baumstruktur nicht berücksichtigen müssen, und eventuell von semantisch irrelevanten Informationen abstrahiert. Es gibt zwei Merkmale von ASGs, die sich in attributierten typisierten Graphen nicht unmittelbar wiederfinden:

- In attributierten typisierten Graphen wird nicht zwischen Baumpfeilen und Nicht-Baumpfeilen unterschieden. Diese Information ist aber nicht für die Bedeutung des Programms relevant.
- Bei ASGs wird die Reihenfolge der Kindknoten im Baum mitrepräsentiert. In attributierten typisierten Graphen kann man das aber mit Pfeilattributen oder zusätzlichen Pfeiltypen (“next”) simulieren.

Da diese beiden Abweichungen nicht bedeutend sind und alle semantisch relevanten Informationen enthalten sind, lässt sich die Aussage vertreten, dass attributierte typisierte Graphen zur Repräsentation der abstrakten Semantik von Programmen bereits ausreichen. Ungeachtet dessen gibt es für beide Unterschiede Graphfeatures, die die jeweilige Information unmittelbar zum Graph hinzufügen, nämlich Containment (Unterabschnitt 3.2.4) bzw. Ordnung (Unterabschnitt 3.2.5).

Da der Formalisierung von Containment Vorrang gegeben wurde, wird Attributierung in dieser Arbeit nicht formalisiert, um den Stoffumfang einzuschränken.

3.2.4. Containment

Hierarchie

In strukturierten Programmiersprachen sind Programme hierarchisch organisiert. In Java enthalten beispielsweise Pakete Unterpakete und Klassen, Klassen enthalten Methoden

und Felder, Methoden enthalten Blöcke, Blöcke enthalten Statements und so weiter. Diese baumartige Hierarchie spielt für die statische Semantik eine Rolle, etwa für Gültigkeitsbereiche von Variablen und die Zugreifbarkeit von Feldern und Methoden, und deshalb macht es Sinn, sie bei der Programmrepräsentation mit Graphen zu berücksichtigen. Außerdem lassen sich etwa im Rahmen der in Unterabschnitt 3.2.3 angesprochen abstrakten Semantikgraphen Algorithmen auf der Baumstruktur ausführen, die schneller sind als Algorithmen auf allgemeinen Graphen, aber durch die zusätzlichen Informationen der nicht zum Baum gehörigen Pfeile bessere Ergebnisse liefern als Algorithmen, die auf dem abstrakten Syntaxbaum (AST) operieren [Rag03; Rag+04]. Bei manchen Anwendungen wie Diff und Merge kann es im Rahmen eines Tradeoffs zwischen Geschwindigkeit und Ergebnisqualität sogar Sinn machen, ausschließlich Baumpfeile zuzulassen.

Containment

Bei Graphen mit Containment (engl. "to contain" enthalten) [KLT07; BET08] wird diese Baumstruktur durch eine spezielle Art von Pfeilen ausgedrückt, nämlich durch sogenannte Containment-Pfeile. Containment drückt aus, welche Knoten in welchen anderen enthalten sind. Deshalb dürfen Containment-Pfeile in Instanzgraphen im Gegensatz zu normalen Pfeilen keine Zyklen bilden und kein Knoten darf Ziel zweier unterschiedlicher Containment-Pfeile sein. Hingegen dürfen die Containment-Pfeiltypen genauso wie normale Pfeiltypen Zyklen bilden und Knotentypen auch Ziel mehrerer unterschiedlicher Pfeiltypen sein, wenn Containment und Typisierung kombiniert werden. Meist werden nicht nur Containmentpfeile erlaubt, sondern zusätzlich auch normale Pfeile, die dann als Querpfeile bezeichnet werden. Formal wird Containment in Unterabschnitt 4.3.3 definiert.

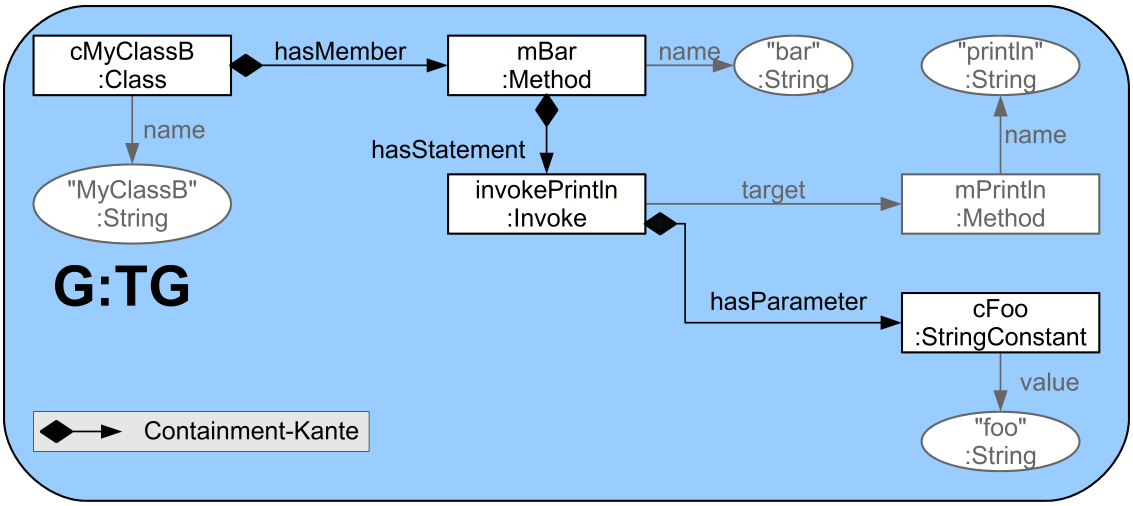


Abbildung 17: Typisierter Graph mit Containment
(Containment-Hierarchie hervorgehoben)

Beispiel: Containment

Den Code aus Abb. 8 kann in typisierten Graphen mit Containment wie in Abb. 17 modelliert werden. Im Wesentlichen drücken die Containmentpfeilen hier aus, welche Sprachelemente in welchen anderen "physisch" enthalten sind. Beispielsweise ist die Definition der Methode `bar` in der Definition der Klasse `MyClassB` enthalten. Deshalb ist der entsprechende Pfeiltyp hasMember ein Containment-Pfeiltyp.

Homomorphismus von Graphen mit Containment

Für Graphen mit Containment gibt es zwei mögliche Homomorphismusbegriffe: Die eine entspricht den normalen Graphhomomorphismen, bei der anderen wird zusätzlich verlangt,

dass ein Knoten mit Verschachtelungstiefe n wieder auf einen Knoten mit Verschachtelungstiefe n abgebildet wird. Für diese Arbeit wurde trotz ihrer Einschränkungen letztere Möglichkeit gewählt, da sie bessere mathematische Eigenschaften mit sich bringt.

Bewertung

Containment macht die hierarchische Struktur von Programmen explizit und erlaubt es, die Containment-Referenzen von Ecore umzusetzen. Außerdem lassen sich durch die Baumstruktur Algorithmen auf heuristische Weise beschleunigen [Rag03]. Aber Containment kann auch einen grundlegenderen Unterschied in der Programmrepräsentation machen, denn man kann auch ausschließlich Containmentpfeile erlauben, wodurch die Graphen im Endeffekt Wälder sind, also eine Menge von Bäumen. Anwendungen wie etwa Diff können ebenso auf Grundlage von allgemeinen Graphen wie auch auf Grundlage von Bäumen definiert werden. Je nach Grundlage unterscheiden sich die Algorithmen in ihrer Komplexität, aber auch in der Qualität der Ergebnisse. Beim Diff von Graphen sind eine längere Laufzeit, aber auch bessere Ergebnisse zu erwarten als bei Bäumen, die etwa keine Namensauflösung erlauben. Auch damit man solche Unterschiede untersuchen kann, soll Containment im Rahmen dieser Arbeit formalisiert werden.

3.2.5. Ordnung

Die Anordnung von Sprachelementen im Programmtext ist für die Semantik von Programmen teils von Bedeutung, wie etwa bei Statements. Sie kann bereits mit typisierten Graphen mit normalen Pfeiltypen repräsentiert werden.

Es macht aber Sinn, sie als Graphfeature in den Formalismus selbst zu integrieren, weil eine Ordnung für Algorithmen den Unterschied zwischen exponentieller und polynomieller Laufzeit ausmachen kann [Leß12]. In anderen Fällen ist die Anordnung von Sprachelementen auch nicht relevant, etwa bei der Anordnung von Java-Methoden innerhalb einer Klasse. Da möglichst nur semantisch relevante Aspekte eines Programms formal repräsentiert werden sollten, sollte die Anordnung nur optional sein.

Es gibt mehrere grundsätzliche Möglichkeiten, die Modellierung von Anordnungen zu erlauben. In Ecore kann man Referenzen als geordnet definieren. Wenn in einem Ecore-Modell eine Referenz r von einer Klasse C_1 zu einer Klasse C_2 definiert wird, enthält die generierte Java-Klasse C_1 eine Liste r vom Typ List$\langle C_2 \rangle$. In die Sprache der typisierten Graphen übersetzt hieße dies, dass alle von einem Knoten ausgehenden Pfeile gleichen Pfeiltyps eine Liste bilden, d.h. relativ zueinander angeordnet sein müssen.

Hier wurde eine andere Repräsentation der Ordnung gewählt, bei der man Pfeile, die vom selben Knoten ausgehen, relativ zueinander anordnen kann, aber nicht muss. Bei der Kombination mit Typisierung kann man dabei auch die Reihenfolge zwischen Pfeilen unterschiedlicher Pfeiltypen repräsentieren, wenn dies im Typgraphen erlaubt wird. Warum dies sinnvoll ist, illustrieren die folgenden Beispiele.

Beispiel: Ordnung

Wenn man im Beispiel in Abb. 18a die Definitionen der Felder `x` und `y` austauscht, entsteht ein ungültiges Programm. Die Methoden `foo` und `bar` könnte man hingegen vertauschen, ohne dass sich die Bedeutung des Programms ändert – und dies ist bei Java-Methoden immer der Fall. Deshalb muss man die Reihenfolge von Feldern auf jeden Fall repräsentieren können, während es bei Methoden sinnvoll wäre, die Reihenfolge nicht unnötigerweise festzulegen.

Ein etwas komplizierter Fall liegt bei den Statements der Methode `f` in Abb. 18b vor:

```
class SomeClass1 {
  private int x = 10;
  private int y = 5 * x;

  void foo() {
    // ...
  }

  void bar() {
    // ...
  }
}
```

(a) Zwischen Feldern und zwischen Methoden

```
class SomeClass2 {
  void f() {
    int a;
    int b;
    b = 2;
  }
}
```

(b) Zwischen Zuweisungen und Definitionen

Abbildung 18: Relevanz der Anordnung von Programmelementen

Man könnte die Variablendefinitionen "`int a;`" und "`int b;`" austauschen[1]. Jedoch darf die Zuweisung `b = 2` nicht vor die Deklaration "`int b;`" gezogen werden, da ansonsten ein ungültiges Programm entsteht. Hier ergibt sich also die Situation, dass die relative Reihenfolge von Variablendeklarationen untereinander irrelevant ist, aber die zwischen Variablendeklarationen und Statements nicht. Daher ist es sinnvoll, dass die relative Reihenfolge zwischen Variablendeklarationen und Statements repräsentiert werden darf, die von Variablendeklarationen ohne Initialisierung untereinander jedoch nicht.

Eine mögliche Repräsentation der Beispieles aus Abb. 18b ist in Abb. 19 zu sehen: Die Definition der Variable `b`, repräsentiert durch den Knoten defB muss vor der Zuweisung `b = 2`, repräsentiert durch den Knoten assignB2 stehen, was durch den gepunkteten Pfeil vom mittleren zum rechten hasStatement-Pfeil dargestellt wird.

Bewertung

Die gewählte Variante deckt die im Beispiel illustrierten Fälle ab, soweit sich die erlaubten Anordnung aus den Typen ergeben. Sie ermöglicht es, relative Anordnungen beliebig zu erlauben und zu verbieten, solange sie sich nicht transitiv aus anderen erlaubten Anordnungen ergeben.

Es wären auch grobgranularere Varianten denkbar: Es gibt eine Graphvariante mit Listen von Knoten als Pfeilen [MR10], ähnlich wie bei Ecore. Diese Art von Ordnung macht bei der Programmrepräsentation mit typisierten Graphen nur in Verbindung mit Vererbung (siehe Unterabschitt 3.2.6) Sinn, denn man kann nur Anordnungen von Knoten repräsentieren, die einen gemeinsamen Obertyp haben. Knoten gleichen Typs kann man immer anordnen – im dem Beispiel in Abb. 18b müsste man sogar zwangsläufig die Anordnung der Variablendefinitionen untereinander repräsentieren, um die Anordnung relativ zu der Zuweisung repräsentieren zu können.

Eine noch grobgranularere Alternative wäre es, nur zwischen geordneten und ungeordneten Pfeiltypen zu unterscheiden. Dadurch bilden alle von einem Knoten ausgehenden geordneten Pfeile unabhängig vom Typ eine Liste, sodass auf Vererbung verzichtet werden kann. Allerdings wird hier noch mehr unnötige Anordnungsinformation erlaubt.

Im Vergleich zu Ecore ist die gewählte Variante wesentlich feingranularer. Aber ähn-

[1] Dies immer möglich solange sie keine Initialisierungswerte haben.

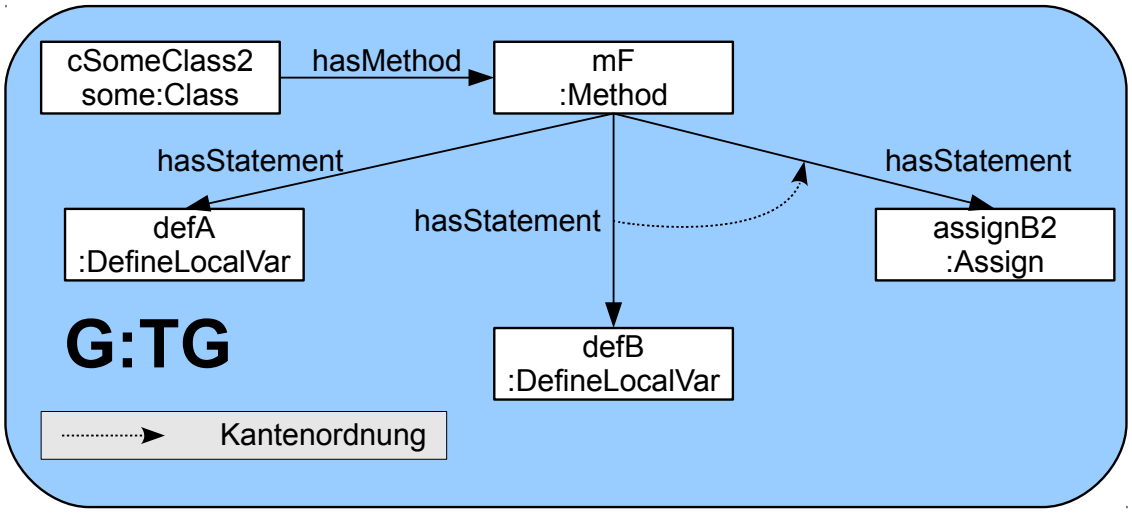


Abbildung 19: Graph zu Abb. 18b

lich wie Ecore es erlaubt, Referenzen als ungeordnet zu definieren, wird hier erlaubt, die Repräsentation einer Ordnung in Graphen zu verbieten. Im Gegensatz zu Ecore, wo bei geordneten Referenztypen eine Liste vorliegt, müssen Pfeile, die angeordnet werden dürfen, nicht zwangsläufig auch tatsächlich angeordnet werden. Um dies zu erzwingen, wären hier Constraints nötig (siehe Abschnitt 3.2.5).

Das Graphfeature Ordnung wird im Rahmen dieser Arbeit nicht formalisiert.

3.2.6. Vererbung

Redundanz im Typgraphen

Um statische Semantik von Programmen auszudrücken, sind typisierte attributierte Graphen bereits ausreichend mächtig. Ein Nachteil an ihnen ist aber, dass gleichartige Pfeiltypen, die von verschiedenen Knotentypen ausgehen bzw. auf verschiedene Knotentypen zeigen, nicht zusammengefasst werden können. Dadurch werden Typgraphen leicht redundant. In Ecore wird dies durch Vererbung zwischen Klassen gelöst. Für Graphen gibt es eine formale Variante der Vererbung [Lar+05].

Lösung: Vererbung

Vererbung führt eine Ober-Untertypbeziehung zwischen Knotentypen ein. Pfeile können dann von jedem Untertyp ihres Quelltyps zu jedem Untertyp ihres Zieltyps gehen. Hierdurch kann man gemeinsame Obertypen der Quell- bzw. Zielknotentypen der gleichartigen Pfeiltypen einführen und diese zu einem einzigen Pfeiltyp zwischen den Obertypen zusammenfassen. Desweiteren gibt es die Möglichkeit, Knotentypen als abstrakt zu definieren, so dass kein Knoten eine unmittelbare Instanz dieses Typs sein kann, sondern Instanz eines nicht-abstrakten Untertypen sein muss.

Beispiel: Vererbung

Um den Unterschied zwischen Typgraphen ohne und mit Vererbung zu verdeutlichen, kann man die entsprechenden Modellierungen von Variablen und Referenzen in Java in Abb. 20 bzw. Abb. 21 vergleichen:

In dem Typgraphen in Abb. 20 sind verschiedene Arten von Variablen und Referenzen mit unterschiedlichen Knoten- und Pfeiltypen modelliert. Diese Arten von Variablen haben gemeinsam, dass sie einen Typ haben (“hasType”) und dass sie referenziert werden können (“references”). Die Modellierung in diesem Beispiel ist deshalb redundant, weil jeweils

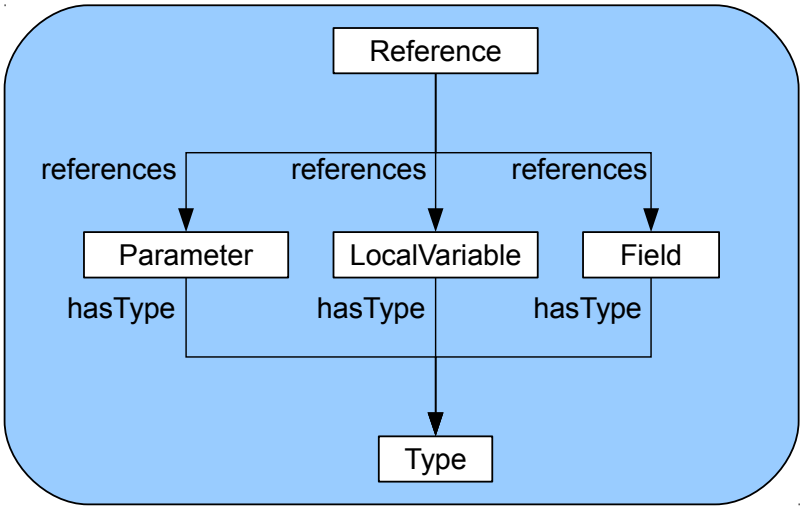


Abbildung 20: Beispiel: Referenzen ohne Vererbung

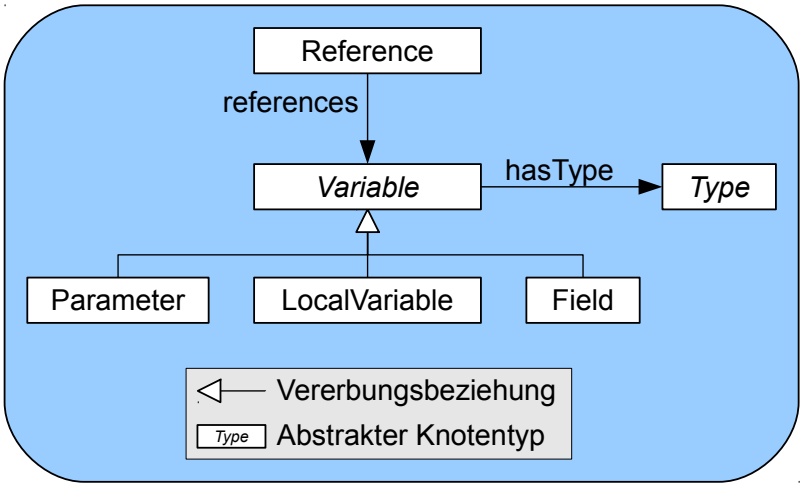


Abbildung 21: Beispiel: Referenzen mit Vererbung

drei verschiedene Pfeiltypen “hasType” bzw. “references” vorhanden sein müssen, die aber jeweils dasselbe ausdrücken.

Vererbung ermöglicht es, diese Gemeinsamkeiten unmittelbar zu modellieren und dadurch diese Redundanz zu vermeiden. Dazu kann man einen Ober-Knotentyp für Variablen definieren, von dem alle drei Variablenarten erben (siehe Abb. 21). Für “hasType” bzw. “references” gibt es dann jeweils nur einen Pfeiltyp, der vom gemeinsamen Ober-Knotentyp ausgeht bzw. auf ihn zeigt.

Bewertung

Vererbung ist ein für praktische Zwecke sehr nützliches Graphfeature, da sie Redundanz in Typgraphen reduziert und diese somit leichter verständlich und besser wartbar werden. Sie fügt aber keine zusätzliche Ausdruckskraft zu den Graphen hinzu und man kann sie auf formaler Ebene mit einer einfachen Konstruktion durch typisierte Graphen ohne Vererbung ersetzen. Deshalb wird sie in dieser Arbeit nicht formal beschrieben.

3.2.7. Constraints

Mit den bisher definierten Graphfeatures kann man bereits alle Informationen eines Programms ausdrücken, aber es gibt auch Graphen, die kein gültiges Programm repräsentieren. Um dies zu verhindern, kann man die Menge der erlaubten Graphen mit Constraints [Ehr+06; TR05] einschränken.

Eine Constraint hat die Form einer Wenn-Dann-Regel. Wenn ein bestimmtes Muster im Graph vorhanden ist, muss es sich zu einem Muster aus einer “Dann”-Liste erweitern lassen, das ebenfalls im Graph vorhanden ist. Falls dies nicht möglich ist, handelt es sich um keinen gültigen Graphen. Die “Dann”-Liste kann auch leer sein, was dann heißt, dass das “Wenn”-Muster in keinem gültigen Graph vorkommen darf.

Beispiel: Constraint

Mit Constraints kann man auch Regeln einer Programmiersprache definieren, die man mit den bisher vorgestellten Graphfeatures nicht ausdrücken kann. In Graphen mit Ordnung kann man beispielsweise mit der in Abb. 22 illustrierten Constraint verlangen, dass die Statements einer Methode geordnet sein müssen: Als “Wenn”-Muster sind zwei Statement-Knoten s_1 und s_2 gegeben, die im selben Methoden-Knoten enthalten sind, ohne dass ihre Anordnung definiert wäre. Wenn dieses Muster im Graph gefunden wird, muss es zu einem von zwei “Dann”-Mustern erweitert werden können, die die beiden Möglichkeiten ausdrücken, die Statements anzuordnen: In dem einen kommt s_1 vor s_2, in dem anderen kommt s_2 vor s_1. Das entspricht gerade der Definition einer totalen Ordnung zwischen den Statements einer Methode.

Ein weiteres Beispiel für Constraints wären Multiplizitäten im Sinne von Ecore und UML. Auch kompliziertere Regeln können mit Constraints ausgedrückt werden, wie etwa die Binderegeln der Programmiersprache.

Bewertung

Mit Constraints lässt sich die statische Semantik feiner modellieren als mit Typgraphen. Insbesondere lässt sich die Existenz bzw. Nichtexistenz bestimmter Knoten und Pfeile fordern. Mit Constraints lassen sich die Multiplizitäten und die Eindeutigkeit von Referenzen aus Ecore modellieren.

Dennoch werden Constraints hier nicht formalisiert, da das den Rahmen dieser Arbeit sprengen würde.

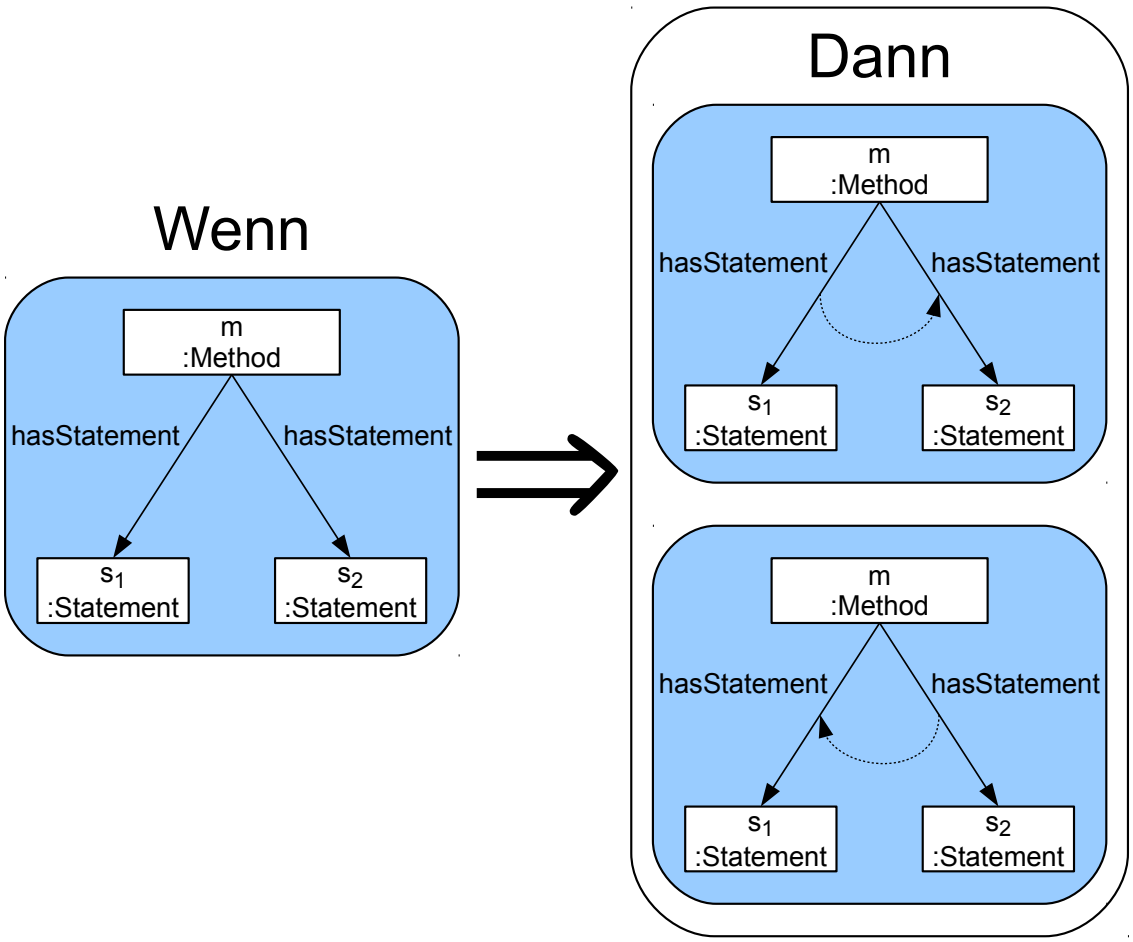


Abbildung 22: Beispiel: Constraint

3.3. Kategorientheorie für die Programmmanipulation

Die Kategorientheorie ist eine Theorie zur einheitlichen Beschreibung und Untersuchung insbesondere mathematischer, logischer und informatischer Themenfelder. Die Vereinheitlichung wird erreicht, indem Konzepte auf einer höchst abstrakten Ebene mit den Begriffen der Kategorientheorie definiert und dann auf die Themenfelder (bzw. Kategorien, die ihnen entsprechen) angewendet werden. In dieser Arbeit wird sie für zwei Dinge benötigt: Sie liefert zum einen das Rahmenwerk, um die den Programmmanipulationen zugrundeliegenden Programmeigenschaften und -beziehungen repräsentationsunabhängig zu beschreiben. Zum anderen wird sie benutzt, um die Graphfeatures zu formalisieren und so eine modulare Definition von Graphvarianten zur Programmrepräsentation zu ermöglichen.

Es folgt ein kurzer Überblick über einige für diese Arbeit relevante Teilbereiche der Kategorientheorie. Formale Definitionen werden in diesem Kapitel vermieden und auf Kapitel 4 verschoben. Eine weitergehende Einführung in die Kategorientheorie mit Bezug auf Softwaretechnologien gibt Fiadeiro [Fia05]. Barr/Wells [BW12] geben eine mathematischere und umfassendere Einführung mit Bezug auf Informatik im Allgemeinen.

3.3.1. Kategorien

Der Begriff der Kategorie ist der wichtigste Grundbegriff der Kategorientheorie. Bei einer Kategorie handelt es sich um eine mathematische Struktur, die ähnlich wie Gruppen, Ringe und Körper durch einige Axiome definiert ist. Während Gruppen, Ringen und Körpern Mengen zugrundliegen, bauen Kategorien auf (untypisierten) Graphen auf. Man kann Kategorien als Graphen mit einer zusätzlichen mathematischen Struktur betrachten, die es ermöglicht, mit Pfeilen zu "rechnen". Aus historischen Gründen bezeichnet man die Knoten einer Kategorie als Objekte und die Rechenvorschrift als Komposition. Die Pfeile werden oft als Morphismen bezeichnet.

Wie in Abb. 23 illustriert ist, dürfen zwei Pfeile f und g genau dann miteinander komponiert werden, wenn sie unmittelbar hintereinander folgen. Das Ergebnis wird hier mit $f \bullet g$ bezeichnet, in der mathematischen Literatur findet man oft $g \circ f$ und in der Informatik $f\,;g$. In einer Kategorie gibt es für jedes Objekt X einen Identitätspfeil, der von X nach X geht. Komposition mit Identitätspfeilen hat keinen Effekt: Das Ergebnis ist wieder der ursprüngliche Pfeil.

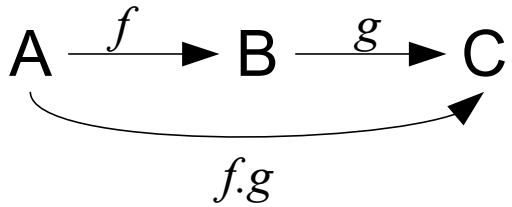


Abbildung 23: Komposition von Pfeilen (Ausschnitt aus einer Kategorie)

In Form einer Kategorie können die verschiedensten Bereiche der Mathematik beschrie-

ben werden. Zur Veranschaulichung werden im Folgenden einige Beispiele von Kategorien erläutert.

Set Ein wichtiges Beispiel ist die Kategorie **Set**, die die Mengenlehre erfasst. In dieser Kategorie sind die Mengen die Objekte und die Funktionen die Pfeile. Als Komposition wird hier die Funktionskomposition $((f \circ g)(x) = f(g(x)))$ verwendet. Im Gegensatz zur üblichen Herangehensweise der Mengenlehre interessiert man sich hier in der Regel nicht für Element- bzw. Teilmengenbeziehungen. Gleich große (isomorphe) Mengen werden als im Wesentlichen gleich betrachtet.

Graphs Ein weiteres, insbesondere für diese Arbeit wichtiges Beispiel ist die Kategorie **Graphs**, die Graphen beschreibt. Die Graphen werden als Objekte und die Graphhomomorphismen als Pfeile benutzt. Mit Hilfe dieser Kategorie lässt sich auch die Kategorie der typisierten Graphen definieren.

Logi Um klarzustellen, dass es sich bei den Objekten nicht zwangsläufig um Mengen mit Struktur handeln muss, wird als letztes Beispiel die Kategorie **Logi** vorgestellt. Diese Kategorie hat Formeln der Aussagenlogik als Objekte. Wenn aus einer Formel A eine Formel B semantisch folgt, dann gibt es genau einen Pfeil von A nach B, ansonsten keinen. Die Komposition entspricht dem Modus ponens: Wenn aus einer Aussage A eine Aussage B folgt, und aus B eine Aussage C folgt, dann folgt C aus A.

Die formalen Definitionen hierzu finden sich in Abschnitt 4.1.1.

3.3.2. Kategorielle Konstruktionen

Wie am Anfang dieses Abschnitts erklärt wurde, erreicht die Kategorientheorie eine einheitliche Beschreibung verschiedener Mathematikbereiche, indem Konzepte abstrakt mit den Begriffen der Kategorientheorie definiert werden. Das bedeutet, dass man ausschließlich von der allgemeinen Definition der Kategorie ausgehend Konstruktionen definiert. Wenn diese kategoriellen Konstruktionen auf eine bestimmte Kategorie angewendet werden, treffen sie oft auf wichtige Konstruktionen des hinter dieser Kategorie stehenden Mathematikbereiches zu.

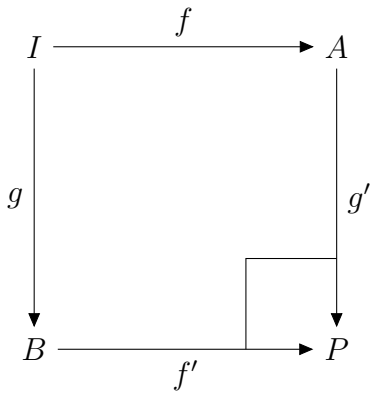


Abbildung 24: Pushout von A und B entlang I mit Pushout-Objekt P

Beispiel: Pushout Ein Beispiel einer kategoriellen Konstruktion ist der Pushout. Der Pushout entspricht in vielen Kategorien einer Verschmelzung von zwei Objekten anhand eines gemeinsamen

Unterobjektes. In diesen Fällen kann man Pushouts anhand Abb. 24 folgendermaßen erklären: A und B sind die Objekte, die miteinander verschmolzen werden sollen. I ist das gemeinsame Unterobjekt von A und B, auch Schnittstellenobjekt oder Klebeobjekt genannt. Das Ergebnis der Verschmelzung ist das Pushout-Objekt P, das in Illustrationen mit einem rechten Winkel markiert wird. Die Pfeile f und g definieren, auf welche Weise das Schnittstellenobjekt I in A bzw. B vorkommt, ähnlich wie man Graphhomomorphismen aus Unterabschnitt 3.2.1 als Vorkommen von Mustern in Graphen verstehen kann. Ebenso zeigen f' und g' an, auf welche Weise A bzw. B im Ergebnis P vorkommen.

Pushouts von Graphen

Pushouts können auf die Kategorie der Graphen und Graphhomomorphismen angewendet werden. Dort bedeutet ein Pushout, dass zwei Graphen mit einem gemeinsamen Untergraphen zu einem Graph zusammengefügt werden. Wenn man die Graphen zur Programmrepräsentation verwendet, kann dies beispielweise benutzt werden, um das Binden von Modulen oder Zusammenfügen von Programmfragmenten zu modellieren. Es handelt sich also schon um eine einfache Programmmanipulation.

In Abbildung 25 ist ein Pushout von zwei Graphen illustriert, die jeweils ein Programmfragment, also z.B. eine Java-Klassendatei, repräsentieren. Der Graph A repräsentiert eine Klasse `MyClassA`, die eine Methode `foo` mit einem Parameter `myB` vom Typ `MyClassB` enthält. In `foo` wird `myB` die in `MyClassB` definierte Methode `bar` aufgerufen. Graph B definiert die Klasse `MyClassB`, die bereits in Abschnitt 2.1 als Beispiel gedient hat. Diese definiert eine Methode `bar`, die die Methode `println` aufruft, um "foo" auszugeben. Der Schnittstellengraph I ist das Schnittstellenobjekt des Pushouts. Er enthält den gemeinsamen Untergraphen von A und B, der das gemeinsame Wissen der beiden Klassendateien repräsentiert, dass die Klasse `MyClassB` eine Methode `bar` enthält. P ist der Ergebnisgraph des Pushouts.

Eine für die Kategorientheorie typische Vorgehensweise ist, dass der gemeinsame Untergraph von A und B nicht durch gemeinsame Knoten und Pfeile festgelegt wird. Stattdessen markieren die Pfeile f und g jeweils ein Vorkommen von I in A bzw. B, wie es in Unterabschnitt 3.2.1 erklärt wurde. In Abb. 25 sind die Vorkommen von I in A, B und P jeweils orange hervorgehoben. Da die Graphen A und B durch den Pushout zum Graphen P zusammengefügt werden, finden auch sie sich in P wieder. Diese Vorkommen in P werden durch die Pfeile g' bzw. f' markiert. g' muss das Vorkommen von I in A auf gleiche Weise in P abbilden, wie f' das Vorkommen von I in B im Graphen P abbildet. In Abb. 25 sind die nur in A vorhandenen Knoten und Pfeile blau hervorgehoben, die von B grün.

Universelle Eigenschaft des Pushout

Zwei wesentliche Eigenschaften von Pushouts lassen sich dadurch in Abb. 25 erkennen:

1. In P werden keine unnötigen neuen Knoten und Pfeile hinzugefügt – es kommen nur Knoten und Pfeile vor, die bereits in A oder B vorkommen.

2. Es werden nur solche Knoten und Pfeile in P gleichgesetzt, die gleichgesetzt werden müssen – im vorliegenden Fall nur die Knoten und Pfeile des Schittstellengraphs I.

Diese beiden Eigenschaften werden zusammen als die universelle Eigenschaft des Pushout von Graphen bezeichnet. Universelle Eigenschaften finden sich bei den meisten kategoriellen Konstruktionen wieder. Konstruktionen mit universellen Eigenschaften bezeichnet man als universelle Konstruktionen. Man kann universelle Eigenschaften auch auf eine kategorienunabhängige Weise formulieren, wodurch es möglich ist, den Begriff der universellen Konstruktion durch die Begriffe des Limes und des Kolimes zu formalisieren. Der oben vorgestellte Pushout ist ein Spezialfall des Kolimes.

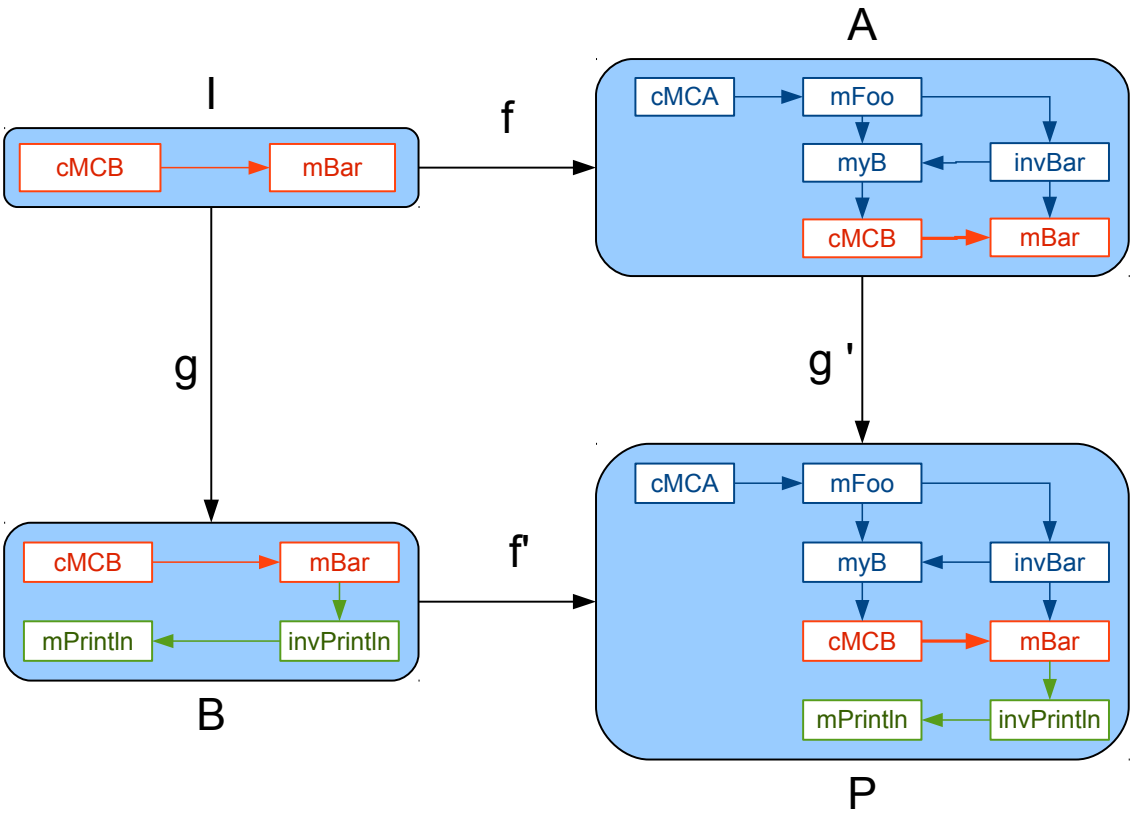


Abbildung 25: Pushout von Graphen

Eindeutigkeit bis auf Isomorphie

Eine weitere in der Kategorientheorie und insbesondere bei universellen Konstruktionen häufig anzutreffende Eigenschaft zeigt sich auch beim Pushout: Er ist nur "bis auf Isomorphie" eindeutig. Das bedeutet, dass man jeder der vier Graphen in Abb. 25 durch einen isomorphen anderen Graphen ersetzen[2] kann. Das Ergebnis ist immer noch ein Pushout und aus kategorientheoretischer Sicht im Wesentlichen dasselbe wie der alte Pushout.

3.4. Zusammenfassung

In diesem Kapitel wurden die Graphfeatures Typisierung, Attributierung, Containment, Ordnung, Vererbung und Constraints informell vorgestellt, die sich zu Graphvarianten zur Programmrepräsentation kombinieren lassen. Es hat sich ergeben, dass attributierte typisierte Graphen für die vollständige Repräsentation der statischen Semantik von Programmen ausreichen. Allerdings werden in dieser Arbeit nur Typisierung und Containment formalisiert, um den Themenumfang zu begrenzen.

Neben den Graphfeatures wurden einige für diese Arbeit relevante Aspekte der Kategorientheorie informell erklärt. Anhand des Pushout wurde gezeigt, wie sich eine kategorielle Konstruktion verwenden lässt, um eine einfache Programmmanipulation, nämlich das Binden, zu formalisieren.

Im nächsten Kapitel folgt eine weitergehende formale Einführung in die Kategorientheorie und die formale Definition von Graphfeatures. Weitere Programmmanipulationen werden in Kapitel 5 formalisiert.

[2] Dabei muss man auch die Pfeile mithilfe eines Graphisomorphismus ersetzen.

Kapitel 4

Formalismus

Nachdem in Kapitel 3 eine informelle Einführung in die Graphvarianten und die dafür notwendige Kategorientheorie gegeben wurde, folgt in diesem Kapitel eine formale Beschreibung. Zunächst wird in Abschnitt 4.1 die Kategorientheorie formal dargestellt. Der davon ausgelagerte Abschnitt 4.2 beschreibt Sketches, die der graphbasierten Definition von Kategorien dienen. In Abschnitt 4.3 wird eine modulare Definition der einzelnen Graphfeatures durch Sketches vorgeschlagen. Eine kurze Ausführung zu den mengentheoretischen Grundlagen und einige Grunddefinitionen finden sich in Anhang A.

4.1. Kategorientheorie

Die Kategorientheorie ermöglicht es, verschiedenste Bereiche der Mathematik, Logik und Informatik mithilfe des Begriffes der Kategorie und darauf aufbauenden kategoriellen Konzepten auf einheitliche Weise zu beschreiben. Zu diesen kategoriellen Konzepten gehören auch die universellen Konstruktionen der Kategorientheorie, nämlich insbesondere Limites und Kolimites, unter die auch der in Unterabschnitt 3.3.2 vorgestellte Pushout fällt. Kategorien und universelle Konstruktionen werden in dieser Arbeit verwendet, um die verschiedenen Graphvarianten formal zu erfassen und um Programmeigenschaften und -beziehungen zu beschreiben.

In diesem Abschnitt wird zunächst der Begriff der Kategorie definiert und erläutert. Dann werden einige nützliche Kategorien definiert, die sich von anderen Kategorien ableiten lassen. Abschließend werden universelle Konstruktionen in Form von Limites und Kolimites vorgestellt.

4.1.1. Kategorien

Wie in der Einleitung zu Kapitel 3 bereits erklärt wurde, werden in dieser Arbeit Kategorien benötigt, um die verschiedenen Graphvarianten zur Programmmodellierung zu erfassen. Bei Kategorien handelt es sich um mathematische Strukturen ähnlich wie Gruppen, Ringe und Körper. Allerdings bauen sie nicht wie diese auf Mengen auf, sondern auf Graphen, oder genauer gesagt auf Köchern (engl. „quiver“), welche eine Verallgemeinerung von Graphen auf echte Klassen sind (siehe Anhang A).

In diesem Unterabschnitt werden zunächst Köcher und Graphen und die zugehörigen Abbildungsbegriffe der Köcher- bzw. Graphhomomorphismen behandelt. Anschließend werden Kategorien auf der Grundlage von Köchern definiert und erläutert. Schließlich werden noch Funktoren als Abbildungen zwischen Kategorien und natürliche Transformationen als Abbildungen zwischen Funktoren beschrieben.

Eine Kategorie ist eine erweiterte Art von Graph, in der man mit den Pfeilen rechnen kann. Allerdings sind aus formalen Gründen die normalen Graphen zu „klein“, um mit ihnen beliebige mathematische Themenfelder zu beschreiben, selbst wenn die Graphen unendlich sein dürfen. Man braucht zunächst den Begriff des Köchers:

Definition 1 (Köcher)

Köcher Sei $\mathbf{G}^0$ eine Klasse, genannt Klasse der Objekte oder Knoten.
Sei $\mathbf{G}^1$ eine Klasse, genannt Klasse der Pfeile, Morphismen oder Kanten[1].
Seien $src_{\mathbf{G}}, tar_{\mathbf{G}} : \mathbf{G}^1 \to \mathbf{G}^0$ Funktionen, die den Pfeilen Knoten zuordnen.

Dann ist $\mathbf{G} = (src_{\mathbf{G}}, tar_{\mathbf{G}})$ ein **Köcher**.
Für einen Pfeil $m \in \mathbf{G}^1$ ist $src(v)$ die **Domäne** oder der **Quellknoten** und $tar(v)$ die **Kodomäne** oder der **Zielknoten** von m.

Für Knoten $A, B \in \mathbf{G}^0$ ist $\mathbf{G}[A,B] = \left\{ m \in \mathbf{G}^1 \;\middle|\; src(m) = A \wedge tar(m) = B \right\} \subseteq \mathbf{G}^1$ die Klasse aller Pfeile **von** A **nach** B in $\mathbf{G}$.
Man schreibt $A \xrightarrow{m} B \in \mathbf{G}$ für einen Pfeil $m \in \mathbf{G}[A,B]$ von A nach B.

Manchmal ist es bequemer, die Pfeile eines Köchers $\mathbf{G}$ knotenweise zu definieren, als die Funktionen *src* und *tar* explizit anzugeben. Hierzu legt man zunächst die Klasse $\mathbf{G}^0$ seiner Knoten fest. Dann definiert man für jedes Paar (A, B) von Knoten die Klasse $\mathbf{G}[A,B]$ der Pfeile von A nach B, wobei die Klassen $\mathbf{G}[A,B]$ paarweise disjunkt sein müssen. Dann ergibt sich die Klasse der Pfeile als Vereinigung dieser Klassen $\mathbf{G}^1 = \bigcup_{A,B \in \mathbf{G}^0} \mathbf{G}[A,B]$.
Für einen Pfeil $m \in \mathbf{G}[A,B]$ ergibt sich die Domäne $src_{\mathbf{G}}(m) = A$ und die Kodomäne $tar_{\mathbf{G}}(B) = B$. □

Die Funktion *src* ordnet jedem Pfeil seine Quelle und *tar* jedem Pfeil sein Ziel zu. Durch die Unterscheidung von Quelle und Ziel sind die Pfeile gerichtet. Auch können unterschiedliche Pfeile dieselbe Quelle und dasselbe Ziel haben, d.h. es gibt Mehrfachkanten.

Bei Köchern handelt es sich also im Grunde um dasselbe wie bei einem Graphen im Sinne von Unterabschnitt 3.2.1, nur dass Köcher auch echte Klassen von Knoten und Pfeilen haben können. Zur Erinnerung: Jede Menge ist eine Klasse. Eine Klasse ist eine Menge, wenn sie nach den Regeln der jeweils benutzten Mengentheorie als Menge definiert werden kann. Im Falle der für diese Arbeit vorgeschlagenen Mengenlehre ARC gilt dies, wenn man sie durch eine „sichere“ Formel definieren kann, oder wenn sie eine Teilklasse einer auf solche Weise definierten Menge ist. „Sicher“ bedeutet hier, dass die Formel nur auf Mengen zutrifft, dass alle darin vorkommenden Variablen Mengen sind, und dass die Konstante $\mathbf{V}$, die die Klasse $\mathfrak{Set}$ aller Mengen bezeichet, nicht vorkommt [Mul01]. Eine Klasse ist eine echte Klasse, wenn sie keine Menge ist.

Graphen sind genau die Köcher, die Mengen von Knoten und Pfeilen haben:

Definition 2 (Graph)

Graph Die Klasse der **Graphen** ist definiert als

$$\mathfrak{Graphs} = \left\{ (src, tar) \;\middle|\; src, tar : \mathbf{G}^1 \to \mathbf{G}^0 \text{ und } \mathbf{G}^0, \mathbf{G}^1 \in \mathfrak{Set} \right\}$$ □

[1] Die Schreibweise $\mathbf{G}^1$ ist an die Tatsache angelehnt, dass jeder Pfeil einem Pfad der Länge 1 entspricht und umgekehrt. Ebenso gibt es für jeden Knoten A genau einen Pfad der Länge 0, nämlich den leeren Pfad von A nach A, wodurch sich die Schreibweise $\mathbf{G}^0$ erklären lässt.

Neben der Unterscheidung zwischen Graphen und Köchern kann man feinere Unterscheidungen bezüglich der Größe von Köchern treffen. Diese Unterscheidungen sind manchmal notwendig, da Köcher für manche Konstruktionen auch zu groß sein können:

Definition 3 (Größen von Köchern)
Größen von Köchern

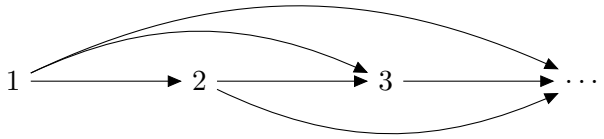
Sei $\mathbf{G}$ ein Köcher.

Dann ist $\mathbf{G}$ genau dann **klein**, wenn er ein Graph ist[2].
$\mathbf{G}$ ist genau dann **lokal klein**, wenn für alle Knoten $A, B \in \mathbf{G}^0$ die Klasse $\mathbf{G}[A, B]$ der Pfeile von A nach B eine Menge ist.
$\mathbf{G}$ ist genau dann **endlich**, wenn $\mathbf{G}^0$ und $\mathbf{G}^1$ endliche Mengen sind. □

Um den Unterschied zwischen den verschiedenen Größen von Graphen und Köchern genauer zu erläutern, seien die Beispiele 4 und 5 aufgeführt:

Beispiel 4 (Graph der natürlichen Zahlen)
Graph der natürlichen Zahlen

Ein Beispiel eines kleinen Köchers ist ein Graph Nat, der als Knoten die natürlichen Zahlen $\mathbb{N}$ hat und jeweils genau einen Pfeil von a nach b hat, wenn $a < b$ ist:



Die Klasse der natürlichen Zahlen ist eine Menge, denn sie lässt sich nach den Regeln der Mengenlehre z.B. nach von Neumann durch $0 = \varnothing$, $1 = \{0\}$, $2 = \{0, 1\}$, $3 = \{0, 1, 2\}$ und so weiter definieren. Sie ist aber keine echte Klasse, eben weil sie eine Menge ist. Da auch die Klasse der Pfeile eine Menge ist, ist der Köcher Nat klein, also ein Graph. □

Damit ein Köcher nicht mehr klein ist, reicht es also nicht, dass er unendlich ist wie Nat, sondern er muss wesentlich größer sein. Dies ist beispielsweise der Fall, wenn die Klasse aller Mengen ins Spiel kommt:

Beispiel 5 (Köcher $\mathbf{Q_{Set}}$ der Mengen und Funktionen)
$\mathbf{Q_{Set}}$

Ein weiteres Beispiel eines Köchers ist der Köcher $\mathbf{Q_{Set}}$ der Mengen und Funktionen. Die Klasse der Knoten von $\mathbf{Q_{Set}}$ ist die Klasse aller Mengen, also $\mathbf{Q}^0_{\mathbf{Set}} = \mathfrak{Set}$. Die Klasse der Pfeile von einem Knoten (also einer Menge) A zu einem Knoten B in $\mathbf{Q_{Set}}$ ist die Menge der Funktionen von A nach B, also $\forall A, B \in \mathbf{Q}^0_{\mathbf{Set}}\colon \mathbf{Q_{Set}}[A, B] = B^A$.

Bei $\mathbf{Q_{Set}}$ handelt es sich um *keinen* Graphen, weil die Klasse aller Mengen selbst keine Menge ist. Allerdings handelt es sich um einen lokal kleinen Köcher, denn die Klasse B^A aller Funktionen zwischen zwei Mengen A und B ist eine Menge. □

Köcher und Graphen sind gut geeignet, um Struktur oder um Beziehungen zwischen mathematischen Objekten formal zu erfassen. Darüber hinaus sind sie aber nicht besonders ausdruckskräftig, solange man wie in der Graphen- und Kategorientheorie üblich die innere

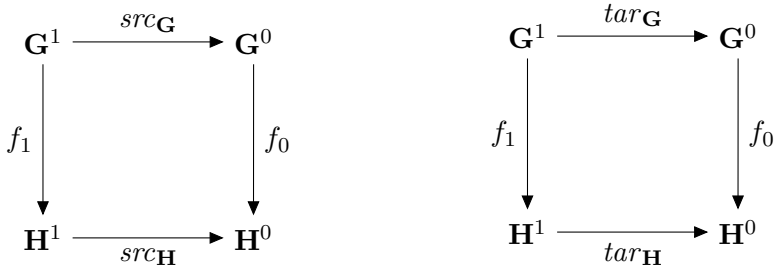
[2] In der Literatur wird teils auch der Begriff „Graph“ für Köcher und „kleiner Graph“ für Graphen mit Mengen von Knoten und Pfeilen benutzt. Dem wurde hier nicht gefolgt, weil das Hauptthema dieser Arbeit die Programmrepräsentation mit *kleinen* Graphen ist, und deshalb kleine Graphen der Normalfall sein sollten.

Struktur von Knoten und Pfeilen nicht betrachtet. Beispielsweise wäre es schwierig, allein mit Köchern Rechenoperationen zu definieren, oder Programme zu repräsentieren, wie in Unterabschnitt 3.2.1 ja bereits festgestellt wurde.

Die kategorientheoretische Herangehensweise ist es, jetzt nicht etwa eine innere Struktur von Knoten und Pfeilen zu definieren, sondern extern zusätzliche Struktur hinzuzufügen, also Köcher als Grundlage für ausdruckskräftigere Konstrukte zu verwenden. Beispielsweise kann man mit Köcherhomomorphismen einen Köcher zu einem anderen in Beziehung setzen. Hierdurch lässt sich, wie in Unterabschnitt 3.2.2 erkannt wurde, Typisierung von Graphen definieren, die eine notwendige Voraussetzung zur vollständigen Repräsentation der konzeptuellen Semantik von Programmen ist.

Köcherhomomorphismus

Definition 6 (Köcherhomomorphismus)
Seien $\mathbf{G}$ und $\mathbf{H}$ Köcher, genannt Domäne bzw. Kodomäne.
Sei $f_0 : \mathbf{G}^0 \to \mathbf{H}^0$ eine Funktion, genannt **Knotenkomponente**, die jedem Knoten von $\mathbf{G}$ einen Knoten von $\mathbf{H}$ zuordnet.
Sei $f_1 : \mathbf{G}^1 \to \mathbf{H}^1$ eine Funktion, genannt **Pfeilkomponente**, die jedem Pfeil von $\mathbf{G}$ einen Pfeil von $\mathbf{H}$ zuordnet.
Sei $f \coloneqq (\mathbf{G}, \mathbf{H}, f_0, f_1)$ das Quadrupel von Domäne, Kodomäne, Knoten- und Pfeilkomponente.



Dann ist f genau dann ein **Köcherhomomorphismus**, geschrieben $f : \mathbf{G} \to \mathbf{H}$, wenn gilt: $f_1 \bullet src_{\mathbf{H}} = src_{\mathbf{G}} \bullet f_0 \wedge f_1 \bullet tar_{\mathbf{H}} = tar_{\mathbf{G}} \bullet f_0$

Falls $\mathbf{G}$ und $\mathbf{H}$ Graphen sind, nennt man f **Graphhomomorphismus**. Für die Menge aller Graphhomomorphismen $f : G \to H$ von einem Graphen G in einen Graphen H schreibt man H^G.

Bei der Anwendung eines Graphhomomorphismus lässt man üblicherweise die Unterscheidung zwischen der Knoten- und Pfeilkomponente fallen. Man schreibt also $f(v)$ anstatt $f_0(v)$ für Knoten $v \in \mathbf{G}^0$ und $f(m)$ anstatt $f_1(m)$ für Pfeile $m \in \mathbf{G}^1$. □

Anschauliche Beispiele von Graphhomomorphismen sind in Unterabschnitt 3.2.1 zu finden.

Eine weitere Möglichkeit, Köcher als Grundlage für eine ausdruckskräftigere Konstruktion zu verwenden, ist es, eine Rechenoperation darauf zu definieren, um eine Kategorie zu erhalten. Kategorien sind Köcher, in denen man mit den Pfeilen rechnen kann, oder in der kategorientheoretischen Ausdrucksweise, die Pfeile komponieren kann. Man kann dabei aber nicht beliebige Pfeile komponieren, sondern nur aufeinanderfolgende – also solche, die einen Pfad der Länge 2 bilden. Um dies formal zu erfassen, hilft die folgende Definition:

Definition 7 (Komponierbares Paar)

Komponierbares Paar

Sei $\mathbf{G}$ ein Köcher.

Dann ist $\mathbf{G}^2 = \{(f,g) \in \mathbf{G}^1 \times \mathbf{G}^1 \mid tar(f) = src(g)\}$ die Klasse der **komponierbaren Paare** von Pfeilen von $\mathbf{G}$.

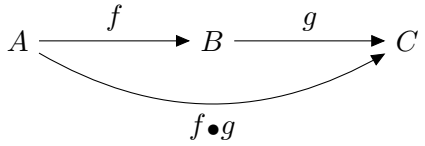
Für ein komponierbares Paar $(f,g) \in \mathbf{G}^2$ wird anschaulicher $A \xrightarrow{f} B \xrightarrow{g} C \in \mathbf{G}$ (oder $A \xrightarrow{f} B \xrightarrow{g} C \in \mathbf{G}^2$) geschrieben. □

Die Grundidee hinter Kategorien ist es, dass man jedes komponierbare Paar von Pfeilen zu einem Pfeil komponieren kann. Daneben ist die Komposition assoziativ und sie hat neutrale Elemente, damit man mit ihr gut rechnen kann:

Definition 8 (Kategorie)

Kategorie

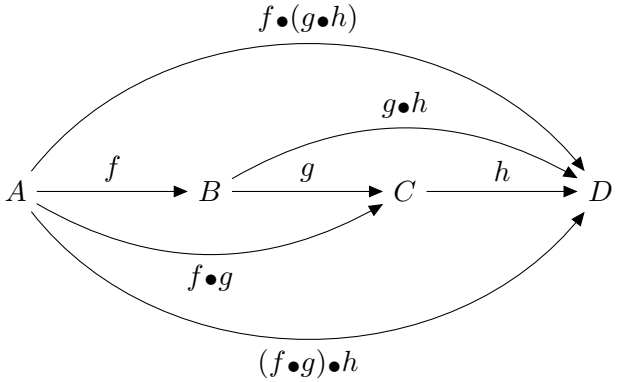
Sei $\mathbf{G}$ ein Köcher (der Kategorie zugrundeliegender Graph[3]).
Sei $\bullet$ eine Funktion $\bullet : \mathbf{G}^2 \to \mathbf{G}^1$, genannt Komposition, die jedes komponierbare Paar $A \xrightarrow{f} B \xrightarrow{g} C \in \mathbf{G}$ zu einem Pfeil $A \xrightarrow{f \bullet g} C \in \mathbf{G}$ komponiert.



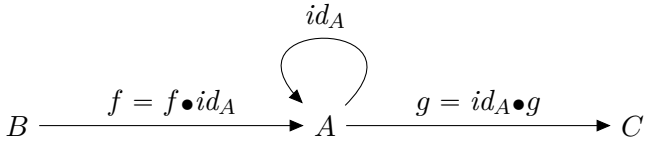
Sei $\mathbf{C} = (\mathbf{G}, \bullet)$.
Definiere $\mathbf{C}^0 := \mathbf{G}^0$, $\mathbf{C}^1 := \mathbf{G}^1$, $\mathbf{C}^2 := \mathbf{G}^2$ und $\mathbf{Q}_\mathbf{C} = \mathbf{G}$.

Dann ist $\mathbf{C}$ genau dann eine Kategorie, wenn

- die Komposition assoziativ ist:
  $\forall f,g,h \in \mathbf{C}^1\colon (f,g) \in \mathbf{C}^2 \wedge (g,h) \in \mathbf{C}^2 \Rightarrow (f \bullet g) \bullet h = f \bullet (g \bullet h)$



- und für jedes Objekt $A \in \mathbf{C}^0$ ein Pfeil $A \xrightarrow{id_A} A$, genannt Identitätspfeil von A, existiert, der ein neutrales Element der Komposition ist:
  $\forall A \in \mathbf{C}^0\colon \exists A \xrightarrow{id_A} A \in \mathbf{C}^1\colon (\forall B \xrightarrow{f} A\colon f \bullet id_A = f) \wedge (\forall A \xrightarrow{g} C\colon id_A \bullet g = g)$



□

Bemerkung 9 (Komposition von Pfaden)

Komposition von Pfaden

Durch die Assoziativität der Komposition wird es möglich, jeden beliebigen nichtleeren Pfad $k_0 \xrightarrow{s_1} k_1 \xrightarrow{s_2} \ldots \xrightarrow{s_{n-1}} k_{n-1} \xrightarrow{s_n} k_n$ aufeinanderfolgender Pfeile zu komponieren, ohne auf die Reihenfolge der Komposition achten zu müssen.

Neben den nichtleeren Pfaden gibt es aber für jedes Objekt A auch einen leeren Pfad von A nach A. Damit die Kompositionsfunktion für alle Pfade definiert ist, setzt man die Komposition des leeren Pfades von A nach A auf den Identitätspfeil $A \xrightarrow{id_A} A$. Formalisiert wird dies in den Definitionen 24 und 31. □

Im Allgemeinen können die Klasse der Objekte und die Klasse der Pfeile wie bei Köchern auch echte Klassen sein, sind also nicht notwendigerweise Mengen. Deshalb kann man Kategorien wie Köcher als klein, lokal klein oder endlich bezeichnen:

Definition 10 (Größen von Kategorien)

Größen von Kategorien

Sei $\mathbf{C}$ eine Kategorie.

Dann ist $\mathbf{C}$ genau dann **klein**, wenn $\mathbf{C}^0$ und $\mathbf{C}^1$ Mengen sind.
$\mathbf{C}$ ist genau dann **lokal klein**, wenn für alle Objekte $A, B \in \mathbf{C}^0$ die Klasse $\mathbf{C}[A, B]$ der Pfeile von A nach B eine Menge bildet.
$\mathbf{C}$ ist genau dann **endlich**, wenn $\mathbf{C}^0$ und $\mathbf{C}^1$ endliche Mengen sind.

Die Klasse aller kleinen Kategorien heißt $\mathfrak{Cat}$ □

Es gibt einige Beispiele sehr einfacher, endlicher Kategorien, die aber dennoch in der Kategorientheorie eine spezielle Rolle einnehmen:

Beispiel 11 (Primitive Kategorien)

Primitive Kategorien

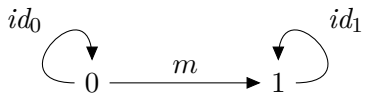
Die Kategorie **0** hat keine Objekte und keine Pfeile.

Die Kategorie **1** hat ein Objekt 0 und der Identitätspfeil $0 \xrightarrow{id_0} 0$.

Die Kategorie $\mathbf{1} + \mathbf{1}$ ist eine Verdoppelung von **1**, hat also zwei Objekte 0 und 1 und deren Identitätspfeile.

[3] Eine Kategorie $\mathbf{C}$ wird oft mit dem ihr zugrundeliegenden Graphen $\mathbf{Q}_\mathbf{C}$ identifiziert. In der Regel ergeben sich daraus keine Probleme.

Von $\mathbf{1} + \mathbf{1}$ verschieden ist die Kategorie **2**, genannt Intervallkategorie. Diese hat auch zwei Objekte 0 und 1, aber neben den Identitätspfeilen auch noch einen Pfeil $0 \xrightarrow{m} 1$.



□

Die Komposition ist in diesen Kategorien trivial, da man die Pfeile, falls vorhanden, nur mit Identitätspfeilen komponieren kann.

Es gibt aber auch wesentlich mächtigere Kategorien. Die wohl wichtigste Kategorie ist die Kategorie **Set** der Mengen und Funktionen. Ihr liegt der Köcher $\mathbf{Q_{Set}}$ der Mengen und Funktionen aus Beispiel 5 zugrunde. Als Komposition besitzt sie die Funktionskomposition:

Beispiel 12 (Kategorie Set)

Kategorie **Set**

Die Kategorie **Set** ist folgendermaßen definiert:

- Die Klasse der Objekte ist die Klasse aller Mengen: $\mathbf{Set}^0 = \mathfrak{Set}$
- Die Klasse der Pfeile zwischen zwei Objekten (Mengen) A und B ist die Menge B^A der Funktionen von A nach B: $\mathbf{Set}[A, B] = B^A$
- Die Komposition der Pfeile (Funktionen) $A \xrightarrow{f} B \xrightarrow{g} C \in \mathbf{Set}^2$ ist die Funktionskomposition, also $f \bullet g : A \to C : x \mapsto g(f(x))$. □

Es ist leicht zu überprüfen, dass es sich bei **Set** um eine Kategorie handelt: Dass die Funktionskomposition assoziativ ist, wird in der Einführungsliteratur zur Mathematik bewiesen und die Rolle des Identitätspfeiles einer Menge X nimmt ihre Identitätsfunktion $id_X : X \to X : x \mapsto x$ ein.

Ein weiteres Beispiel einer Kategorie bilden die Graphen zusammen mit den Graphhomomorphismen (siehe Definition 6 und Unterabschnitt 3.2.1):

Beispiel 13 (Kategorien Graphs und Quiv)

Kategorien **Graphs** *und* **Quiv**

Die Kategorie **Graphs** ist folgendermaßen definiert:

- Die Klasse der Objekte ist die Klasse der Graphen: $\mathbf{Graphs}^0 = \mathfrak{Graphs}$
- Die Klasse der Pfeile zwischen zwei Objekten (also Graphen) G und H ist die Menge H^Gder Graphhomomorphismen von G nach H: $\mathbf{Graphs}[G, H] = H^G$
- Die Komposition zweier Pfeile (also Graphhomomorphismen) $f : G \to H$ und $g : H \to I$ aus $\mathbf{Graphs}^1$ mit $f = (G, H, f_0, f_1)$ und $g = (H, I, g_0, g_1)$, also mit Objektkomponenten f_0 bzw. g_0 und Pfeilkomponenten f_1 bzw. g_1 ist die komponentenweise Funktionskomposition:
  $f \bullet g = (G, H, f_0, f_1) \bullet (H, I, g_0, g_1) = (G, I, f_0 \bullet g_0, f_1 \bullet g_1)$

In der für diese Arbeit vorgeschlagenen Mengenlehre ARC kann man analog zu **Graphs** auch eine Kategorie **Quiv** definieren, die nicht nur die Graphen enthält, sondern alle Köcher, die höchstens gleichmächtig zur Klasse **Set** aller Mengen sind. □

Ähnlich wie Köcher haben auch Kategorien einen zugehörigen Homomorphismusbegriff, nämlich den der Funktoren. Mit Funktoren kann man unter anderem Konstruktionen wie beispielsweise die Bildung der Pfadkategorie beschreiben, die die Pfade eines Graphen umfasst (siehe Definition 27). Funktoren sind formal in folgender Weise definiert:

Definition 14 (Funktor)

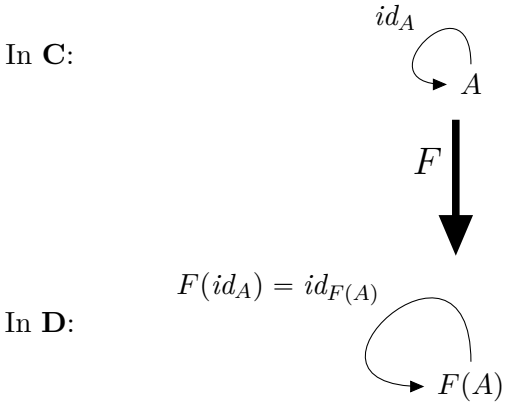
Funktor

Seien $\mathbf{C}$ und $\mathbf{D}$ Kategorien, genannt Domäne bzw. Kodomäne.
Sei $F_0 : \mathbf{C}^0 \to \mathbf{D}^0$ eine Funktion, genannt **Objektkomponente**, die jedem Objekt von $\mathbf{C}$ ein Objekt von $\mathbf{D}$ zuordnet.
Sei $F_1 : \mathbf{C}^1 \to \mathbf{D}^1$ eine Funktion, genannt **Pfeilkomponente**, die jedem Pfeil von $\mathbf{C}$ einen Pfeil von $\mathbf{D}$ zuordnet.
Sei $\bar{F} = (\mathbf{Q_C}, \mathbf{Q_D}, F_0, F_1)$ ein Köcherhomomorphismus $\bar{F} : \mathbf{Q_C} \to \mathbf{Q_D}$ zwischen den zugrundeliegenden Köchern von $\mathbf{C}$ und $\mathbf{D}$ (zugrundeliegender Köcherhomomorphismus).

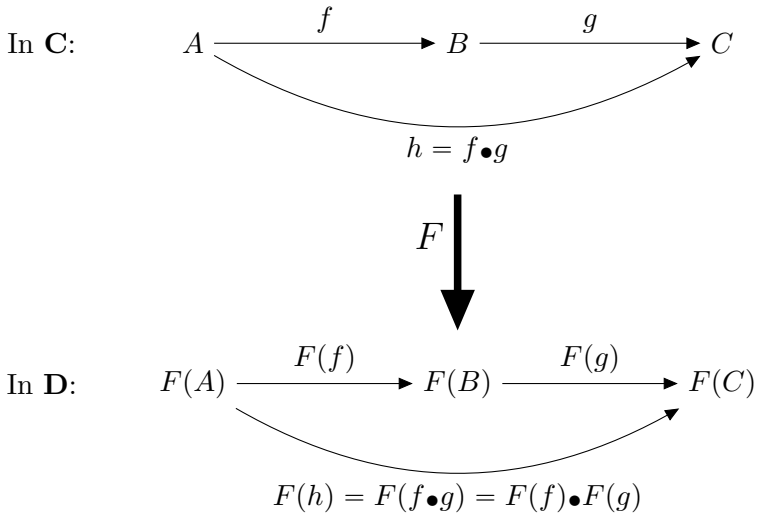
Sei $F := (\mathbf{C}, \mathbf{D}, F_0, F_1)$ das Quadrupel von Domäne, Kodomäne und den Objekt- und Pfeilkomponenten des zugrundeliegenden Köcherhomomorphismus.

Dann ist F genau dann ein **Funktor** von $\mathbf{C}$ nach $\mathbf{D}$, wenn

1. F mit den Identitätspfeilen verträglich ist: $\forall A \in \mathbf{C}^0$: $F(id_A) = id_{F(A)}$



2. und F mit der Komposition verträglich ist:
$\forall A \xrightarrow{f} B \xrightarrow{g} C \in \mathbf{C}^2$: $F(f \bullet g) = F(f) \bullet F(g)$



Auch bei Funktoren wird die Schreibweise $F : \mathbf{C} \to \mathbf{D}$ verwendet.
$\bar{F}$ ist der zugrundeliegende Köcherhomomorphismus von F, mit dem F häufig identifiziert wird.
Die Klasse aller Funktoren von $\mathbf{C}$ nach $\mathbf{D}$ wird mit $\mathfrak{Fun}(\mathbf{C}, \mathbf{D})$ bezeichnet. □

Ein Funktor ist also im Wesentlichen ein Köcherhomomorphismus zwischen Kategorien, der mit den Identitätspfeilen und der Komposition verträglich ist. Zwei primitive Fälle von Funktoren sind die konstanten Funktoren und die Identitätsfunktoren:

Beispiel 15 (Konstanter Funktor)
Seien $\mathbf{C}$ und $\mathbf{D}$ Kategorien und $D \in \mathbf{D}^0$ ein Objekt von $\mathbf{D}$.

Identitätsfunktor

Der **konstante Funktor** $\Delta_D : \mathbf{C} \to \mathbf{D}$ von $\mathbf{C}$ nach D ordnet jedem Objekt von $\mathbf{C}$ das Objekt D und jedem Pfeil von $\mathbf{C}$ den Identitätsmorphismus id_D von D zu. □

Beispiel 16 (Identitätsfunktor)
Sei $\mathbf{C}$ eine Kategorie.

Identitätsfunktor

Der **Identitätsfunktor** $id_{\mathbf{C}} : \mathbf{C} \to \mathbf{C}$ von $\mathbf{C}$ nach $\mathbf{C}$ ordnet jedem Objekt und jedem Pfeil sich selbst zu, also $\forall A \in \mathbf{C}^0$: $id_{\mathbf{C}}(A) = A$ und $\forall A \xrightarrow{m} B \in \mathbf{C}$: $F(m) = m$. □

Beispiel 17 (Kategorie Cat)
Zusammen mit den Funktoren bilden die kleinen Kategorien selber eine Kategorie, nämlich die Kategorie **Cat**:

Kategorie **Cat**

- Die Klasse der Objekte ist die Klasse der kleinen Kategorien: $\mathbf{Cat}^0 = \mathfrak{Cat}$
- Die Klasse der Pfeile zwischen zwei Objekten (also Kategorien) A und B ist die Menge der Funktoren von A nach B: $\mathbf{Cat}[A, B] = B^A$
- Die Komposition zweier Pfeile (also Funktoren) $f = (\mathbf{C}, \mathbf{D}, f_0, f_1)$ und $g = (\mathbf{D}, \mathbf{E}, g_0, g_1)$ mit Objektkomponenten f_0 bzw. g_0 und Pfeilkomponenten f_1 bzw. g_1 ist die komponentenweise Funktionskomposition $f \bullet g = (\mathbf{C}, \mathbf{E}, f_0 \bullet g_0, f_1 \bullet g_1)$.
- Der Identitätspfeil eines Objektes von $A \in \mathbf{Cat}$ ist der in Beispiel 16 definierte Identitätsfunktor id_A.

Diese Kategorie ist unter anderem deshalb nützlich, weil man so Funktoren definieren kann, die aus Objekten einer Kategorie kleine Kategorien erzeugen.

In der für diese Arbeit vorgeschlagenen Mengenlehre ARC kann man analog zu **Cat** auch eine Kategorie **CAT** definieren, die nicht nur die kleinen Kategorien enthält, sondern alle Kategorien, die höchstens gleichmächtig zur Klasse **Set** aller Mengen sind. □

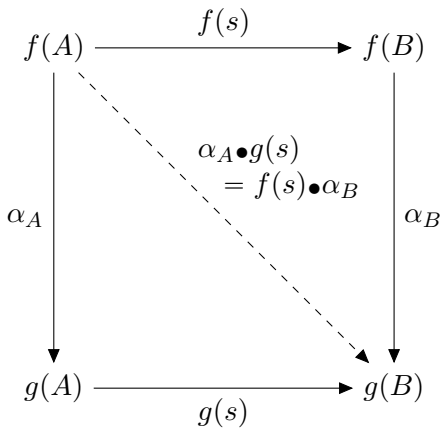
Ein weiterer zentraler Begriff der Kategorientheorie ist der der natürlichen Transformation. Dabei handelt es sich um eine Verallgemeinerung verschiedener Arten von Homomorphismen. Eine natürliche Transformation ist eine Abbildung zwischen Abbildungen, nämlich zwischen parallelen Köcherhomomorphismen, deren Kodomäne eine Kategorie ist. Natürliche Transformationen zwischen parallelen Funktoren sind ein Spezialfall hiervon, weil auch Funktoren Köcherhomomorphismen mit einer Kategorie als Kodomäne sind.

Definition 18 (Natürliche Transformation)

Natürliche Transformation

Sei $\mathbf{G}$ ein Köcher.
Sei $\mathbf{C}$ eine Kategorie.
Seien $f : \mathbf{G} \to \mathbf{C}$ und $g : \mathbf{G} \to \mathbf{C}$ Köcherhomomorphismen von $\mathbf{G}$ nach $\mathbf{C}$.
Sei $\alpha = (\alpha_A)_{A \in \mathbf{G}^0}$ eine Familie von Pfeilen $f(A) \xrightarrow{\alpha_A} g(A)$ von den Bildern der Knoten von $\mathbf{G}$ unter f zu ihren Bildern unter g.

Dann ist α genau dann eine natürliche Transformation $\alpha : f \to g$, wenn für jeden Pfeil $A \xrightarrow{s} B \in \mathbf{G}^1$ gilt: $f(s) \bullet \alpha_B = \alpha_A \bullet g(s)$



Die Klasse aller natürlichen Transformationen von f nach g heißt $\mathfrak{NatTrans}(f, g)$, formal:

$$\mathfrak{NatTrans}(f,g) = \left\{ \alpha \in \prod_{A \in \mathbf{G}^0} (g(A))^{f(A)} \;\middle|\; \forall A \xrightarrow{s} B \in \mathbf{G}^1 \colon f(s) \bullet \alpha_B = \alpha_A \bullet g(s) \right\}$$ □

Eine natürliche Transformation kann man als zusammengesetzte Abbildung betrachten, die pro Objekt des Formköchers $\mathbf{G}$ eine Komponente (d.h. einen Pfeil von $\mathbf{C}$) hat. Die definierende Eigenschaft ist, dass sie mit den Bildern $f(s)$ und $g(s)$ der Pfeile $s \in \mathbf{G}^1$ des Formköchers verträglich ist, was durch die Kommutativitätsbedingung ausgedrückt wird. In Abschnitt 4.2 findet eine genauere Diskussion hiervon statt.

Ein wichtiges Prinzip der Kategorientheorie ist das Dualitätsprinzip. Aus jeder Aussage über Kategorien ergibt sich eine dazu duale Aussage, und aus jedem Beweis ein dazu dualer Beweis. Die Grundlage dafür kann man mit Köchern formulieren. Ein Köcher induziert immer einen dazu dualen Köcher, in dem die Pfeile in umgekehrte Richtung zeigen:

Definition 19 (Dualer Köcher)

Dualer Köcher

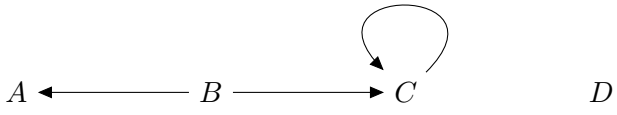
Sei $\mathbf{G} = (src_{\mathbf{G}}, tar_{\mathbf{G}})$ ein Köcher.

Der **duale Köcher** $\mathbf{G^{op}}$ von $\mathbf{G}$ definiert sich durch $\mathbf{G^{op}} = (tar_{\mathbf{G}}, src_{\mathbf{G}})$, also:

- $(\mathbf{G^{op}})^0 = \mathbf{G}^0$
- $\mathbf{G^{op}}[A, B] = \mathbf{G}[B, A]$ für $A, B \in \mathbf{G}^0$ □

Beispielsweise ist zu diesem Graphen (Köcher)

der folgende Graph dual:



Definition 20 (Duale Kategorie)
Sei $\mathbf{C}$ eine Kategorie.

Duale Kategorie

Die **duale Kategorie $\mathbf{C^{op}}$** von $\mathbf{C}$ definiert sich durch:

- $(\mathbf{C^{op}})^0 = \mathbf{C}^0$
- $\mathbf{C^{op}}[A, B] = \mathbf{C}[B, A]$ für $A, B \in \mathbf{C}^0$
- Für $A \xrightarrow{f} B \xrightarrow{g} C \in (\mathbf{C^{op}})^1$ ist die Komposition $f \bullet g$ von $A \xrightarrow{f} B$ und $B \xrightarrow{g} C$ in $\mathbf{C^{op}}$ definiert als die Komposition $g \bullet f$ von $C \xrightarrow{g} B$ und $B \xrightarrow{f} A$ in $\mathbf{C}$. □

Die duale Kategorie $\mathbf{C^{op}}$ repräsentiert im Grunde dasselbe wie $\mathbf{C}$. Man betrachtet dieselben Objekte und Pfeile, aber aus einer anderen Perspektive. Beispielsweise kann man einen Graphhomomorphismus $h : G \to H$, der ja ein Vorkommen von G in H repräsentiert, in beide Richtungen interpretieren (siehe Unterabschnitt 3.2.1):

Normalerweise betrachtet man h in der Richtung von G nach H. In diesem Fall kann man sagen, dass h neue Knoten und Pfeile zu G *hinzufügt* und bestehende Knoten und Pfeile von G miteinander *verschmilzt*. Hinzugefügt werden diejenigen, die in H vorhanden sind, aber kein Urbild in G haben. Wenn unterschiedliche Knoten und Pfeile von G auf denselben Knoten bzw. auf denselben Pfeil von H abgebildet werden, d.h. die Urbilder eines Knotens bzw. eines Pfeiles von H werden miteinander verschmolzen. Die Knoten und Pfeile in H, die genau ein Urbild in G haben bleiben hingegen unverändert.

In der dualen Kategorie betrachtet man h in der entgegengesetzten Richtung, von H nach G. In dieser Betrachtungsweise kann h Knoten und Pfeile aus H *entfernen* und einzelne Knoten und Pfeile in mehrere *aufspalten*, im Sinne von vervielfältigen. Entfernt werden diejenigen Knoten und Pfeile, die nicht zu dem Vorkommen von G in H gehören, also solche, die kein Urbild unter h haben. Knoten und Pfeile von H, die mehrere Urbilder unter h besitzen, werden in ihre Urbilder aufgespalten. Diejenigen Knoten und Pfeile, die genau ein Urbild haben, bleiben auch hier unverändert erhalten.

Durch die Existenz der dualen Kategorien kann man sämtliche Aussagen, die sich auf Kategorien im allgemeinen beziehen, auch auf die dualen Kategorien anwenden. Aus einer Aussage $P(\mathbf{C})$ mit einer freien Variable für eine Kategorie $\mathbf{C}$ lässt sich immer eine duale Aussage $P^{\mathbf{op}}$ mit $P^{\mathbf{op}}(\mathbf{C}) \Leftrightarrow P(\mathbf{C^{op}})$ ableiten. Insbesondere gibt es zu jedem kategoriellen Konzept, das durch eine Aussage $P(\mathbf{C})$ definiert wird, ein duales Konzept, das durch $P^{\mathbf{op}}(\mathbf{C})$ definiert wird. Eigenschaften eines Konzeptes übertragen sich dadurch auf duale Eigenschaften des dualen Konzeptes und aus jedem Beweis ergibt sich ein dualer Beweis, was die Effektivität von Beweisen in der Kategorientheorie letztlich verdoppelt.

4.1.2. Pfade und Diagramme

Ein wichtiger Baustein der kategorientheoretischen Arbeitsweise sind Diagramme bzw. kommutierende Diagramme.

Diagramme

Mit Diagrammen kann man einen Ausschnitt aus einer Kategorie oder aus einem Köcher beschreiben, wie es im vorigen Unterabschnitt bereits geschehen ist, ohne explizit von Diagrammen zu reden. So wurden in Definition 6 Graphhomomorphismen mit der Grafik aus Abb. 26 illustriert. Eine solche Grafik kann man als ein Diagramm D auffassen.

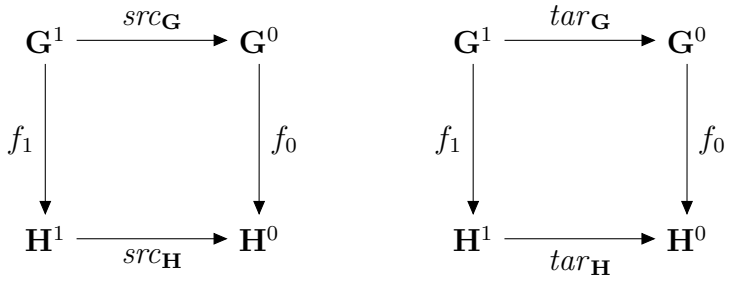


Abbildung 26: Diagramm für Graphhomomorphismen

Kommutierende Diagramme

Bei *kommutierenden* Diagrammen handelt es sich um eine Möglichkeit, Zusammenhänge zwischen den Pfeilen einer Kategorie visuell auszudrücken, die man auch mittels Gleichungen erfassen könnte. Die Idee ist, dass jeder Pfad zwischen zwei Knoten eines kommutierenden Diagrammes zum selben Pfeil komponieren muss. Mehrfachvorkommen von Knoten werden dabei nicht berücksichtigt. Beispielsweise bedeutet die Aussage, dass das obige Diagramm D kommutiert, dass die Gleichungen $f_1 \bullet src_{\mathbf{H}} = src_{\mathbf{G}} \bullet f_0$ (linke Hälfte) und $f_1 \bullet tar_{\mathbf{H}} = tar_{\mathbf{G}} \bullet f_0$ (rechte Hälfte) gelten. Nach Definition 6 könnte man genau so gut sagen, dass $(\mathbf{G}, \mathbf{H}, f_0, f_1)$ ein Graphhomomorphismus ist, also kann man den Begriff des Graphhomomorphismus mithilfe der Kommutativität von Diagrammen definieren.

Überblick

Diagramme und deren Kommutativität sind aber nicht nur ein Werkzeug, mit dem man Gleichungen ausdrücken kann, sondern man kann beides formal erfassen und mathematisch untersuchen, was im Folgenden geschehen wird. Zunächst werden Diagramme formalisiert. Danach werden Pfade und deren Komposition untersucht, die notwendig sind um anschließend Kommutativität von Diagrammen formal zu definieren.

Diagramme sind formal lediglich eine andere Betrachtungsweise von Köcherhomomorphismen, ähnlich wie Familien eine andere Betrachtungsweise von Funktionen sind:

Definition 21 (Diagramm)

Diagramm

Seien $\mathbf{I}, \mathbf{G}$ Köcher.

Ein **Diagramm** D der Form $\mathbf{I}$ in $\mathbf{G}$ ist ein Köcherhomomorphismus $D : \mathbf{I} \to \mathbf{G}$.

$\mathbf{I}$ wird auch als Formköcher von D bezeichnet, oder als Formgraph falls er klein ist.

Ein Diagramm ist genau dann **klein**, wenn $\mathbf{I}$ klein ist, also wenn $\mathbf{I} \in \mathfrak{Graphs}$. □

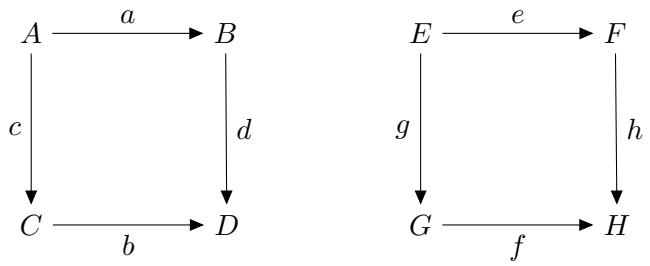
Wie diese Definition mit einem Diagramm wie dem in Abb. 26 zusammenhängt, soll das folgende Beispiel erläutern:

Beispiel 22 (Diagramm)

Diagramm

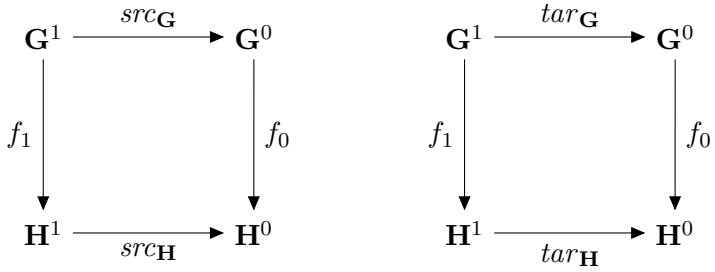
Ein endliches Diagramm $D : I \to \mathbf{C}$ der Form I visualisiert man, indem man im Grunde die Knoten und Pfeile des Formköchers I zeichnet, diese aber nicht mit den eigenen Bezeichnungen markiert, sondern mit den Bezeichnungen ihrer Bilder unter D.

Beispielsweise könnte das Diagramm D in Abb. 26 den folgenden Formköcher I haben:



Da das Diagramm D einen Ausschnitt aus der Kategorie **Set** der Mengen und Funktionen abbildet, handelt es sich formal gesehen um einen Köcherhomomorphismus $D : I \to \mathbf{Set}$. Dabei bildet D die Knoten nach der Vorschrift $A \mapsto \mathbf{G}^1$, $B \mapsto \mathbf{G}^0$, $C \mapsto \mathbf{H}^1$, $D \mapsto \mathbf{H}^0$, $E \mapsto \mathbf{G}^1$, $F \mapsto \mathbf{G}^0$, $G \mapsto \mathbf{H}^1$, $H \mapsto \mathbf{H}^0$ ab und die Pfeile nach der Vorschrift $a \mapsto src_{\mathbf{G}}$, $b \mapsto src_{\mathbf{H}}$, $c \mapsto f_1$, $d \mapsto f_0$, $e \mapsto tar_{\mathbf{G}}$, $f \mapsto tar_{\mathbf{H}}$, $g \mapsto f_1$, $h \mapsto f_0$.

Wenn man D nun wie angegeben zeichnet, erhält man genau das gewünschte Diagramm:



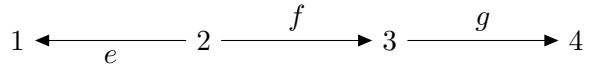
□

Beispiel 23 (Bedeutung von Diagrammen)

Bedeutung von Diagrammen

Ein Diagramm repräsentiert einen Ausschnitt aus einer Kategorie. Es wählt eine Zusammenstellung von Objekten und Pfeilen der Kategorie aus. Hierbei stehen die ausgewählten Objekte in bestimmten Beziehungen, die durch die ausgewählten Pfeile repräsentiert werden. Durch den Formköcher **G** wird festgelegt, *welche* Objekte und *welche* Beziehungen zwischen *welchen* Objekten erfasst werden. Die *Bedeutung* der ausgewählten Objekte und Beziehungen hängt hingegen von der Kategorie **C** ab.

Sei beispielsweise der folgende Formgraph **G** gegeben:



Ein Diagramm $D : \mathbf{G} \to \mathbf{C}$ dieser Form in einer beliebigen Kategorie **C** wählt vier Objekte $D(1)$, $D(2)$, $D(3)$ und $D(4)$ und drei Pfeile $D(2) \xrightarrow{D(e)} D(1)$, $D(2) \xrightarrow{D(f} D(3)$ und $D(3) \xrightarrow{D(g)} D(4)$ aus. Die Pfeile e, f und g stehen für Beziehungen zwischen den Objekten, deren Natur von **C** abhängt. Wenn beispielsweise $\mathbf{C} = \mathbf{Graphs}$ die Kategorie der Graphen und Graphhomomorphismen ist, kann man einen Pfeil $X \xrightarrow{m} Y \in \mathbf{Graphs}^1$ als Vorkommen des Graphen X im Graphen Y interpretieren, wie in Unterabschnitt 3.2.1 erklärt wurde. Somit steht in diesem Fall jeder Pfeil des Formgraphen für ein Vorkommen eines Graphen in einem anderen. □

Wie bereits erwähnt, kann man mit Diagrammen Ausschnitte von Kategorien beschreiben. Kommutative Diagramme bauen diese Art der Beschreibung aus, indem sie Zusammenhänge zwischen Pfeilen der Kategorie herstellen. Die Bedeutung eines kommutativen Diagramms wird mithilfe der Komposition von Pfaden definiert. Zwei Pfade eines Diagrammes mit gleichen Anfangs- und Endknoten kommutieren genau dann, wenn ihre Komposition jeweils denselben Pfeil ergibt. Ein Diagramm kommutiert genau dann, wenn alle darin enthaltenen Pfade mit gleichen Anfangs- und Endknoten kommutieren.

Zur formalen Erfassung von Kommutativität müssen deshalb die Komposition von Pfaden und hierfür wiederum die Pfade formalisiert werden. Deshalb werden hier zunächst Pfade und damit zusammenhängende Konstruktionen untersucht.

Formal lassen sich Pfade folgendermaßen erfassen:

Definition 24 (Pfad)

Pfad

Sei $\mathbf{G}$ ein Köcher.
Sei $\mathbf{G}^0$ die Klasse der Knoten von $\mathbf{G}$.
Sei $\mathbf{G}^1$ die Klasse der Pfeile von $\mathbf{G}$.
Sei $\mathbf{G}^2$ die Klasse der komponierbaren Paare von $\mathbf{G}$.

$\mathbf{G}^0$ wird auch bezeichnet als Klasse der **Pfade der Länge 0**, $\mathbf{G}^1$ als Klasse der **Pfade der Länge 1** und $\mathbf{G}^2$ als Klasse der **Pfade der Länge 2** in $\mathbf{G}$.
Für $n \in \mathbb{N}\backslash\{0,1\}$ ist

$$\mathbf{G}^{n+1} := \left\{(s_1, s_2, \ldots, s_n, s_{n+1}) \;\middle|\; (s_1, s_2, \ldots, s_n) \in \mathbf{G}^n \wedge s_{n+1} \in \mathbf{G}^1 \wedge src(s_{n+1}) = tar(s_n)\right\}$$

die Klasse der **Pfade der Länge n + 1** in $\mathbf{G}$.
Die Klasse der **Pfade** in $\mathbf{G}$ ist die disjunkte Vereinigung $\mathbf{G}^* = \biguplus_{n\in\mathbb{N}} \mathbf{G}^n$.
Die Klasse der **nichtleeren Pfade** in $\mathbf{G}$ ist $\mathbf{G}^+ = \biguplus_{\substack{n\in\mathbb{N}\\ n>0}} \mathbf{G}^n$.

Einen Pfad $p \in \mathbf{G}^*$ der Länge $n \in N$ kann man anschaulicher folgendermaßen schreiben:

$$p = (k_0 \xrightarrow{s_1} k_1 \xrightarrow{s_2} \ldots \xrightarrow{s_{n-1}} k_{n-1} \xrightarrow{s_n} k_n)$$

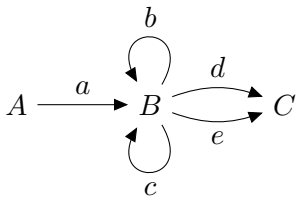
Für einen solchen Pfad p definiere $len(p) := n$, $src(p) := k_0$ und $tar(p) := k_n$.
Falls $p \in \mathbf{G}^+$ (also $len(p) > 0$), definiere:

- $head(p) = (k_0 \xrightarrow{s_1} k_1) \in \mathbf{G}[k_0, k_1]$ ist der Anfangspfeil von p
- $tail(p) = (k_1 \xrightarrow{s_2} k_2 \xrightarrow{s_3} \ldots \xrightarrow{s_{n-1}} k_{n-1} \xrightarrow{s_n} k_n) \in \mathbf{P}_\mathbf{G}[k_1, k_n]$ ist der Restpfad von p.
- $foot(p) = (k_{n-1} \xrightarrow{s_n} k_n) \in \mathbf{G}[k_{n-1}, k_n]$ ist der Endpfeil von p
- $body(p) = (k_0 \xrightarrow{s_1} k_1 \xrightarrow{s_2} \ldots \xrightarrow{s_{n-2}} k_{n-2} \xrightarrow{s_{n-1}} k_{n-1}) \in \mathbf{P}_\mathbf{G}[k_0, k_{n-1}]$ ist der Rumpf von p. □

Beispiel 25 (Pfade)

Pfade

Sei der folgende Graph G gegeben:



In G sind unter anderem folgende Pfade enthalten:

1. Länge 0: $A,\ B,\ C$
2. Länge 1: $A \xrightarrow{a} B,\ B \xrightarrow{b} B,\ B \xrightarrow{c} B,\ B \xrightarrow{d} C,\ B \xrightarrow{E} C$
3. Länge 2: $A \xrightarrow{a} B \xrightarrow{d} C,\ B \xrightarrow{b} B \xrightarrow{c} B$
4. Länge 7: $A \xrightarrow{a} B \xrightarrow{b} B \xrightarrow{b} B \xrightarrow{c} B \xrightarrow{b} B \xrightarrow{c} B \xrightarrow{e} C$ □

Aus den Pfaden $p_1 = (A \xrightarrow{a} B)$ und $p_2 = (B \xrightarrow{d} C)$ aus Beispiel 25 kann man $p_3 = (A \xrightarrow{a} B \xrightarrow{d} C)$ konstruieren, indem man sie zu einem neuen Pfad zusammenfügt. Dieses Zusammenfügen kann man als eine Rechenoperation $p_3 = p_1 \parallel p_2$ auffassen, die allgemein auf Paare von Pfaden angewendet werden kann, von denen der erste im Anfangsknoten des zweiten endet:

Definition 26 (Verkettung von Pfaden)
Verkettung von Pfaden

Sei $\mathbf{G}$ ein Köcher.

Seien $p = (a \xrightarrow{s_1} k_1 \xrightarrow{s_2} \dots \xrightarrow{s_n} b)$ und $q = (b \xrightarrow{t_1} l_1 \xrightarrow{t_2} \dots \xrightarrow{t_m} c)$ Pfade in $\mathbf{G}$ mit $tar(p) = src(q) = b \in \mathbf{G}^0$.

Dann ist

$$p \parallel q := (a \xrightarrow{s_1} k_1 \xrightarrow{s_2} \dots \xrightarrow{s_n} b \xrightarrow{t_1} l_1 \xrightarrow{t_2} \dots \xrightarrow{t_m} c)$$

die **Verkettung** von p und q. □

Die Pfade in einem Köcher $\mathbf{G}$ bilden selbst einen Köcher, weil jeder Pfad einen Anfangs- und einen Endknoten hat. Zusammen mit der Verkettung von Pfaden bilden sie sogar eine Kategorie, nämlich die Pfadkategorie $\mathbf{P_G}$:

Definition 27 (Pfadkategorie)
Pfadkategorie

Sei $\mathbf{G}$ ein Köcher.

Definiere die **Pfadkategorie** (auch: die **freie** Kategorie) $\mathbf{P_G}$ über $\mathbf{G}$ folgendermaßen:

- Die Objekte sind die Knoten von $\mathbf{G}$ und stehen für die Anfangs- und Endknoten der Pfade: $\mathbf{P}^0_\mathbf{G} = \mathbf{G}^0$
- Die Pfeile sind die Pfade p beliebiger endlicher Länge: $\mathbf{P}^1_\mathbf{G} = \mathbf{G}^*$
  Für einen Pfad $p = (k_0 \xrightarrow{s_1} k_1 \xrightarrow{s_2} \dots \xrightarrow{s_n} k_n)$ ist $src(p) = k_0$ und $tar(p) = k_n$.
- Die Komposition von zwei Pfeilen (also Pfaden) $a \xrightarrow{p} b$ und $b \xrightarrow{q} c$ ist die Verkettung von p und q: $p \bullet q = p \parallel q$. □

Wenn man eine Pfadkategorie bildet, erhält man von einem Köcher ausgehend eine Kategorie. Diese Konstruktion lässt sich auf Köcherhomomorphismen ausdehnen: Einen Köcherhomomorphismus kann man auf Pfade erweitern und erhält so eine Abbildung zwischen den Pfadkategorien, die im Rahmen von funktionalen Programmiersprachen oft mit dem Namen *map* bezeichnet wird:

Definition 28 (Map)

Map

Seien $\mathbf{G}$ und $\mathbf{H}$ Köcher.
Sei $f : \mathbf{G} \to \mathbf{H}$ ein Köcherhomomorphismus.

Definiere $\mathbf{P}_f : \mathbf{P_G} \to \mathbf{P_H}$ folgendermaßen:

- Da die Objekte einer Köchers und seiner Pfadkategorie gleich sind, kann man sie nach der Vorschrift von f abbilden, also für $A \in (\mathbf{P_G})^0 = \mathbf{G}^0$:
  $\mathbf{P}_f(A) := f(A) \in \mathbf{H}^0 = (\mathbf{P_H})^0$
- Die Pfade der Pfadkategorie bestehen aus Pfeilen von G, auf die man ebenfalls f anwenden kann, also definiere für einen Pfad $(k_0 \xrightarrow{s_1} k_1 \xrightarrow{s_2} \ldots \xrightarrow{s_n} k_n) \in \mathbf{G}^*$:

$$\mathbf{P}_f(k_0 \xrightarrow{s_1} k_1 \xrightarrow{s_2} \ldots \xrightarrow{s_n} k_n) :=$$
$$(f(k_0) \xrightarrow{f(s_1)} f(k_1) \xrightarrow{f(s_2)} \ldots \xrightarrow{f(s_n)} f(k_n)) \quad \in \quad \mathbf{H}^*$$ □

Beweis 29 ($\mathbf{P}_f$ ist ein Funktor)

$\mathbf{P}_f$ ist ein Funktor

Für einen Köcherhomomorphismus $f : \mathbf{G} \to \mathbf{H}$ ist die Abbildung $\mathbf{P}_f : \mathbf{P_G} \to \mathbf{P_H}$ ein Funktor: Wie man leicht überprüft, ist $\mathbf{P}_f$ ein Köcherhomomorphismus und bildet Identitätspfeile auf Identitätspfeile (also leere Pfade auf leere Pfade) ab. Außerdem ist $\mathbf{P}_f$ mit der Komposition verträglich, denn es gilt für beliebige Pfade $p = (a \xrightarrow{s_1} k_1 \xrightarrow{s_2} \ldots \xrightarrow{s_n} b)$ und $q = (b \xrightarrow{t_1} l_1 \xrightarrow{t_2} \ldots \xrightarrow{t_m} c)$ in $\mathbf{P_G}$:

$$\begin{aligned}
&\mathbf{P}_f(p \parallel q) \\
&= \mathbf{P}_f(a \xrightarrow{s_1} k_1 \xrightarrow{s_2} \ldots \\
&\xrightarrow{s_n} b \xrightarrow{t_1} l_1 \xrightarrow{t_2} \ldots \xrightarrow{t_m} c) \\
&= (f(a) \xrightarrow{f(s_1)} f(k_1) \xrightarrow{f(s_2)} \ldots \\
&\xrightarrow{f(s_n)} f(b) \xrightarrow{f(t_1)} f(l_1) \xrightarrow{f(t_2)} \ldots \xrightarrow{f(t_m)} f(c)) \\
&= (f(a) \xrightarrow{f(s_1)} f(k_1) \xrightarrow{f(s_2)} \ldots \\
&\xrightarrow{f(s_n)} f(b)) \parallel (f(b) \xrightarrow{f(t_1)} f(l_1) \xrightarrow{f(t_2)} \ldots \xrightarrow{f(t_m)} f(c)) \\
&= \mathbf{P}_f(p) \parallel \mathbf{P}_f(q)
\end{aligned}$$

□

Mithilfe der vorangehenden Definitionen kann man aus einem Graphen G eine kleine Kategorie $\mathbf{P}_G$ und aus einem Graphhomomorphismus $f : \mathbf{G} \to \mathbf{H}$ einen Funktor $\mathbf{P}_f : \mathbf{P}_G \to \mathbf{P}_H$ konstruieren. Anders formuliert: Man kann aus Objekten und Pfeilen der Kategorie **Graphs** Objekte und Pfeile der Kategorie **Cat** konstruieren. Diese Konstruktion lässt sich durch einen Funktor von **Graphs** nach **Cat** beschreiben:

Definition 30 (Freier Funktor)
Definiere $\mathbf{P}_{(\cdot)} : \mathbf{Graphs} \to \mathbf{Cat}$ folgendermaßen: *Freier Funktor*

- Jeder Graph $G \in \mathbf{Graphs}^0$ wird auf seine Pfadkategorie $\mathbf{P}_G \in \mathbf{Cat}^0$ abgebildet.
- Jeder Graphhomomorphismus $f : G \to H \in \mathbf{Graphs}^1$ wird auf Pfade erweitert, also abgebildet auf $\mathbf{P}_f : \mathbf{P}_G \to \mathbf{P}_H \in \mathbf{Cat}^1$.

$\mathbf{P}_{(\cdot)}$ heißt freier Funktor und ist auch tatsächlich ein Funktor [BW12, S. 67].
Deshalb gilt für komponierbare Paare von Graphhomomorphismen $G \xrightarrow{g} H \xrightarrow{h} K$:

$$\mathbf{P}_{g \bullet h} = \mathbf{P}_g \bullet \mathbf{P}_h$$

In der für diese Arbeit vorgeschlagenen Mengenlehre ARC kann man $\mathbf{P}_{(\cdot)}$ auch analog zu dieser Definition als Funktor der Form $\mathbf{P}_{(\cdot)} : \mathbf{Quiv} \to \mathbf{CAT}$ definieren, der einer zu $\mathfrak{Set}$ gleichmächtigen Köcher seine zugehörige Pfadkategorie zuordnet. □

Da nun Pfade formalisiert sind, kann man jetzt die formale Definition der Komposition von Pfaden angehen, um schließlich die Kommutativität von Diagrammen zu formalisieren.

Definition 31 (Komposition eines Pfades)
Sei $\mathbf{C}$ eine Kategorie. *Komposition eines Pfades*

Die Komposition $\mathbf{C}(\cdot) : \mathbf{C}^* \to \mathbf{C}^1$ eines Pfades $(k_0 \xrightarrow{f_1} k_1 \xrightarrow{f_2} \dots \xrightarrow{f_n} k_n)$ von $\mathbf{C}$ ist definiert durch $\mathbf{C}(k_0 \xrightarrow{f_1} k_1 \xrightarrow{f_2} \dots \xrightarrow{f_n} k_n) = f_1 \bullet f_2 \bullet \dots \bullet f_n$, oder formaler:

- Die Komposition des leeren Pfades von $A \in \mathbf{C}^0$ nach A ist die Identität von A: $\mathbf{C}(A) = id_A : A \to A$
- Um die Komposition eines nichtleeren Pfades $p = (k_0 \xrightarrow{f_1} k_1 \xrightarrow{f_2} \dots \xrightarrow{f_n} k_n)$ zu definieren, wird zunächst der Anfangspfad $body(p) = (k_0 \xrightarrow{f_1} k_1 \xrightarrow{f_2} \dots \xrightarrow{f_{n-1}} k_{n-1})$ rekursiv komponiert und dann das Ergebnis mithilfe der binären Komposition $\bullet$ mit dem letzten Pfeil $foot(p) = f_n$ komponiert: $\mathbf{C}(p) = \mathbf{C}(body(p)) \bullet foot(p)$ □

Beispiel 32 (Komposition von Pfaden)
Die Komposition $\mathbf{C}(\cdot)$ von Pfaden ist eine Verallgemeinerung der binären Komposition $\bullet$ einer Kategorie $\mathbf{C}$ auf beliebige Pfade. Während $\bullet$ nur komponierbaren Paaren, also Pfaden der Länge 2 einen Pfeil in $\mathbf{C}$ als Ergebnis der Komposition zuordnet, ordnet $\mathbf{C}(\cdot)$ allen Pfaden einen Pfeil zu. *Komposition von Pfaden*

Beispielsweise ist die Komposition eines Pfades $p = (A \xrightarrow{f} B \xrightarrow{g} C \xrightarrow{h} D \xrightarrow{j} E)$ der Länge 4 in einer Kategorie $\mathbf{C}$ nach obiger Definition gleich $\mathbf{C}(p) = ((f \bullet g) \bullet h) \bullet j = f \bullet g \bullet h \bullet j$.

Um $\mathbf{C}(\cdot)$ zurecht als Verallgemeinerung von $\bullet$ zu bezeichnen, sollte seine Einschränkung auf komponierbare Paare auch mit $\bullet$ übereinstimmen, was tatsächlich der Fall ist: Wenn $p = (A \xrightarrow{f} B \xrightarrow{g} C) \in \mathbf{C}^2$ ein komponierbares Paar ist, dann gilt:

$$\mathbf{C}(p) = \mathbf{C}(A \xrightarrow{f} B)\bullet g = \mathbf{C}(A)\bullet f \bullet g = id_A \bullet f \bullet g = f \bullet g$$

Einem einzelnen Pfeil $A \xrightarrow{f} B$, also einem Pfad der Länge 1, ordnet $\mathbf{C}(\cdot)$ sinnvollerweise sich selbst zu (denn $\mathbf{C}(A \xrightarrow{f} B) = \mathbf{C}(A)\bullet f = id_A \bullet f = f$) – ähnlich verhalten sich beispielsweise auch verallgemeinerte Verknüpfungen wie die Summe $\sum$ oder die verallgemeinerte Multiplikation $\prod$, die einer einelementigen Menge deren einziges Element zuordnen.

Da die leeren Pfade die Identitätspfeile in der Pfadkategorie $\mathbf{P_C}$ einer Kategorie $\mathbf{C}$ sind, ist es naheliegend, einem leeren Pfad $(A) \in \mathbf{C}^*$ wie definiert den Identitätspfeil id_A in $\mathbf{C}$ zuzuordnen. Da leere Pfade in $\mathbf{C}^*$ per Definition dasselbe sind wie die Objekte von $\mathbf{C}$, ergibt sich hieraus der Vorteil, dass durch die Definition von $\mathbf{C}(\cdot)$ die Zuordnung der Identitätspfeile zu den einzelnen Objekten explizit gemacht wird, während in der Definition von Kategorien nur ihre Existenz gefordert wird. Ein weiterer Vorteil hiervon wird sich in Definition 34 zeigen. □

Bemerkung 33 (Notation $\mathbf{C}(\cdot)$)

Notation $\mathbf{C}(\cdot)$

Die Notation $\mathbf{C}(\cdot)$ ist angelehnt an die Tatsache, dass eine Kategorie im Wesentlichen dasselbe ist wie ihre Komposition, da sich die Kategorie von der Komposition ableiten lässt und umgekehrt. □

Die Pfade $\mathbf{C}^*$ einer Kategorie $\mathbf{C}$ bilden nach Definition 27 die Pfeile der Pfeilkategorie $\mathbf{P_C}$ von $\mathbf{C}$. Die Komposition beliebiger Pfade $\mathbf{C}(\cdot)$ einer Kategorie $\mathbf{C}$ ist die Pfeilkomponente eines gleichnamigen Funktors von $\mathbf{P_C}$ nach $\mathbf{C}$, der in der funktionalen Programmierung oft als *reduce* oder *fold* bezeichnet wird:

Definition 34 (Kompositionsfunktor)

Kompositionsfunktor

Sei $\mathbf{C}$ eine Kategorie.

Definiere den Kompositionsfunktor $\mathbf{C}(\cdot) : \mathbf{P_C} \to \mathbf{C}$ folgendermaßen:

- Da die Objekte von $\mathbf{P_C}$ und $\mathbf{C}$ identisch sind, lässt sich definieren, dass jedes Objekt auf sich selbst abgebildet wird: $\mathbf{C}(A) = A$ für $A \in \mathbf{P}^0_\mathbf{C} = \mathbf{C}^0$
- Die Pfeilkomponente ist die Kompositionsfunktion $\mathbf{C}(\cdot)$ aus Definition 31. □

Beweis 35 (Die Komposition von Pfaden ist ein Funktor)

Die Komposition von Pfaden ist ein Funktor

Die Komposition von Pfaden ist ein Funktor: Sei $\mathbf{C}$ eine Kategorie. Für $A \in \mathbf{P}^0_\mathbf{C} = \mathbf{C}^0$ ist $\mathbf{C}(id_A) = \mathbf{C}((A)) = id_A = id_{\mathbf{C}(A)}$, also erhält $\mathbf{C}(\cdot)$ die Identitätspfeile. Für Pfade $p \in \mathbf{P_C}[A, B]$, $q \in \mathbf{P_C}[B, C]$ ist $\mathbf{C}(p\bullet q) = \mathbf{C}(p \parallel q) = \mathbf{C}(p)\bullet\mathbf{C}(q)$, denn per Induktion über $len(q)$ gilt:

- Falls $len(q) = 0$, dann ist $\mathbf{C}(p\bullet q) = \mathbf{C}(p \parallel q) = \mathbf{C}(p) = \mathbf{C}(p)\bullet id_B = \mathbf{C}(p)\bullet\mathbf{C}(q)$

- Gelte nun die Behauptung für alle Pfade der Länge kleiner $n \in \mathbb{N}$ und sei $len(q) = n$. Dann gilt:

$$\begin{aligned}
\mathbf{C}(p \bullet q) &= \mathbf{C}(p \parallel q) \\
&= \mathbf{C}(p \parallel (body(q) \parallel (foot(q)))) \\
&= \mathbf{C}((p \parallel body(q)) \parallel foot(q)) \\
&= \mathbf{C}(p \parallel body(q)) \bullet foot(q) \\
&\overset{\text{IV}}{=} (\mathbf{C}(p) \bullet \mathbf{C}(body(q))) \bullet foot(q) \\
&= \mathbf{C}(p) \bullet (\mathbf{C}(body(q)) \bullet foot(q)) \\
&\overset{\text{IV}}{=} \mathbf{C}(p) \bullet \mathbf{C}(body(q) \parallel (foot(q))) \\
&= \mathbf{C}(p) \bullet \mathbf{C}(q),
\end{aligned}$$

  denn $len(body(q)) = len(q) - 1 < n$, wodurch sich die Induktionsvoraussetzung an den mit IV markierten Stellen anwenden lässt.

Also erhält $\mathbf{C}(\cdot)$ auch die Komposition und ist somit ein Funktor. □

Definition 36 (Komposition von Pfaden in Diagrammen)

Komposition von Pfaden in Diagrammen

Diagramme in einer Kategorie sind Köcherhomomorphismen $D : \mathbf{I} \to \mathbf{C}$ mit einer Kategorie $\mathbf{C}$ als Kodomäne und bilden einen Spezialfall der Köcherhomomorphismen, der für die Definition der Kommutativität von Diagrammen bedeutsam ist.

Indem man zur Pfadkategorie übergeht, erhält man einen Funktor $\mathbf{P}_D : \mathbf{P}_\mathbf{I} \to \mathbf{P}_\mathbf{C}$. Da der Kompositionsfunktor $\mathbf{C}(\cdot) : \mathbf{P}_\mathbf{C} \to \mathbf{C}$ die Pfadkategorie von $\mathbf{C}$ als Domäne hat, kann man die beiden Funktoren komponieren zu $\mathbf{C}_D(\cdot) := \mathbf{P}_D \bullet \mathbf{C}(\cdot)$ und erhält so einen Funktor, der Pfade eines Diagrammes komponiert: $\mathbf{C}_D(\cdot) : \mathbf{P}_\mathcal{I} \to \mathbf{C}$ □

Beispiel 37 (Komposition von Pfaden in Diagrammen)

Komposition von Pfaden in Diagrammen

Wenn man das Diagramm $D : \mathbf{I} \to \mathbf{C}$ aus Beispiel 22 betrachtet, wird beispielsweise der Pfad $p_1 = (A \xrightarrow{c} C \xrightarrow{b} D) \in \mathcal{I}^*$ aus dem Formköcher $\mathbf{I}$ zu $\mathbf{C}_D(p_1) = f_1 \bullet src_H$ und der Pfad $p_2 = (A \xrightarrow{a} B \xrightarrow{d} D) \in \mathcal{I}^*$ zu $\mathbf{C}_D(p_2) = src_G \bullet f_0$ komponiert. □

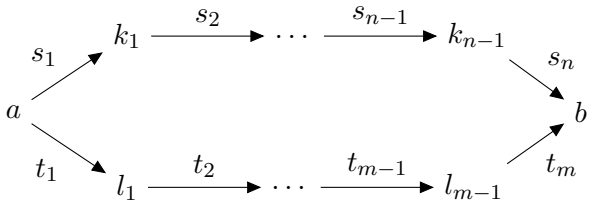
Hiermit kann man nun Kommutativität von Diagrammen formalisieren:

Definition 38 (Kommutatives Diagramm [BW12])

Kommutatives Diagramm [BW12]

Sei $\mathbf{I}$ ein Köcher.
Sei $\mathbf{C}$ eine Kategorie.
Sei $D : \mathbf{I} \to \mathbf{C}$ ein Diagramm der Form $\mathbf{I}$ in $\mathbf{C}$.

D kommutiert (oder: ist kommutativ) in $\mathbf{C}$ genau dann, wenn für jedes Paar $A, B \in \mathbf{I}^0$ von Knoten des Formköchers $\mathbf{I}$ die Komposition von je zwei Pfaden $p, q \in \mathbf{P}_\mathbf{I}[A, B]$ von A nach B immer denselben Pfeil ergibt, also wenn $\mathbf{C}_D(p) = \mathbf{C}_D(q)$ für alle Pfade p, q von A nach B. D kommutiert also genau dann, wenn für jedes Paar von Pfaden



im Formköcher $\mathbf{I}$ mit gleichen Anfangs- und Endknoten a und b gilt:

$$D(s_1)\bullet D(s_2)\bullet \ldots \bullet D(s_{n-1})\bullet D(s_n) = D(t_1)\bullet D(t_2)\bullet \ldots \bullet D(t_{m-1})\bullet D(t_m)$$

Wenn D kommutiert, wird hier in Formeln auch $comm_{\mathbf{C}}(D)$ geschrieben. Wenn D selbst in der Illustration eines anderen Diagrammes als Pfeil mit Ziel $\mathbf{C}$ vorkommt, wird dort ${}^cD$ anstatt D geschrieben, um auszusagen, dass D in $\mathbf{C}$ kommutiert. Um in der Illustration eines Diagrammes auszusagen, dass die vollständig an einer leeren Fläche angrenzenden Pfade kommutieren, wird die leere Fläche mit $\sim$ markiert. □

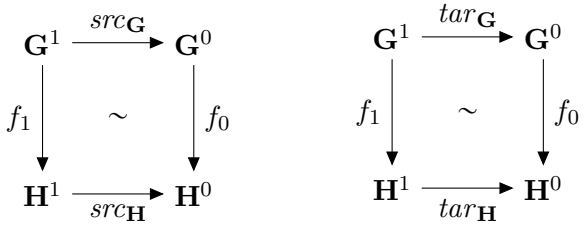
Beispiel 39 (Kommutatives Diagramm)

Kommutatives Diagramm

Das Diagramm $D : \mathbf{I} \to \mathbf{C}$ aus Beispiel 22 bzw. dessen Formköcher $\mathbf{I}$ enthält zwei Paare von Pfaden mit gleichem Endknoten, nämlich $p_1 = (A \xrightarrow{c} C \xrightarrow{b} D) \in \mathcal{I}^*$ und $p_2 = (A \xrightarrow{a} B \xrightarrow{d} D) \in \mathcal{I}^*$ bzw. $q_1 = (E \xrightarrow{g} G \xrightarrow{f} H) \in \mathcal{I}^*$ und $q_2 = (E \xrightarrow{e} f \xrightarrow{h} H) \in \mathcal{I}^*$.

Wenn D kommutiert, müssen nach Definition 38 diese beiden Paare von Pfaden jeweils zu dem selben Anfangsknoten komponieren, d.h. $\mathbf{C}_D(p_1) = \mathbf{C}_D(p_2)$ und $\mathbf{C}_D(q_1) = \mathbf{C}_D(q_2)$. Da p_1 und p_2 zu $\mathbf{C}_D(p_1) = f_1\bullet src_H$ bzw. $\mathbf{C}_D(p_2) = src_G\bullet f_0$ und q_1 und q_2 zu $\mathbf{C}_D(q_1) = f_1\bullet tar_H$ bzw. $\mathbf{C}_D(q_2) = tar_G\bullet f_0$ komponieren, sind diese Aussagen gleichbedeutend zu $f_1\bullet src_H = src_G\bullet f_0$ bzw. $f_1\bullet tar_H = tar_G\bullet f_0$. Da dies der Definition eines Graphhomomorphismus $(\mathbf{G}, \mathbf{H}, f_0, f_1)$ entspricht, ist somit die Definition von Graphhomomorphismen selbst als ein mathematisches Objekt erfasst, nämlich durch das Diagramm D und seine Pfade.

Nach den in Definition 38 eingeführten Konventionen würde man D als kommutatives Diagramm folgendermaßen illustrieren:



□

Beispiel 40 (Zyklen und Schleifen in kommutierenden Diagrammen)

Zyklen und Schleifen in kommutierenden Diagrammen

Eine Besonderheit ergibt sich bei Zyklen in kommutativen Diagrammen. Sei beispielsweise $z = (A = k_0 \xrightarrow{s_1} k_1 \xrightarrow{s_2} \ldots \xrightarrow{s_{n-1}} k_{n-1} \xrightarrow{s_n} k_n = A) \in \mathbf{P}_{\mathbf{I}}[A, A]$ ein Pfad von A nach A im Formgraphen eines kommutativen Diagrammes $D : \mathbf{I} \to \mathbf{C}$.

Da der leere Pfad (A) ebenso wie z ein Pfad von A nach A ist, muss nach Definition 38 die Komposition von z denselben Pfeil ergeben wie die Komposition von (A), also den Identitätspfeil $D(A) \xrightarrow{id_{D(A)}} D(A)$. Da auch Schleifen Zyklen sind, steht insbesondere jede Schleife in einem kommutativen Diagramm für den entsprechenden Identitätspfeil. □

4.1.3. Limites und Kolimites

Limites und Kolimites (Singular: Limes bzw. Kolimes) sind zueinander duale kategorientheoretische Arten, mathematische Konstruktionen unabhängig von einer bestimmten Kategorie zu beschreiben. Sie werden in dieser Arbeit gebraucht, um Kategorien zu repräsentieren, mit denen später Graphvarianten definiert werden. Darüber hinaus kann man ausgehend von einer Kategorie von Graphen Limites und Kolimites in der Kategorie anwenden, um Programmanipulationen und -analysen damit zu beschreiben. Spezialfälle von Limites und Kolimites sind beispielsweise Pushouts zur Modellierung von Programmmanipulationen und -analysen, Pullbacks, initiale Objekte, Equalizer und Koprodukte, die zur Modellierung von Containment verwendet werden.

Überblick

In diesem Unterabschnitt werden zunächst anhand des Pushouts von Graphen, der ein Spezialfall des Kolimes ist, die relevanten Aspekte eines Kolimes und der zugehörigen Konzepte erklärt. Anschließend werden allgemeine Kolimites formal definiert. Daraufhin werden die zu Pushouts und Kolimes dualen Konstruktionen, nämlich Pullback und Limes untersucht. Abschließend werden einige Arten von Limites und Kolimites definiert, die für die Definition der Graphvarianten oder für die Modellierung der Programmmanipulationen und -analysen benötigt werden.

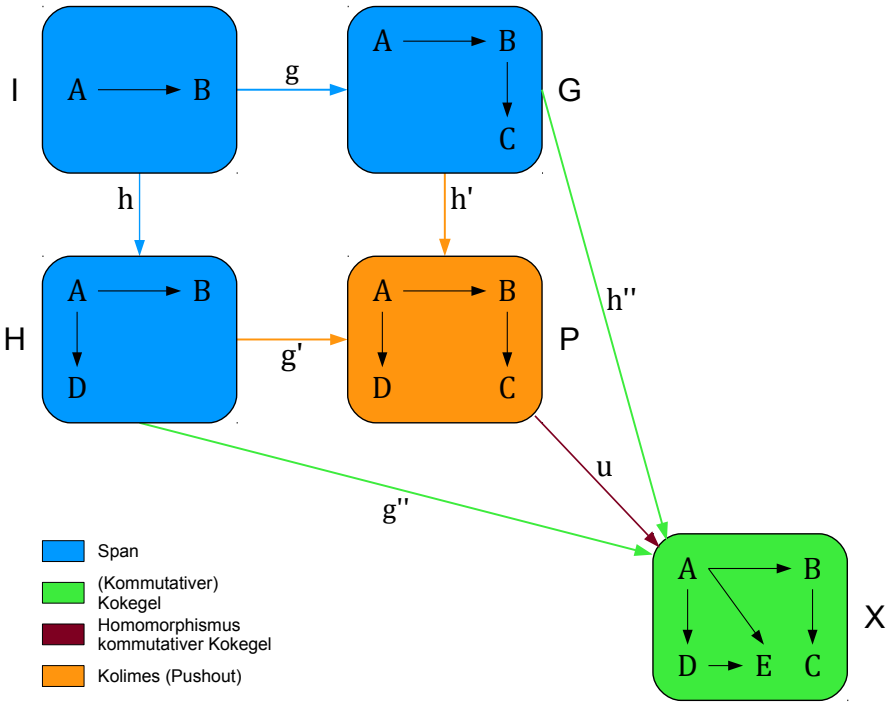


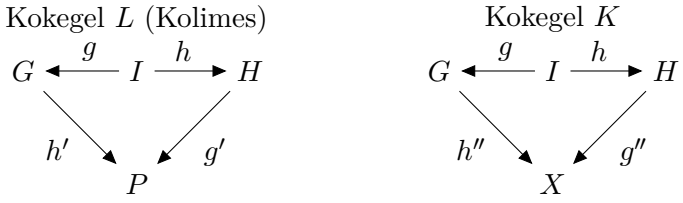
Abbildung 27: Pushout von Graphen

Pushouts

Bei Pushouts in der Kategorie der Graphen handelt es sich um die Verschmelzung zweier Graphen entlang eines gemeinsamen Teilgraphen. Beispielsweise illustriert Abb. 27 die Verschmelzung der Graphen G und H entlang des gemeinsamen Teilgraphen I zu einem Pushoutgraphen P. Genauer gesagt enthält der Pushoutgraph P Vorkommen der Graphen G und H, welche durch die Graphhomomorpismen h' und g' repräsentiert werden. Es sei in

Erinnerung gerufen, dass allgemein ein Vorkommen eines Teilgraphen in einem anderen Graphen durch einen Graphhomomorphismus repräsentiert wird (siehe Unterabschnitt 3.2.1). Beispielsweise repräsentiert in Abb. 27 der Graphhomomorphismus $g : I \to A$ das Vorkommen von I in G, indem er dem Knoten A in I den Knoten A in H, dem Knoten H in G den Knoten H in H und den unbenannten Pfeil zwischen A und H in I die unbenannten Pfeile zwischen den Knoten A und H in G zuordnet[4].

Die Eingabedaten eines Pushout sind die Graphen I, G und H zusammen mit den Homomorphismen g und h, die die Vorkommen von I in G und H repräsentieren. Sie bilden einen sogenannten Span, ein Diagramm S von der Art $G \xleftarrow{g} I \xrightarrow{h} H$, das man als Diagramm mit dem Formgraphen $\mathrm{PO} \coloneqq (1 \xleftarrow{l} 0 \xrightarrow{r} 2)$ auffassen kann. Das Ergebnis L des Pushout heißt Kolimes über S, oder einfach Pushout über S. Es besteht aus dem Span S, dem Pushoutgraph P, und den Homomorphismen $H \xrightarrow{g'} P$ und $G \xrightarrow{h'} P$, die die Vorkommen von H bzw. G in P repräsentieren. Allgemeiner wird eine solche Zusammenstellung als Kokegel über dem Span S mit Scheitel P bezeichnet. In Abb. 27 ist neben L auch noch ein weiterer Kokegel K über dem Span S mit Scheitel X abgebildet. Diese beiden Kokegel kann man also als Diagramme wie folgt darstellen:



Ein Kokegel kann aber den Teilgraphen I mehrmals enthalten, wenn die Vorkommen von I in G bzw. H auf unterschiedliche Vorkommen von I im Scheitelgraphen abgebildet werden. Ein Kokegel, der I nur einmal enthält, wird als kommutativer Kokegel bezeichnet; andernfalls heißt er nichtkommutativ. Beispielsweise sind in Abb. 28 zwei kommutative Kokegel K_1 und K_2 und einen nichtkommutativen Kokegel K_3 abgebildet. Formal betrachtet ist ein Kokegel wie K kommutativ, wenn $g \bullet h''$ und $h \bullet g''$ das selbe Vorkommen von I in X repräsentieren, also wenn den das obige Diagramm kommutiert.

Ein Kolimes (Pushout) über einem Span von Graphen ist ein kommutativer Kokegel, der die in 3.3.2 erläuterte universelle Eigenschaft erfüllt. Diese besagt, dass P nur Knoten und Pfeile hat, die in G oder H vorkommen und der außerdem keine Knoten oder Pfeile miteinander verschmilzt, es sei denn es ist aufgrund von f oder g zwingend nötig. Diese Beschreibung von Pushouts ist aber nur für Graphen und nicht für beliebige Kategorien anwendbar. Für eine kategorienunabhängige formale Definition werden sogenannte Homomorphismen kommutativer Kokegel als Hilfskonstruktionen benötigt. In Abb. 27 und Abb. 28 sind neben dem Span S und den verschiedenen Kokegeln solche Homomorphismen kommutativer Kokegel u, u_1, u_2 und u_3 illustriert. Zur Definition hiervon seien die folgenden kommutativen Kokegel K_1 und K_2 über S mit Scheiteln X_1 bzw. X_2 und ein Pfeil (Graphhomomorphismus) $X_1 \xrightarrow{m} X_2$ gegeben:

[4] Für Abb. 27 soll $g(A) = A$, $g(B) = B$, $g'(D) = D$, $u(C) = C$ usw. gelten.

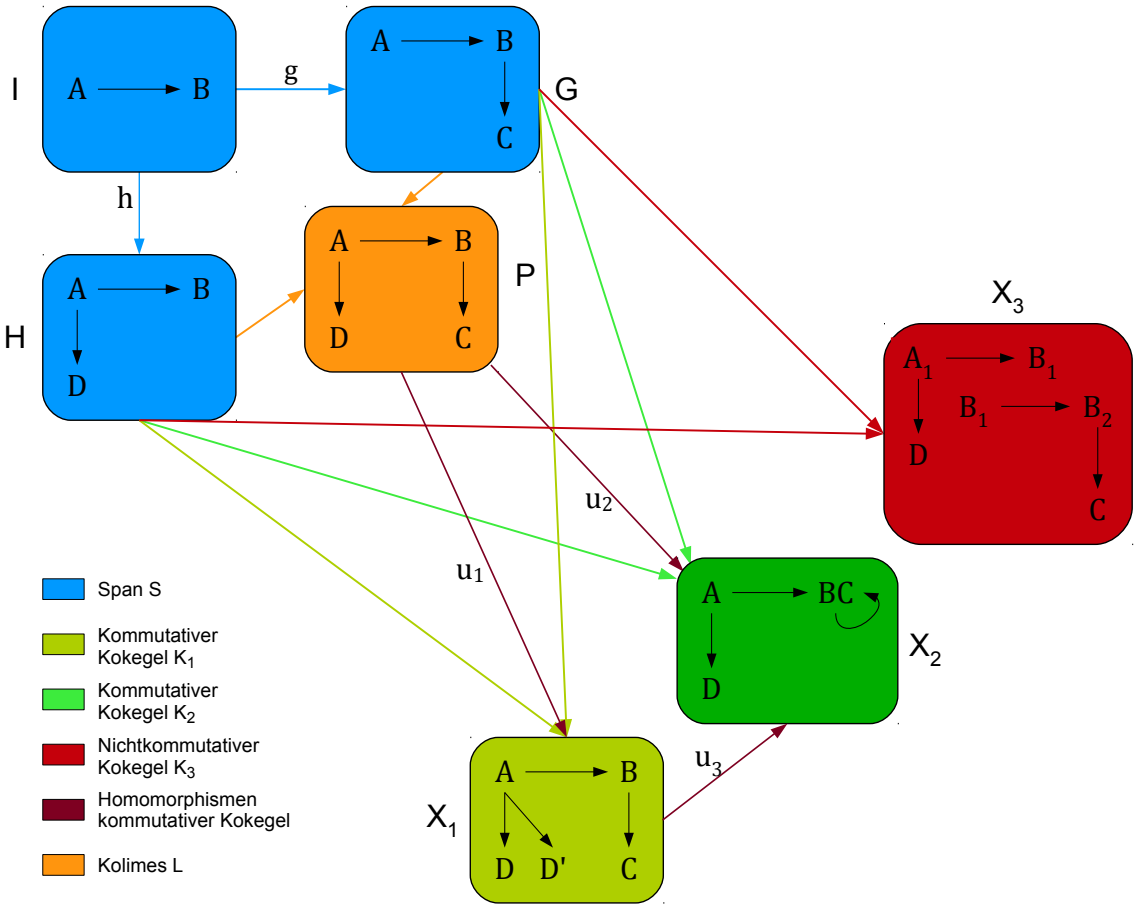
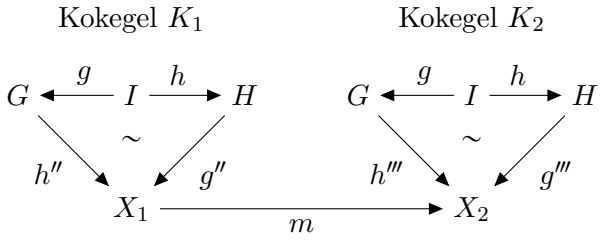
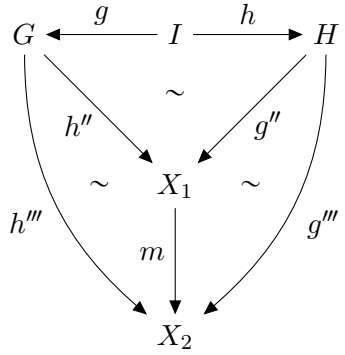


Abbildung 28: Nichtkommutative Kokegel, kommutative Kokegel und Homomorphismen kommutativer Kokegel



(K_1, K_2, m) ist genau dann ein Homomorphismus kommutativer Kokegel von K_1 nach K_2, geschrieben $m : K_1 \to K_2$, wenn m die Vorkommen von G und H in X_1 auf die Vorkommen von G bzw. H in X_2 abbildet. Formal wird das durch die Bedingungen $h'' \bullet m = h'''$ für die Vorkommen von G und $g'' \bullet m = g'''$ für die Vorkommen von H ausgedrückt, oder grafisch:

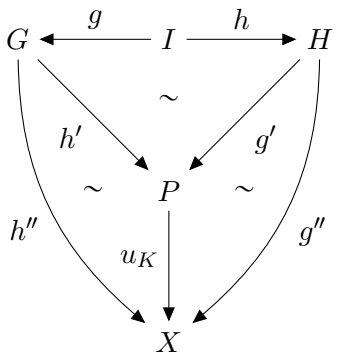


Ein Kolimes L zeichnet sich unter den kommutativen Kokegeln dadurch aus, dass er zu jedem weiteren Kokegel K genau einen Homomorphismus kommutativer Kokegel $u_K : L \to K$ hat. Diese Eigenschaft ist nämlich eine andere Formulierung der universellen Eigenschaft aus Unterabschnitt 3.3.2. Im Gegensatz dazu hat ein kommutativer Kokegel, der unnötige Knoten oder Pfeile hat, oder der Knoten oder Pfeile unnötigerweise miteinander verschmilzt, zu irgendeinem anderen kommutativen Kokegel entweder gar keinen oder gleich mehrere Homomorphismen kommutativer Kokegel.

Beispielsweise hat der Kolimes L in Abb. 28 zu den beiden anderen kommutativen Kokegeln K_1 und K_2 genau die abgebildeten Homomorphismen kommutativer Kokegel u_1 bzw. u_2. Auch zu jedem anderen kommutativen Kokegel hat er genau einen Homomorphismus, da sein Scheitel keine überflüssigen Knoten enthält wie $D' \in X_1^0$ und keine Knoten unnötigerweise verschmolzen werden wie $BC \in X_2^0$. Von K_1 nach L gibt es hingegen mehrere Homomorphismen kommutativer Kokegel, die in der Abbildung nicht illustriert sind. Denn $D' \in X_3^0$ kann sowohl auf $D \in P^0$ als auch auf $B \in P^0$ abgebildet werden, wodurch sich zwei unterschiedliche Homomorphismen ergeben. Von K_2 nach K_1 existiert überhaupt kein Homomorphismus kommutativer Kokegel, denn er müsste aufgrund der Definition der Homomorphismen kommutativer Kokegel den Knoten $BC \in X_2^0$ sowohl auf den Knoten $B \in X_1^0$ als auch auf den Knoten $C \in X_1^0$ abbilden, was aber nicht erlaubt ist. Aus demselben Grund existiert auch kein Homomorphismus kommutativer Kokegel von K_2 nach L.

Deshalb kann man definieren, dass ein Kolimes ein kommutativer Kokegel L mit einem Scheitel P ist, der zu jedem weiteren kommutativen Kokegel K mit einem Scheitel X genau einen Homomorphismus kommutativer Kokegel $u_K : L \to K$ hat. Für jeden kommutativen

Kokegel K mit Scheitel X gibt es also genau ein kommutatives Diagramm der folgenden Form:



Pushouts von Graphen lassen sich also durch die Existenz eines eindeutigen Homomorphismus kommutativer Kokegel zu jedem weiteren kommutativen Kokegel charakterisieren. Mithilfe dieser Charakterisierung kann man Pushouts auf beliebige Kategorien verallgemeinern. Darüber hinaus kann man Pushouts zu allgemeinen Kolimites verallgemeinern.

In Abb. 29 findet sich eine Gegenüberstellung der einzelnen Schritte hin zu einem Pushout bzw. allgemeinen Kolimes, ausgehend von den Eingabedaten. Der wesentliche Unterschied zwischen Pushouts und allgemeinen Kolimites besteht darin, dass ein allgemeiner Kolimes Diagramme beliebiger Form $\mathbf{G}$ als Eingabedaten hat, während für einen Pushout nur Spans, also Diagramme der Form $\bullet \leftarrow \bullet \rightarrow \bullet$ erlaubt sind. In einem ersten Schritt hin zum Pushout bzw. Kolimes definiert man die Kokegel über einem Span bzw. Diagramm. Hiervon werden im zweiten Schritt die kommutativen Kokegel ausgesondert. Anschließend werden Homomorphismen kommutativer Kokegel zwischen diesen betrachtet. Als Kolimites werden schließlich diejenigen kommutativen Kokegel definiert, die genau einen Homomorphismus kommutativer Kokegel zu jedem weiteren kommutativen Kokegel haben. Diese Schritte werden im folgenden formal definiert und genauer erläutert.

Für den Übergang von Spans zu beliebigen Diagrammen muss man die von Pushouts bekannten Kokegel zu allgemeinen Kokegeln generalisieren: Ein Kokegel für Pushouts besteht aus einem Span $S = (G \xleftarrow{g} I \xrightarrow{h} H)$, einem Scheitel X und zwei Morphismen $G \xrightarrow{h''} X$ und $H \xrightarrow{g''} X$. Ein allgemeiner Kokegel hat ebenso einen Scheitel $X \in \mathbf{C}^0$, aber anstatt eines Spans ein beliebiges Diagramm $D : \mathbf{G} \to \mathbf{C}$. Während sich bei Kokegeln für Pushouts als Pfeile von I nach X nur implizit $I \xrightarrow{g \bullet h''} X$ und $I \xrightarrow{h \bullet g''} X$ ergeben, ist bei allgemeinen Kokegeln für jeden Knoten von $\mathbf{G}$ ein Pfeil nach X explizit vorhanden (siehe Abb. 29):

Definition 41 (Kokegel) *Kokegel*

Sei $\mathbf{G}$ ein Köcher,
$\mathbf{C}$ ein Köcher (in der Regel eine Kategorie),
$D : \mathbf{G} \to \mathbf{C}$ ein Diagramm,
$X \in \mathbf{C}^0$ ein Objekt von $\mathbf{C}$ und
$i = (i_v)_{v \in \mathbf{G}^0}$ eine Familie von Pfeilen $D(v) \xrightarrow{i_v} X$ aus $\mathbf{C}^1$ für $v \in \mathbf{G}^0$.

Dann ist $W = (D, X, i)$ ein **Kokegel** mit Basis D und Scheitel X in $\mathbf{C}$.

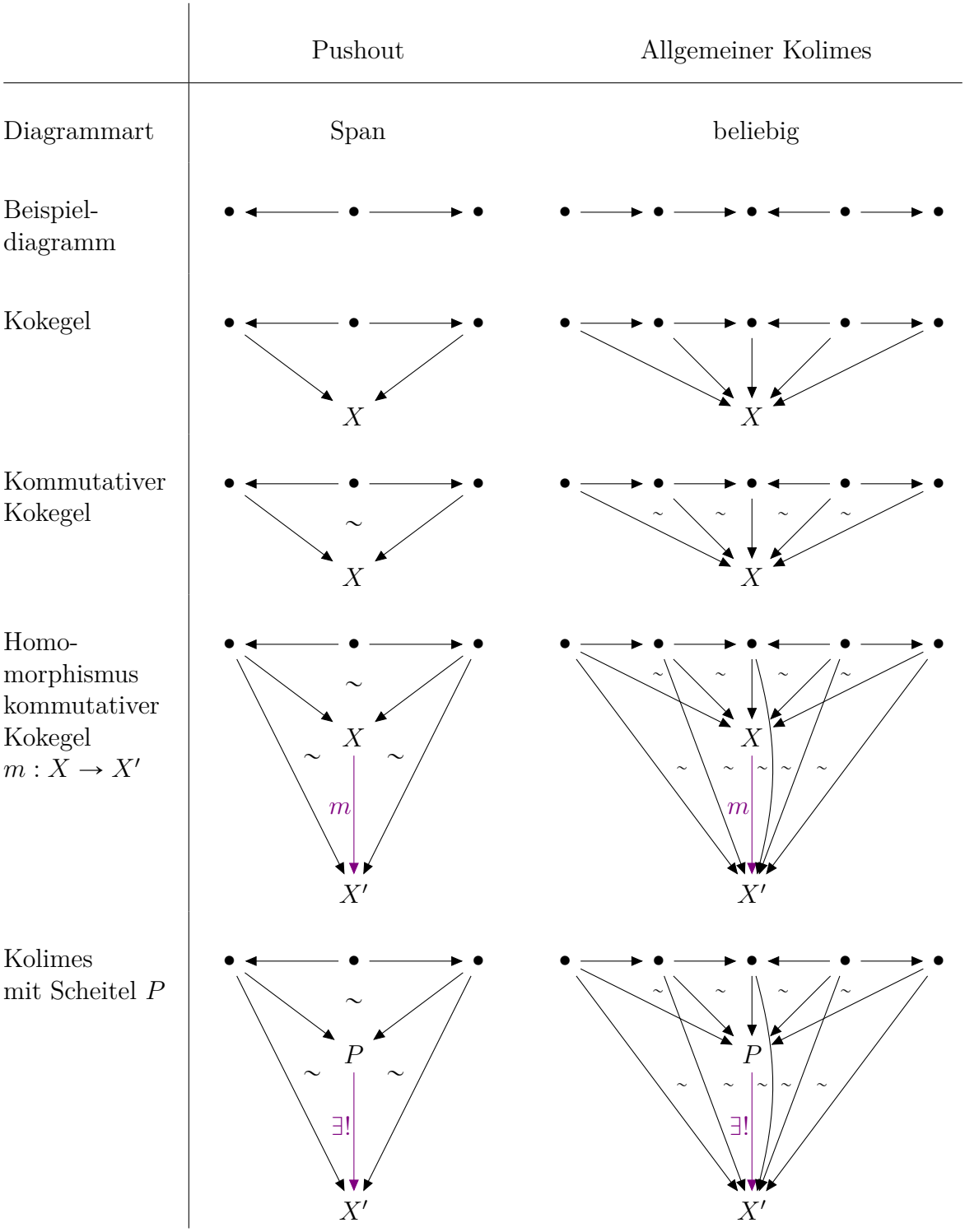
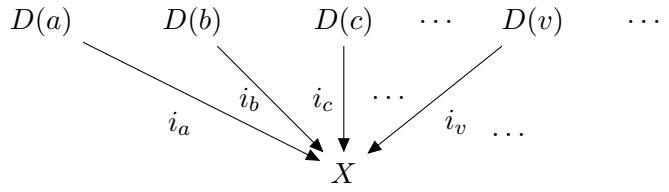


Abbildung 29: Vergleich Pushout – Allgemeiner Kolimes



Für $v \in \mathbf{G}^0$ heißt $i_v \in \mathbf{C}^1$ die v-**Komponente** oder auch v-Injektion von K.

Für Köcher $\mathbf{C}$ und $\mathbf{G}$ definiere die Klasse aller Kokegel der Form $\mathbf{G}$ in $\mathbf{C}$ folgendermaßen:
$\mathfrak{CoCone}(\mathbf{G}, \mathbf{C}) = \left\{ (D, X, i) \;\middle|\; D \in \mathbf{C}^{\mathbf{G}} \wedge X \in \mathbf{C}^0 \wedge i \in (\mathbf{C}^1)^{\mathbf{G}^0} \wedge \forall v \in \mathbf{G}^0 \colon i_v \in \mathbf{C}[D(v), X] \right\}$
□

In Fall von Graphen ($\mathbf{C} = \mathbf{Graphs}$) ist jede Komponente $i_v : D(v) \to X$ ein Graphhomomorphismus, d.h. jeder durch D ausgewählte Graph hat ein Vorkommen in X.

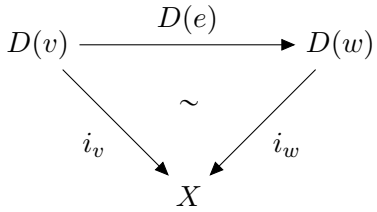
Wie bei Pushouts kommen für Kolimites nur kommutative Kokegel in Frage:

Definition 42 (Kommutativer Kokegel)

Kommutativer Kokegel

Sei $\mathbf{C}$ eine Kategorie.
Sei (D, X, i) ein Kokegel mit Basis $D : \mathbf{G} \to \mathbf{C}$ und Scheitel X in $\mathbf{C}$.

Dann ist (D, X, i) genau dann ein **kommutativer Kokegel** in $\mathbf{C}$, wenn für jeden Pfeil $v \xrightarrow{e} w$ in $\mathbf{G}^1$ gilt: $D(e) \bullet i_w = i_v$, d.h. wenn das Diagramm



für jeden solchen Pfeil kommutativ ist.
Für einen kommutativen Kokegel (D, X, i) schreibt man auch $i : D \to X$.

Zu beachten ist, dass das Diagramm D für sich *nicht* kommutativ sein muss. □

Auf Graphen angewendet heißt dies ähnlich wie bei Pushouts, dass i_w das durch den Graphhomomorphismus $D(e)$ repräsentierte Vorkommen des Graphen $D(v)$ in $D(w)$ auf das durch i_v repräsentierte Vorkommen von $D(v)$ in X abbilden muss.

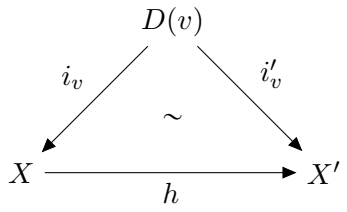
Um Kolimites kategorienunabhängig zu definieren, kann man ähnlich wie bei Pushouts Homomorphismen kommutativer Kokegel verwenden:

Definition 43 (Homomorphismus kommutativer Kokegel)

Homomorphismus kommutativer Kokegel

Sei $\mathbf{G}$ ein Köcher.
Sei $\mathbf{C}$ eine Kategorie.
Sei $D : \mathbf{G} \to \mathbf{C}$ ein Diagramm der Form $\mathbf{G}$ in $\mathbf{C}$.
Sei $i : D \to X$ und $i' : D \to X'$ Kokegel mit Basis D in $\mathbf{C}$.
Sei $X \xrightarrow{h} X' \in \mathbf{C}^1$ ein Pfeil von X nach X' in $\mathbf{C}$

Dann ist (i, i', h) genau dann ein **Homomorphismus kommutativer Kokegel** von i nach i', wenn für jeden Knoten $v \in \mathbf{G}^0$ gilt: $i_v = i'_v \bullet f$, d.h. wenn



kommutativ ist.

Für einen Homomorphismus (i, i', h) kommutativer Kokegel schreibt man $h : i \to i'$. □

Im Fall von Graphen bedeutet dies genau dasselbe wie bei Pushouts: h muss das Vorkommen von $D(v)$ in X auf das Vorkommen von $D(v)$ in X' abbilden.

Ein Kolimes ist wie bei Pushouts ein kommutativer Kokegel, der zu jedem weiteren kommutativen Kokegel genau einen Homomorphismus kommutativer Kokegel u besitzt:

Definition 44 (Kolimes)

Kolimes

Sei **C** eine Kategorie.
Sei $i : D \to X$ ein kommutativer Kokegel mit Basis D in **C**.

$i : D \to X$ ist ein **Kolimes** (oder: **universeller** kommutativer Kokegel) von D genau dann, wenn es für jeden kommutativen Kokegel $i' : X \to D$ genau einen Homomorphismus kommutativer Kokegel $u : i \to i'$ gibt. □

Mit diesem Kolimes-Begriff lässt sich der Pushout auf folgende Weise allgemein erfassen:

Beispiel 45 (Pushout)

Pushout

Bereits weiter oben wurde festgestellt, dass Spans Diagrammen mit dem folgenden Formgraphen PO entsprechen:

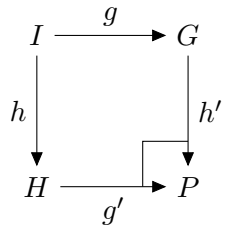
$$1 \xleftarrow{\ l\ } 0 \xrightarrow{\ r\ } 2$$

Deshalb wird ein Span in einer Kategorie **C** hier auch als **Pushoutdiagramm** in **C** bezeichnet, definiert als ein Diagramm $S : \mathrm{PO} \to \mathbf{C}$ der Form PO.
Ein Kolimes eines Pushoutdiagramm $S : \mathrm{PO} \to \mathbf{C}$ heißt **Pushout** über S.

Angewendet auf die Kategorie **Graphs** der Graphen und Graphhomomorphismen ergibt sich tatsächlich der Pushout wie er am Anfang des Abschnittes beschrieben wurde. Die beiden zu vereinigenden Graphen werden durch das Diagramm D mit $D(1)$ bzw. $D(2)$ und der Schnittstellengraph mit $D(0)$ ausgewählt. Der zugehörige Kolimes $l : D \to X$ liefert als Ergebnis den Graphen X und die Injektionen $D(1) \xrightarrow{i_1} X$ und $D(2) \xrightarrow{i_2} X$. Die dritte Injektion $D(0) \xrightarrow{i_0} X$ des Schnittstellengraphen ergibt sich immer aus den anderen beiden als $i_0 = D(l)\bullet i_1 = D(r)\bullet i_2$, weshalb sie am des Abschnittes nicht als Teil des Pushouts erwähnt wurde.

Dies lässt sich für einen beliebigen Kolimes $i : D \to X$ verallgemeinern auf jede Komponente $D(a) \xrightarrow{i_a} X$ eines Knotens a, der Quelle mindestens eines Pfeiles $m : a \to b$ des Formgraphen ist. Durch einen solchen Diagrammpfeil und die b-Komponente $D(b) \xrightarrow{i_b} X$ ist die a-Komponente eindeutig festgelegt als $i_a = D(m)\bullet i_b$. Es besteht also keine Wahlmöglichkeit mehr für i_a, sobald i_b festgelegt ist.

In Illustrationen von Diagrammen haben Pushouts eine besondere Notation. Sie werden als Quadrate gezeichnet und durch einen rechten Winkel in der Ecke des Pushoutobjektes markiert:



□

Bemerkung 46 (Eindeutigkeit und Existenz von Kolimites)

Eindeutigkeit und Existenz von Kolimites

Im Gegensatz zu klassischen mathematischen Konstruktionen ist das Ergebnis kategorieller Konstruktionen wie des Pushouts oder allgemeiner Kolimites nicht eindeutig. Über einem Diagramm D kann es mehrere Kolimites $i : D \to X_1,\ j : D \to X_2,\ k : D \to X_3$ usw. geben. Allerdings sind diese Kolimites im Wesentlichen gleich; bei Graphen könnte man sagen, dass lediglich die Knoten und Pfeile umbenannt werden, und zwar durch die eindeutigen Homomorphismen u zwischen den Kolimites. Deshalb wird oft trotz Mehrdeutigkeit von *dem* Kolimes oder *dem* Pushout über einem Diagramm D gesprochen.

Außerdem existiert nicht in jeder Kategorie jeder Kolimes über jedem Diagramm. Auch existiert nicht in jeder Kategorie ein Pushout über jedem Span. In **Graphs** und in **Set** ist beides allerdings sehr wohl der Fall. □

Dualität bei Graphhomomorphismen

Wie bei Definition 20 erklärt wurde, erhält man zu jeder kategoriellen Konstruktion eine duale Konstruktion, indem man in ihrer Definition die Richtung der Pfeile umdreht. Tatsächlich betrachtet man dadurch nur dieselben Pfeile aus einer anderen Perspektive. Während in der Kategorie **Graphs** ein Graphhomomorphismus $f : G \to H$ das Verschmelzen von Knoten und Pfeilen aus G und das Hinzufügen von Knoten und Pfeilen zu G repräsentiert, repräsentiert er in der dualen Kategorie $\mathbf{Graphs}^{\mathbf{op}}$ das Aufspalten (Vervielfältigen) von Knoten und Pfeilen von H und das Entfernen von Knoten und Pfeilen aus H. In dieser unterschiedlichen Betrachtungsweise besteht der wesentliche Unterschied zwischen Kolimites und den dazu dualen Limites. Beispielsweise ist der Pullback der zum Pushout duale Limes. Das bedeutet, dass ein Pullback in der Kategorie **Graphs** im Wesentlichen dasselbe ist wie ein Pushout der dualen Kategorie $\mathbf{Graphs}^{\mathbf{op}}$.

Im Folgenden werden zunächst Pullbacks von Graphen anhand des Beispiels aus Abb. 30 illustriert. Darauf aufbauend werden dann allgemeine Limites definiert.

Ein Pullback von Graphen ist die Aufspaltung eines Graphen anhand zweier Graphen, die in ihm enthalten sind. Als Eingabedaten hat ein Pullback einen sogenannten Kospan – ein Diagramm der Art $S = (G \xrightarrow{g} I \xleftarrow{h} H)$, also einen Span, in dem die Pfeile umgedreht sind. Das Ergebnis ist ein Limes oder Pullback über S, der aus einem Pullbackgraphen P und Graphhomomorphismen $G \xrightarrow{g'} I$ und $H \xrightarrow{h'} I$ besteht. Diese Graphhomomorphismen repräsentieren das Entfernen oder Aufspalten von Knoten und Pfeilen aus I: Knoten und Pfeile werden in ihre Urbilder unter g bzw. h aufgespalten, oder entfernt falls sie kein Urbild unter g bzw. h haben. In P sind alle Knoten und Pfeile, die in G oder H entfernt

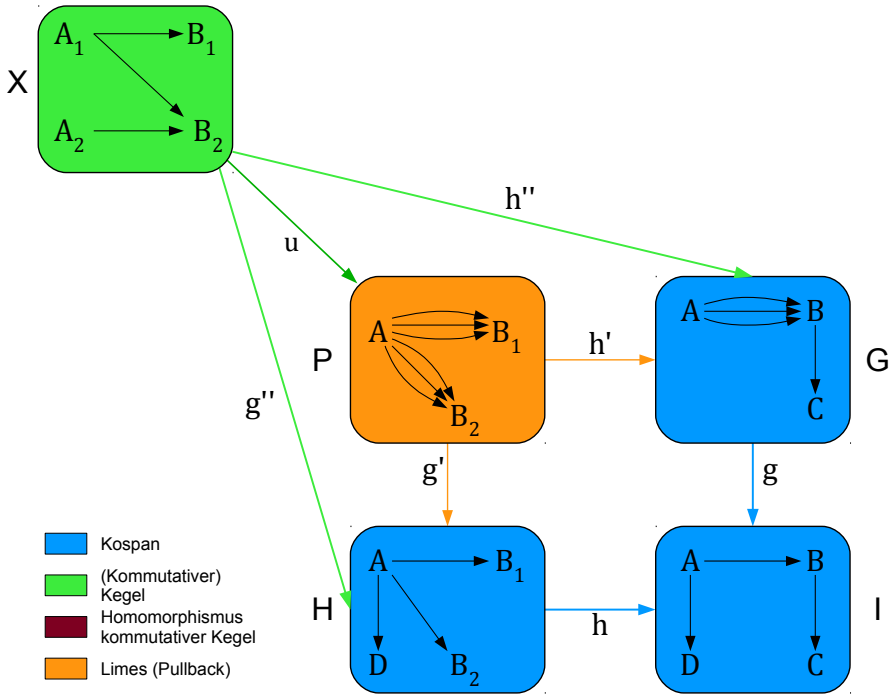


Abbildung 30: Pullback von Graphen

wurden, auch entfernt. Knoten und Pfeile, die in G ver-n-facht und in H ver-m-facht werden, sind in P dann $n \cdot m$-fach vorhanden, haben also für jede Kombination von Urbildern in G mit Urbildern in H genau eine Aufspaltung.

In Abb. 30 entfernt h den Knoten C und den Pfeil von B nach C, und es spaltet den Knoten B in zwei Knoten B_1 und B_2 und den unbenannten Pfeil zwischen A und B in zwei Pfeile auf. H hingegen entfernt den Knoten D und den Pfeil von A nach D, und verdreifacht den unbenannten Pfeil von A nach B, ohne den Knoten B aufzuspalten. Der Pullback kombiniert diese beiden Änderungen, so dass P sie beide enthält: Sowohl C als auch D und die darauf zeigenden Pfeile sind entfernt, und die Aufspaltungen des Pfeiles von A nach B aus I werden kombiniert, so dass sie in P insgesamt sechsfach vorhanden ist.

Genauer erkennt man die Wirkungsweise der Änderungen anhand der Graphhomomorphismen g' und h': Ebenso wie g aus I den Knoten D und den Pfeil von A nach D entfernt und den Pfeil von A nach B verdreifacht, entfernt und vervielfältigt auch g' aus H die Entsprechungen dieser Knoten und Pfeile. Und ebenso wie h entfernt h' die Entsprechung des Knotens D und des Pfeiles von A nach D und verdoppelt die Entsprechungen des Knotens B und des Pfeiles von A nach B.

Diese Betrachtungsweise von Pullbacks in **Graphs** ergibt sich daraus, dass man sie als Pushouts in der dualen Kategorie $\mathbf{Graphs}^{\mathbf{op}}$ betrachten kann. Man kann sie aber auch ohne Zuhilfenahme der Dualität unmittelbar erklären. Wie in Beispiel 52 genauer erläutert wird, erfüllen in den Kategorien **Set** und **Graphs** eingeschränkte kartesische Produkte zusammen mit den kanonischen Projektionsfunktionen bzw. -homomorphismen die Eigenschaften eines Pullbacks. Deshalb bezeichnet man g' und h' auch als Projektionen.

Ähnlich wie Kokegel, kommutative Kokegel und Homomorphismen kommutativer Koke-

gel bei Pushouts und Kolimites die Zwischenschritte zur Definition sind, sind es bei Pullbacks und Limites Kegel, kommutative Kegel und Homomorphismen kommutativer Kegel. Da die dazu dualen Begriffe bereits bekannt sind, werden sie im Folgenden unmittelbar für allgemeine Limites definiert und Pullbacks werden als Spezialfall des Limes behandelt.

Beispiel 47 (Pullback-Diagramm)

Pullback-Diagramm

Wie bereits bekannt ist, sind die Eingabedaten eines Pullback durch einen Kospan gegeben, den man als Diagramm mit dem folgenden Formgraphen PB auffassen kann:

$$1 \xrightarrow{l} 0 \xleftarrow{r} 2$$

Ein Diagramm $D : PB \to \mathbf{C}$ der Form PB in einer Kategorie $\mathbf{C}$ heißt **Pullback-Diagramm** (oder Kospan) in $\mathbf{C}$. □

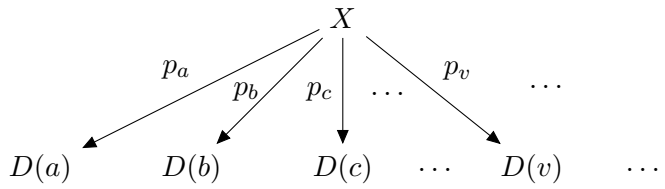
Kegel sind das zu Kokegeln duale Konzept:

Definition 48 (Kegel)

Kegel

Sei $\mathbf{G}$ ein Köcher.
Sei $\mathbf{C}$ eine Kategorie (oder ein Köcher).
Sei $X \in \mathbf{C}^0$ (Scheitel).
Sei $p = (p_v)_{v \in \mathbf{G}^0}$ eine Familie von Pfeilen $X \xrightarrow{p_v} D(v)$.
Sei $D : \mathbf{G} \to \mathbf{C}$ ein Diagramm der Form $\mathbf{G}$ in $\mathbf{C}$.

Dann ist $V = (X, D, p)$ ein Kegel mit Basis D in $\mathbf{C}$.



Für $v \in \mathbf{G}^0$ heißt p_v die v-**Komponente** oder auch v-Projektion von K.

Für Köcher $\mathbf{C}$ und $\mathbf{G}$ definiere die Klasse aller Kegel der Form $\mathbf{G}$ in $\mathbf{C}$ folgendermaßen:
$\mathfrak{Cone}(\mathbf{G}, \mathbf{C}) = \left\{ (X, D, p \;\middle|\; D \in \mathbf{C}^{\mathbf{G}} \wedge X \in \mathbf{C}^0 \wedge p \in (\mathbf{C}^1)^{\mathbf{G}^0} \wedge \forall v \in \mathbf{G}^0 \colon p_v \in \mathbf{C}[X, D(v)] \right\}$
□

Bei Graphen repräsentiert (in der dualen Sichtweise) jede Komponente p_v des Kegels Aufspaltungen und Entfernungen von Knoten und Pfeilen aus dem Graphen $D(v)$.

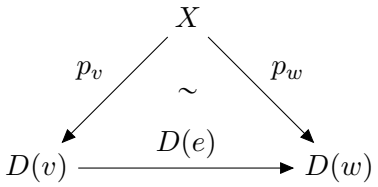
Allerdings können bei Kegeln diese Aufspaltungen und Entfernungen in „undisziplinierter“ Weise ablaufen. Ähnlich wie bei Kokegeln von Pushouts der Schnittstellengraph I mehrfach im Scheitel vorkommen können, berücksichtigen auch Kegel den Schnittstellengraph I von Pullbacks nicht immer. Abhilfe schaffen kommutative Kegel:

Definition 49 (Kommutativer Kegel)

Kommutativer Kegel

Sei $\mathbf{C}$ eine Kategorie.
Sei (X, D, p) ein Kegel mit Basis D in $\mathbf{C}$, wobei $src(D) = \mathbf{G}$.

(X, D, p) ist genau dann ein **kommutativer Kegel** in $\mathbf{C}$ wenn für jeden Pfeil $v \xrightarrow{e} w$ in $\mathbf{G}$ gilt: $p_v \bullet D(e) = p_w$, d.h. wenn



für jeden Pfeil e kommutativ ist.

Für einen kommutativen Kegel (X, D, p) schreibt man $p : X \to D$. □

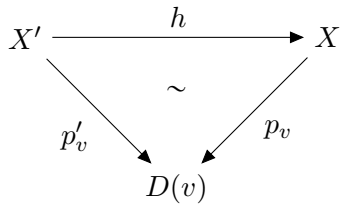
Kommutative Kegel von Graphen können immer noch zu viele Knoten und Pfeile entfernen, oder sie zu sehr oder zu wenig aufspalten. Ein Limes entfernt und spaltet gerade diese Knoten und Pfeile auf, die er aufgrund des Diagrammes entfernen bzw. aufspalten muss. Da eine solche Definition nicht auf beliebige Kategorien ausgedehnt werden kann, stützt man sich auch hier auf einen passenden Homomorphismusbegriff:

Definition 50 (Homomorphismus kommutativer Kegel)

Homomorphismus kommutativer Kegel

Sei $\mathbf{G}$ ein Köcher.
Sei $\mathbf{C}$ eine Kategorie.
Sei $D : \mathbf{G} \to \mathbf{C}$ ein Diagramm der Form $\mathbf{G}$ in $\mathbf{C}$.
Sei $p : X \to D$ und $p' : X' \to D$ Kegel mit Basis D in $\mathbf{C}$.
Sei $X' \xrightarrow{h} X \in \mathbf{C}^1$ ein Pfeil von X' nach X in $\mathbf{C}$

Dann ist f genau dann ein **Homomorphismus kommutativer Kegel** von p' nach p, geschrieben $h : p' \to p$, wenn für jeden Knoten $v \in \mathbf{G}^0$ gilt: $p'_v = f \bullet p_v$, d.h. wenn



kommutativ ist. □

Ein Homomorphismus kommutativer Kegel ist also ein Homomorphismus zwischen den Scheiteln der Kegel, der mit den Komponenten der Kegel verträglich ist. Bei Graphen heißt diese Definition, dass h im Vergleich zu p_v zusätzliche Knoten und Pfeile von $D(v)$ aufspaltet oder löscht, und am Ende dasselbe Ergebnis erhält wie p'.

Mittels Eindeutigkeit von Homomorphismen kommutativer Kegel kann man nun Limites kategorienunabhängig definieren:

Definition 51 (Limes)

Limes

Sei $\mathbf{C}$ eine Kategorie.
Sei $p : X \to D$ ein kommutativer Kegel mit Basis D in $\mathbf{C}$.

p ist ein **Limes** (oder: **universeller** kommutativer Kegel) genau dann, wenn es für jeden kommutativen Kegel $p' : X \to D$ genau einen Homomorphismus kommutativer Kegel $u : p' \to p$ gibt. □

Im Sinne der obigen Interpretation von Homomorphismen kommutativer Kegel hat ein Limes also die Eigenschaft, dass man aus ihm jeden anderen kommutativen Kegel auf eindeutige Weise erhält, indem man zusätzliche Knoten und Pfeile entfernt und aufspaltet.

Ein Limes eines Pullback-Diagrammes $D : PB \to \mathbf{C}$ in einer Kategorie $\mathbf{C}$ heißt **Pullback** von $D(1)$ und $D(2)$ mit Schnittstellenobjekt $D(0)$ in $\mathbf{C}$.

Beispiel 52 (Pullbacks in Set und Graphs)

Pullbacks in **Set** *und* **Graphs**

Für die Zwecke dieser Arbeit sind Pullbacks in **Set** und in **Graphs** besonders nützlich. In **Set** ist das eingeschränkte kartesische Produkt $A \times_{f=g} B := \{(a,b) \in A \times B \mid f(a) = g(b)\}$ der Scheitel eines Pullbacks über dem Kospan $S = (A \xrightarrow{f} I \xleftarrow{g} B)$. Die Komponenten dieses Pullbacks sind die Projektionsfunktionen $g' : A \times_{f=g} B \to A : (a,b) \mapsto a$ und $h' : A \times_{f=g} B \to B : (a,b) \mapsto b$.

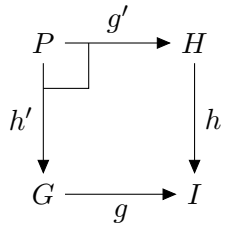
Der Pullback hat in **Set** also eine ähnliche Wirkungsweise wie ein Equi-Join $A \bowtie_{f=g} B$ in relationalen Datenbanken. Im Bereich der Kategorientheorie wird aber für den Pullback die oben benutzte Notation $A \times_{f=g} B$ oder unsauberer $A \times_I X$ verwendet.

In **Graphs** lässt sich der Pullback über einem Kospan $S = (G \xrightarrow{g} I \xleftarrow{h} H)$ konstruieren, indem man die obige Konstruktion sowohl auf die Menge der Knoten als auch auf die Menge der Pfeile anwendet. Der Scheitelgraph $G \times_{g=h} H$ des Pullback kann folgendermaßen konstruiert werden:

- $(G \times_{g=h} H)^0 = \left\{(v,w) \in G^0 \times H^0 \;\middle|\; g(v) = h(w)\right\}$
- $(G \times_{g=h} H)^1 = \left\{(a,b) \in G^1 \times H^1 \;\middle|\; g(a) = h(b)\right\}.$

Zusammen mit zwei Projektionshomomorphismen, die den Paaren von Knoten und Pfeilen jeweils ihre linke bzw. rechte Komponente zuordnen, erfüllt $G \times_{g=h} H$ die Eigenschaften eines Pullbacks über S.

Ebenso wie bei Pushouts gibt es für einen Pullback mit Scheitel P und Komponenten h' und g' eine spezielle Notation in Illustrationen, wobei die Komponente $P \xrightarrow{p_0} I$ wiederum weggelassen wird, da sie sich als $p_0 = h' \bullet g = g' \bullet h$ ergibt:



□

Beispiel 53 (Beliebige Limites in Set)

Beliebige Limites in **Set**

Ähnlich wie sich Pullbacks in **Set** als eingeschränkte kartesische Produkte konstruieren lassen, kann man Limites über kleinen Diagrammen als eine eingeschränkte Familie konstruieren. Für ein Diagramm $D : \mathbf{G} \to \mathbf{Set}$ mit Formgraphen $\mathbf{G}$ ist die folgende Menge X der Scheitel eines Limes von D:

$$X = \left\{(x_v)_{v \in \mathbf{G}^0} \;\middle|\; (\forall v \in \mathbf{G}^0 \colon x_v \in D(v)) \wedge (\forall v \xrightarrow{e} w \in \mathbf{G}^1 \colon (D(e))(x_v) = x_w)\right\}$$

Zusammen mit den Projektionshomomorphismen $p_w : X \to D(w) : (x_v)_{v \in \mathbf{G}^0} \mapsto x_w$ für jeden Knoten $w \in \mathbf{G}^0$ bildet X einen Limes über D in **Set**. □

Beispiele von Limites und Kolimites

Im Folgenden werden einige weitere Beispiele von Limites und Kolimites vorgestellt, die im Laufe der Arbeit benötigt werden.

Beispiel 54 (Initiales Objekt)

Initiales Objekt

Die wohl einfachsten Kolimites sind die initialen Objekte. Sei eine Kategorie $\mathbf{C}$ und der leere Graph $\mathbf{0}$ gegeben.

Da der Graph $\mathbf{0}$ leer ist, gibt es nur ein einziges Diagramm $IO : \mathbf{0} \to \mathbf{C}$ der Form $\mathbf{0}$ in $\mathbf{C}$. Ein Kolimes $i : IO \to O$ dieses Diagrammes IO heißt **initiales Objekt** von $\mathbf{C}$.

Der Grund für die Bezeichnung des ganzen Kolimes als initiales *Objekt* liegt darin, dass einzige relevante Information eines Kokegels $i : IO \to X$ das Scheitelobjekt X ist – er hat keine Komponenten und auch das Diagramm O enthält keine zusätzliche Information. Deshalb gibt es für jedes Objekt X von $\mathbf{C}$ genau einen kommutativen Kokegel $i_X : IO \to X$ und umgekehrt. Daher kann man die (kommutativen) Kokegel über O mit den Objekten von $\mathbf{C}$ gleichsetzen, wodurch sich die Bezeichnung „initiales Objekt“ rechtfertigen lässt.

Da ein initiales Objekt $i : IO \to O$ als Kolimes genau einen Pfeil zu jedem kommutativen Kokegel hat und man es mit seinem Scheitelobjekt O gleichsetzen kann, entspricht es einem Objekt, das zu jedem Objekt der Kategorie genau einen Pfeil hat. Durch diese Eigenschaft spielen initiale Objekte eine wichtige Rolle in den Kategorien, in denen sie existieren. Der eindeutige Pfeil von O zu einem Objekt A wird mit $O \xrightarrow{\lozenge} A$ bezeichnet. In **Set** nimmt die leere Menge $\varnothing$ die Rolle eines initialen Objektes ein, was bei der Modellierung von Containment ausgenutzt wird. □

Beispiel 55 (Terminale Objekte und globale Elemente)

Terminale Objekte und globale Elemente

Der zum initialen Objekt duale Limes ist das terminale Objekt. Da der leere Graph $\mathbf{0}$ zu sich selbst dual ist, handelt es sich hierbei ebenfalls um Limites über leeren Diagrammen, welche man wiederum mit Objekten von $\mathbf{C}$ gleichsetzen kann. Während es sich bei initialen Objekten um Objekte handelt, die zu jedem weiteren Objekt genau einen *ausgehenden* Pfeil haben, haben terminale Objekte von jedem weiteren Objekt genau einen *eingehenden* Pfeil.

In **Set** ist jede einelementige Menge $\{a\}$ ein terminales Objekt, und der von einer weiteren Menge X ausgehende eindeutige Pfeil ist die konstante Funktion $\Delta_a : X \to \{A\} : x \mapsto a$. Da alle terminalen Objekte zueinander isomorph sind, wählt man üblicherweise eine beliebige einelementige Menge $\{*\}$ aus (z.B. mit $* = \varnothing$), die man – ebenso wie terminalen Objekte – mit 1 bezeichnet. Diese Schreibweise ist ein Hinweis darauf, dass man ein terminales Objekt auch als ein nullstelliges kategorielles Produkt betrachten kann.

Eine wichtige Anwendung von terminalen Objekten sind die sogenannten globalen Elemente eines Objekts Y. Dabei handelt es sich um die Pfeile, die von einem terminalen Objekt nach Y, d.h. die Pfeile der Form $1 \xrightarrow{f} Y$. In **Set** entspricht ein globales Element $f : \{*\} \to Y$ einer einelementigen Teilmenge von Y und somit einem Element von Y, und zwar $f(*)$. □

Beispiel 56 (Koprodukte und Kotupel)

Koprodukte und Kotupel

Sei COPR der Graph mit zwei Knoten 1 und 2 und ohne Pfeile:

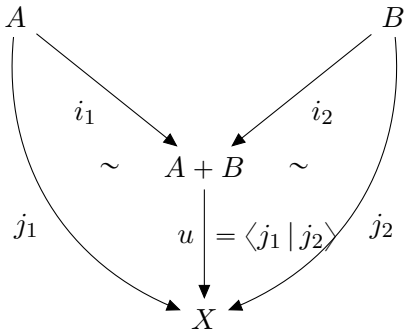
1 2

Ein binäres **Koproduktdiagramm** ist ein Diagramm $D : \mathrm{COPR} \to \mathbf{C}$ der Form COPR in einer Kategorie $\mathbf{C}$ und wird hier als $D = (D(1) \quad D(2))$ notiert.
Ein binäres **Koprodukt** ist ein Kolimes über einem solchen Diagramm D. Um ein Koprodukt zweier Objekte A und B in einer Illustration zu markieren, wird der Scheitel üblicherweise als $A + B$ bezeichnet.

Da binäre Koproduktdiagramme keine Pfeile enthalten, sind Kokegel über Koproduktdiagrammen immer trivialerweise kommutativ.

Für die Zwecke dieser Arbeit ist vor allem das Koprodukt von Mengen interessant, denn es ist zur Modellierung von Graphvarianten mit Sketches nützlich. In **Set** gibt es zwei konkrete Konstruktionen für das Koprodukt: Für beliebige Mengen A und B bildet die disjunkte Vereinigung $A + B = (\{1\} \times A) \cup (\{2\} \times B)$ von A und B zusammen mit den Injektionen $i_1 : A \to A + B : a \mapsto (1, a)$ und $i_2 : B \to A + B : b \mapsto (2, b)$ ein Koprodukt von A und B. Daneben ist für den Fall, dass A und B disjunkt sind, auch die normale Mengenvereinigung $A \cup B$ zusammen mit den Inklusionen $\imath_1 : A \to A \cup B : a \mapsto a$ und $\imath_2 : B \to A \cup B : b \mapsto b$ ein Koprodukt von A und B.

Beim Koprodukt sind die eindeutigen Homomorphismen in andere kommutative Kokegel von besonderer Bedeutung. Deshalb soll hier ein solcher Homomorphismus $u : i \to j$ bzw. $u : A + B \to X$ für einen anderen kommutativen Kokegel $j : D \to X$ betrachtet werden. Ein kommutativer Kokegel mit einer Scheitelmenge X ist durch beliebige Funktionen $j_1 : A \to X$ und $j_2 : B \to X$ gegeben:



Der eindeutige Homomorphismus u von i nach j wird als Kotupel $\langle j_1 \mid j_2 \rangle$ von j_1 und j_2 bezeichnet. In **Set** nimmt $\langle j_1 \mid j_2 \rangle$ eine Fallunterscheidung zwischen A und B vor. Wenn man die disjunkte Vereinigung als Koprodukt verwendet, gilt nämlich:

$$u = \langle j_1 \mid j_2 \rangle : A + B \to X : x \mapsto \begin{cases} j_1(a) & \text{falls } x = (1, a) \text{ mit } a \in A \\ j_2(b) & \text{falls } x = (2, b) \text{ mit } b \in B \end{cases}$$

Auch Koprodukte von Graphen und von kleinen Diagrammen (d.h. Köcherhomomorphismen mit kleinen Formgraphen) können auf ähnliche Weise konstruiert werden und die eindeutigen Homomorphismen u kann man ebenfalls als Kotupel von Graphhomomorphismen bezeichnen. Bei kleinen Diagrammen $D_1 : G_1 \to \mathbf{C}$ und $D_2 : G_2 \to \mathbf{C}$ mit Formgraphen $G_1, G_2 \in \mathfrak{Graphs}$ kann man durch Bildung des Kotupels $\langle D_1 \mid D_2 \rangle : G_1 + G_2 \to \mathbf{C}$ die Diagramme vereinigen. Das Kotupel $\langle D_1 \mid D_2 \rangle$ ist nämlich selbst ein Diagramm und kommutiert genau dann, wenn sowohl D_1 als auch D_2 kommutieren.

Man kann auch Koprodukte und Kotupel einer beliebigen Menge von Objekten bzw. Pfeilen definieren. Formal handelt es sich dabei um Koprodukte über einem Diagramm ohne Pfeile, d.h. einem Diagramm $D : \mathbf{G} \to \mathbf{C}$ mit $\mathbf{G}^1 = \varnothing$. Ein solches Diagramm heißt **diskretes** Diagramm und Kokegel darüber nennt man einen **diskreten** Kokegel. Ein diskretes Diagramm ist im Wesentlichen dasselbe wie die Klasse der Objekte, die es auswählt. Ein diskreter Kokegel ist immer kommutativ und ist im Wesentlichen dasselbe wie ein Objekt X zusammen mit einer Klasse von Pfeilen nach X. Ein Kolimes über einem diskreten Diagramm heißt Koprodukt.

Der Scheitel eines Koproduktes über einem diskreten Diagramm $D : \mathbf{I} \to \mathbf{C}$ mit $I = \mathbf{I}^0$ wird geschrieben als:

$$\coprod_{A \in I} D(A)$$

Das Kotupel eines diskreten Kokegels $j : D \to X$ über D mit Scheitel X wird geschrieben als:

$$\langle j_A \rangle_{A \in I} : \coprod_{A \in I} D(A) \to X$$

□

Falls $I = \varnothing$, dann ist das Koprodukt das initiale Objekt und das (eindeutige) Kotupel ist der eindeutige Morphismus (siehe Beispiel 54).

Beispiel 57 (Beliebige Kolimites in Set)

Beliebige Kolimites in **Set**

Auch Kolimites über beliebigen Diagrammen lassen sich in **Set** auf allgemeine Weise konstruieren. Die Konstruktion ist aber komplexer als die für Limites.

Für ein Diagramm $D : \mathbf{G} \to \mathbf{Set}$ mit Formgraphen $\mathbf{G}$ ignoriert man zunächst die Pfeile und konstruiert zunächst das Koprodukt. In **Set** ist dies die disjunkte Vereinigung $U \coloneqq \biguplus_{v \in \mathbf{G}^0} \{v\} \times D(v) = \big\{(v, a) \bigm| v \in \mathbf{G}^0 \wedge a \in D(v)\big\}$ der durch D ausgewählten Objekte.

Um den Kolimes von D zu erhalten, müssen aber eventuell noch Elemente von U aufgrund der Pfeile des Diagrammes miteinander verschmolzen werden, ähnlich wie bei Pushouts. Zwei Elemente von U müssen miteinander verschmolzen werden, wenn eine der Funktionen im Diagramm das eine Element auf das andere abbildet. Hierzu definiert man zunächst eine Relation $R : U \nrightarrow U$ auf der disjunkten Vereinigung, die diese Paare von Elementen in Beziehung setzt, also mit $(v_1, a_1)\, R\, (v_2, a_2)$ gdw. $\exists v_1 \xrightarrow{f} v_2 \in \mathbf{G}^1$: $(D(f))(a_1) = a_2$.

Da sich hierbei Ketten von Elementen bilden können, die miteinander verschmolzen werden müssen, wird in einem dritten Schritt die Äquivalenzhülle $S \coloneqq R^{\equiv}$ gebildet. In dieser stehen sämtliche Elemente unmittelbar miteinander in Beziehung, die in R nur mittelbar miteinander verbunden sind (oder graphentheoretisch ausgedrückt: in derselben schwachen Zusammenhangskomponente von R sind).

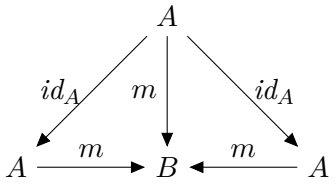
Die Menge U/S der Äquivalenzklassen von S bildet nun einen Kolimes über D. Die Injektionen sind dabei die Funktionen $i_v : D(v) \to U/S : x \mapsto [(v, x)]_S$ für $v \in \mathbf{G}^0$, die jedem Element seine Äquivalenzklasse zuordnen. □

Beispiel 58 (Monomorphismen mittels Pullbacks)

Monomorphismen mittels Pullbacks

Ein Pfeil $A \xrightarrow{m} B \in \mathbf{C}^1$ einer Kategorie $\mathbf{C}$ ist genau dann ein Monomorphismus (oder: monomorph), wenn für jedes Paar $C \underset{g}{\overset{f}{\rightrightarrows}} A$ paralleler Pfeile nach A, deren Komposition mit m dasselbe Ergebnis $f \bullet m = g \bullet m$ ergibt, die Pfeile $f = g$ identisch sein müssen. Beispielsweise sind die injektiven Funktionen die Monomorphismen von **Set**, und die injektiven Graphhomomorphismen die Monomorphismen von **Graphs**.

Man kann Monomorphismen auch mittels eines Pullbacks beschreiben. Ein Morphismus $A \xrightarrow{m} B \in \mathbf{C}^1$ ist genau dann ein Monomorphismus, wenn der folgende Kegel ein Limes ist: [BW12, S. 268]



□

Beispiel 59 (Equalizer)

Equalizer

Equalizer (auch: Egalisatoren) modellieren im Falle der Kategorie **Set** die Teilmenge einer Menge A, auf der zwei von A ausgehende Funktionen $f, g : A \to B$ miteinander übereinstimmen. Kategorientheoretisch lassen sie sich als Limites über Diagrammen mit dem folgenden Formgraphen EQL erfassen:

$$1 \underset{b}{\overset{a}{\rightrightarrows}} 2$$

Ein **Equalizer-Diagramm** in einer Kategorie $\mathbf{C}$ ist ein Diagramm der Form EQL in $\mathbf{C}$. Ein Limes $p : X \to D$ eines Equalizer-Diagrammes $D : \mathrm{EQL} \to \mathbf{C}$ in $\mathbf{C}$ heißt **Equalizer** von $D(a)$ und $D(b)$ in $\mathbf{C}$.

Für ein Equalizer-Diagramm $D = \left(S \underset{g}{\overset{f}{\rightrightarrows}} T \right)$ in **Set** ist $C := \{s \in S \mid f(s) = g(s)\}$ der Scheitel eines Limes $p : C \to D$. Dabei sind die Komponenten p_1 und p_2 des Limes die Inklusionsfunktionen von C in A bzw. B, also $p_1 : C \to A : x \mapsto x$ und $p_1 : C \to B : x \mapsto x$. □

Neben den vorgestellten Limites und Kolimites gibt es zahlreiche weitere, insbesondere auch die jeweils dualen Limites zu Koprodukt, initialem Objekt und Equalizer. Die im vorangehenden Unterabschnitt aufgeführten Limites und Kolimites werden aber für diese Arbeit gebraucht, um mit Hilfe der im nächsten Abschnitt vorgestellten Sketches die verschiedenen Graphfeatures zu definieren.

4.2. Sketchbasierte Definition von Kategorien

Die auf Sketches basierende Definition von Kategorien ist eine Methode, mithilfe von Graphen, kommutativen Diagrammen und Limites und Kolimites neue Kategorien auf Grundlage bestehender Kategorien zu definieren. In dieser Arbeit werden Sketches verwendet, um die verschiedenen Graphfeatures (bzw. die entsprechenden Kategorien) zu definieren

und diese zu Graphvarianten für die Programmrepräsentation zu kombinieren. Es gibt verschiedene Arten von Sketches mit unterschiedlicher Ausdrucksfähigkeit und zwar triviale Sketches, linearen Sketches, allgemeinen Sketches und typisierten Sketches unterschieden.

Sketches sind eine Art Schablone für Ausschnitte aus einer bestehenden Kategorie, die die Objekte einer neuen Kategorie bilden und als Diagramme formalisiert werden. Diese mittels Sketches definierten Kategorien heißen Instanzkategorien. Die Instanzkategorie eines Sketches hat Diagramme mit einem festen Formgraphen G in einer bestehenden Kategorie C als Objekte, die man als Instanzen bezeichnet[5]. Ob dabei alle Diagramme der Form G in C Instanzen eines Sketches sind oder nur bestimmte, hängt von der Art des Sketches ab. Die Instanzkategorien aller Sketcharten haben als Pfeile natürliche Transformationen zwischen den Diagrammen und als Komposition die komponentenweise Komposition natürlicher Transformationen. Darüber hinaus bilden die Sketches jeder Sketchart zusammen mit einem geeigneten Homomorphismusbegriff selbst eine Kategorie. Indem man für jedes Graphfeature einen Sketch definiert, kann man sie dann durch Pushouts in dieser Sketchkategorie zu Graphvarianten kombinieren.

In den folgenden Unterabschnitten werden die vier Sketchartarten, ihre Instanzen und ihre Instanzkategorien formal definiert und anhand eines Beispiels erläutert.

4.2.1. Triviale Sketches

Die einfachste und am wenigsten ausdruckskräftigste Art von Sketches sind die trivialen Sketches. Ein trivialer Sketch ist nichts anderes als ein Graph $S = (src_S, tar_S) \in \mathfrak{Graphs}$ im Sinne von Definition 2. Die Klasse der trivialen Sketches ist also die Klasse $\mathfrak{Graphs}$ der Graphen. Im Kontext von Unterabschnitt 4.1.2 würde man einen trivialen Sketch S als Formgraph seiner Instanzen bezeichnen, die allesamt Diagramme der Art $I : S \to \mathbf{C}$ sind. Triviale Sketches stellen keine weiteren Anforderungen an ihre Instanzen, so dass sämtliche Diagramme mit Formgraph S Instanzen von S sind. Formal gilt also:

Definition 60 (Trivialer Sketch)

Trivialer Sketch

Ein **trivialer Sketch** ist ein Graph $S \in \mathfrak{Graphs}$.

Ein Knoten $v \in S^0$ wird als **formales Objekt** von S und ein Pfeil $e \in S^1$ als **formaler Pfeil** bezeichnet. □

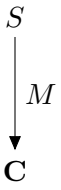
Definition 61 (Instanz eines trivialen Sketches)

Instanz eines trivialen Sketches

Sei $S \in \mathfrak{Graphs}$ ein trivialer Sketch.
Sei $\mathbf{C}$ eine Kategorie.

Eine **Instanz** von S in $\mathbf{C}$ ist ein Diagramm $I : S \to \mathbf{C}$ der Form S in C.
Falls $\mathbf{C} = \mathbf{Set}$ ist, dann heißt I einfach Instanz von S.

[5] In der Literatur üblicherweise „Modell“, aber auch „Interpretation“ oder „Ausprägung“. Ähnlich wie in [RWL08] wird hier der Begriff „Modell“ wegen der Nähe zur modellgetriebenen Entwicklung vermieden. Dort ist er nämlich bereits auf eine Weise besetzt, die eher auf Sketches als auf ihre Instanzen zutrifft.



Im Folgenden werden die Bestandteile einer Instanz $I : S \to \mathbf{C}$ eines trivialen Sketches S auf eine andrere Weise geschrieben als bei Diagrammen üblich, nämlich:

- Für ein formales Objekt $\mathtt{A} \in S^0$ wird $\mathtt{A}_I$ anstatt von $I(\mathtt{A}) \in \mathbf{C}^0$ geschrieben. $\mathtt{A}_I$ heißt Interpretation von $\mathtt{A}$ unter I.
- Für einen formalen Pfeil $\mathtt{A} \xrightarrow{\mathtt{s}} \mathtt{B} \in S^1$ wird $\mathtt{s}_I$ anstatt $I(\mathtt{s}) \in \mathbf{C}^1$ geschrieben. $\mathtt{s}_I$ heißt Interpretation von $\mathtt{s}$ unter I. □

Der Zweck von Sketches ist es, neue Kategorien zu definieren. Wie bereits in der Einleitung zum Abschnitt erklärt wurde, sind die Pfeile dieser Kategorien die natürlichen Transformationen zwischen den Instanzen. Diese sind zwar schon aus Definition 18 bekannt, aber man kann sie auf eine für Sketches geeignetere Art notieren:

Definition 62 (Natürliche Transformation von Instanzen eines Sketches)

Natürliche Transformation von Instanzen eines Sketches

Sei $S \in \mathfrak{Graphs}$ ein trivialer Sketch.
Sei $\mathbf{C}$ eine Kategorie.
Seien $I : S \to \mathbf{C}$ und $J : S \to \mathbf{C}$ Instanzen von S in $\mathbf{C}$.
Sei $\alpha = (\alpha_{\mathtt{A}})_{\mathtt{A} \in S^0}$ eine Familie von Pfeilen $\mathtt{A}_I \xrightarrow{\alpha_{\mathtt{A}}} \mathtt{A}_J$ aus $\mathbf{C}^1$, die von den Interpretationen der formalen Objekte von S unter I hin zu ihren Interpretationen unter J gehen.

Dann ist (I, J, α) genau dann eine natürliche Transformation von I nach J, geschrieben $\alpha : I \to J$, wenn für jeden formalen Pfeil $\mathtt{A} \xrightarrow{\mathtt{s}} \mathtt{B} \in S^1$ gilt: $\mathtt{s}_I \bullet \alpha_{\mathtt{B}} = \alpha_{\mathtt{A}} \bullet \mathtt{s}_J$

$$\begin{array}{ccc} \mathtt{A}_M & \xrightarrow{\mathtt{s}_M} & \mathtt{B}_M \\ {\scriptstyle \alpha_{\mathtt{A}}}\downarrow & \sim & \downarrow{\scriptstyle \alpha_{\mathtt{B}}} \\ \mathtt{A}_N & \xrightarrow[\mathtt{s}_N]{} & \mathtt{B}_N \end{array}$$

Die Menge aller natürlichen Transformationen zwischen zwei Instanzen I und J eines Sketches heißt $\mathfrak{NatTrans}(I, J)$.

Zwei aufeinanderfolgende natürliche Transformationen $\alpha : I \to J$, $\beta : J \to P$ zwischen Modellen $I, J, P : S \to \mathbf{C}$ von S kann man miteinander komponieren zu

$$\alpha \bullet \beta : I \to P \text{ mit } \forall \mathtt{A} \in S^0 \colon (\alpha \bullet \beta)_{\mathtt{A}} = \alpha_{\mathtt{A}} \bullet \beta_{\mathtt{A}}$$ □

Eine natürliche Transformation von Instanzen eines (trivialen) Sketches besteht also aus einer Familie von Funktionen, nämlich für jedes formale Objekt des Sketches eine. Diese Funktionen nennt man **Komponenten**. Eine natürliche Transformation hat die Eigenschaft, dass sie mit jedem Pfeil im Sketch (bzw. dessen Interpretation) „verträglich“ ist. Das

bedeutet, dass die Beziehungen der im Sketch repräsentierten Elemente bei der Anwendung einer natürlichen Transformation erhalten bleiben, was formal durch die Kommutativitätsbedingung aus Definition 62 ausgedrückt wird.

Die Instanzen eines trivialen Sketches und die natürlichen Transformationen zwischen diesen Instanzen liegen einer Kategorie zugrunde:

Definition 63 (Kategorie der Instanzen eines trivialen Sketches)

Kategorie der Instanzen eines trivialen Sketches

Sei $\mathbf{C}$ eine Kategorie.
Sei $S \in \mathfrak{Graphs}$ ein trivialer Sketch.

Dann ist die Kategorie $\mathbf{I}_\mathbf{C}(S)$ der Instanzen von S in C folgendermaßen definiert:

- Die Objekte sind die Instanzen von S in C, d.h. alle Diagramme der Form S in $\mathbf{C}$: $(\mathbf{I}_\mathbf{C}(S))^0 = \mathbf{C}^S$
- Die Pfeile zwischen zwei Instanzen sind die natürlichen Transformationen zwischen ihnen: $(\mathbf{I}_\mathbf{C}(S))[I, J] = \mathfrak{NatTrans}(I, J)$
- Die Komposition zweier natürlicher Transformationen $\alpha : I \to J$ und $\beta : J \to P$ ist ihre komponentenweise Komposition $\alpha \bullet \beta : I \to P$, die definiert ist durch $(\alpha \bullet \beta)_A = \alpha_A \bullet \beta_A$ für $A \in S^0$.

Falls $\mathbf{C} = \mathbf{Set}$ ist, wird anstatt $\mathbf{I}_\mathbf{C}(S)$ auch $\mathbf{I}(S)$ geschrieben. □

Auf diese Weise lassen sich mithilfe eines trivialen Sketches Kategorien definieren, beispielsweise eine Kategorie für untypisierte Graphen.

Für untypisierte Graphen ist bereits eine mögliche Formalisierung bekannt, nämlich die Graphen aus Definition 2. Diese Formalisierung ist aber nicht dafür geeignet, sie als Graphfeature aufzufassen, das man mit anderen Graphfeatures zu Graphvarianten zur Programmrepräsentation kombinieren kann. Eine für diesen Zweck geeignetere Formalisierung der untypisierten Graphen basiert auf einem trivialen Sketch:

Beispiel 64 (Untypisierte Graphen mithilfe von Sketches)

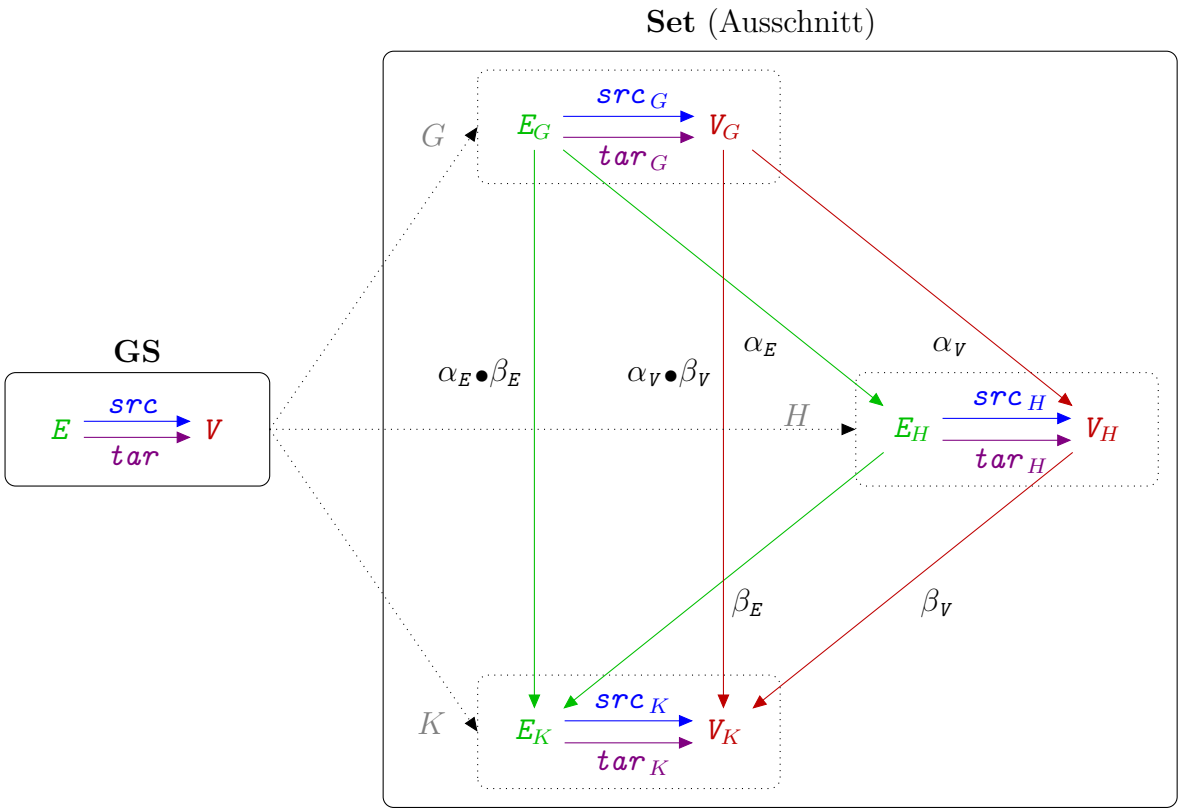
Untypisierte Graphen mithilfe von Sketches

Zur sketchbasierten Definition untypisierter Graphen wird der folgende triviale Sketch **GS** verwendet, der als **Sketch der untypisierten Graphen** bezeichnet wird:

$$E \underset{tar}{\overset{src}{\rightrightarrows}} V$$

GS beschreibt der untypisierten Graphen – eine Instanz von **GS** entspricht einem untypisierten Graphen. Dies ist zu erkennen, indem man Abb. 31 betrachtet. Darin sind der triviale Sketch **GS** und drei Instanzen G, H und K von **GS** in der Kategorie **Set** abgebildet. Eine solche Instanz, also ein Diagramm der Form **GS**, ordnet den formalen Objekten E und V beliebige Mengen und den formalen Pfeilen src und tar beliebige Funktionen zu. Diese kann man als Knoten- und Pfeilmengen bzw. als Quell- und Zielfunktion eines Graphen betrachten. Es könnte sich beispielsweise um die in der Tabelle in Abb. 31 beschriebenen Mengen und Funktionen handeln.

Die Pfeile der Instanzkategorie sind die natürlichen Transformationen, wie sie auf Seite 60 definiert wurden. Wendet man die Definition der natürlichen Transformationen auf die Instanzen G und H von **GS** an, ergibt sich, dass eine natürliche Transformation



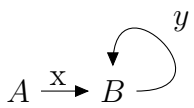
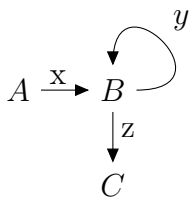
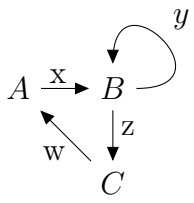
Instanzname	Instanz	Illustration
G	$V_G = \{A, B\}$, $E_G = \{x, y\}$, $src_G = \{x \mapsto A, y \mapsto B\}$, $tar_G = \{x \mapsto B, y \mapsto B\}$	
H	$V_H = \{A, B, C\}$, $E_H = \{x, y, z\}$, $src_H = \{x \mapsto A, y \mapsto B, z \mapsto B\}$, $tar_H = \{x \mapsto B, y \mapsto B, z \mapsto C\}$	
K	$V_K = \{A, B, C\}$, $E_K = \{x, y, z, w\}$, $src_K = \{x \mapsto A, y \mapsto B, z \mapsto B, w \mapsto C\}$, $tar_K = \{x \mapsto B, y \mapsto B, z \mapsto C, w \mapsto A\}$	

Abbildung 31: Instanzen G, H und K des trivialen Sketch **GS** in **Set**

von G nach H aus zwei Pfeilen $E_G \xrightarrow{\alpha_E} E_H$ und $V_G \xrightarrow{\alpha_V} V_H$ von **Set** (also Funktionen) besteht, die mit den Interpretationen von $\mathtt{src}$ und $\mathtt{tar}$ verträglich sind, oder formal: $\mathtt{src}_G \bullet \alpha_V = \alpha_E \bullet src_G$ und $tar_G \bullet \alpha_V = \alpha_E \bullet \mathtt{tar}_H$. Das entspricht aber gerade der Definition genau eines Graphhomomorphismus von G nach H. Auch die komponentenweise Komposition solcher natürlicher Transformationen entspricht der Komposition von Graphhomomorphismen. Da in der Instanzkategorie von **GS** also die Objekte mit den untypisierten Graphen, die Pfeile mit den Graphhomomorphismen und die Komposition mit der Komposition von Graphhomomorphismen korrespondieren, beschreibt der Sketch **GS** also tatsächlich die Kategorie der untypisierten Graphen. □

Sketches eignen sich unter anderem zur Definition der Graphfeatures, weil man sie durch Pushouts miteinander zu Graphvarianten kombinieren kann. Außerdem ist es für die spätere Definitionen nützlich, Teilsketches von Sketches zu betrachten. Für beides benötigt man einen Begriff des Homomorphismus von Sketchen und Kategorien von Sketches. Im Fall von trivialen Sketches handelt es sich dabei um Graphhomomorphismen bzw. die Kategorie **Graphs**.

Kategorie der trivialen Sketches

Definition 65 (Kategorie der trivialen Sketches)
Die Kategorie der trivialen Sketches ist die Kategorie **Graphs** der Graphen und Graphhomomorphismen. □

Homomorphismus trivialer Sketches

Definition 66 (Homomorphismus trivialer Sketches)
Seien $S_1, S_2 \in \mathfrak{Graphs}$ triviale Sketches.

Ein Homomorphismus h trivialer Sketches von S_1 nach S_2, geschrieben $h : S_1 \to S_2$, ist ein Graphhomomorphismus von S_1 nach S_2. □

Da man S_1 als Teilsketch von S_2 betrachten kann, kann man eine Instanz von S_2 zu einer Instanz von S_1 einschränken, indem man nur die Interpretation der formalen Objekte und Pfeile betrachtet, die in S_1 vorhanden sind. Formal handelt es sich bei einer solchen Einschränkung um einen Funktor:

Einschränkung einer Instanz eines trivialen Sketches

Definition 67 (Einschränkung einer Instanz eines trivialen Sketches)
Sei **C** eine Kategorie.
Sei $h : S_1 \to S_2$ ein Homomorphismus typisierter Sketches.
Sei $I : S_2 \to \mathbf{C}$ eine Instanz von S_2 in **C**.

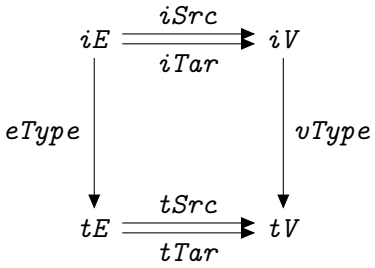
Dann ist $h_I \coloneqq h \bullet I$ die Einschränkung von I auf S_1 mittels h. □

Wie Beispiel 64 gezeigt hat, sind triviale Sketches ausdrucksstark genug für untypisierte Graphen, aber ingesamt reichen sie für die Zwecke dieser Arbeit nicht aus:

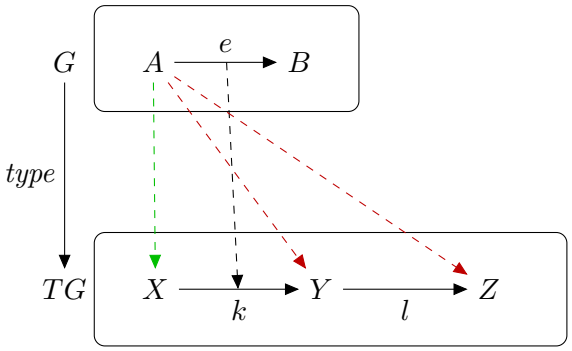
Ausdruckskraft trivialer Sketches

Bemerkung 68 (Ausdruckskraft trivialer Sketches)
Mit trivialen Sketches lassen sich einige wichtige Kategorien wie die untypisierten Graphen, oder auch die natürlichen Zahlen, modellieren. Aber bereits für typisierte Graphen sind triviale Sketches nicht ausdrucksstark genug, wie die folgenden Ausführungen zeigen.

Die wichtigste „Zutat“ für einen typisierten Graphen ist sein Typisierungshomomorphismus *type*. Als Graphhomomorphismus besteht dieser im Wesentlichen aus einer Pfeilkomponente, die Pfeilen ihren Pfeiltyp zuordnet, und einer Knotenkomponente, die Knoten ihren Knotentyp zuordnet. Da diese Komponenten Funktionen sind, kann man sie in einer Instanz eines trivialen Sketches in der Kategorie **Set** erfassen. Hierfür verdoppelt man **GS** – einmal für den Instanzgraphen, einmal für den Typgraphen – und fügt Pfeile $\mathtt{eType}$ und $\mathtt{vType}$ für die beiden Komponenten des Typisierungshomomorphismus ein. So erhält man den folgenden trivialen Sketch für typisierte Graphen $\mathbf{TGS}_{triv}$:



Ein typisierter Graph GT mit einem Typisierungshomomorphismus $type : G \to TG$, Typgraphen TG und Instanzgraphen G lässt sich mit einer Instanz $I : \mathbf{TGS}_{triv} \to \mathbf{Set}$ dieses Sketches vollständig erfassen[6]. In umgekehrter Richtung entspricht aber nicht jede Instanz $I : \mathbf{TGS}_{triv} \to \mathbf{Set}$ von $\mathbf{TGS}_{triv}$ einem typisierten Graphen, d.h. es gibt „ungültige“ Instanzen. Zwar ergeben sich immer ein Instanzgraph G und ein Typgraph TG; ein typisierter Graph ergibt sich aber nur, wenn $\mathtt{eType}_I$ und $\mathtt{vType}_I$ einen Graphhomomorphismus bilden, den man als Typisierungshomomorphismus *type* verwenden kann. Hierfür müssen sie die definierende Regel von Graphhomomorphismen erfüllen. Dies ist aber nicht zwangsläufig der Fall, da in einer Instanz von $\mathbf{TGS}_{triv}$ die Funktionen $\mathtt{eType}_I$ und $\mathtt{vType}_I$ vollkommen beliebig gewählt werden können.



Wenn beispielsweise in einer Instanz I von $\mathbf{TGS}_{triv}$ die Funktion $\mathtt{eType}_I$ einem Pfeil $A \xrightarrow{e} B \in \mathtt{iE}_I$ den Pfeiltyp $X \xrightarrow{k} Y \in \mathtt{tE}_I$ zuordnet, müsste $\mathtt{vType}_I$ bei typisierten Graphen ihrem Quellknoten A den Knotentyp X zuordnen. Wie in der obigen Grafik illustriert ist, gibt es aber auch Instanzen, in denen $\mathtt{vType}_I$ dem Knoten A den Knotentyp Y oder Z zuordnet, was in typisierten Graphen nicht zulässig ist.

[6] Die Knotenmenge des Instanzgraphen G wird mit $\mathtt{iV}_I$ repräsentiert, die Pfeilmenge mit $\mathtt{iE}_I$, die Quell- und die Zielknotenfunktion mit $\mathtt{iSrc}_I$ bzw. $\mathtt{iTar}_I$. Ebenso wird Die Knotenmenge des Typgraphen TG wird mit $\mathtt{tV}_I$, die Pfeilmenge mit $\mathtt{tE}_I$, die Quell- und die Zielknotenfunktion mit $\mathtt{tSrc}_I$ bzw. $\mathtt{tTar}_I$ repräsentiert. Die Pfeilkomponente von *type* wird mit $\mathtt{eType}$ und die Knotenkomponente mit $\mathtt{vType}$ erfasst.

Der trivale Sketch $\mathbf{TGS}_{triv}$ lässt also zu viele Instanzen in **Set** zu, nämlich auch solche, bei denen eType_I und vType_I keinen Graphhomomorphismus bilden. Triviale Sketches geben aber auch keine Möglichkeit, auszudrücken, dass es sich dabei um einen Graphhomomorphismus handeln muss. Hierfür sind ausdrucksstärkere Varianten von Sketches nötig, die in den nächsten Unterabschnitten vorgestellt werden. □

4.2.2. Lineare Sketches

Wie in Bemerkung 68 dargelegt wurde, kann man mit trivialen Sketches die Klasse der Instanzen nicht ausreichend einschränken, um typisierte Graphen exakt zu erfassen. Ähnliche Probleme gibt es bei allen betrachteten Graphfeatures außer den untypisierten Graphen. Eine Lösungsmöglichkeit hierfür bieten lineare Sketches. Ein linearer Sketch besteht aus einem trivialen Sketch S und einer Menge von Diagrammen der Art $L : K \to S$ in S. Eine Instanz des linearen Sketches ist eine Instanz $I : S \to \mathbf{C}$ des trivialen Sketches S, für die jedes der Diagramme kommutiert, d.h. für die $L \bullet I$ ein kommutatives Diagramm ist. Hierdurch lassen sich beispielsweise die typisierten Graphen durch die Instanzkategorie eines linearen Sketches exakt erfassen:

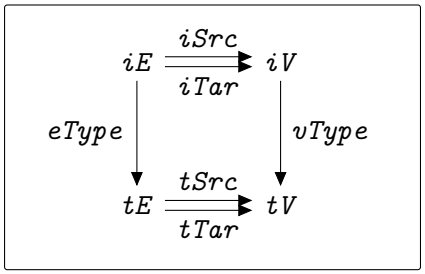
Beispiel 69 (Sketch der typisierten Graphen)

Sketch der typisierten Graphen

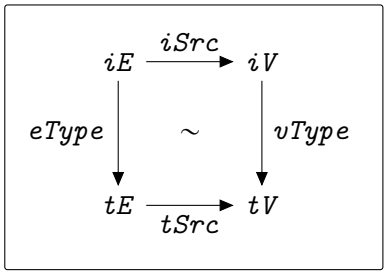
Das Problem im Beispiel zu Bemerkung 68 war, dass es Instanzen von $\mathbf{TGS}_{triv}$ gibt, die keinem typisierten Graphen entsprechen, weil die Interpretationen von eType und vType keinen Graphhomomorphismus bilden. Man kann solche Instanzen in trivialen Sketches aber auch nicht ausschließen. In linearen Sketches kann man dies hingegen mit kommutativen Diagrammen erreichen. Mit diesen kann man nämlich wie in Beispiel 39 festlegen, dass eType und vType einen Graphhomomorphismus bilden müssen.

Der Sketch der typisierten Graphen **TGS** ist ein linearer Sketch über dem trivialen Sketch $\mathbf{TGS}_{triv}$ der typisierten Graphen. Er enthält die zwei kommutativen Diagramme $\mathit{typePreservesSrc} : K_1 \to \mathbf{TGS}_{triv}$ und $\mathit{typePreservesTar} : K_2 \to \mathbf{TGS}_{triv}$.

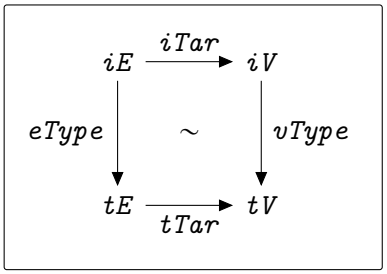
$\mathbf{TGS}_{triv}$ (triv. Sketch):



typePreservesSrc (KD):



typePreservesTar (KD):



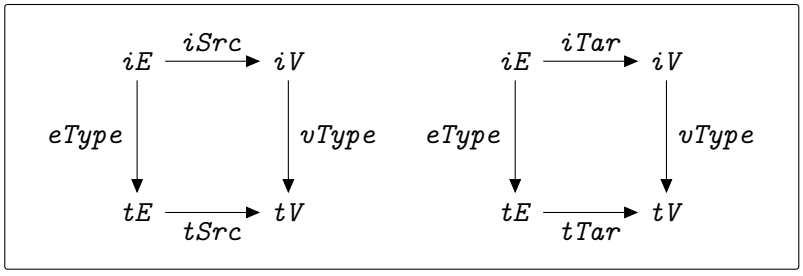
Im Beispiel aus Bemerkung 68 kommutiert *typePreservesSrc* nur in der Instanz I von $\mathbf{TGS}_{triv}$, in der dem Knoten A der Typ X zugeordnet wird, denn aus der Kommutativität folgt $\mathit{iSrc}_I \bullet \mathit{vType}_I = \mathit{eType}_I \bullet \mathit{tSrc}_I$ und somit: $\mathit{vType}_I(A) = \mathit{vType}_I(\mathit{iSrc}_I(e)) = \mathit{iSrc}_I \bullet \mathit{vType}_I(e) = \mathit{eType}_I \bullet \mathit{tSrc}_I(e) = \mathit{tSrc}_I(\mathit{eType}_I(e)) = \mathit{tSrc}_I(k) = X$. In den Instanzen von $\mathbf{TGS}_{triv}$, die A den Typ Y oder Z zuordnen, kommutiert deshalb *typePreservesSrc* nicht und somit sind sie keine Instanzen von **TGS**.

Deshalb entspricht jeder typisierte Graph einer Instanz von **TGS**, denn der zugehörige Typisierungshomomorphismus ist ein Graphhomomorphismus und deshalb kommutieren die beiden Diagramme. In umgekehrter Richtung sind in jeder Instanz $I : \mathbf{TGS} \to \mathbf{Set}$ die Funktionen eType_I und vType_I die Komponenten eines Graphhomomorphismus. In G kommutieren nämlich die Diagramme *typePreservesSrc* und *typePreservesTar*, was nach Beispiel 39 der Definition eines Graphhomomorphismus gleichkommt. Deshalb repräsentiert eine Instanz $I : \mathbf{TGS} \to \mathbf{Set}$ den (eindeutigen) typisierten Graphen, der diesen Graphhomomorphismus als Typisierungshomomorphismus hat. Somit lässt sich eine Kategorie aller typisierten Graphen mit linearen Sketches definieren.

Allerdings interessieren oft nicht über beliebigen Typgraphen typisierte Graphen, sondern man will einen Typgraphen TG fixieren und nur über TG typisierte Graphen betrachten. Mit linearen Sketches allein ist dies nicht möglich. Im Unterabschnitt 4.2.4 wird eine Lösung hierzu vorgeschlagen. □

Für die formale Definition linearer Sketches reicht es, lediglich ein einziges Diagramm zu betrachten. Man kann nämlich eine beliebige Menge von Diagrammen zu einem einzigen Diagramm $L : K \to S$ zusammenfassen, ohne dass sich die Klasse der Instanzen des Sketches ändert[7]: Formal betrachtet fasst man eine Menge $\mathcal{D}$ von Diagrammen in S mit kleinen Formgraphen zusammen, indem man das Kotupel $L = \langle D \rangle_{D \in \mathcal{D}}$ bildet, das ja selbst ein Diagramm ist. Der Formgraph von L ist die disjunkte Vereinigung der Formgraphen der Diagramme von $\mathcal{D}$, also $K = \biguplus_{D \in \mathcal{D}} \mathit{src}(D)$ (siehe Beispiel 56). Das Kotupel einer Menge von Diagrammen kommutiert genau dann, wenn jedes einzelne Diagramm kommutiert. Deshalb ändert sich durch das Zusammenfassen der Diagramme die Menge der Pfade, die kommutieren müssen, nicht. Beispielsweise kann man die beiden Diagramme aus Beispiel 69 zu dem folgenden Diagramm $\mathit{typeIsHom} := \langle \mathit{typePreservesSrc} \,|\, \mathit{typePreservesTar} \rangle$ zusammenfassen, das offensichtlich genau dann kommutiert, wenn die beiden Teildiagramme kommutieren:

typeIsHom (KD):



[7] Dies ist möglich, weil **Graphs** eine sogenannte kovollständige Kategorie ist. Dies bedeutet, dass es zu jedem Diagramm mit einem (kleinen) Formgraphen in **Graphs**, also insbesondere auch zu Koproduktdiagrammen, einen Kolimes gibt [BW12].

Da sich auch der zugrundeliegende lineare Sketch $S = tar(D)$ als Kodomäne von D ableiten lässt, ist ein linearer Sketch formal dasselbe wie ein Diagramm in einem trivialen Sketch:

Definition 70 (Linearer Sketch)

Linearer Sketch

Sei $S \in \mathfrak{Graphs}$ ein trivialer Sketch.
Sei $K \in \mathfrak{Graphs}$ ein Graph.

Dann ist ein linearer Sketch über S ein Diagramm $L : K \to S$ in S.

$$K \xrightarrow{\quad L \quad} S$$

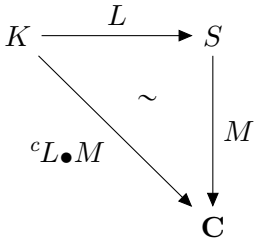
Definiere $\mathfrak{Sk}_{lin} = \bigcup_{K,S \in \mathfrak{Graphs}} S^K$ als die Klasse aller linearen Sketches. □

Definition 71 (Instanz eines linearen Sketches)

Instanz eines linearen Sketches

Sei $L : K \to S$ ein linearer Sketch über S.
Sei $\mathbf{C}$ eine Kategorie.
Sei $I : S \to \mathbf{C}$ eine Instanz von S in $\mathbf{C}$.

Dann ist I genau dann eine Instanz von L in $\mathbf{C}$, wenn gilt: $comm(L \bullet I)$



Für eine Instanz I von L in $\mathbf{C}$ wird $I : L \to \mathbf{C}$ geschrieben.
Die Klasse der Instanzen von L in $\mathbf{C}$ heißt $\mathbf{C}^L$. □

Obwohl diese Definition kürzer ist, ist es zur Definition und zur Erläuterung linearer Sketches ganz nützlich, die einzelnen Diagramme auseinanderzuhalten. Deshalb werden im Folgenden bei Definitionen von Sketches mehrere Diagramme $D_1, D_2, \ldots, D_n$ genannt, die aber formal als ein einziges betrachtet werden, nämlich als ihr Kotupel $L = \langle D_1 \mid D_2 \mid \ldots \mid D_n \rangle$.

Die Instanzkategorie eines linearen Sketches hat alle Instanzen des Sketches als Objekte und definiert sich ansonsten genau wie die Instanzkategorie eines trivialen Sketches:

Definition 72 (Kategorie der Instanzen eines linearen Sketches)

Kategorie der Instanzen eines linearen Sketches

Sei $\mathbf{C}$ eine Kategorie.
Sei $L : K \to S$ ein linearer Sketch.

Dann ist die Kategorie $\mathbf{I}_\mathbf{C}(L)$ der Instanzen von L in C folgendermaßen definiert:

- Die Objekte sind die Instanzen von L in C, d.h. alle Diagramme I der Form S in $\mathbf{C}$, für die das Diagramm $L \bullet I : K \to \mathbf{C}$ kommutiert: $(\mathbf{I}_\mathbf{C}(L))^0 = \mathbf{C}^L$
- Die Pfeile zwischen zwei Instanzen sind die natürlichen Transformationen zwischen ihnen: $(\mathbf{I}_\mathbf{C}(L))[I, J] = \mathfrak{NatTrans}(I, J)$

- Die Komposition natürlicher Transformationen $\alpha : I \to J$ und $\beta : J \to P$ ist ihre komponentenweise Komposition $\alpha \bullet \beta : I \to P$ mit $(\alpha \bullet \beta)_{\mathtt{A}} = \alpha_{\mathtt{A}} \bullet \beta_{\mathtt{A}}$ für $\mathtt{A} \in S^0$.

Falls $\mathbf{C} = \mathbf{Set}$ ist, wird anstatt $\mathbf{I}_{\mathbf{C}}(L)$ auch $\mathbf{I}(L)$ geschrieben. □

Außerdem gibt es auch für lineare Sketches selbst einen Homomorphismusbegriff und die Kategorie der linearen Sketches.

Definition 73 (Homomorphismus linearer Sketches)

Homomorphismus linearer Sketches

Seien $L_1 : K_1 \to S_1$ und $L_2 : K_2 \to S_2$ lineare Sketches.
Seien $h_K : K_1 \to K_2$ und $h_S : S_1 \to S_2$ Graphhomomorphismen.

Dann ist $h = (L_1, L_2, h_K, h_S)$ genau dann ein **Homomorphismus linearer Sketches**, wenn gilt: $L_1 \bullet h_S = h_K \bullet L_2$.

$$\begin{array}{ccc} K_1 & \xrightarrow{L_1} & S_1 \\ {\scriptstyle h_K}\downarrow & \sim & \downarrow{\scriptstyle h_S} \\ K_2 & \xrightarrow[L_2]{} & S_2 \end{array}$$

Man schreibt hierfür $h : L_1 \to L_2$. Die Klasse aller Homomorphismen linearer Sketches von L_1 nach L_2 ist definiert als $L_2^{L_1} = \{(L_1, L_2, h_K, h_S) \mid L_1 \bullet h_S = h_K \bullet L_2\}$. □

Definition 74 (Kategorie der linearen Sketches)

Kategorie der linearen Sketches

Die **Kategorie der linearen Sketches** heißt **LSk** und ist folgendermaßen definiert:

- Die Objekte sind die linearen Sketches, d.h. die Graphhomomorphismen: $\mathbf{LSk}^0 = \mathfrak{Sk}_{lin}$
- Die Pfeile zwischen zwei linearen Sketches $L_1 : K_1 \to S_1$ und $L_2 : K_2 \to S_2$ sind die Homomorphismen linearer Sketches: $\mathbf{LSk}[L_1, L_2] = {L_2}^{L_1}$
- Die Komposition zweier Homomorphismen $h = (L_1, L_2, h_K, h_S)$ und $g = (L_2, L_3, g_K, g_S)$ ist komponentenweise definiert: $h \bullet g = (L_1, L_3, h_K \bullet g_K, h_S \bullet g_S)$

In der Literatur ist diese Kategorie auch bekannt als die **Pfeilkategorie** von **Graphs**. □

Wie bei primitiven Sketches kann man Homomorphismen linearer Sketches verwenden, um Instanzen auf Teilsketches einzuschränken:

Definition 75 (Einschränkung einer Instanz eines linearen Sketches)

Einschränkung einer Instanz eines linearen Sketches

Sei $\mathbf{C}$ eine Kategorie.
Seien $L_1 : K_1 \to S_1$ und $L_2 : K_2 \to S_2$ lineare Sketches.
Sei $h : L_1 \to L_2$ ein Homomorphismus linearer Sketches von L_1 nach L_2.
Sei dabei $h = (L_1, L_2, h_K, h_S)$.
Sei $I : L_2 \to \mathbf{C}$ eine Instanz von S_2 in $\mathbf{C}$.

Dann ist $h_I \coloneqq h_S \bullet I$ die Einschränkung von I auf L_1 mittels h. □

Beweis 76 (Die Einschränkung eines linearen Sketches ist eine Instanz)

Die Einschränkung eines linearen Sketches ist eine Instanz

Es ist nicht sofort offensichtlich, dass die Einschränkung eines linearen Sketches auf einen Teilsketch wie sie oben definiert ist tatsächlich eine Instanz des Teilsketches ist. Dies ist aber der Fall: Seien $L_1 : K_1 \to S_1$ und $L_2 : K_2 \to S_2$ lineare Sketches. Sei $h : L_1 \to L_2$ ein Homomorphismus linearer Sketches und $I : L_2 \to \mathbf{C}$ eine Instanz von L_2 in einer Kategorie $\mathbf{C}$. D.h. I ist eine Instanz $I : S_2 \to \mathbf{C}$ des zugrundeliegenden trivialen Sketches von L_2 für die das Diagramm $L_2 \bullet I$ kommutiert. Die Einschränkung von L_2 auf L_1 mittels h ist $h_I = h_S \bullet I$, also ein Diagramm $h_I : S_1 \to \mathbf{C}$ der Form I in $\mathbf{C}$, und somit ein Modell des zugrundeliegenden trivialen Sketches von L_1. Zu zeigen ist, dass $L_1 \bullet h_I$ kommutiert, so dass h_I eine Instanz von L_1 ist.

Dies trifft zu, denn: Da $L_2 \bullet I$ kommutiert, ergibt für jedes Paar $A' \overset{p'}{\underset{q'}{\rightrightarrows}} B' \in \mathbf{P}_{K_2}$ von parallelen Pfaden in K_2 die Komposition denselben Wert $\mathbf{C}_{L_2 \bullet I}(p') = \mathbf{C}_{L_2 \bullet I}(q')$. Es ist $L_1 \bullet h_S = h_K \bullet L_2$, da h ein Homomorphismus linearer Sketches ist. Dadurch gilt:

$$\begin{aligned}\mathbf{C}_{L_1 \bullet h_I} &= \mathbf{C}_{L_1 \bullet h_S \bullet I} = \mathbf{C}_{h_K \bullet L_2 \bullet I} \\ &= \mathbf{P}_{h_K \bullet L_2 \bullet I} \bullet \mathbf{C}(\cdot) = \mathbf{P}_{h_K} \bullet \mathbf{P}_{L_2 \bullet I} \bullet \mathbf{C}(\cdot) \\ &= \mathbf{P}_{h_K} \bullet \mathbf{C}_{L_2 \bullet I}\end{aligned}$$

Sei nun $A \overset{p}{\underset{q}{\rightrightarrows}} B \in \mathbf{P}_{K_1}$ ein Paar paralleler Pfade in K_1. Da $L_2 \bullet I$ kommutiert, gilt:

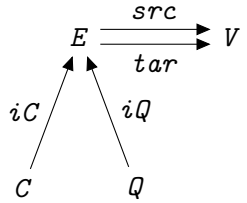
$$\mathbf{C}_{L_1 \bullet h_I}(p) = \mathbf{P}_{h_K} \bullet \mathbf{C}_{L_2 \bullet I}(p) = \mathbf{P}_{h_K} \bullet \mathbf{C}_{L_2 \bullet I}(q) = \mathbf{C}_{L_1 \bullet h_I}(q)$$

Somit kommutiert $L_1 \bullet h_I$, weshalb h_I ein Modell von L_1 ist. □

Bemerkung 77 (Ausdruckskraft linearer Sketches)

Ausdruckskraft linearer Sketches

Desweiteren kann man beispielsweise disjunkte Vereinigungen oder Pullbacks nicht ausdrücken, die für Containment und Attributierung benötigt werden. Dies illustriert das folgende Beispiel: Gegeben ein linearer Sketch $L : 0 \to S$ mit dem man die Graphvariante der untypisierten Graphen mit Containment modellieren kann:

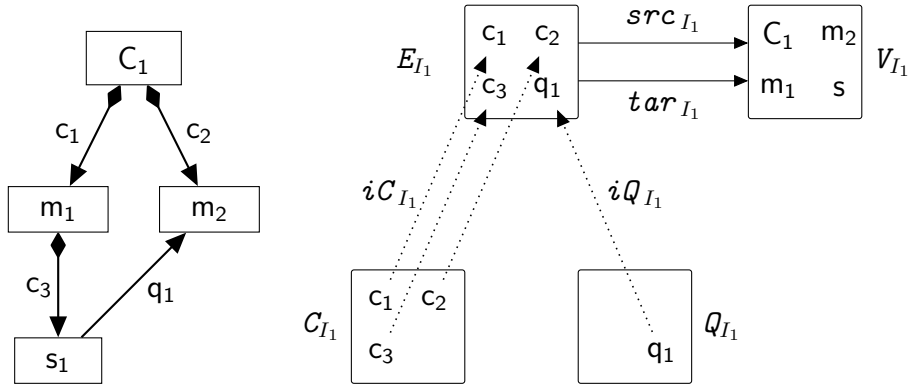


Dabei steht das formale Objekt `Q` für die Menge Q der Querpfeile, `C` für die Menge C Containmentpfeile und `E` für die Menge E aller Pfeile in einem untypisierten Graphen mit Containment. Die Mengen Q und C sind Teilmengen von E. Diese Teilmengenbeziehungen oder genauer gesagt die Vorkommen von Q beziehungsweise C in E werden im Sketch durch die Funktionen `iC` und `iQ` dargestellt.

Im Folgenden wird die Qualität dieses Sketches anhand von zwei Eigenschaften des Sketches untersucht; nämlich ob die Modellierung stark genug ist um jeden Graphen zu repräsentieren und präzise genug ist um keine unsinningen Instanzen zuzulassen, also solche die keinen Graphen repräsentieren:

1. **Stärke der Modellierung:** Gegeben sei ein untypisierter Graph mit Containment G. Die Frage ist ob es eine Instanz I dieses Sketches gibt, die G repräsentiert. Dies ist möglich, indem man eine Instanz I von G ableitet, die die formalen Objekte und Pfeile des Sketches auf folgende Weise interpretiert:
   - V, E, C, Q durch die Knoten- und Pfeilmengen,
   - src und tar durch die Quell- und Zielknotenfunktionen, und
   - iC und iQ durch die Inklusionsfunktionen $iC_I : C_I \to E_I : c \mapsto c$ und $iQ_I : Q_I \to E_I : q \mapsto q$.

   Zur Veranschaulichung dieser Ableitung sei der Graph in Abb. 32a gegeben. Man könnte diesen Graph als Modellierung eines Programmes mit einer Klasse $\mathsf{C_1}$ verstehen, die die Methoden $\mathsf{m_1}$ und $\mathsf{m_2}$ enthält, wobei $\mathsf{m_1}$ ein Statement $\mathsf{s_1}$ enthält, das $\mathsf{m_2}$ aufruft. Von diesem Graphen lässt sich die Instanz I_1 in Abb. 32b ableiten.



(a) Graph mit Containment G

(b) Von G abgeleitete Instanz I_1

Abbildung 32: Ableitung einer Instanz

2. **Präzision der Modellierung:** Gegeben sei eine Instanz I des Sketches. Hier ist die Frage, ob es einen Graph G gibt, den I repräsentiert. Damit der Sketch ausreichend präzise ist, müsste es immer einen solchen Graphen geben. Eine Eigenschaft von Graphen mit Containment ist, dass dort E immer die disjunkte Vereinigung von C und Q ist. Dementsprechend müsste E_I immer die disjunkte Vereinigung der Mengen C_I und Q_I oder zumindest ein Koprodukt von C_I und Q_I sein. Da der gegebene Sketch diese Eigenschaft aber nicht berücksichtigt, ist dies nicht immer der Fall:

   Das Gegenbeispiel in Abb. 33 zeigt eine Instanz I_2. des linearen Sketches L. Es handelt sich um eine Instanz, weil man die Interpretationen das formale Objekte und Pfeile beliebig wählen kann, da L keine kommutativen Diagramme enthält. Sie repräsentiert aber aus drei Gründen keinen gültigen Graphen mit Containment: Der Pfeil f ist gleichzeitig ein Containment- und ein Querpfeil, der Pfeil g ist keines von beiden und dem Pfeil e entsprechen gleich zwei Containmentpfeile $\mathsf{e_1}$ und $\mathsf{e_2}$. Aus ebendiesen Gründen ist auch E_{I_2} kein Koprodukt von C_{I_2} und Q_I.

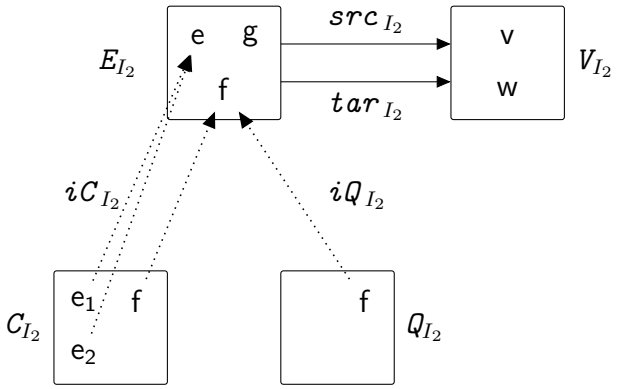


Abbildung 33: Unsinnige Instanz I_2

Der gegebene Sketch lässt also Instanzen I zu, in denen E_I kein Koprodukt von C_I und Q_I ist. Das Problem mit linearen Sketches ist, dass sie auch keine Möglichkeit bieten, derartige Instanzen auszuschließen. Man kann zwar mit kommutativen Diagrammen einzelne kommutative (Ko-)Kegel und Homomorphismen dazwischen ausdrücken; die Existenz eines eindeutigen Homomorphismus kann man aber nicht erzwingen, und deshalb kann man nicht sicherstellen, dass ein bestimmter kommutativer (Ko-)Kegel ein (Ko-)Limes ist. Deshalb sind lineare Sketches nicht geeignet, um Graphen mit Containment zu modellieren. Eine Möglichkeit, dieses Problem zu lösen, folgt in Unterabschnitt 4.2.3.

Desweiteren reichen lineare Sketches zwar aus, um die Kategorie aller typisierten Graphen zu definieren, wie Beispiel 69 gezeigt hat; allerdings lässt sich die Kategorie aller über einem festen Typgraphen TG typisierten Graphen nicht mit einem linearen Sketch modellieren, der von der konkreten Wahl von TG unabhängig ist. Eine Lösung hierfür wird in Unterabschnitt 4.2.4 vorgeschlagen. □

4.2.3. Allgemeine Sketches

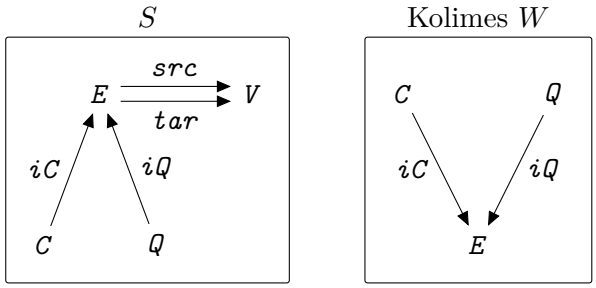
Wie in Bemerkung 77 illustriert wurde, werden disjunkte Vereinigungen für die Modellierung von Containment benötigt. Mit linearen Sketches allein kann man Kolimites nicht modellieren, weil man die Instanzkategorie nicht ausreichend genau einschränken kann. Ähnliches gilt für Limites, die beispielsweise für attributierte Graphen benötigt werden, um Algebren (oder eine eingeschränkte Version von Sketches) mit Sketches zu modellieren.

Allgemeine Sketches bauen deshalb auf linearen Sketches auf, enthalten aber zusätzlich „Schablonen“ für Limites und Kolimites, sogenannte formale Limites und Kolimites. Ein formaler Limes bzw. Kolimes ist lediglich ein Kegel bzw. Kokegel im zugrundeliegenden trivialen Sketch. Eine Instanz des zugrundeliegenden linearen Sketches ist genau dann eine Instanz des allgemeinen Sketches, wenn sie jeden formalen Limes durch einen Limes und jeden formalen Kolimes durch einen Kolimes interpretiert.

Beispiel 78 (Allgemeiner Sketch)

Allgemeiner Sketch

Beispielsweise kann man den Sketch aus Bemerkung 77 zu dem folgenden allgemeinen Sketch erweitern:



Dieser allgemeine Sketch enthält einen formalen Kolimes W über einem binären Koproduktdiagramm (siehe Beispiel 56). Durch diesen formale Kolimes wird gefordert, dass E_I in jeder Instanz I ein Koprodukt von C_I und Q_I mit Injektionen iC_I und tiQ_I ist. Da die Vereinigung zweier disjunkter Mengen zusammen mit den Inklusionsfunktionen immer ein Koprodukt ist, ist die Instanz I_1 aus Bemerkung 77 auch eine Instanz dieses Sketches. Die unsinnige Instanz I_2 wird hingegen ausgeschlossen, weil es sich bei E_{I_2} nicht wie gefordert um ein Koprodukt von C_{I_2} und Q_{I_2} handelt.

Es sei noch darauf hingewiesen, das damit nicht sämtliche Instanzen ausgeschlossen sind, die keinen Graphen mit Containment repräsentieren. Hierfür müssen nämlich zusätzlich noch die Containmentpfeile einen Wald bilden, was durch den obigen Sketch noch nicht sichergestellt wird. Ein allgemeiner Sketch, der dies ebenfalls erreicht, wird in Abschnitt 4.3.3 vorgestellt. □

Formal lässt sich ein allgemeiner Sketch folgendermaßen definieren:

Definition 79 (Allgemeiner Sketch)

Allgemeiner Sketch

Sei $L : K \to S$ ein linearer Sketch über dem trivialen Sketch S.
Sei $V = (V_j)_{j \in \widehat{J}}$ eine durch eine Menge $\widehat{J}$ indizierte Familie von Kegeln.
Sei dabei $V_j = (\widehat{P}_j, \widehat{D}_j, p_j)$ für $j \in \widehat{J}$ ein Kegel mit Scheitel $\widehat{P}_j \in S^0$ über einem kleinen Diagramm $\widehat{D}_j : \widehat{G}_j \to S$ und mit Projektionen $p_{j,v} : X \to \widehat{D}_j(v)$ für $v \in \widehat{G}_j^0$.
Sei $W = (W_j)_{j \in \check{J}}$ eine durch eine Menge $\check{J}$ indizierte Familie von Kokegeln.
Sei dabei $W_j = (\check{D}_j, \check{P}_j, i_j)$ für $j \in \check{J}$ ein Kokegel mit Scheitel $\check{P}_j \in S^0$ über einem kleinen Diagramm $\check{D}_j : \check{G}_j \to S$ und mit Injektionen $i_{j,v} : \check{D}_j(v) \to X$ für $v \in \check{G}_j^0$ und $j \in \check{J}$.

Dann ist $\mathcal{S} = (L, V, W)$ ein allgemeiner Sketch über L mit Limites V und Kolimites W.

Definiere also die Klasse der allgemeinen Sketches als:

$$\mathfrak{Sk} = \bigcup_{\substack{K,S \in \mathfrak{Graphs} \\ \widehat{J},\check{J} \in \mathfrak{Set}}} \left\{ (L,V,W) \in S^K \times \Big(\bigcup_{j \in \widehat{J}} \mathfrak{Cone}(\widehat{G}_j, S)\Big)^{\widehat{J}} \times \Big(\bigcup_{j \in \check{J}} \mathfrak{CoCone}(\check{G}_j, S)\Big)^{\check{J}} \;\middle|\; \forall j \in \widehat{J}\colon V_j \in \mathfrak{Cone}(\widehat{G}_j, S) \wedge \forall j \in \check{J}\colon W_j \in \mathfrak{CoCone}(\check{G}_j, S) \right\}$$

□

Instanzen sind dann folgendermaßen zu formalisieren:

Definition 80 (Instanz eines allgemeinen Sketches)

Instanz eines allgemeinen Sketches

Sei $\mathcal{S} = (L, V, W)$ ein allgemeiner Sketch über dem linearen Sketch L mit Limites V und Kolimites W wie in Definition 79.
Sei $I : L \to \mathbf{C}$ eine Instanz von L in einer Kategorie $\mathbf{C}$.
Definiere die Interpretation der Projektion p_j als $(p_j)_I := ((p_{j,v})_I)_{v \in V}$ für $j \in \widehat{J}$.
Definiere die Interpretation der Injektion i_j als $(i_j)_I := ((i_{j,v})_I)_{v \in V}$ für $j \in \breve{J}$.

Dann ist I genau dann eine Instanz von $\mathcal{S}$, wenn die Interpretation jedes formalen Limes V_j für $j \in \widehat{J}$ ein Limes und die Interpretation jedes formalen Kolimes W_j für $j \in \breve{J}$ ein Kolimes ist, oder formal

$$\forall j \in j_V\colon (p_j : (\widehat{P}_j)_I \to \widehat{D}_j \bullet I) \wedge \forall X \in \mathbf{C}^0\colon \forall (p' : X \to \widehat{D}_j \bullet I)\colon \exists! u\colon (u : p' \to (p_j)_I)$$

und

$$\forall j \in j_W\colon (i_j : \breve{D}_j \bullet I \to (\breve{P}_j)_I) \wedge \forall X \in \mathbf{C}^0\colon \forall (i' : \breve{D}_j \bullet I \to X)\colon \exists! u\colon (u : (i_j)_I \to i')$$

Die Klasse der Instanzen von $\mathcal{S}$ wird auch hier als $\mathbf{C}^{\mathcal{S}}$ bezeichnet. □

Wenn $j \in \widehat{J}$ der Index eines formalen Limes des Sketches ist, besitzt die Instanz I einen Limes über dem Diagramm $\widehat{D}_j \bullet I : \widehat{G}_j \to \mathbf{C}$. Dieser Limes hat als Scheitel das Objekt $(\widehat{P}_j)_I \in \mathbf{C}^0$ und für jeden Knoten $v \in \widehat{G}_j$ aus dem Formgraphen des Diagrammes die v-Projektion $(p_{j,v})_I : (\widehat{P}_j)_I \to (\widehat{D}_j(v))_I$ wie oben definiert. Analoges gilt für die Kolimites.

Mit formalen Limites und Kolimites in Sketches kann man die selben Arten von Limites und Kolimites bilden wie sie in Unterabschnittes 4.1.3 beschrieben wurden. Hierzu benutzt man die Formgraphen aus den dortigen Beispielen von Limites und Kolimites. Beispielsweise erhält man einen Equalizer indem man einen formalen Limes V_j über einem Diagramm mit Formgraphen $\widehat{G}_j = \mathrm{EQL}$ hinzufügt (siehe Beispiel 59).

Die Instanzkategorie eines allgemeinen Sketches definiert sich ähnlich wie bei trivialen und linearen Sketches:

Definition 81 (Kategorie der Instanzen eines allgemeinen Sketches)

Kategorie der Instanzen eines allgemeinen Sketches

Sei $\mathbf{C}$ eine Kategorie.
Sei $\mathcal{S}$ ein allgemeiner Sketch.

Dann ist die Kategorie $\mathbf{I}_{\mathbf{C}}(\mathcal{S})$ der Instanzen von $\mathcal{S}$ in C folgendermaßen definiert:

- Die Objekte sind die Instanzen von $\mathcal{S}$ in C: $(\mathbf{I}_{\mathbf{C}}(\mathcal{S}))^0 = \mathbf{C}^{\mathcal{S}}$
- Die Pfeile zwischen zwei Instanzen sind die natürlichen Transformationen zwischen ihnen: $(\mathbf{I}_{\mathbf{C}}(\mathcal{S}))[I, J] = \mathfrak{NatTrans}(I, J)$
- Die Komposition natürlicher Transformationen $\alpha : I \to J$ und $\beta : J \to P$ ist ihre komponentenweise Komposition $\alpha \bullet \beta : I \to P$ mit $(\alpha \bullet \beta)_A = \alpha_A \bullet \beta_A$ für $A \in S^0$.

Falls $\mathbf{C} = \mathbf{Set}$ ist, wird anstatt $\mathbf{I}_{\mathbf{C}}(\mathcal{S})$ auch $\mathbf{I}(\mathcal{S})$ geschrieben. □

Etwas komplizierter ist der Homomorphismusbegriff für allgemeine Sketches. Dies liegt daran, dass man mit ihm die formalen Limites und Kolimites zweier allgemeiner Sketches aufeinander abbilden können muss. Da jeder allgemeine Sketch beliebige Indexmengen für diese hat, hat er auch beliebig viele formale Limites und Kolimites. Im Gegensatz zu kommutativen Diagrammen kann man aber mehrere formale Limites nicht zu einem formalen Limes kombinieren, und ebenso wenig formale Kolimites. Da für jeden formalen Limes und Kolimes jeweils eine Komponente für den Formgraphen des Diagrammes nötig ist, kann deshalb ein Homomorphismus allgemeiner Sketches beliebig viele Komponenten haben.

Definition 82 (Homomorphismus allgemeiner Sketches)

Homomorphismus allgemeiner Sketches

Seien $\mathcal{S}_1, \mathcal{S}_2$ allgemeine Sketches mit Bezeichnungen wie in Definition 79.
Sei $h_S : S_1 \to S_2$ ein Homomorphismus der zugrundeliegenden trivialen Sketches.
Sei $h_K : K_1 \to K_2$ ein Homomorphismus der Formgraphen der zugrundeliegenden linearen Sketches.
Sei $h_{\hat{J}} : \hat{J}_1 \to \hat{J}_1$ eine Funktion zwischen den Indexmengen der formalen Limites.
Sei $h_{\check{J}} : \check{J}_2 \to \check{J}_2$ dasselbe für die formalen Kolimites.
Sei $h_{\hat{G}} = (h_{\hat{G},j})_{j \in \hat{J}_1}$ eine Familie von Homomorphismen der Formgraphen der formalen Limites, d.h. $h_{\hat{G},j} : \hat{G}_{1,j} \to \hat{G}_{2,h_{\hat{J}}(j)}$ für $j \in \hat{J}_1$.
Sei $h_{\check{G}} = (h_{\check{G},j})_{j \in \check{J}_1}$ eine Familie von Homomorphismen der Formgraphen der formalen Kolimites, d.h. $h_{\check{G},j} : \check{G}_{1,j} \to \check{G}_{2,h_{\hat{J}}(j)}$ für $j \in \check{J}_1$.

Dann ist $h = (\mathcal{S}_1, \mathcal{S}_2, h_S, h_K, h_{\hat{J}}, h_{\check{J}}, h_{\hat{G}}, h_{\check{G}})$ genau dann ein Homomorphismus allgemeiner Sketches, wenn gilt:

- h_K und h_S sind miteinander verträglich und sind somit die Komponenten eines Homomorphismus h_L der zugrundeliegenden linearen Sketches:

$$L_1 \bullet h_S = h_K \bullet L_2$$

- Die Formgraphen der formalen Limites sind isomorph:

$$\forall j \in \hat{J}_1 \colon \exists (r : \hat{G}_{2,h_{\hat{J}}(j)} \to \hat{G}_{1,j}) \colon h_{\hat{G},j} \bullet r = id_{\hat{G}_{2,h_{\hat{J}}(j)}} \wedge r \bullet h_{\hat{G},j} = id_{\hat{G}_{1,j}}$$

- h_S ist mit jedem formalen Limes, d.h. mit den Diagrammen, den Scheiteln und den Komponenten verträglich:
  - $\forall j \in \hat{J}_1 \colon \hat{D}_{1,j} \bullet h_S = h_{\hat{G},j} \bullet \hat{D}_{2,h_{\hat{J}}(j)}$
  - $\forall j \in \hat{J}_1 \colon h_S(\hat{P}_{1,j}) = \hat{P}_{2,h_{\hat{J}}(j)}$
  - $\forall j \in \hat{J}_1 \colon \forall v \in \hat{G}^0_{1,j} \colon h_S(p_{1,j,v}) = p_{2,h_{\hat{J}}(j),h_{\hat{G},j}(v)}$
- Die Formgraphen von formalen Kolimites sind isomorph:

$$\forall j \in \check{J}_1 \colon \exists (r : \check{G}_{2,h_{\check{J}}(j)}) \to \check{G}_{1,j} \colon h_{\check{G},j} \bullet r = id_{\check{G}_{2,h_{\check{J}}(j)}} \wedge r \bullet h_{\check{G},j} = id_{\check{G}_{1,j}}$$

- h_S ist mit jedem formalen Kolimes, d.h. mit den Diagrammen, den Scheiteln und den Komponenten verträglich:

- $\forall j \in \check{J}_1$: $\check{D}_{1,j} \bullet h_S = h_{\check{G},j} \bullet \check{D}_{2,h_{\check{J}}(j)}$
- $\forall j \in \check{J}_1$: $h_S(\check{P}_{1,j}) = \check{P}_{2,h_{\check{J}}(j)}$
- $\forall j \in \check{J}_1$: $\forall v \in \check{G}^0_{1,j}$: $h_S(p_{1,j,v}) = p_{2,h_{\check{J}}(j),h_{\check{G},j}(v)}$

Die Klasse aller Homomorphismen zwischen von $\mathcal{S}_1$ nach $\mathcal{S}_2$ wird mit ${\mathcal{S}_2}^{\mathcal{S}_1}$ bezeichnet. □

Auch bei allgemeinen Sketches ist die Einschränkung auf Teilsketches ein wichtiger Einsatzzweck von Homomorphismen, der für typisierte Sketches relevant sein wird:

Definition 83 (Einschränkung eines allgemeinen Skeches)

Einschränkung eines allgemeinen Skeches

Seien $\mathcal{S}_1, \mathcal{S}_2$ allgemeine Sketches mit Bezeichnungen wie in Definition 79.
Sei $h : \mathcal{S}_1 \to \mathcal{S}_2$ ein Homomorphismus allgemeiner Sketches mit Bezeichnungen wie in Definition 82.
Sei $\mathbf{C}$ eine Kategorie.
Sei $I : \mathcal{S}_2 \to \mathbf{C}$ eine Instanz von $\mathcal{S}_2$.
Sei h_{LI} die Einschränkung Einschränkung des $\mathcal{S}_2$ zugrundeliegenden linearen Sketches $L_2 : K_2 \to S_2$ mittels des h zugrundeliegenden Homomorphismus linearer Skeches h_L.

Dann ist die Einschränkung von I auf $\mathcal{S}_1$ mittels h dasselbe wie die Einschränkung von I auf S_1 mittels h_L:

$$h_I \coloneqq (h_L)_I$$

□

Diese Definition ist möglich, da eine Instanz von $\mathcal{S}_2$ immer auch eine Instanz des zugrundeliegenden linearen Sketches S_2 ist. Allerdings ist noch zu beweisen, dass diese Instanz von S_1 auch eine Instanz von $\mathcal{S}_1$ ist, also dass die formalen Limites als Limites und die formalen Kolimites als Kolimites interpretiert werden. Dieser Beweis findet sich im Anhang auf Seite 157.

Definition 84 (Kategorie der allgemeinen Sketches)

Kategorie der allgemeinen Sketches

Die **Kategorie der allgemeinen Sketches** heißt **Sk** und ist folgendermaßen definiert:

- Die Objekte sind die allgemeinen Sketches: $\mathbf{Sk}^0 = \mathfrak{Sk}$
- Die Pfeile zwischen zwei allgemeinen Sketches $L_1 : K_1 \to S_1$ und $L_2 : K_2 \to S_2$ sind die Homomorphismen linearer Sketches: $\mathbf{Sk}[\mathcal{S}_1, \mathcal{S}_2] = {\mathcal{S}_2}^{\mathcal{S}_1}$
- Die Komposition zweier Homomorphismen $h = (\mathcal{S}_1, \mathcal{S}_2, h_S, h_K, h_{\hat{J}}, h_{\check{J}}, h_{\hat{G}}, h_{\check{G}})$ und $g = (\mathcal{S}_2, \mathcal{S}_3, g_S, g_K, g_{\hat{J}}, g_{\check{J}}, g_{\hat{G}}, g_{\check{G}})$ ist komponentenweise definiert:
  $h \bullet g = (\mathcal{S}_1, \mathcal{S}_3, h_S \bullet g_S, h_K \bullet g_K, h_{\hat{J}} \bullet g_{\hat{J}}, h_{\check{J}} \bullet g_{\check{J}}, h_{\hat{G}} \bullet g_{\hat{G}}, h_{\check{G}} \bullet g_{\check{G}})$ □

Bemerkung 85 (Ausdruckskraft allgemeiner Sketches)

Ausdruckskraft allgemeiner Sketches

Mit allgemeinen Sketches lassen sich sämtliche vorgesehenen Graphvarianten ausdrücken.

Insbesondere lässt sich auch die Kategorie der über einem Typgraph TG typisierten Graphen durch einen allgemeinen Sketch auf Grundlage des linearen Sketches **TGS** der typisieren Graphen ausdrücken. Hierzu muss man **TGS** so erweitern, dass der Typgraph, also die Mengen `tV` und `tE` und die Funktionen `tSrc` und `tTar` bis auf Isomorphie eindeutig festgelegt sind.

Dies ist möglich dank der Tatsache, dass jede Menge M die Vereinigung ihrer (natürlich paarweise disjunkten) einelementigen Teilmengen ist. In **Set** kann man Vereinigungen disjunkter Mengen als Koprodukte (siehe Beispiel 56) und einelementige Mengen durch ein terminales Objekt[8] (siehe Beispiel 55) modellieren. Das terminale Objekt wird dabei mit einem formalen Limes über dem leeren Diagramm spezifiziert (siehe Beispiel 55). Es ergibt sich der folgende Sketch:

Daher kann man M in einem allgemeinen Sketch als formales Koprodukt über einem Diagramm D mit einem diskreten Formgraphen mit Knotenmenge M, Injektionen $\mathit{i}[\mathit{a}]$ für $a \in M$ und einem Scheitel M modellieren. Dieses Diagramm wählt für jeden dieser Knoten ein terminales Objekt $\mathit{1}$ aus, das mit einem formalen Limes über dem leeren Diagramm spezifiziert wird (siehe Beispiel 55). In einer Instanz I in **Set** wird $\mathit{1}$ immer als eine einelementige Menge interpretiert, deren (einziges) Element man oft mit $*$ bezeichnet, also $\mathit{1}_I = \{*\}$. Dadurch besteht M_I in jeder Instanz I des Sketches in **Set** aus Kopien der einelementigen Menge $\mathit{1}_I$, und zwar genau einer für jedes Element $a \in M$. Eine solche Kopie wird hier Repräsentation von a in M genannt. Eine von $\mathit{1}_I$ ausgehende Funktion wählt gewissermaßen ein Element aus einer anderen Menge aus, indem sie es dem Element $*$ als Wert zuordnet. In der beschriebenen Konstruktion wählt also die Injektion $\mathit{i}[\mathit{a}]$ das Element $\mathit{i}[\mathit{a}](*) \in \mathit{M}_I$ als Repräsentation von a in M aus. Wie man dies verwendet, um einen Typgraphen festzulegen, wird im Folgenden gezeigt.

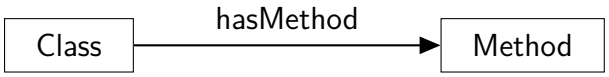


Abbildung 34: Typgraph TG

Sei beispielsweise der Typgraph TG aus Abb. 34 gegeben. Um die über TG typisierten Graphen zu modellieren kann man den Sketches **TGS** aus Beispiel 69 wie oben beschrieben erweitern. Man erhält den allgemeinen Sketch $\mathbf{TGS}_{TG}$ aus Abb. 35.

Der zugrundeliegende triviale Sketch wird im Vergleich zu **TGS** um ein formales Objekt $\mathit{1}$ erweitert, das für ein terminales Objekt steht. Dass es sich dabei tatsächlich um ein terminales Objekt handelt, wird mit einem formalen Limes über dem leeren Diagramm mit Scheitel $\mathit{1}$ sichergestellt. Außerdem werden formale Pfeile $\mathit{itV}[\mathsf{Class}]$, $\mathit{itV}[\mathsf{Method}]$ und $\mathit{itE}[\mathsf{hasMethod}]$ hinzugefügt. Diese sind die Injektionen zweier formaler Kolimites, die wie oben beschrieben die Mengen der Knoten und Pfeile festlegen. Mit zwei zusätzlichen kommutativen Diagrammen werden der Quell- und Zielknotentyp des formalen Pfeils hasMethod spezifiziert. Dabei wird die oben beschriebene Tatsache ausgenutzt, dass die Funktionen $\mathit{itV}[\mathsf{Class}]$, $\mathit{itV}[\mathsf{Method}]$ die Repräsentationen der Knotentypen Class und Method und mit $\mathit{itE}[\mathsf{hasMethod}]$ die Repräsentation des Pfeiltyps Method auswählen. Man kann die kommutativen Diagramme als Gleichungen lesen: $\mathit{itE}[\mathsf{hasMethod}]_I \bullet \mathit{tSrc}_I = \mathit{itV}[\mathsf{Class}]_I$ und $\mathit{itE}[\mathsf{hasMethod}]_I \bullet \mathit{tTar}_I = \mathit{itV}[\mathsf{Method}]_I$, d.h. die Quellknotentyp von hasMethod ist Class und der Zielknotentyp ist Method.

Der Nachteil dieser Herangehensweise ist, dass der konstruierte Sketch vom jeweils verwendeten Typgraphen abhängig ist. Das heißt, dass sich für jeden Typgraphen ein eigener Sketch ergibt. Es wäre aber wünschenswert, einen einzigen Sketch für beliebige

[8] Da einelementige Mengen/terminale Objekte zueinander isomorph und daher aus kategorientheoretischer Sicht im Wesentlichen gleich sind, reicht hierbei ein einziges terminales Objekt.

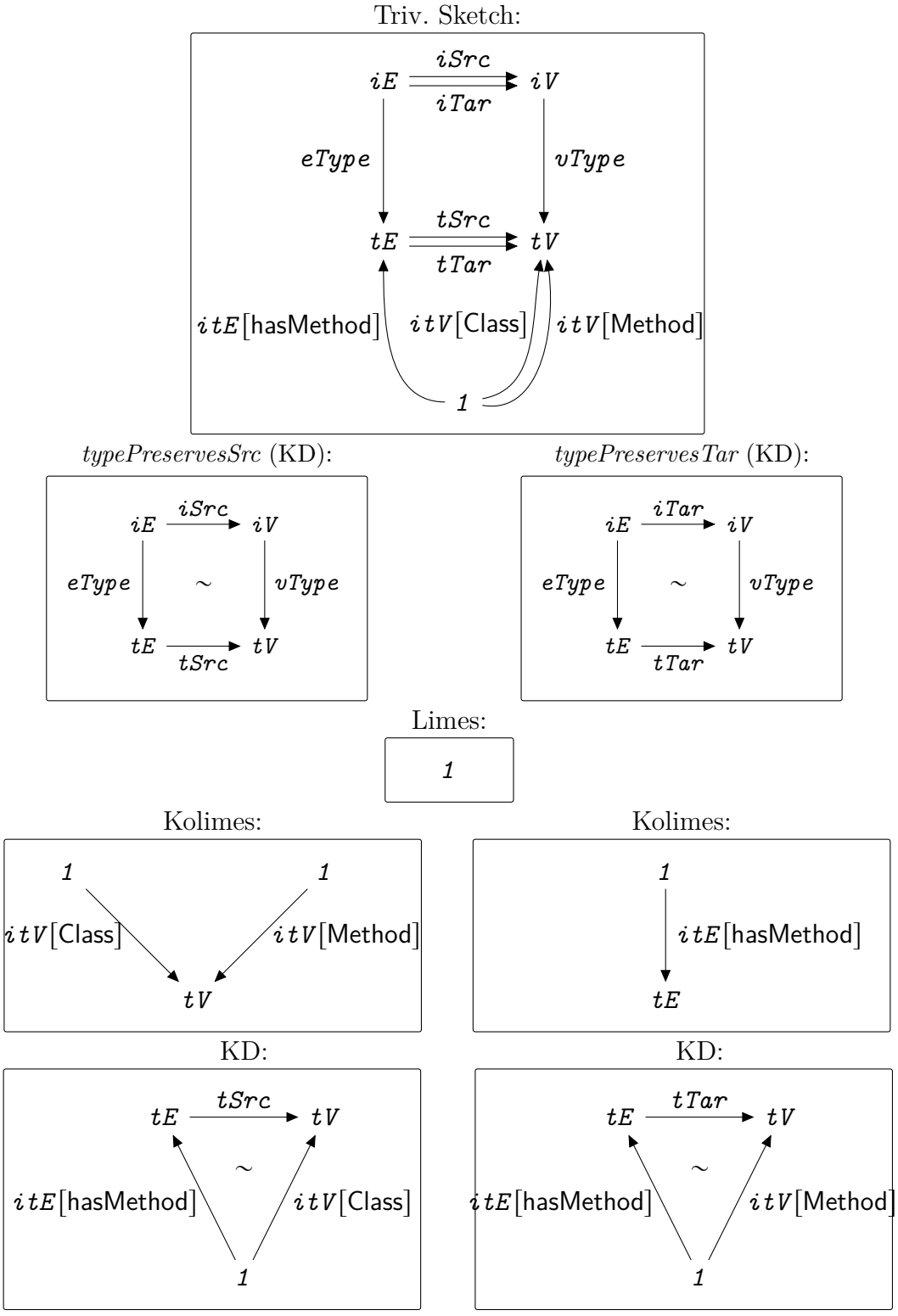
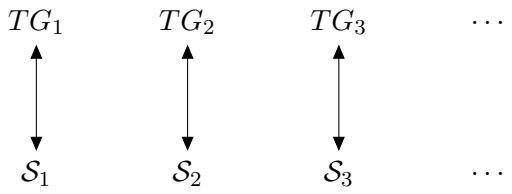


Abbildung 35: Sketch $\mathbf{TGS}_{TG}$ der über dem Typgraphen TG aus Abb. 34 typisierten Graphen

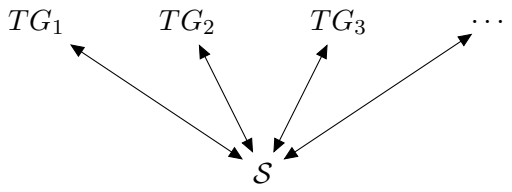
Typgraphen zu haben. Zur Auflösung diese Abhängigkeit werden in Unterabschnitt 4.2.4 typisierte Sketches vorgeschlagen. □

4.2.4. Typisierte Sketches

In Bemerkung 85 im vorhergehenden Unterabschnitt wurde festgestellt, dass man mit allgemeinen Sketches zwar typisierte Graphen über einem festen Typgraphen ausdrücken kann, dabei aber der Sketch vom jeweiligen Typgraphen abhängt. Das heißt, dass man für jeden Typgraphen einen eigenen Sketch konstruieren muss:



Im folgenden werden typisierte Sketches als eine Möglichkeit vorgeschlagen, diese Abhängigkeit aufzulösen. Mit diesen benötigt man nur einen einzigen Sketch, mit dem man beliebige Typgraphen erfassen kann:



Einem typisierten Sketch liegt ein allgemeiner Sketch zugrunde. Darüber hinaus enthält er einen Teilsketch dieses zugrundeliegenden Sketches, das sogenannte **Typfragment**. Dieses Typfragment ist ebenfalls ein allgemeiner Sketch und entspricht im Fall der typisierten Graphen dem Typgraphen. Die Idee ist, dass man eine Instanz des Typfragmentes, eine sogenannte **Typinstanz**, im Vorhinein festlegt. Man betrachtet nur solche Instanzen des allgemeinen Sketches als Objekte der Instanzkategorie, die das Typfragment auf dieselbe Weise interpretieren wie diese Typinstanz. Auch die Menge der Morphismen wird auf diejenigen natürlichen Transformationen eingeschränkt, die die Interpretation des Typfragmentes unverändert belassen.

Das folgende Beispiel erläutert diese Vorgehensweise:

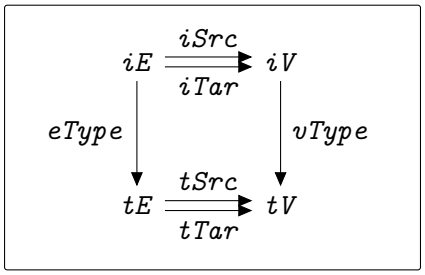
Beispiel 86 (Typisierter Sketch der typisierten Graphen)

Typisierter Sketch der typisierten Graphen

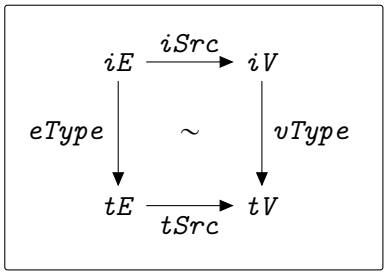
Beispielsweise kann man auf Grundlage des linearen Sketches **TGS** der typisierten Graphen aus Unterabschnitt 4.2.2 einen typisierten Sketch TS definieren, der die über einem beliebigen Typgraphen TG typisierten Graphen modelliert. Hierbei nutzt man die Tatsache aus, dass der Teilsketch $\mathtt{tE} \underset{\mathtt{tTar}}{\overset{\mathtt{tSrc}}{\rightrightarrows}} \mathtt{tV}$ von **TGS** dem Typgraphen entspricht. Diesen kann man als Typfragment T von TS benutzen:

Zugrundeliegender Sketch:

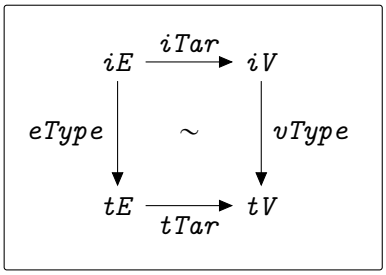
TGS$_{triv}$ (triv. Sketch):



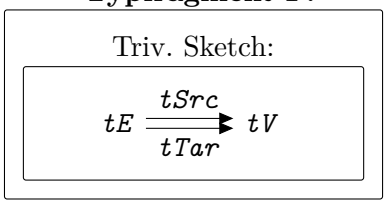
typePreservesSrc (KD):



typePreservesTar (KD):



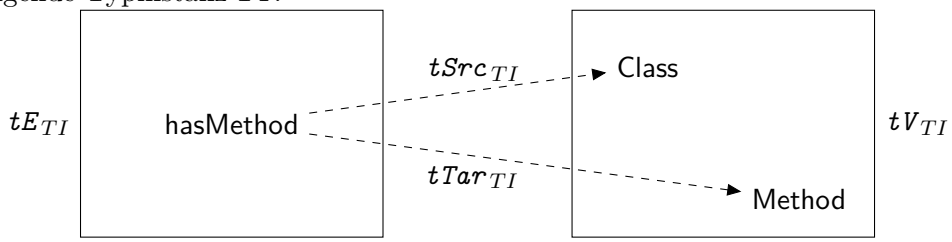
Typfragment T:



Der zugrundeliegende Sketch entspricht dem linearen Sketch **TGS** der typisierten Graphen. Sowohl im zugrundeliegenden Sketch als auch im Typfragment steht *tV* für die Knoten- und *tE* für die Pfeiltypen, und *tSrc* und *tTar* ordnen jedem Pfeiltyp ihren Quell- bzw. Zielknotentyp zu.

Im Allgemeinen könnten im zugrundeliegenden Sketch beliebige kommutative Diagramme und formale Limites und Kolimites enthalten sein, und ein Teil davon kann sich auch im Typfragment wiederfinden. Im vorliegenden Fall sind aber keine formalen Limites und Kolimites und im Typfragment auch keine kommutativen Diagramme notwendig und werden deshalb weggelassen.

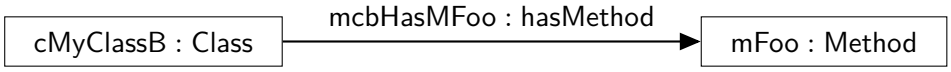
Um die über einem bestimmten Typgraphen typisierten Graphen zu betrachten, muss man nun eine Typinstanz festlegen. Im Fall des Typgraphen TG aus Abb. 34 ergibt sich die folgende Typinstanz TI:



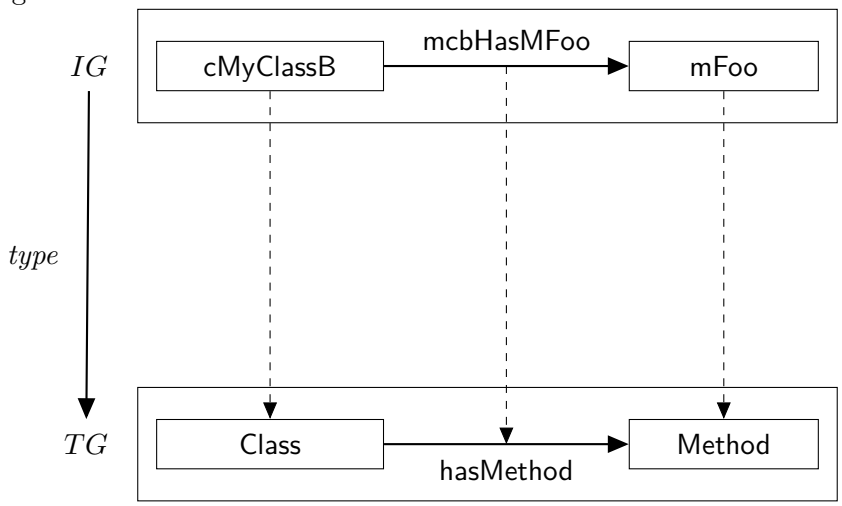
Dabei ist $tSrc_{TI}$(hasMethod) = Class und $tTar_{TI}$(hasMethod) = Method.

Die über TG typisierten Graphen sind nun diejenigen Instanzen des zugrundeliegenden Sketches, die auf dem Typfragment mit der Typinstanz übereinstimmen. Sei beispiels-

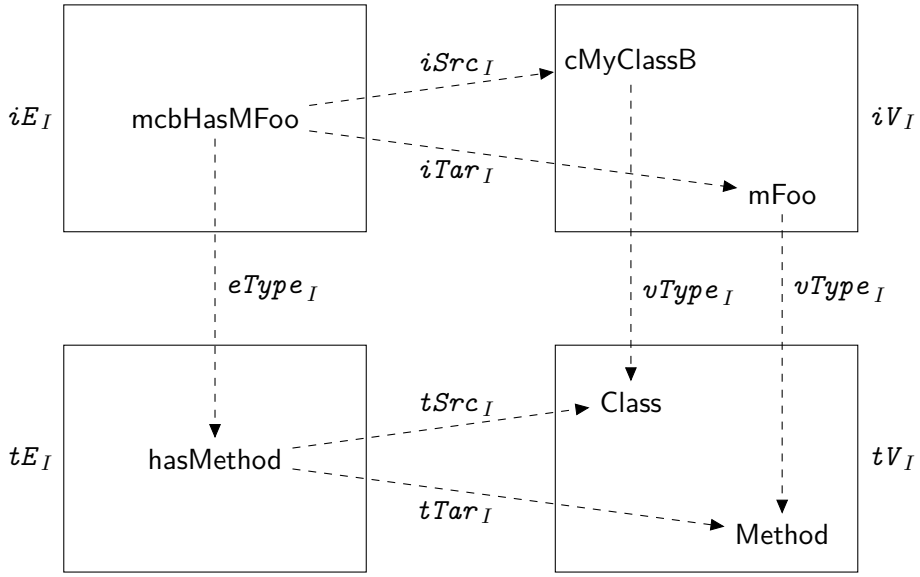
weise der folgende über TG typisierte Graph gegeben:



Wenn man Instanz- und Typgraph wie in Unterabschnitt 3.2.2 explizit macht, ergibt sich dabei die folgende Konstellation:



Als Instanz von **TGS** wird dieser typisierte Graph folgendermaßen repräsentiert:



Wenn man das Typfragment in dieser Instanz – also die unteren beiden Mengen tE_I und tV_I und die Funktionen dazwischen – mit der Typinstanz vergleicht, stellt man fest, dass sie übereinstimmen. Deshalb ist diese Instanz eine Instanz des typisierten Graphen TS über der Typinstanz TI.

Man könnte dieselben Schritte auch auf einen anderen Typgraphen TG' anwenden. Der wesentliche Unterschied zur Konstruktion aus 85 ist, dass sich der Sketch TS unverändert bleibt – es wird lediglich eine andere Typinstanz TI' benutzt. □

Formal sind typisierte Sketches folgendermaßen definiert:

Definition 87 (Typisierter Sketch)

Typisierter Sketch

Sei $\mathcal{S} = (L, V, W)$ ein allgemeiner Sketch.
Sei $\mathcal{TF} = (TL, TV, TW)$ ein Teilsketch von $\mathcal{S}$, modelliert durch einen injektiven Homomorphismus $\mathcal{TS} : \mathcal{TF} \hookrightarrow \mathcal{S}$ allgemeiner Sketches (Typfragment). Injektiv heißt dabei, dass alle Komponenten injektiv sind.

Dann ist $\mathcal{TS}$ ein **typisierter Sketch** zu $\mathcal{S}$ mit Typfragment $\mathcal{TF}$.

$$\mathcal{TF} \xhookrightarrow{\mathcal{TS}} \mathcal{S}$$

□

Ein typisierter Sketch wird in dieser Definition gleichgesetzt mit einem injektiven Homomorphismus typisierter Sketches. Zu beachten ist, dass nicht verlangt wird, dass die Menge der formalen Objekt bzw. Pfeile von $\mathcal{TF}$ eine Teilmenge der Objekt- bzw. Pfeilmenge von $\mathcal{S}$ sind – da $\mathcal{TS}$ injektiv ist, sind sie aber isomorph zu einer Teilmenge der Objekt- bzw. Pfeilmenge von $\mathcal{S}$.

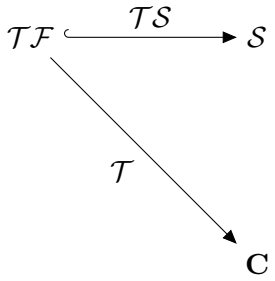
Eine Typinstanz ist eine beliebige Interpretation des Typfragmentes:

Definition 88 (Typinstanz eines typisierten Sketches)

Typinstanz eines typisierten Sketches

Sei $\mathcal{TS} : \mathcal{TF} \hookrightarrow \mathcal{S}$ ein typisierter Sketch zu S mit Typfragment $\mathcal{TF}$.
Sei $\mathbf{C}$ eine Kategorie.
Sei $\mathcal{T} : \mathcal{TF} \to \mathbf{C}$ eine Instanz des Typfragmentes $\mathcal{TF}$ in $\mathbf{C}$.

Dann ist $\mathcal{T}$ eine **Typinstanz** von $\mathcal{TS}$ in $\mathbf{C}$.



□

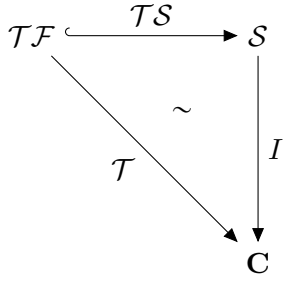
Instanzen eines typisierten Sketches sind folgendermaßen definiert:

Definition 89 (Instanz eines typisierten Sketches)

Instanz eines typisierten Sketches

Sei $\mathcal{TS} : \mathcal{TF} \hookrightarrow \mathcal{S}$ ein typisierter Sketch zu S mit Typfragment $\mathcal{TF}$.
Sei $\mathbf{C}$ eine Kategorie.
Sei $(\mathcal{TS}, \mathcal{T})$ mit $\mathcal{T} : \mathcal{TF} \to \mathbf{C}$ eine Typinstanz von $\mathcal{TS}$ in $\mathbf{C}$.
Sei $I : \mathcal{S} \to \mathbf{C}$ eine Instanz des zugrundeliegenden Sketches von $\mathcal{TS}$.

Dann ist I genau dann eine über $\mathcal{T}$ typisierte Instanz von $\mathcal{TS}$, wenn ihre Einschränkung auf $\mathcal{TF}$ mit $\mathcal{T}$ übereinstimmt, also wenn gilt: $\mathcal{TS}_I = \mathcal{TS} \bullet I = \mathcal{T}$



Man schreibt hierfür auch $I : (\mathcal{TS}, \mathcal{T}) \to \mathbf{C}$.

Die Klasse aller über $\mathcal{T}$ typisierten Instanzen von $\mathcal{TS}$ in $\mathbf{C}$ heißt $\mathbf{C}^{(\mathcal{TS},\mathcal{T})}$. □

Definition 90 (Katgorie der Instanzen eines typisierten Sketches)

Katgorie der Instanzen eines typisierten Sketches

Sei $\mathcal{TS} : \mathcal{TF} \hookrightarrow \mathcal{S}$ ein typisierter Sketch zu S mit Typfragment $\mathcal{TF}$.
Sei dabei TF der zugrundeliegende triviale Sketch von $\mathcal{TF}$.
Sei $\mathbf{C}$ eine Kategorie.
Sei $\mathcal{T} : \mathcal{TF} \to \mathbf{C}$ eine Typinstanz von $\mathcal{TS}$ in $\mathbf{C}$.

Dann ist die Kategorie $\mathbf{I}_\mathbf{C}((\mathcal{TS}, \mathcal{T}))$ der über $\mathcal{T}$ typisierten Instanzen von $\mathcal{TS}$ in $\mathbf{C}$ folgendermaßen definiert:

- Die Objekte sind über die $\mathcal{T}$ typisierten Instanzen von $\mathcal{TS}$ in $\mathbf{C}$:

$$(\mathbf{I}_\mathbf{C}((\mathcal{TS}, \mathcal{T})))^0 = \mathbf{C}^{(\mathcal{TS},\mathcal{T})}$$

- Die Pfeile zwischen zwei Instanzen sind diejenigen natürlichen Transformationen zwischen ihnen, deren Einschränkung auf das Typfragment die Identität ist:

$$(\mathbf{I}_\mathbf{C}(\mathcal{S}))[I, J] = \left\{\alpha : I \to J \in \mathfrak{NatTrans}(I, J) \;\middle|\; \forall v \in TF^0\text{: } \alpha_{\mathcal{T}(v)} = id_{\mathcal{T}(V)}\right\}$$

- Die Komposition natürlicher Transformationen $\alpha : I \to J$ und $\beta : J \to P$ ist ihre komponentenweise Komposition $\alpha \bullet \beta : I \to P$ mit $(\alpha \bullet \beta)_{\mathtt{A}} = \alpha_{\mathtt{A}} \bullet \beta_{\mathtt{A}}$ für $\mathtt{A} \in S^0$.

Falls $\mathbf{C} = \mathbf{Set}$ ist, wird anstatt $\mathbf{I}_\mathbf{C}(\mathcal{S})$ auch $\mathbf{I}(\mathcal{S})$ geschrieben. □

Auch zwischen typisierten Sketches gibt es Homomorphismen, die ähnlich wie Homomorphismen linearer Sketches definiert sind:

Definition 91 (Homomorphismus typisierter Sketches)

Homomorphismus typisierter Sketches

Seien $\mathcal{TS}_1 : \mathcal{TF}_1 \hookrightarrow \mathcal{S}_1$ und $\mathcal{TS}_2 : \mathcal{TF}_2 \hookrightarrow \mathcal{S}_2$ typisierte Sketches.
Seien $h_{\mathcal{TF}} : \mathcal{TF}_1 \to \mathcal{TF}_2$ und $h_\mathcal{S} : \mathcal{S}_1 \to \mathcal{S}_2$ Homomorphismen allgemeiner Sketches.

Dann ist $h = (\mathcal{TS}_1, \mathcal{TS}_2, h_{\mathcal{TF}}, h_\mathcal{S})$ genau dann ein **Homomorphismus typisierter Sketches**, wenn gilt: $\mathcal{TS}_1 \bullet h_\mathcal{S} = h_{\mathcal{TS}} \bullet \mathcal{TS}_2$.

$$\begin{array}{ccc} \mathcal{TF}_1 & \xrightarrow{\mathcal{TS}_1} & \mathcal{S}_1 \\ \Big\downarrow{\scriptstyle h_{\mathcal{TF}}} & \sim & \Big\downarrow{\scriptstyle h_\mathcal{S}} \\ \mathcal{TF}_2 & \xrightarrow[\mathcal{TS}_2]{} & \mathcal{S}_2 \end{array}$$

Man schreibt hierfür $h : \mathcal{TS}_1 \to \mathcal{TS}_2$.

Die Klasse aller Homomorphismen typisierter Sketches von $\mathcal{TS}_1$ nach $\mathcal{TS}_2$ ist

$$\mathcal{TS}_2^{\mathcal{TS}_1} = \{(\mathcal{TS}_1, \mathcal{TS}_2, h_{\mathcal{TF}}, h_{\mathcal{S}}) \mid \mathcal{TS}_1 \bullet h_{\mathcal{S}} = h_{\mathcal{TF}} \bullet \mathcal{TS}_2\}$$ □

Mit dieser Definition kann man auch die Kategorie der typisierten Sketches definieren, so dass man auch Pushouts typisierter Sketches ausführen kann:

Definition 92 (Kategorie der typisierten Sketches)

Kategorie der typisierten Sketches

Die **Kategorie der typisierten Sketches** heißt **TSk** und ist folgendermaßen definiert:

- Die Objekte sind die typisierten Sketches: $\mathbf{TSk}^0 = \mathfrak{Sk}_{typ}$
- Die Pfeile zwischen zwei typisierten Sketches $\mathcal{TS}_1 : \mathcal{TF}_1 \hookrightarrow \mathcal{S}_1$ und $\mathcal{TS}_2 : \mathcal{TF}_2 \hookrightarrow \mathcal{S}_2$ sind die Homomorphismen typisierten Sketches: $\mathbf{TSk}[\mathcal{TS}_1, \mathcal{TS}_2] = {\mathcal{TS}_2}^{\mathcal{TS}_1}$
- Die Komposition zweier Homomorphismen $h = (\mathcal{TS}_1, \mathcal{TS}_2, h_{\mathcal{TF}}, h_{\mathcal{S}})$ und $g = (\mathcal{TS}_2, \mathcal{TS}_3, g_{\mathcal{TF}}, g_{\mathcal{S}})$ ist komponentenweise definiert: $h \bullet g = (\mathcal{TS}_1, \mathcal{TS}_3, h_{\mathcal{TF}} \bullet g_{\mathcal{TF}}, h_{\mathcal{S}} \bullet g_{\mathcal{S}})$ □

Daneben kann man auch typisierte Sketches einschränken:

Definition 93 (Einschränkung eines typisierten Skeches)

Einschränkung eines typisierten Skeches

Seien $\mathcal{TS}_1 : \mathcal{TF}_1 \hookrightarrow \mathcal{S}_1$ und $\mathcal{TS}_2 : \mathcal{TF}_2 \hookrightarrow \mathcal{S}_2$ typisierte Sketches.
Sei $h : \mathcal{TS}_1 \to \mathcal{TS}_2$ ein Homomorphismus typisierter Sketches.
Sei $\mathbf{C}$ eine Kategorie.
Sei $\mathcal{T} : \mathcal{TF} \to \mathbf{C}$ eine Typinstanz von $\mathcal{TS}$ in $\mathbf{C}$.
Sei $I : (\mathcal{TS}, \mathcal{T}) \to \mathbf{C}$ eine über $\mathcal{T}$ typisierte Instanz von $\mathcal{TS}$.

Dann ist die Einschränkung von I auf $\mathcal{TS}_1$ mittels h dasselbe wie die Einschränkung von I auf $\mathcal{S}_1$ mittels $h_{\mathcal{S}}$:

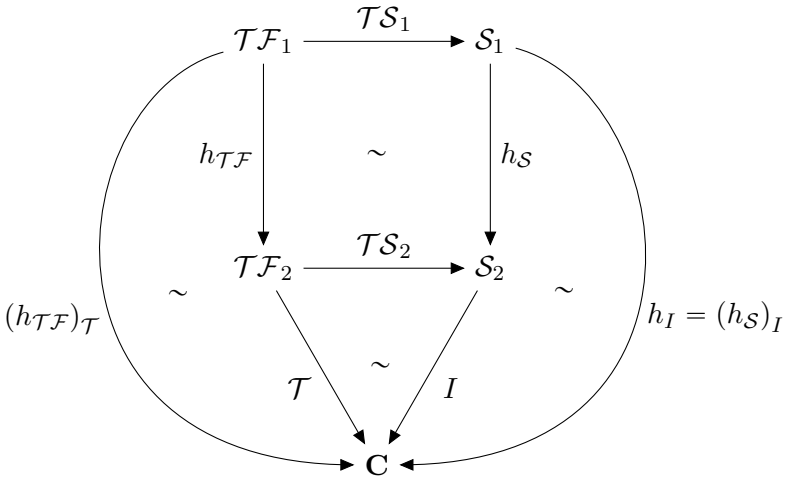
$$h_I \coloneqq (h_{\mathcal{S}})_I$$ □

Es muss noch gezeigt werden, dass die Einschränkung $(h_{\mathcal{S}})_I$ selbst eine Instanz ist. Genauer gesagt ist sie über der Einschränkung $(h_{\mathcal{TF}})_{\mathcal{T}}$ von $\mathcal{T}$ auf $\mathcal{TF}_1$ mittels $h_{\mathcal{TF}}$ typisiert, wie der folgende Beweis zeigt:

Beweis 94 (Die Einschränkung eines typisierten Sketches ist eine Instanz)

Die Einschränkung eines typisierten Sketches ist eine Instanz

Seien $\mathcal{TS}_1$, $\mathcal{TS}_2$, h $\mathbf{C}$, $\mathcal{T}$ und I gegeben wie in Definition 93. Das folgende kommutative Diagramm beweist, dass $\mathcal{TS}_1 \bullet h_I = (h_{\mathcal{TF}})_{\mathcal{T}}$ gilt, was per Definition bedeutet, dass h_I über $(h_{\mathcal{TF}})_{\mathcal{T}}$ typisiert ist.



□

4.3. Graphfeatures und Graphvarianten

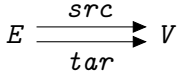
In dieser Arbeit werden Graphvarianten zur Programmrepräsentation untersucht, die als Grundlage für die Beschreibung von graphbasierten Programmmanipulationen und -analysen dienen. Der Ansatz dieser Arbeit ist es, Gemeinsamkeiten zwischen den Graphvarianten als Graphfeatures zu betrachten und diese unabhängig voneinander durch (allgemeine) Sketches umzusetzen. Diese Sketches kann man dann mittels Pushouts in der Kategorie der allgemeinen Sketches verschmelzen und zusammen mit geeigneten Typfragmenten zu Graphvarianten kombinieren.

Im folgenden Abschnitt werden, neben den untypisierten Graphen als Basisfeature, die Graphfeatures Typisierung und Containment formal als Sketches umgesetzt. Neben der Definition der Sketches selbst werden dabei auch Teilsketches davon betrachtet, die als Schnittstellenobjekte für den Pushout dienen können. Abschließend wird anhand eines Beispiels beschrieben, wie man aus mehreren Graphfeatures eine Graphvariante ableitet.

4.3.1. Untypisierte Graphen

Untypisierte Graphen sind ein Graphfeature und gleichzeitig die einfachste Graphvariante, welche als Grundlage für die anderen Graphvarianten verwendet wird. Sie werden durch den Sketch **GS** beschrieben, der bereits aus Beispiel 64 bekannt ist und hier als allgemeiner Sketch betrachtet mit **G** bezeichnet wird:

Trivialer Sketch:

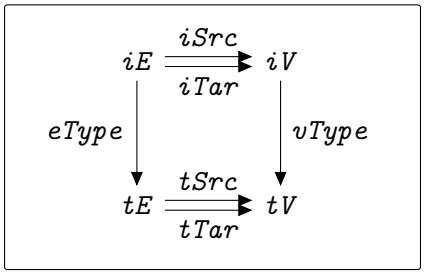


Zusammen mit leeren Mengen von kommutativen Diagrammen und formalen Limites und Kolimites und Typfragment kann man diesen trivialen Sketch auch als allgemeinen Sketch interpretieren. Eine genauere Erläuterung findet sich im oben genannten Beispiel.

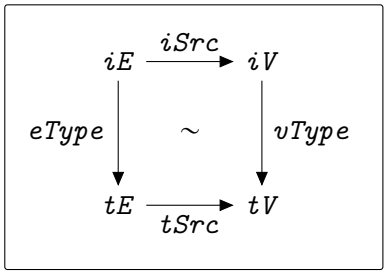
4.3.2. Typisierung

Durch Typisierung wird jedem Knoten und jedem Pfeil ein Knoten- bzw. Pfeiltyp zugeordnet. Wie in Unterabschnitt 3.2.2 beschrieben werden diese Typen bei der Programmrepräsentation benötigt, um die Rollen der repräsentierten Programmelemente auszudrücken. Der zugehörige Sketch **T** ist ein allgemeiner Sketch, der dem linearen Sketch der typisierten Graphen entspricht (siehe Beispiel 69).

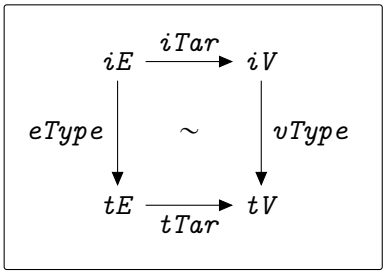
$\mathbf{TGS}_{triv}$ (triv. Sketch):



typePreservesSrc (KD):

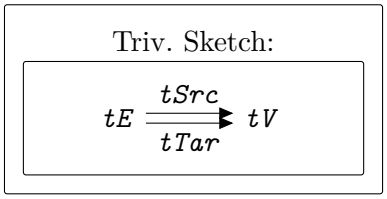


typePreservesTar (KD):

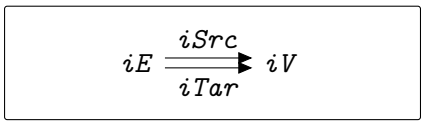


In diesem Sketch sind zwei wichtige Teilsketches enthalten, die man zur Bildung von Pushouts verwenden kann. Der eine modelliert den Typgraphen, der andere den Instanzgraphen. Diese Teilsketches haben beiden die Form des Sketches **G**, da es sich bei Typ- und Instanzgraph um untypisierte Graphen handelt. Sie unterscheiden sich dadurch, dass ihre Vorkommen in **T** durch zwei unterschiedliche Homomorphismen typisierter Skeches repräsentiert werden. Diese Vorkommen kann man ähnlich wie Diagramme illustrieren.

Der Teilsketch, der den Typgraphen modelliert, wurde bereits in Beispiel 86 als Typfragment verwendet und entspricht dem folgenden Vorkommen von **G** in **T**, das mit $TG : \mathbf{G} \to \mathbf{T}$ bezeichnet wird:



Der Instanzgraph entspricht dem folgenden Vorkommen von **G** in **T**, das mit $IG : \mathbf{G} \to \mathbf{T}$ bezeichnet wird:



Die Einschränkung einer Instanz I von $\mathbf{T}$ auf $\mathbf{G}$ anhand des Homomorphismus TG entspricht dem Typgraphen des repräsentierten Graphen. Die gewählten Bezeichnungen und Schreibweisen haben den Nebeneffekt, dass diese Einschränkung mit TG_I bezeichnet wird. Ebenso wird der Instanzgraph mit IG_I bezeichnet.

Ein weiterer Teilsketch ist der Sketch $\mathit{iV} \xrightarrow{\mathit{vType}} \mathit{tV}$, der durch einen Homomorphismus $\Delta[\mathit{vType}] : \mathbf{2} \to \mathbf{T}$ repräsentiert wird, wobei $\mathbf{2}$ ein allgemeiner Sketch ist, der lediglich einen formalen Pfeil $1 \xrightarrow{e} 2$ im zugrundeliegenden trivialen Sketch enthält. Allgemein kann ein Homomorphismus $\Delta[e] : \mathbf{2} \to \mathbf{T}$ dieser Art verwendet werden, um einen formalen Pfeil e aus einem Sketch auszuwählen.

4.3.3. Containment

Containment erlaubt es, die hierarchische Struktur von Programmen explizit im Graph zu repräsentieren. Es teilt sich in zwei wesentliche Gesichtspunkte auf, die man unabhängig voneinander modellieren kann. Der erste Gesichtspunkt ist die Unterscheidung zwischen zwei Arten von Pfeilen, die hier als Containment- und Querpfeile bezeichnet werden. Diese Unterscheidung wurde bereits in Beispiel 78 behandelt. Der zweite Gesichtspunkt ist eine hierarchische, also waldartige Menge von Pfeilen. Die Waldartigkeit besagt, dass kein Knoten Ziel zweier solcher Pfeile ist, und dass es keine Zyklen gibt. Die beiden Gesichtspunkte kann man in separaten Sketches modellieren. Hierdurch kann man bei Bedarf auch eine Graphvariante modellieren, die keine Querpfeile erlaubt.

Der in Abb. 36 illustrierte Sketch $\mathbf{D}$ modelliert eine Unterscheidung zwischen zwei Arten von Pfeilen und erlaubt es gleichzeitig, die Menge aller Pfeile zu betrachten. Dies wurde bereits in Beispiel 78 genauer erklärt. Zu dem dortigen Sketch werden neben dem in $\mathbf{D}$ lediglich noch zwei Pfeilpaare $\mathit{cSrc}/\mathit{cTar}$ und $\mathit{qSrc}/\mathit{qTar}$ hinzugefügt, die den beiden Pfeilarten C und Q unmittelbar ihre Quell- und Zielknoten zuordnen:

In diesem Sketch kommen drei Teilsketches der Form $\mathbf{G}$ vor, die drei Einschränkungen einer Instanz entsprechen, nämlich:

- $\mathit{CG}\colon \mathit{C} \overset{\mathit{cSrc}}{\underset{\mathit{cTar}}{\rightrightarrows}} \mathit{V}$
- $\mathit{QG}\colon \mathit{Q} \overset{\mathit{qSrc}}{\underset{\mathit{qTar}}{\rightrightarrows}} \mathit{V}$
- $\mathit{EG}\colon \mathit{C} \overset{\mathit{src}}{\underset{\mathit{tar}}{\rightrightarrows}} \mathit{V}$

Man kann diese Homomorphismen in Pushouts hernehmen, etwa um den Containmentpfeilen, den Querpfeilen bzw. allen Pfeilen neue Eigenschaften hinzuzufügen. Außerdem kann man damit Instanzen von $\mathbf{D}$ zu untypisierten Graphen einschränken: Eine Einschränkung auf CG enthält nur die Containmentpfeile, eine Einschränkung auf QG nur die Querpfeile und eine Einschränkung auf EG hebt die Unterscheidung zwischen Containment- und Querpfeilen auf.

Wenn nicht gewünscht ist, dass die Menge aller Pfeile modelliert wird, reicht für diesen Zweck auch der einfachere Sketch $\mathbf{D}'$ in Abb. 37 aus. In diesem Sketch kommen nur die folgenden zwei Teilsketches der Form $\mathbf{G}$ vor:

- $\mathit{CG}\colon \mathit{C} \overset{\mathit{cSrc}}{\underset{\mathit{cTar}}{\rightrightarrows}} \mathit{V}$

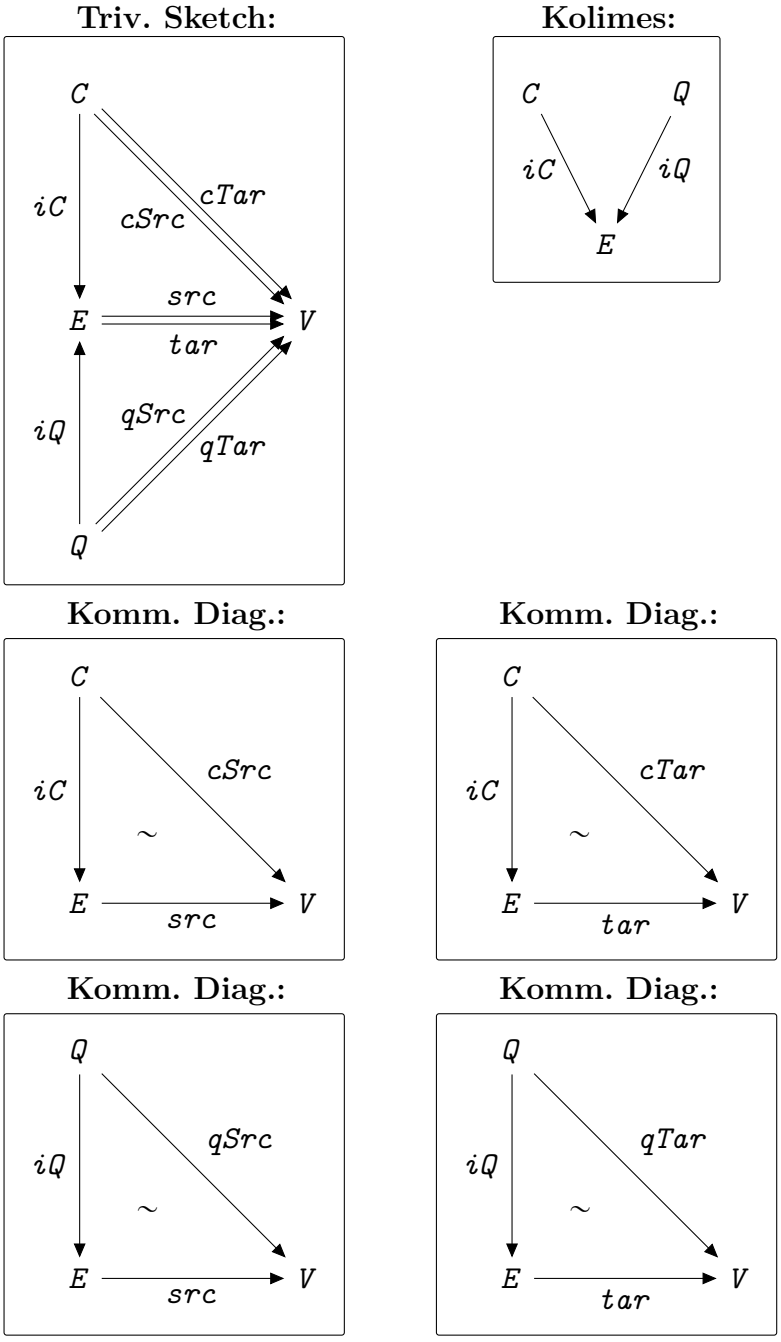


Abbildung 36: Sketch **D** zur Unterscheidung zweier Pfeilarten `C` und `Q`

- $QG\colon Q \underset{qTar}{\overset{qSrc}{\rightrightarrows}} V$

Dabei ist zu beachten, dass hier und an anderer Stelle Bezeichnungen wie CG und QG überladen werden – es ist immer auf den Zielsketch (hier $\mathbf{D}'$) zu achten.

Der zweite Gesichtspunkt von Containment ist eine waldartige Pfeilmenge, die zur Repräsentation der hierarchischen Struktur von Programmen verwendet werden kann. Um in einem geeigneten Sketch $\mathbf{F}$ zu modellieren, dass die Containmentpfeile der Menge C_I in jeder Instanz einen Wald bilden, muss man zwei Eigenschaften sicherstellen:

1. Jeder Knoten darf höchstens einen Vaterknoten haben, d.h. er darf Ziel von höchstens einem Pfeil von C_I sein.

2. Es darf keine Zyklen von Pfeilen in C_I geben, d.h. kein Knoten darf sich selbst enthalten.

Die erste Eigenschaft wird automatisch erreicht, wenn man die Knotenmenge in zwei disjunkte Mengen von Wurzelknoten und Kindknoten aufteilt. Wurzelknoten sind dann kein Ziel irgendeines Containmentpfeiles, während Kindknoten Ziel genau eines Containmentpfeiles sind, wodurch die Menge der Kindknoten isomorph zur Menge des Containmentpfeile ist. Eine dritte Art von Knoten wird in der Modellierung nicht erlaubt, so dass es keine Knoten mit mehr als einem Elternknoten geben kann. Alternativ könnte man statt einer Unterscheidung zwischen Wurzel- und Kindknoten ähnlich wie in Beispiel 58 verlangen, dass die Zielknotenfunktion des Containmentpfeile injektiv ist.

Die zweite Eigenschaft erfordert etwas mehr Aufwand. Man kann dafür einen der beiden folgenden Ansätze verwenden:

1. Man kann versuchen, eine Menge CG_I^+ der nichtleeren Pfade aus Containmentpfeilen zu modellieren, und dann mithilfe eines formalen Equalizer-Diagramms und eines initialen Objektes die Gleichung $\left\{p \in CG_I^+ \;\middle|\; src(p) = tar(p)\right\} = \varnothing$ auszudrücken. Diese besagt, dass es keinen nichtleeren Pfad aus Containmentpfeilen von einem Knoten zu sich selbst geben kann. Man muss bei der Modellierung der Pfade darauf achten, dass man keine unendlichen Pfade und insbesondere keine Zyklen in der Menge der Pfade zulässt. Diese Modellierung ist auch ohne explizite Unterscheidung zwischen Wurzel- und Kindknoten möglich.

2. Wenn man wie oben beschrieben zwischen Wurzel- und Kindknoten unterscheidet, kann es nur solche Zyklen geben, die ausschließlich aus Kindknoten bestehen und die man von keinem Wurzelknoten aus erreichen kann. Wenn ein Zyklus von einem Wurzelknoten aus erreichbar wäre, müsste nämlich mindestens einer der beteiligten Knoten zwei Elternknoten haben, wie Abb. 38 illustriert. Dies ist aber bei Containment nicht erlaubt.

   Auf diese Weise kann man die explizite Modellierung der (möglicherweise exponentiell großen) Menge von Pfaden vermeiden, wenn man auf anderen Wegen verhindert, dass es Zyklen von Kindknoten gibt. In einem Wald ist jeder Knoten und jeder Pfeil Teil eines Baumes, der von einem Wurzelknoten ausgeht. Hierzu kann man jedem Knoten und jedem Pfeil den Wurzelknoten seines zugehörigen Baumes mit einer Funktion $vRoot_I$ bzw. $cRoot_I$ zuordnen. Da es sich dabei um keinen Kindknoten handeln kann, ist sichergestellt, dass es sich um keinen Zyklus von Kindknoten handelt.

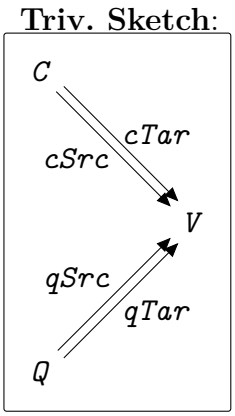


Abbildung 37: Vereinfachter Sketch $\mathbf{D}'$ zur Unterscheidung zweier Pfeilarten C und Q

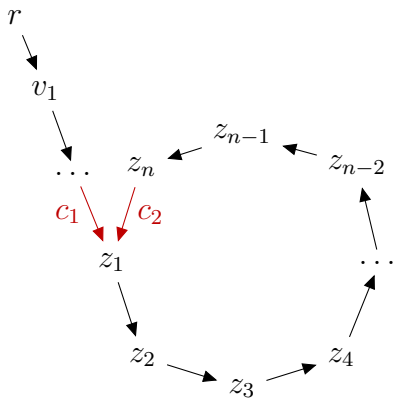


Abbildung 38: Von einer Wurzel r aus erreichbarer Zyklus von Containmentpfeilen

Für diese Arbeit wurde letztere Möglichkeit gewählt.

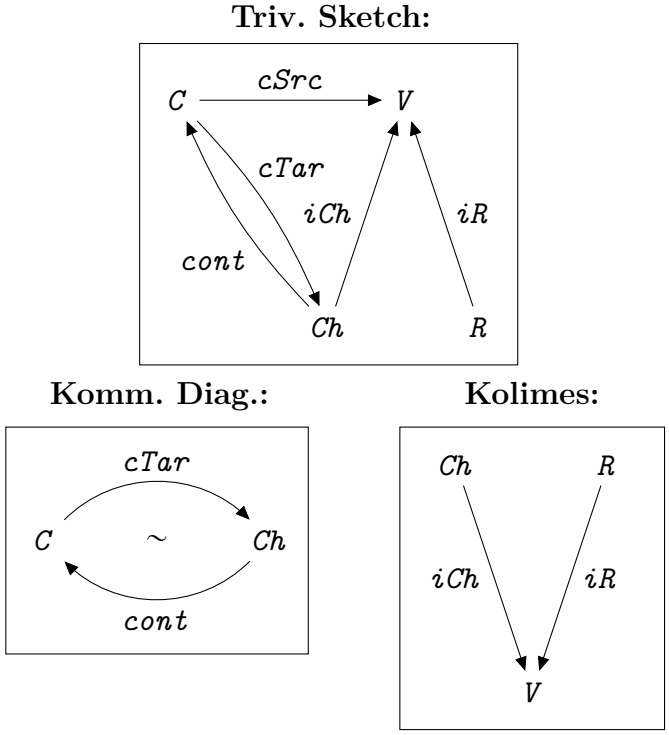
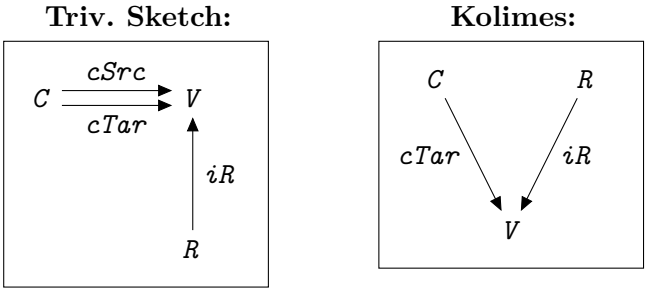


Abbildung 39: Vorläufiger Sketch zur Modellierung einer waldartigen Pfeilmenge

Um den Containment-Wald mit einem Sketch auf die vorher festgelegte Art zu erfassen, muss man zunächst die Unterscheidung zwischen Wurzel- und Kindknoten modellieren. Hierfür ergibt sich vorläufig der Sketch in Abb. 39: Der Kolimes stellt sicher, dass jeder Knoten entweder ein Wurzelknoten oder ein Kindknoten, aber nicht beides ist. Das kommutative Diagramm drückt aus, dass es für jeden Kindknoten genau eine Containmentpfeil gibt, deren Ziel er ist und umgekehrt, d.h. dass C_I und Ch_I isomorph sind. Aufgrund dieser Isomorphie reicht es eigentlich aus, nur einen Knoten im Sketch zu haben, der für beides steht. Der Kolimes sagt dann aus, dass ein Knoten entweder ein Wurzelknoten oder Ziel eines Containmentpfeiles ist:



Nun gilt es noch sicherzustellen, dass jeder Knoten bzw. Containmentpfeil zu einem Baum gehört, und nicht etwa zu einem Zyklus von Containmentpfeilen und Kindknoten. Hierzu

kann man die Menge Tr aller im Containment-Wald enthaltenen Bäume durch einen sogenannten Coequalizer modellieren. Ein Coequalizer ist ein Kolimes über einem Diagramm der Form $A \underset{f'}{\overset{f}{\rightrightarrows}} B$. In **Set** verschmilzt ein Coequalizer diejenigen Elemente $a, a' \in A$ miteinander bzw. fasst sie zu einer Menge zusammen, für die $f(a) = f'(a')$ ist.

Wenn man dies auf die Quell- und die Zielknotenfunktion von Containmentpfeilen anwendet, verschmilzt man jeweils genau die in einem Baum des Containmentwaldes enthaltenen Pfeile/Knoten zu einem Element einer Menge Tr. Diese Elemente von Tr entsprechen gerade den Bäumen des Containmentwaldes. Da aber jeder solche Baum genau einen Wurzelknoten enthält, und jeder Wurzelknoten in genau einem Baum enthalten ist, sind Tr und R_I isomorph. Daher kann man sie in der Modellierung gleichsetzen und R anstatt Tr als Scheitel des Coequalizers verwenden. Zwei weitere kommutative Diagramme stellen sicher, dass jeder Wurzelknoten seiner eigenen Komponente zugeordnet wird. Es ergibt sich der in Abb. 40 dargestellte Sketch **F**.

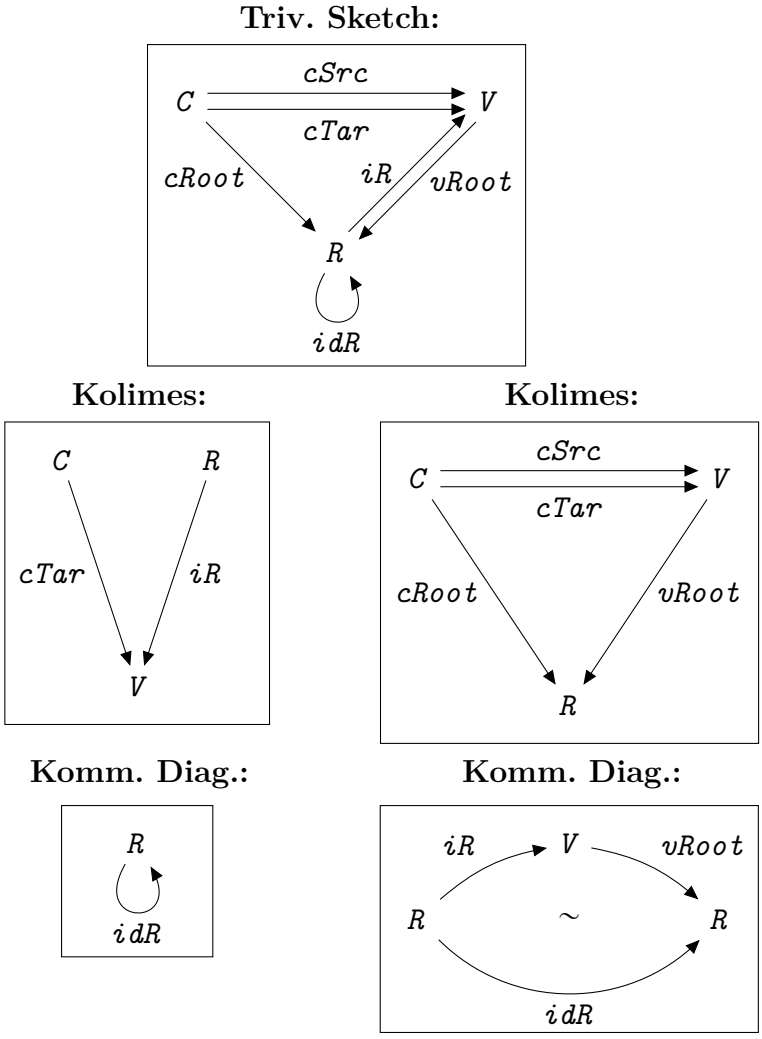


Abbildung 40: Sketch **F** zur Modellierung einer waldartigen Pfeilmenge

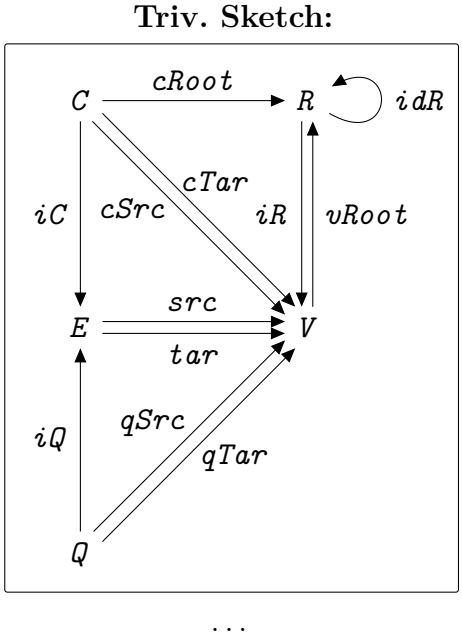
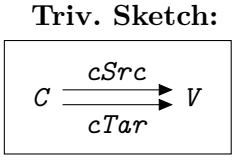


Abbildung 41: Sketch **C** für Containment mit Modellierung der Menge aller Kanten (Rest siehe Abb. 36 und 40)

In diesem Sketch ergibt sich ein Teilsketch $CG : \mathbf{G} \to \mathbf{F}$ der Form **G**, der die Unterscheidung zwischen Wurzel- und Kindknoten aufhebt:



Um die beiden Gesichtspunkte von Containment zu kombinieren, kann man die Sketches **D** und **F** entlang der Homomorphismen $CG : \mathbf{G} \to \mathbf{D}$ und $CG : \mathbf{G} \to \mathbf{F}$ durch einen Pushout verschmelzen. Der sich ergebende Sketch **C** besteht aus dem trivialen Sketch in Abb. 41 und den kommutativen Diagrammen und formalen Kolimites aus den Abbildungen 36 und 40: Ähnlich kann man auch mit dem Sketch $\mathbf{D}'$ verfahren.

4.3.4. Graphvarianten

Die in den vorherigen Unterabschnitten vorgestellen Sketches zur Umsetzung der Graphfeatures kann man mittels Pushouts zu größeren Sketches zusammenfügen, die Graphvarianten beschreiben. Beispielsweise kann man aus den Sketches für Typisierung und für Containment einen Sketch für typisierte Graphen mit Containment ableiten.

Hierzu muss man zunächst die benötigten Sketches und dann die Klebeobjekte für die Pushouts identifizieren. Anschließend wird durch Pushouts schrittweise der gewünschte allgemeine Sketch erzeugt. Schließlich wird noch das Typfragment identifiziert.

Typisierte Graphen mit Containment unterscheiden zwischen Querpfeilen und Containmentpfeilen. Um diese Unterscheidung zu ermöglichen, aber das Ergebnis noch einigerma-

ßen einfach zu halten, wird hierfür der Sketch $\mathbf{D'}$ benutzt. Außerdem wird zwischen Querpfeiltypen und Containmentpfeiltypen unterschieden, wofür ein weiteres Mal der Sketch $\mathbf{D'}$ benutzt wird. Die Containmentpfeile müssen einen Containment-Wald formen, was durch die Anwendung des Sketches $\mathbf{F}$ auf die Containmentpfeile sichergestellt wird. Die Containmentpfeiltypen dürfen im Gegensatz zu den Containmentpfeilen Zyklen bilden, weshalb an dieser Stelle $\mathbf{F}$ nicht eingesetzt wird. Für die Typisierung wird der Sketch $\mathbf{T}$ benötigt, und zwar zweimal: Einmal für die Containmentpfeiltypen und einmal für die Querpfeiltypen.

Nun kann man schrittweise den Sketch konstruieren: Zunächst wird der Sketch $\mathbf{D'}$ zur Unterscheidung zwischen Querpfeilen und Containmentpfeilen mit dem Sketch für eine waldartige Pfeilmenge verschmolzen, so dass die Containmentpfeile einen Wald bilden müssen. Als Schnittstellenobjekt dient der gemeinsame Teilgraph $\mathit{C} \overset{\mathit{cSrc}}{\underset{\mathit{cTar}}{\rightrightarrows}} \mathit{V}$, der durch die Homomorphismen $CG : \mathbf{G} \to \mathbf{D'}$ und $CG : \mathbf{G} \to \mathbf{F}$ markiert wird. Als Eingabe des Pushout dient also der Span $\mathbf{D'} \xleftarrow{CG} \mathbf{G} \xrightarrow{CG} \mathbf{F}$ und das Ergebnis ist der Sketch $\mathbf{V_1}$ in Abb. 42. Für die Typisierung werden Querpfeiltypen und Containmentpfeiltypen benötigt. Hierzu kann man zunächst eine Kopie von $\mathbf{D'}$ durch ein Koprodukt, also einen Pushout mit dem leeren Sketch als Schnittstellenobjekt, hinzufügen. Es ist sinnvoll, die Bezeichnungen der Knoten und Pfeile an dieser Stelle ähnlich wie bei typisierten Graphen abzuändern zu iV, tV, iSrc, tSrc, iCSrc, tCSrc etc., was bei einem Pushout ohne weiteres möglich ist. Hierdurch ergibt sich der Sketch $\mathbf{V_2}$ in Abb. 43.

$\mathbf{V_2}$ enthält die folgenden Teilsketches der Form $\mathbf{G}$:

- CIG: $\mathit{iC} \overset{\mathit{iCSrc}}{\underset{\mathit{iCTar}}{\rightrightarrows}} \mathit{iV}$, also der Containment-Instanzgraph.
- QIG: $\mathit{iQ} \overset{\mathit{iQSrc}}{\underset{\mathit{iQTar}}{\rightrightarrows}} \mathit{iV}$, also der Querpfeil-Instanzgraph.
- CTG: $\mathit{tC} \overset{\mathit{tCSrc}}{\underset{\mathit{tCTar}}{\rightrightarrows}} \mathit{tV}$, also der Containment-Typgraph.
- QTG: $\mathit{tQ} \overset{\mathit{tQSrc}}{\underset{\mathit{tQTar}}{\rightrightarrows}} \mathit{tV}$, also der Querpfeil-Typgraph.

Die Typisierung lässt sich durch den Sketch $\mathbf{T}$ in zwei Schritten hinzufügen. Zunächst wird der Pushout über dem Span $\mathbf{T} \xleftarrow{\langle \mathit{IG} \mid \mathit{TG} \rangle} \mathbf{G} + \mathbf{G} \xrightarrow{\langle \mathit{CIG} \mid \mathit{CTG} \rangle} \mathbf{V_2}$ gebildet, um einen Sketch V_3 zu erhalten, dessen zugrundeliegender trivialer Sketch und zusätzliche kommutative Diagramme in Abb. 44 abgebildet sind. Dabei steht $\mathbf{G} + \mathbf{G}$ für einen Sketch mit zwei Kopien von $\mathbf{G}$, von der die erste dem Instanzgraphen und die zweite dem Typgraphen entspricht. $\langle \mathit{IG} \mid \mathit{TG} \rangle$ und $\langle \mathit{CIG} \mid \mathit{CTG} \rangle$ sind Kotupel, die in $\mathbf{T}$ die Teilsketches IG und TG und in $\mathbf{V_2}$ die Teilsketches CIG und CTG auswählen.

Abschließend wird noch die Typisierung für die Querpfeile hinzugefügt. Dabei ist darauf zu achten, dass es nur eine Art von Knotentyp gibt und deshalb der Pfeil vType in das Schnittstellenobjekt mit aufgenommen werden muss. Deshalb werden zusätzlich noch die Homomorphismen $\Delta[\mathit{vType}] : \mathbf{2} \to \mathbf{F}$ und $\Delta[\mathit{vType}] : \mathbf{2} \to \mathbf{V_3}$ verwendet, die diesen Pfeil auswählen (siehe Unterabschnitt 4.3.2). Es ist also der Pushout über dem folgenden Span zu bilden:

$$\mathbf{T} \xleftarrow{\langle \mathit{IG} \mid \mathit{TG} \mid \Delta[\mathit{vType}] \rangle} \mathbf{G} + \mathbf{G} + \mathbf{2} \xrightarrow{\langle \mathit{CIG} \mid \mathit{CTG} \mid \Delta[\mathit{vType}] \rangle} \mathbf{V_2}$$

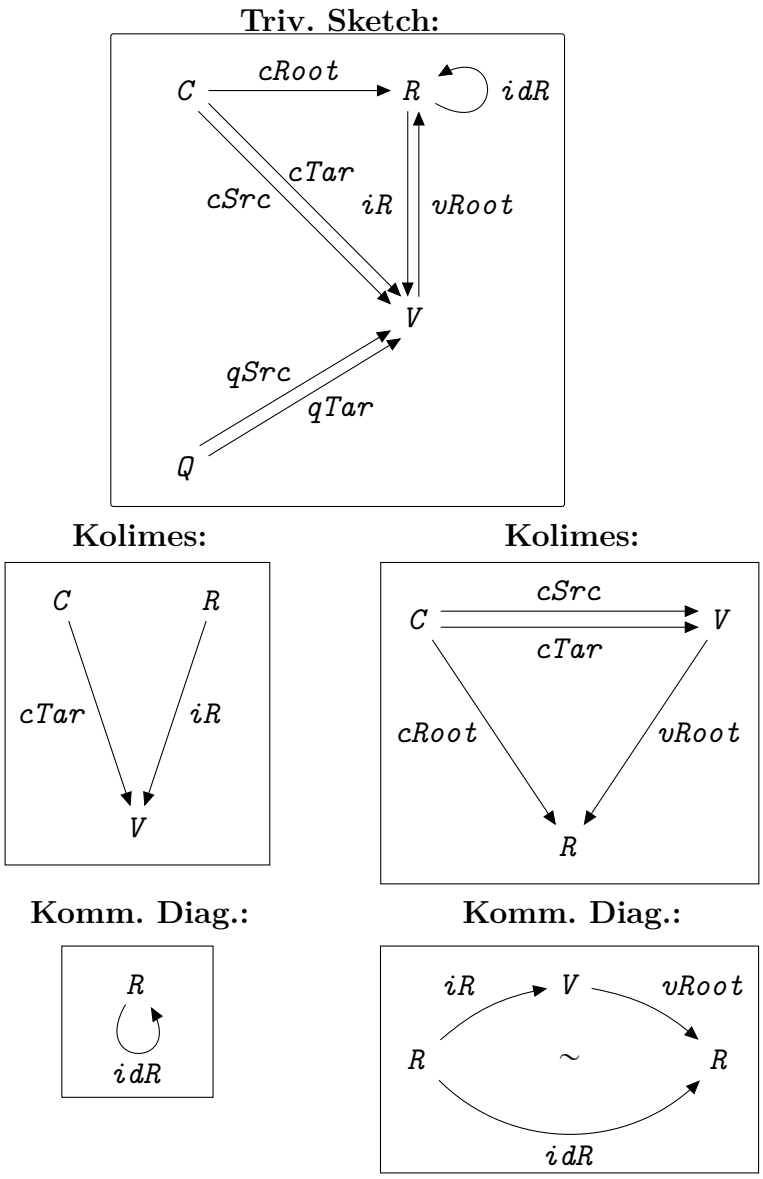
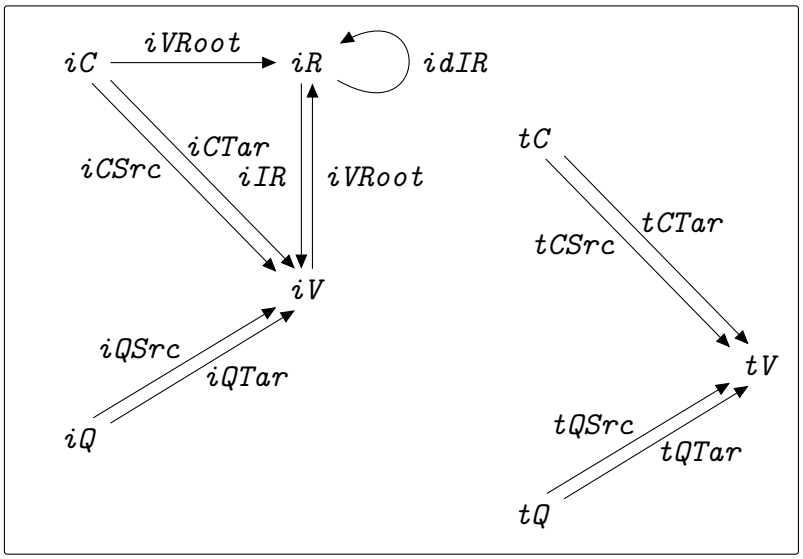
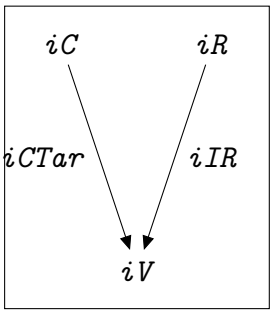


Abbildung 42: Sketch $\mathbf{V_1}$

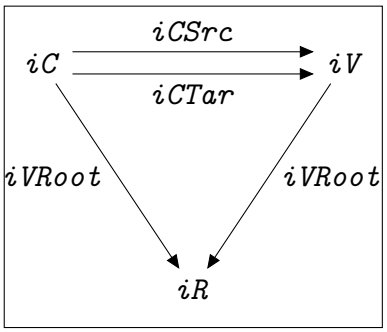
Triv. Sketch:



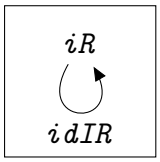
Kolimes:



Kolimes:



Komm. Diag.:



Komm. Diag.:

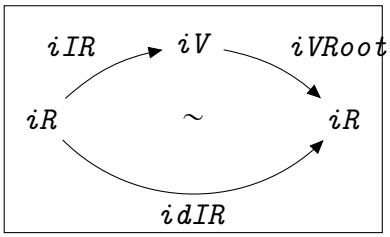
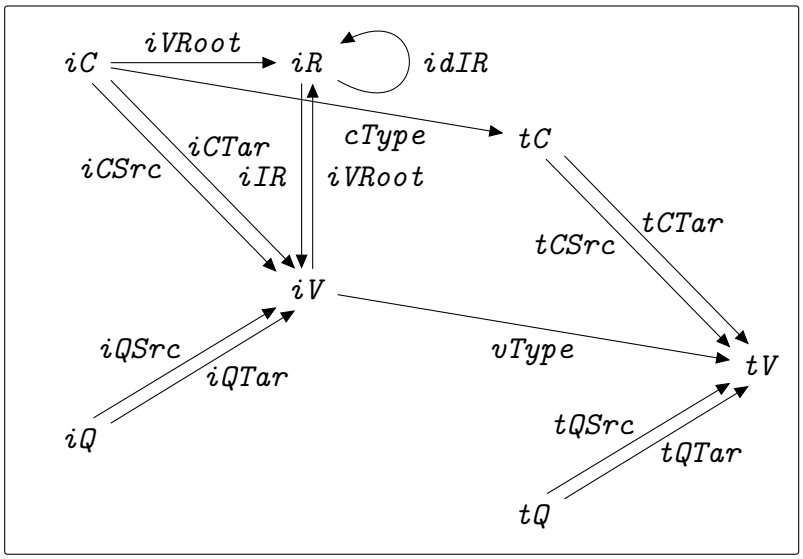
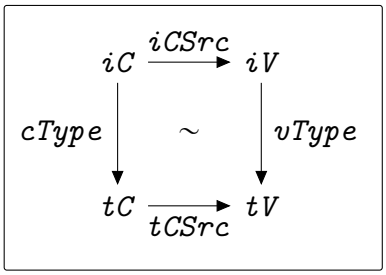


Abbildung 43: Sketch $\mathbf{V_2}$

Triv. Sketch:



Komm. Diag.:



Komm. Diag.:

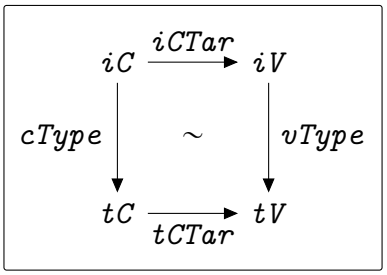
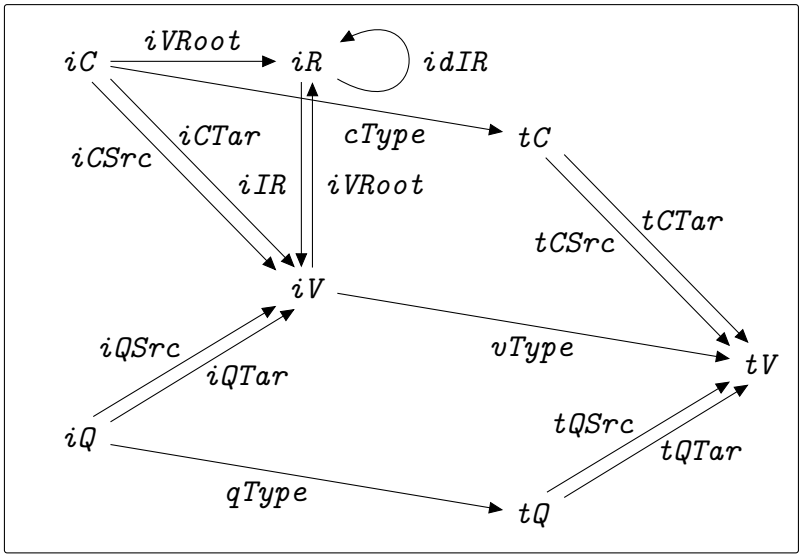
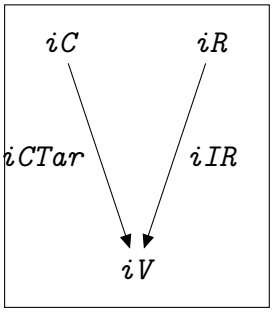


Abbildung 44: Sketch $\mathbf{V_3}$
(Änderungen im Vergleich zu V_2 in Abb. 43)

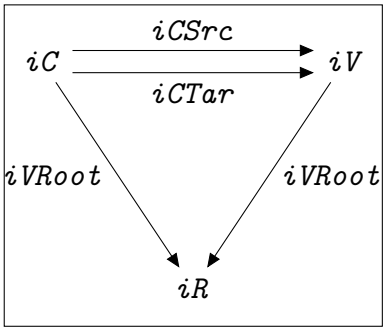
Triv. Sketch:



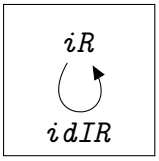
Kolimes:



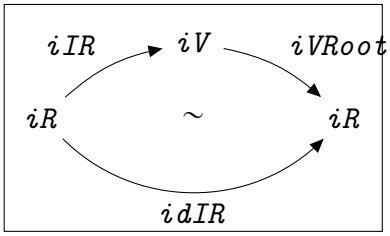
Kolimes:



Komm. Diag.:



Komm. Diag.:



Sketch **V** der typisierten Graphen mit Containment
(Fortsetzung nächste Seite)

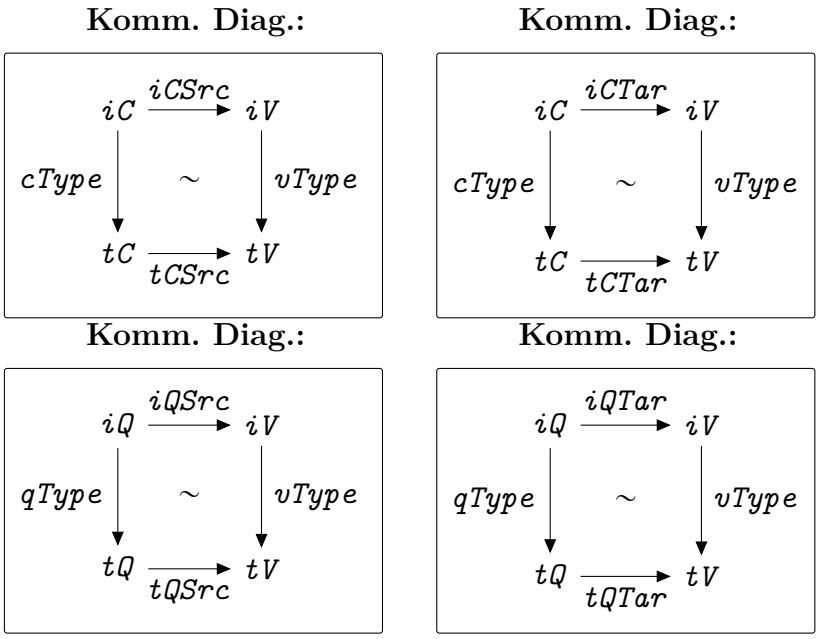


Abbildung 45: Sketch **V** der typisierten Graphen mit Containment

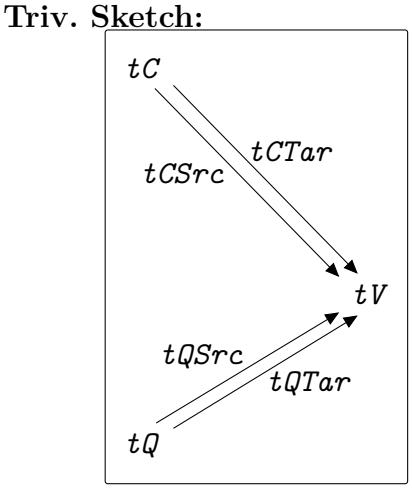


Abbildung 46: Typfragment für den Sketch **V** der typisierten Graphen mit Containment

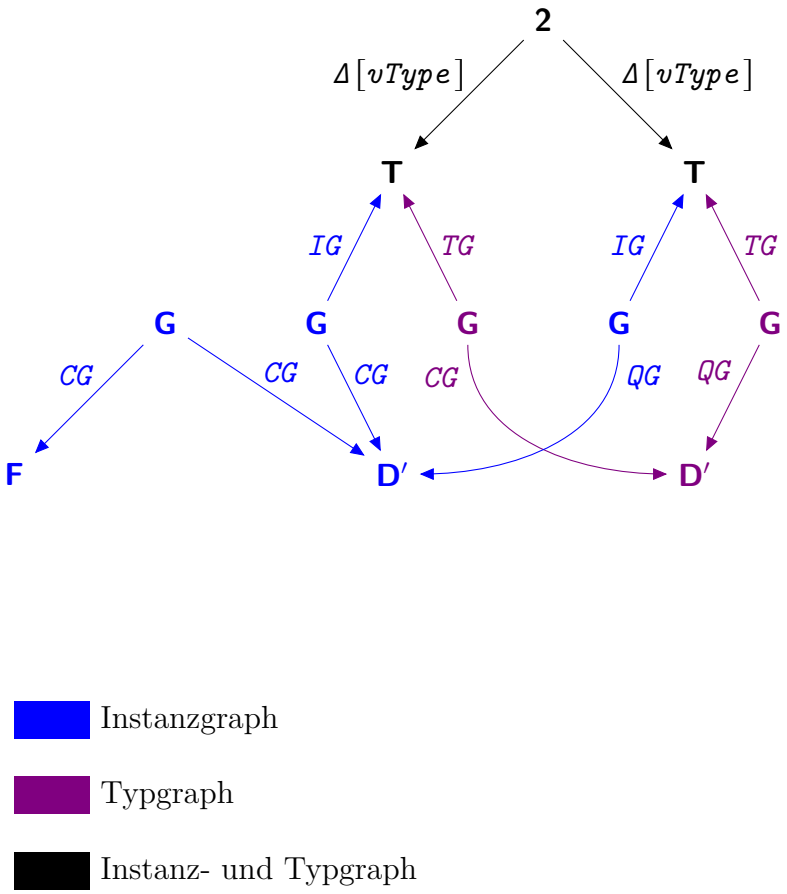


Abbildung 47: Kolimes-Diagramm D zur alternativen Konstruktion der typisierten Graphen mit Containment

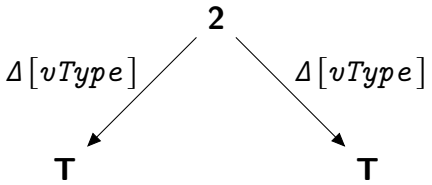


Abbildung 48: Diagramm D_t für das Typfragment aus Abb. 46

Das Ergebnis ist der Sketch **V** in Abb. 45. Als Typfragment wird der Teilsketch mit den Containment- und Querpfeiltypen, den Knotentypen und den zugehörigen Pfeilen gewählt, der in Abb. 46 abgebildet ist.

Anstatt den Sketch **V** in mehreren Schritten zu konstruieren, kann man auch einen einzigen Kolimes mit Scheitel **V** über dem Diagramm D in Abb. 47 verwenden. Dieser Kolimes ist dennoch pushoutartig, da zwischen keinen zwei Knoten des Diagrammes mehr als ein Pfeil vorhanden ist. Deswegen kann man das Diagramm als eine Menge von Teilspans betrachten, die Schnittstellenobjekte (gemeinsame Teilsketches) zwischen zu verschmelzenden Sketches definieren: Im Diagramm D konstruiert der linke Teilspan $\mathbf{D'} \xleftarrow{CG} \mathbf{G} \xrightarrow{CG} \mathbf{F}$ den Instanzgraphen. Die mittleren Teilspans $\mathbf{T} \xleftarrow{IG} \mathbf{G} \xrightarrow{CG} \mathbf{D'}$ und $\mathbf{T} \xleftarrow{TG} \mathbf{G} \xrightarrow{CG} \mathbf{D'}$ entsprechen dem zweiten Schritt der Konstruktion, fügen also die Typisierung für die Containmentpfeile hinzu. Die rechten Teilspans $\mathbf{T} \xleftarrow{IG} \mathbf{G} \xrightarrow{QG} \mathbf{D'}$ und $\mathbf{T} \xleftarrow{TG} \mathbf{G} \xrightarrow{QG} \mathbf{D'}$ bewirken dasselbe für die Querpfeile. Der obere Teilspan $\mathbf{T} \xleftarrow{\Delta[vType]} \mathbf{2} \xrightarrow{\Delta[vType]} \mathbf{T}$ sorgt dabei dafür, dass es im Ergebnis nur eine Art von Knotentyp gibt.

Um in dieser Modellierung das Typfragment zu erhalten, kann man das in Abb. 48 dargestellte Teildiagram D_t verwenden: Der Scheitel des Kolimes über D_t entspricht dem Typfragment $\mathcal{TF}$. Dessen Vorkommen in **V** erhält man, indem man die Projektionen des Kolimes über dem Diagramm D verwendet, um einen kommutativen Kokegel über $\mathbf{D'}$ zu bilden. Der eindeutige Homomorphismus kommutativer Kokegel von $\mathcal{TF}$ nach **V** modelliert das gewünschte Vorkommen $\mathcal{TS} : \mathcal{TF} \to \mathbf{V}$. Dieses ist formal gesehen dasselbe wie der gesuchte typisierte Sketch der typisierten Graphen mit Containment.

Auf diese Weise kann man also wahlweise mit mehreren Pushouts oder mit einem einzigen pushoutartigen Kolimes Sketches für Graphvarianten aus Sketches für Graphfeatures konstruieren.

Bewertung

Sketches erlauben es, Graphfeatures isoliert voneinander zu definieren und durch eine Spezifikation in Form von Spans oder allgemeineren Kolimes-Diagrammen zu Graphvarianten zusammenzufügen. Demonstriert wurde dies für die Typisierung und für Containment. Zumindest für die Attributierung sollte eine solche Modellierung ebenfalls möglich sein. Die Datentypen werden in [EPT04; Ehr+06] durch algebraische Signaturen und Algebren modelliert. Die dort verwendeten mehrsortigen algebraische Signaturen haben dieselbe Ausdruckskraft wie Sketches mit einer endlichen Anzahl von endlichen Produkten als Limites und ohne kommutative Diagramme und Kolimites [BW12, Abschnitt 7.7]. Algebren entsprechen Instanzen solcher Sketches. Zur Umsetzung der Attributierung kann man nun mit Sketches wahlweise algebraische Signaturen und Algebren oder eine eingeschränkte Art von Sketches und Instanzen davon modellieren.

Für Ordnung, Vererbung und Constraints ist keine sketchbasierte Modellierung bekannt. Ordnung ist vermutlich ohne größere Schwierigkeiten modellierbar. Für Vererbung sind zwei Möglichkeiten denkbar: Zum einen könnte man die Sketches nicht Mengen und Funktionen in **Set** zu interpretieren, sondern durch eine geeignete Art partieller Ordnungen und Abbildungen dazwischen – dann unterliegen aber alle formalen Objekte und Pfeile des Sketches der Vererbung. Alternativ könnte man versuchen, eine geeignete Art partieller Ordnungen mit einem Sketch modellieren und Knoten und Pfeile nicht mehr mit einem Typisierungshomomorphismus einen Typ, sondern auf andere Weise mehrere Typen zuzuordnen. Um Constraints als Graphfeature mit Sketches zu Modellieren ist möglicherweise eine ausdruckskräftigere Art von Sketches nötig. Alternativ kann man vielleicht manche

Constraints unmittelbar in den Sketch codieren, allerdings geht dadurch die Sprachunabhängigkeit verloren.

4.4. Zusammenfassung

In diesem Kapitel wurde demonstriert, dass man Graphfeatures mit Sketches voneinander isoliert formal definieren und dann miteinander zu Graphvarianten kombinieren kann. Mit typisierten Sketches ist es möglich, Graphvarianten zur sprachunabhängigen Programmrepräsentation zu definieren. Konkret wurden Typisierung und Containment mit Sketches formalisiert, für Ordnung und Vererbung wäre dies vermutlich ebenfalls möglich.

Im folgenden Kapitel werden noch Beispiele von Programmmanipulationen und -analysen gegeben, die auf solchen Programmrepräsentationen aufbauen.

Kapitel 5

Anwendungsbeispiele

In diesem Kapitel werden die in Kapitel 4 vorgestellten Mittel der Kategorientheorie angewendet, um Programmmanipulationen zu beschreiben. Die Programmrepräsentation erfolgt durch Objekte einer beliebigen Kategorie, wobei die Objekte im Folgenden dennoch als Graphen bezeichnet werden, weil vor allem die in Abschnitt 4.3 formalisierten Graphvarianten dafür vorgesehen sind. Auch die Mittel zur formalen Erfassung von Programmeigenschaften und -beziehungen sind bereits aus Abschnitt 4.1 bekannt, nämlich Pfeile der Kategorien, kommutative Diagramme und Limites und Kolimites. Die Operationen werden hier nicht beschrieben, da die Ausführung von Programmmanipulationen nicht Teil dieser Arbeit ist.

Die im folgenden vorgeschlagenen Beschreibungen von Programmmanipulationen ist repräsentationsunabhängig. Das heißt, dass sie nicht festlegen, **wie** Programme repräsentiert werden, und dass man die Beschreibung auf verschiedene Programmrepräsentationen anwenden kann – prinzipiell sogar welche, die nicht graphbasiert sind. Dennoch sind Annahmen darüber nötig, **was** genau repräsentiert wird, um die Programmeigenschaften und -beziehungen zu formalisieren. Außerdem kann die Formalisierung nur auf Kategorien angewendet werden, in denen die zur Beschreibung der Programmeigenschaften und -beziehungen verwendeten Limites und Kolimites existieren.

Beim Merge wird davon ausgegangen, dass die Graphen Programme oder Fragmente davon repräsentieren. Bei der Featurekomposition wird angenommen, dass Graphen Produktvarianten oder Fragmenten davon entsprechen. Auf diesen Annahmen aufbauend wird beim Merge die Ableitungsbeziehung zwischen Programmen formalisiert. Bei der Featurekomposition wird die Eigenschaft, ob eine Produktvariante ein bestimmtes Feature enthält, und die Verfeinerungsbeziehung zwischen Produktvarianten beschrieben.

Auf Grundlage dieser Annahmen werden in den folgenden beiden Abschnitten die Eigenschafen und Beziehungen von Merge und Featurekomposition formalisiert und anhand von Beispielen erläutert.

5.1. Merge

Der Merge ist eine Programmtransformation, mit der zwei geänderte Programmversionen aufgrund einer gemeinsamen Grundversion zusammengeführt werden und wurde schon in Abschnitt 2.2 erläutert und in [Tae+10; Rut+09; EET11] ähnlich wie hier formalisiert. In der folgenden Formalisierung wird angenommen, dass ein Graph einer Programmversion entspricht. Vereinfachend wird davon ausgegangen, dass man mit den Pfeilen der zur Programmrepräsentation verwendeten Kategorie ähnlich wie Graphhomomorphismen das Hinzufügen und Entfernen von Programmelementen modellieren kann (siehe Unterabschnitte 3.2.1 und 4.1.3). Außerdem wird nur der Fall betrachtet, dass keine Konflikte auftreten. Für die Erkennung von Konflikten sei auf [Tae+10] verwiesen.

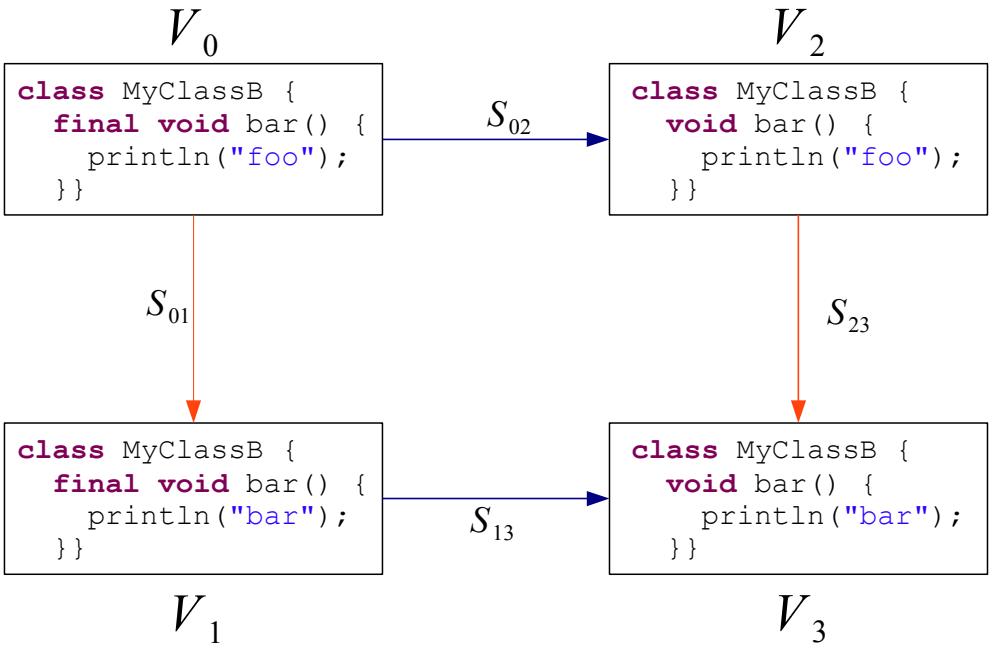


Abbildung 49: Merge von Java-Programmen

Eine Änderung zwischen zwei Programmversionen besteht darin, dass zunächst Elemente entfernt und dann hinzugefügt werden. Dieses Entfernen und Hinzufügen wird durch einen Span beschrieben (vgl. Unterabschnitt 4.1.3). Beispielsweise beschreibt ein Span $V_0 \xleftarrow{r_{01}} C_{01} \xrightarrow{a_{01}} V_1$ eine Änderung einer Grundversion V_0 zu einer abgeänderten Version V_1. C_{01} enthält die Gemeinsamkeiten von V_0 und V_1, r_{01} beschreibt das Entfernen der Elemente aus V_0, die in V_0 nicht mehr vorhanden sind. Mit a_{01} wird das Hinzufügen der neuen Elemente von V_1 beschrieben.

Beim Merge sind als Eingabe zwei Programmversionen V_1 und V_2 gegeben, die durch Änderungen an einer Grundversion V_0 entstehen. Beispielsweise könnten die in Abb. 49 gegebenen Programmversionen (in einer graphbasierten Repräsentation) gegeben sein, die bereits aus Abschnitt 2.2 bekannt sind. Die Grundvariante V_0 enthält eine Methode `bar`, die als `final` markiert ist und `"foo"` ausgibt. In V_1 wird der String `"foo"` durch `"bar"` ersetzt und in V_2 der Modifier `final` entfernt. Formal sind diese Änderungen durch Spans S_{01} und S_{02} gegeben:

$$S_{01} = (V_0 \xleftarrow{r_{01}} C_{01} \xrightarrow{a_{01}} V_1)$$

$$S_{02} = (V_0 \xleftarrow{r_{02}} C_{02} \xrightarrow{a_{02}} V_2)$$

Im Beispiel aus Abb. 49 würde C_{01} das String-Literal `"foo"` und C_{02} den Modifier `final` nicht mehr enthalten, wie in Abb. 51 illustriert ist. Mit r_{01} bzw. r_{02} wird das Entfernen von Elementen aus V_0 beschrieben, im Beispiel also das Entfernen des String-Literals `"foo"` bzw. des Schlüsselwortes `final`. Mit a_{01} bzw. a_{02} wird das Hinzufügen von Elementen in V_1 bzw. V_2 beschrieben. Im Beispiel fügt a_{01} das String-Literal `"bar"` in V_1 hinzu, während a_{02} in V_2 nichts Neues hinzufügt.

Gesucht

Gesucht ist eine Änderung, beschrieben durch einen Span S_{03}, die die Änderungen S_{01}

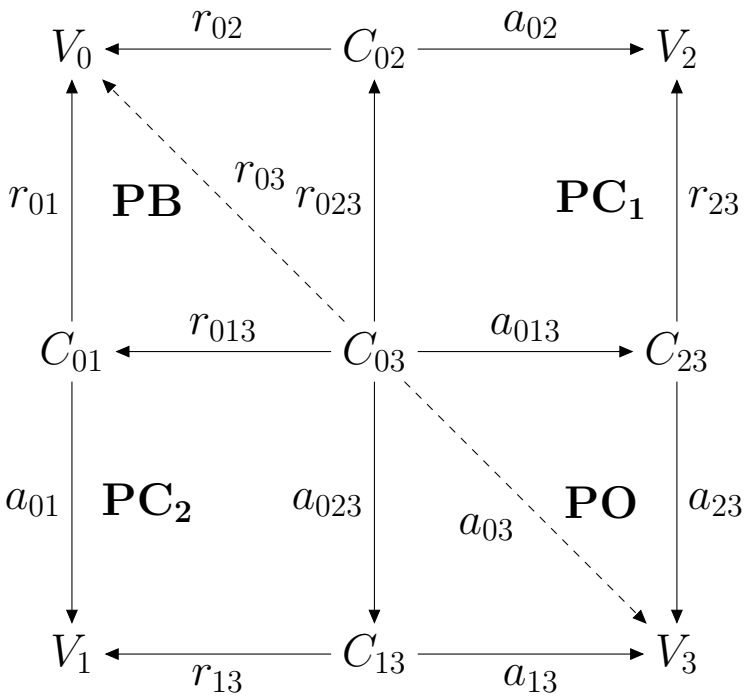


Abbildung 50: Merge durch kategorielle Konstrukationen

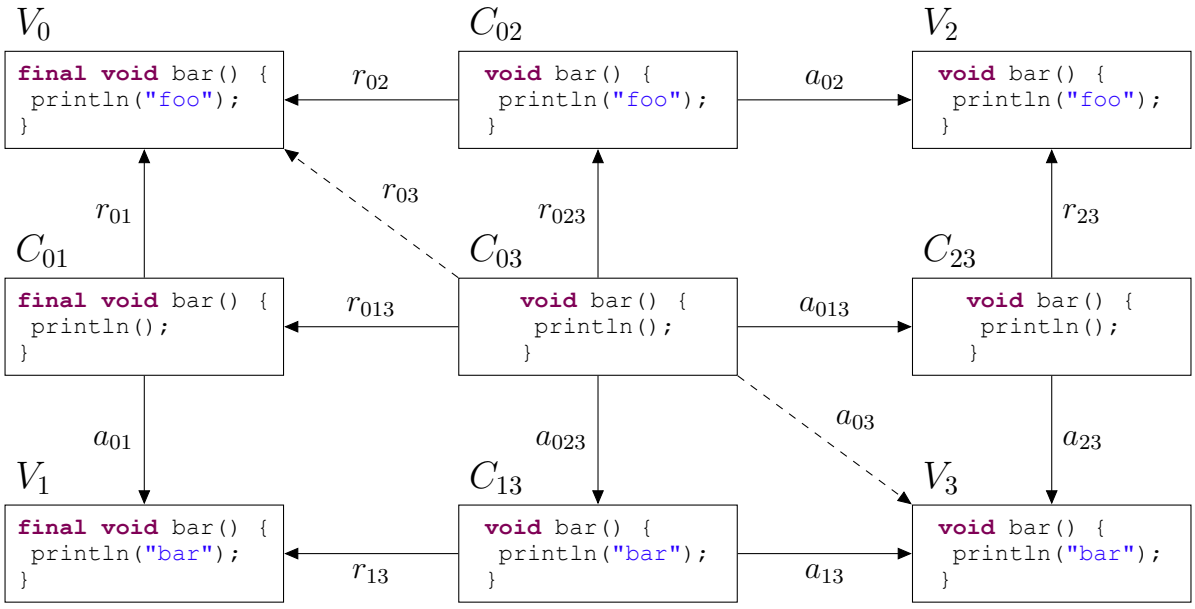


Abbildung 51: Formalisierung des Merge aus Abb. 49

und S_{02} kombiniert, und eine Programmversion V_3 ergibt:

$$S_{03} = (V_0 \xleftarrow{r_{03}} C_{03} \xrightarrow{a_{03}} V_3)$$

C_{03} beschreibt die gemeinsamen Elemente von V_0, V_1 und V_2. r_{03} entfernt aus V_0 die Elemente, die r_{01} oder r_{02} entfernt haben. a_{03} fügt in V_3 die Elemente hinzu, die a_{01} in V_1 oder a_{02} in V_2 hinzugefügt hat.

Vorgehensweise

Das Ziel ist es, die Konstruktion des Spans S_{03} mit kategorientheoretischen Mitteln zu beschreiben. Alle Elemente, die in V_1 oder V_2 entfernt wurden, müssen in V_3 entfernt werden, und alle Elemente, die in V_1 oder V_2 hinzugefügt wurden, müssen in V_3 hinzugefügt werden. Wie in Abb. 50 illustriert ist, kann man dazu im ersten Schritt r_{03} mittels eines Pullbacks **PB** und anschließend a_{03} mithilfe von zwei Pushout-Komplementen $\mathbf{PC_1}$ und $\mathbf{PC_2}$ und einem Pushout **PO** konstruieren. Ein Pushout-Komplement ist eine Konstruktion, bei der versucht wird, zwei gegebene Pfeile $I \xrightarrow{f} G_1 \xrightarrow{g'} P$ durch zwei weitere Pfeile $I \xrightarrow{g} G_2 \xrightarrow{f'} P$ zu einem Pushout über dem Span $G_1 \xleftarrow{f} I \xrightarrow{g} G_2$ mit Scheitel P zu vervollständigen. Die Pushout-Komplemente sind dabei nötig, da V_1 Elemente enthalten kann, die V_2 entfernt hat und umgekehrt. Das Ergebnis von $\mathbf{PC_1}$ enthält die in V_1 hinzugefügten Elemente, die in V_2 entfernten aber nicht mehr. Ebenso enthält das Ergebnis von $\mathbf{PC_2}$ die in V_2 hinzugefügten Elemente und entfernt die, die in V_1 entfernt wurden. Im Beispiel fügt $\mathbf{PC_1}$ das neue String-Literal `"bar"` hinzu und entfernt den `final`-Modifier, wie in Abb. 51 zu sehen ist. $\mathbf{PC_2}$ fügt keine neuen Elemente hinzu und entfernt das String-Literal `"foo"`.

Im Folgenden werden diese Schritte zur Konstruktion des Spans S_{03} im Einzelnen beschrieben:

1. C_{03} und r_{03} bestimmen mittels eines Pullback **PB**:
   Gegeben ist der Kospan $C_{01} \xrightarrow{r_{01}} V_0 \xleftarrow{r_{02}} C_{02}$. Das Ergebnis des Pullback besteht aus dem Scheitel C_{03} und den Pfeilen r_{013}, r_{023} und r_{03}. Der Pfeil r_{013} entfernt aus C_{01} die Elemente von V_0, die r_{01} nicht entfernt hat, die aber r_{02} entfernt. Ebenso entfernt Pfeil r_{023} aus C_{02} die Elemente von V_0, die r_{02} nicht entfernt hat, die aber r_{01} entfernt. Im Beispiel entfernt r_{013} das Schlüsselwort `final` aus C_{01} und r_{023} das String-Literal `"foo"` aus C_{02}. Der Pfeil r_{03} entfernt im Beispiel beides aus V_0.

2. C_{13} und a_{023} bestimmen mittels eines Pushout-Komplements $\mathbf{PC_1}$:
   Gegeben sind die Pfeile und Objekte $C_{03} \xrightarrow{r_{013}} C_{01} \xrightarrow{a_{01}} V_1$. Ergebnis sind die Pfeile und Objekte $C_{03} \xrightarrow{a_{023}} C_{13} \xrightarrow{r_{13}} V_1$. C_{13} enthält die Elemente aus C_{03} und die Elemente, die a_{01} zu V_1 hinzugefügt hat. a_{023} fügt in C_{13} die Elemente hinzu, die a_{01} in V_1 hinzugefügt hat. Der Pfeil r_{13} entfernt aus V_1 diejenigen Elemente, die r_{013} aus C_{01} entfernt hat. Im Beispiel wird das String-Literal `"bar"` hinzugefügt und der Modifier `final` entfernt.

3. In ähnlicher Weise werden C_{23} und a_{013} durch ein Pushout-Komplement $\mathbf{PC_2}$ bestimmt. Im Beispiel entfernt r_{23} das String-Literal `"foo"` aus V_2. a_{013} fügt nichts hinzu.

4. $V3$ und a_{03} bestimmen mittels eines Pushouts **PO**:
   Gegeben ist der Span $C_{13} \xleftarrow{a_{023}} C_{03} \xrightarrow{a_{013}} C_{23}$. Das Ergebnis des Pushouts besteht aus dem Scheitel V_3 und den Pfeilen a_{13}, a_{23} und a_{03}. Der Pfeil a_{13} fügt in V_3

diejenigen Elemente hinzu, die a_{013} bzw. a_{02} zu V_2 hinzugefügt haben. a_{23} fügt in V_3 diejenigen Elemente hinzu, die a_{023} bzw. a_{01} zu V_21 hinzugefügt haben. Im Beispiel ergibt sich daraus die gemergte Version V_3, die den Modifier `final` nicht enthält und `"bar"` anstelle von `"foo"` ausgibt.

5.2. Featurekomposition

Featurekomposition ist eine Programmtransformation, die zu bestehenden Produktvarianten einer Produktlinie neue Features hinzufügt und wurde schon in Abschnitt 2.3 vorgestellt. Dort wurde festgestellt, dass ihr eine Eigenschaft und eine Beziehung zugrundeliegen, nämlich die Eigenschaft, ob eine Produktvariante ein Feature enthält oder nicht, und eine Verfeinerungsbeziehung zwischen Produktvarianten.

Bei der folgenden Formalisierung der Featurekomposition wird angenommen, dass ein Graph eine Produktvariante oder ein Fragment einer Produktvariante repräsentiert. Ein Spezialfall von Produktvarianten sind dabei die Featuremodule, durch die die Features implementiert werden. Es wird nicht davon ausgegangen, dass die Graphen Informationen über die Produktlinie und das Domänenmodell enthalten, also darüber, welche Features vorhanden sind und in welcher Weise sie kombiniert werden dürfen. Deshalb müssen im Folgenden zunächst das Domänenmodell und die Produktlinie separat formalisiert werden, bevor die Featurekomposition selbst definiert wird.

Domänenmodell

In dieser Beschreibung der Featurekomposition wird eine vereinfachte Art von Domänenmodell benutzt, bei der alle Features optional sind und bei der sich Features paarweise ausschließen können. Die Features der Produktlinie (bzw. ihre Namen) werden durch eine endliche Menge F repräsentiert. Die Paare von Features, die miteinander komponiert werden dürfen, sich also **nicht** gegenseitig ausschließen, werden durch eine Menge P von zweielementigen Mengen $p = \{a, b\} \in P$ von Features $a, b \in F$ beschrieben. Das Domänenmodell wird durch das Paar $D = (F, P)$ vollständig erfasst.

Produktlinie

In einer Produktlinie mit einem solchen Domänenmodell D wird jedes Feature $f \in F$ durch ein Featuremodul implementiert, das durch einen Graphen G_f repräsentiert wird. Dadurch ergibt sich eine Familie $G = (G_f)_{f \in F}$ von Featuremodulen (Graphen). Featuremodule kann man durch Pushouts über geeigneten Spans komponieren, sofern das Domänenmodell die Kombination der zugehörigen Features nicht verbietet. Für jedes Paar $\{a, b\} \in P$ ist daher ein Schnittstellengraph $I_{\{a,b\}} = I_{a,b} = I_{b,a}$ und ein Span $S_{a,b} = (G_a \xleftarrow{i_{a,b}} I_{a,b} \xrightarrow{i_{b,a}} G_b)$ gegeben. Diese Spans bilden die Familie $S = (S_p)_{p \in P}$. Sie legen fest, in welcher Form die Featuremodule miteinander komponiert werden und ergeben sich aus den Binderegeln der Programmiersprache und der Kompositionsreihenfolge der Features. Die Produktlinie wird somit durch das Tripel $L = (D, G, S)$ modelliert.

Schrittweise Verfeinerung

Bei der Featurekomposition durch schrittweise Verfeinerung [BSR04] wird ein Featuremodul nach dem anderen inkrementell hinzugefügt. Ein Verfeinerungsschritt, bei dem eine Produktvariante v um Feature f verfeinert wird, lässt sich durch einen Pushout beschreiben, wie in Abb. 52 illustriert ist. Vorausgesetzt ist hier, dass alle in v enthaltenen Features mit f kombiniert werden dürfen. Die Produktvariante v wird durch einen Graphen G_v repräsentiert, das Featuremodul von f durch einen Graphen G_f. Die Eingabedaten des Pushout liefert ein Span $S_{v,f} = (G_v \xleftarrow{i_{v,f}} I_{v,f} \xrightarrow{i_{f,v}} G_f)$. Dieser legt fest, wie die Produktvariante v mit dem Feature f komponiert wird. Das Ergebnis des Pushout besteht

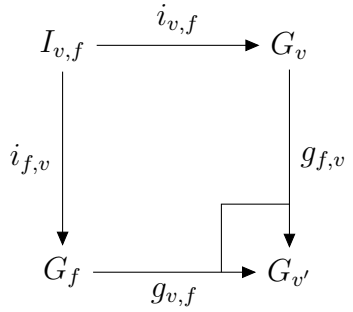


Abbildung 52: Verfeinerungsschritt:
der Produktvariante v um das Feature f

aus dem Scheitelgraph $G_{v'}$ und den Pfeilen $g_{f,v}$ und $g_{v,f}$. $G_{v'}$ repräsentiert die verfeinerte Produktvariante v'. Der Pfeil $G_f \xrightarrow{g_{f,v}} G_{v'}$ drückt die Eigenschaft aus, dass die berechnete Produktvariante v' das Feature f enthält. Der andere Pfeil $G_v \xrightarrow{g_{v,f}} G_{v'}$ drückt die Verfeinerungsbeziehung von v nach v' aus. Diese beiden Pfeile entsprechen somit den Eigenschaften und Beziehungen, die der Featurekomposition zugrundeliegen.

Aus der obigen Beschreibung ergibt sich die Frage, wie man den Eingabespan $S_{v,f}$ erhält. Im Folgenden wird erklärt, wie man ihn rekursiv konstruieren kann. Gegeben ist eine Produktvariante v mit n durchnummerierten Features 1, 2, ..., $n-1$, n. Das Ziel ist es eine Produktvariante v' zu komponieren, die sich aus der Verfeinerung der Produktvariante v um das Feature $n+1$ ergibt. Gesucht wird dazu der Span $S_{1...n,n+1} = (G_{1...n} \longleftarrow I_{1...n,n+1} \longrightarrow G_{n+1})$.

1. **Rekursionsanfang:**
   Falls $n = 1$, dann ist der gesuchte Span $S_{1,2}$ im Domänenmodell enthalten.

2. **Rekursionsschritt** $n \to n+1$**:**
   Konstruiere rekursiv die Spans

$$S_{1...n-1,n} = (G_{1...n-1} \longleftarrow I_{1...n-1,n} \longrightarrow G_n)$$

$$S_{1...n-1,n+1} = (G_{1...n-1} \longleftarrow I_{1...n-1,n} \longrightarrow G_n)$$

   a) Durch einen Pushout über dem Span $S_{1...n-1,n}$ erhält man den Graph $G_{1...n}$ und die Pfeile $G_n \xrightarrow{g_{1...n-1,n}} G_{1...n}$ und $G_{1...n-1} \xrightarrow{g_{n,1...n-1}} G_{1...n}$ (Abb. 53a)

   b) Um diese beiden Spans zu kombinieren, berechne die gemeinsamen Programmelemente der Schnittstellengraphen durch einen Pullback über dem Kospan

$$I_{1...n-1,n+1} \xrightarrow{i_{n+1,1...n-1}} G_{n+1} \xleftarrow{i_{n+1,n}} I_{n,n+1}$$

   mit Scheitel $C_{1...n}$ (Abb. 53b).

   c) Bilde den Pushout über dem Span $I_{1...n-1,n+1} \xleftarrow{l_{1...n-1,n+1}} C_{1...n} \xrightarrow{l_{n,n+1}}$, um einen Schnittstellengraphen $I_{1...n,n+1}$ zu erhalten (Abb. 53c).

d) Konstruiere den rechten Pfeil $I_{1...n} \xrightarrow{i_{n+1,1...n}} G_{n+1}$ des gesuchten Span als den eindeutigen Homomorphismus kommutativer Kokegel von dem Pushout aus dem vorherigen Schritt zu dem kommutativen Kokegel in Abb. 53d. Hierbei handelt es sich aufgrund des Pullback in Schritt b) um einen kommutativen Kokegel.

e) Konstruiere den linken Pfeil $I_{1...n} \xrightarrow{i_{1...n,n+1}} G_{n+1}$ des gesuchten Span als den eindeutigen Homomorphismus kommutativer Kokegel von dem Pushout aus dem vorherigen Schritt zu dem kommutativen Kokegel in Abb. 53e mit Pfeilen $c_1 = i_{1...n-1,n+1} \bullet g_{n,1...n-1}$ und $c_2 = i_{n,n-1} \bullet g_{1...n-1,n}$. Auch hier kann man beweisen, dass es sich um einen kommutativen Kokegel handelt.

Hierdurch ist der gesuchte Span $S_{1...n,n+1}$ konstruiert und v kann mit f wie in Abb. 52 verfeinert werden.

Beispiel 95 (Featurekomposition)

Featurekomposition

Sei eine Produktlinie mit dem Domänenmodell in Abb. 54 gegeben. Die Produktvariante base soll um das Feature weights zu der Produktvariante base•weights verfeinert werden.

Das formalisierte Domänenmodell enthält einen Schnittstellengraph $I_{\text{base,weights}}$. Die Featurekomposition wird durch einen Pushout über dem Span $S_{\text{base,weights}}$ beschrieben, der in Abb. 55 illustriert ist. Die erzeugte Produktvariante base•weights wird durch den Scheitelgraph $G_{\text{base}\bullet\text{weights}}$ des Pushout repräsentiert. □

Bewertung

Durch die in diesem Abschnitt gegebene Beschreibung der Featurekomposition wird die schrittweise Verfeinerung von Produktvarianten formalisiert. Die Beschreibung ist repräsentationsunabhängig, da sie auf beliebige Kategorien angewendet werden kann, in denen die benutzten Limites und Kolimites existieren. Sie erfasst die in 2.3 identifizierte Eigenschaft, ob eine Produktvariante ein Feature enthält, und die Verfeinerungsbeziehung zwischen Produktvarianten.

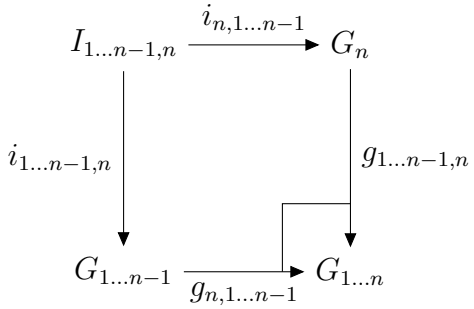
Die Produktlinie, in der die Featurekomposition stattfindet, wurde formalisiert, aber nicht auf kategorientheoretische Weise. Das Domänenmodell der Produktlinie wurde dabei in einer vereinfachten Form berücksichtigt, in der alle Features optional sind und Features sich paarweise ausschließen können.

Die Spans für die Komposition müssen bei der hier gezeigten Vorgehensweise extern geliefert werden. Sie ergeben sich aus den Binderegeln der Programmiersprache und der Kompositionsreihenfolge der Features.

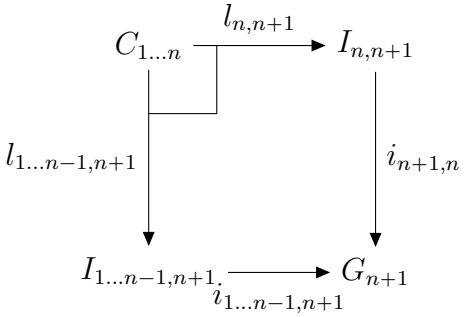
Die Reihenfolge, in der die Pushouts zur Featurekomposition ausgeführt werden, spielt keine Rolle. Wenn sie in der zu repräsentierenden Programmiersprache relevant ist, muss sie deshalb in den Graphen mit ausgedrückt werden. Formal kann man hierzu beispielsweise ein künstliches, für die jeweilige Featureauswahl generiertes Featuremodul hinzufügen, das die gewünschte Reihenfolge festlegt. Alternativ könnte eine auch eine Integration des Domänenmodells als Graphfeature nützlich sein, solange es nicht gewünscht ist, dass eine Programmmanipulation das Domänenmodell ändert.

5.3. Zusammenfassung

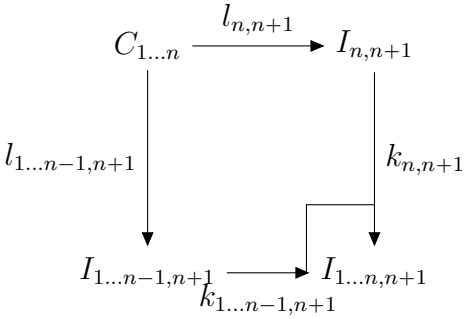
In diesem Kapitel wurden der Merge und die Featurekomposition formal beschrieben und jeweils anhand eines Beispiels erläutert. Dabei wurde beim Merge nur der konfliktfreie



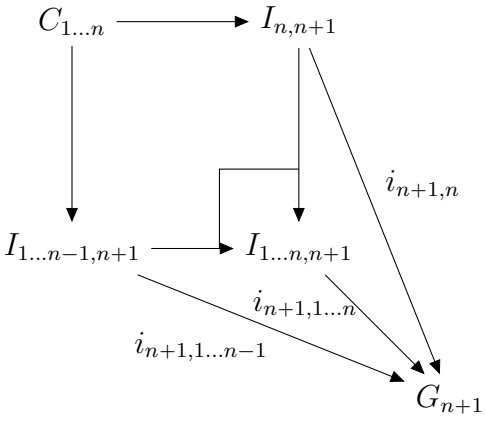
(a) Komponiere Programmvariante v



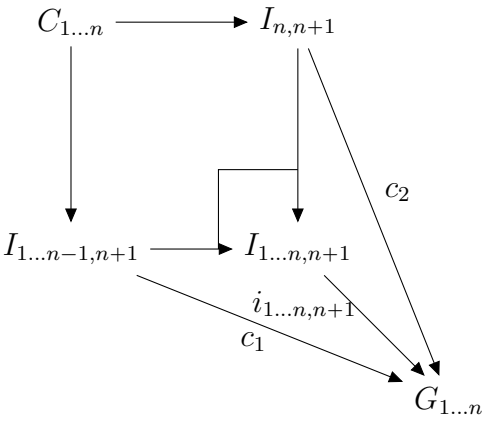
(b) Gemeinsamkeiten der Schnittstellengraphen konstruieren



(c) Gesuchten Schnittstellengraphen konstruieren



(d) Ersten Pfeil des Span konstruieren



(e) Zweiten Pfeil des Span konstruieren

Abbildung 53: Rekursive Konstruktion des Spans $S_{1...n,n+1}$

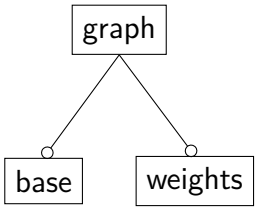


Abbildung 54: Domänenmodell

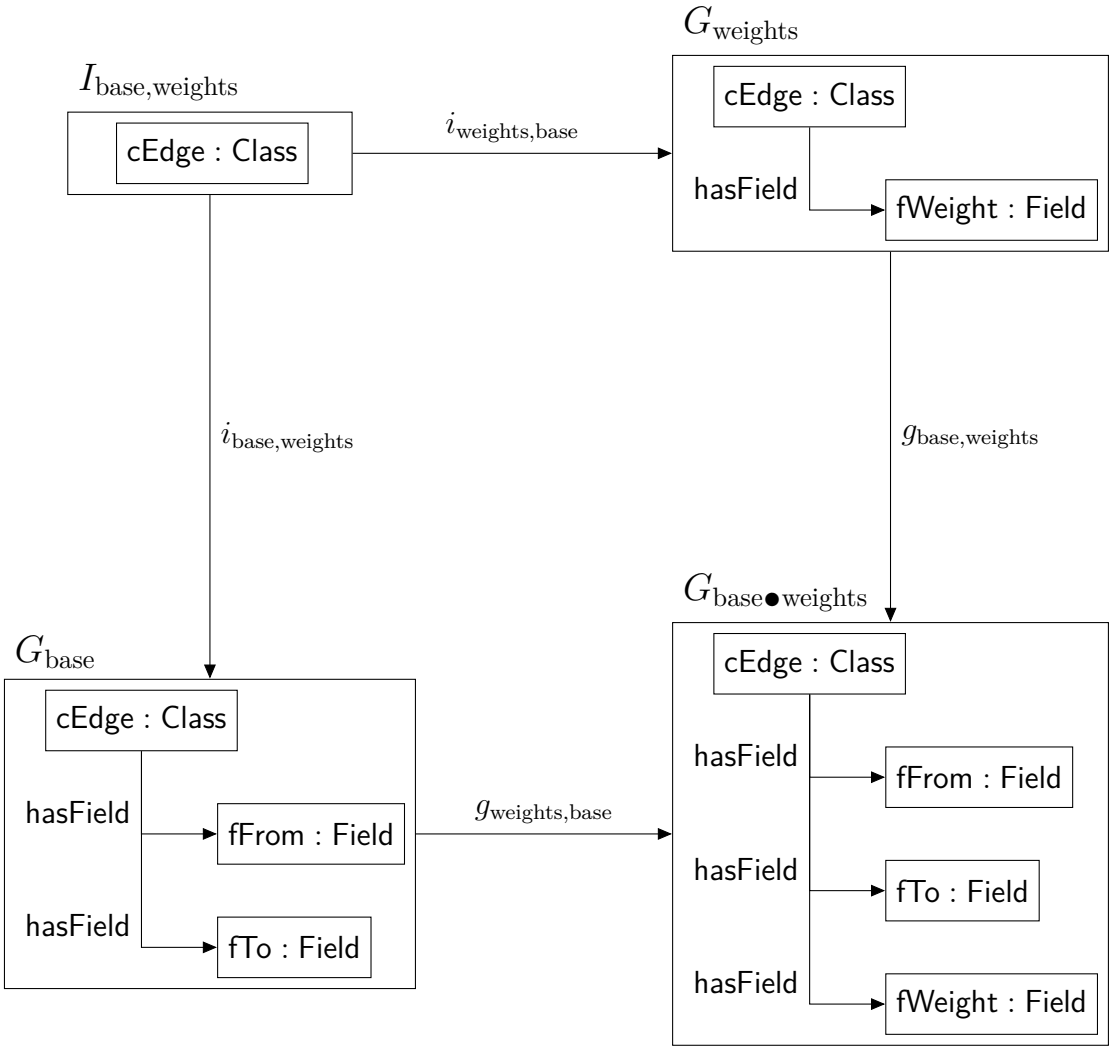


Abbildung 55: Verfeinerung des Produkts base um das Feature weights

Fall betrachtet, aber es existiert auch für die Konfliktbehandlung schon eine kategorientheoretische Beschreibung. Für die Featurekomposition wurde eine vereinfachte Art von Domänenmodell betrachtet.

Kapitel 6

Zusammenfassung und Ausblick

6.1. Zusammenfassung

In dieser Arbeit wurden formale Grundlagen für die implementierungs-, sprach- und repräsentationsunabhängige Beschreibung von Programmmanipulationen untersucht.

Als Grundlage hierfür wurde eine Auswahl von Programmmanipulationen untersucht, nämlich Diff, Merge, Featurekomposition, Interaktionsanalyse und Refactoring. Dabei wurde ein informelles Schema entwickelt, mit dem Programmmanipulationen anhand der Merkmale Programmrepräsentation, -eigenschaften, -beziehungen und Operationen charakterisiert werden können. Dieses Schema hat sich für die systematische Konzeption des Formalismus als sehr nützlich erwiesen.

Aufgrund des Schemas wurde die formale Umsetzung der ersten drei Merkmale mit den Mitteln der Kategorientheorie untersucht. Hierzu wurden für die Programmrepräsentation relevante Graphvarianten aus der algebraischen Graphtransformationstheorie analysiert. Es wurde vorgeschlagen, einzelne charakteristische Eigenschaften von Graphvarianten als Graphfeatures aufzufassen. Als Richtschnur für die Entscheidung, welche Graphfeatures für die Programmrepräsentation relevant sind, wurde das auf dem Modellierungsframework EMF aufbauende Werkzeug EMFText genutzt. Hierfür wurden die Modellierungselemente der EMF-Modellierungssprache Ecore identifiziert, die EMFText verwendet. Weiterhin wurden Graphfeatures vorgestellt, mit denen man die so ausgewählten Modellierungselemente von Ecore formal umsetzen kann, nämlich Typisierung, Attributierung, Containment, Ordnung, Vererbung und Constraints. Dabei wurde festgestellt, dass typisierte attributierte Graphen für die Repräsentation der statischen Semantik von Programmen ausreichen.

Als formale Grundlage für die Umsetzung von Programmrepräsentation, -eigenschaften und -beziehungen wurden Grundbegriffe der Kategorientheorie eingeführt, insbesondere auch Limites und Kolimites. Zur formalen Umsetzung der Graphfeatures wurden Sketches vorgestellt. Mit Sketches kann man Graphvarianten auf flexible Weise modular definieren, indem man sie durch Pushouts miteinander kombiniert. Hierdurch sind Definitionen komplexer Graphvarianten besser handhabbar als bei der in der algebraischen Graphtransformationstheorie üblichen Vorgehensweise, auch komplexe Graphvarianten als Tupel zu definieren. Außerdem wurden dabei typisierte Sketches definiert, die die Definition sprachunabhängiger Programmrepräsentationen unterstützen sollen, indem man für jede Programmiersprache eine eigene Typinstanz verwenden kann. Im Rahmen dieser Arbeit wurden Typisierung und Containment als Sketches formalisiert. Die Formalisierung von Containment wurde gegenüber der Attributierung vorgezogen, damit man in darauf aufbauenden Arbeiten die durch die Baumstruktur variierende Komplexität genauer untersuchen kann.

Zur Formalisierung von Programmeigenschaften und -beziehungen wurde die Verwendung kategorieller Mittel vorgeschlagen. Insbesondere sind hierfür neben den Pfeilen der

zur Programmrepräsentation verwendeten Kategorien auch Limites und Kolimites gedacht. Die Verwendung der Kategorientheorie hat vor allem dazu beigetragen, dass die Programmmanipulationen repräsentationsunabhängig beschrieben werden konnten. Es wurde die Formalisierung des Merge nach [Tae+10] im Rahmen der Programmmanipulation vorgestellt und eine Formalisierung der Featurekomposition durch schrittweise Verfeinerung entwickelt. Beim Merge wurde die Konfliktbehandlung nicht berücksichtigt. Eine Konflikterkennung nach [Tae+10] kann mit Pullbacks und Pushouts ausgedrückt werden. Die Behandlung von Konflikten nach [EET11] erfordert sogenannte initiale Pushouts. Da diese keine (Ko-)Limites sind, müsste untersucht werden, wie gut sie sich in die Konzeption des Formalismus fügen. Für die Featurekomposition wäre es wünschenswert auch das Domänenmodell kategorientheoretisch und mit der Beschreibung der Featurekomposition integriert zu formulieren. Beim Merge wie bei der Featurekomposition wird vorausgesetzt, dass außerhalb des Formalismus berechnete Schnittstellengraphen gegeben sind, die die Gemeinamkeiten zwischen Programmen erfassen. Eine Alternative zu den Schnittstellengraphen wäre es, die Binderegeln der jeweiligen Programmiersprache in den Typgraphen etwa durch Constraints auszudrücken.

6.2. Ausblick

In dieser Arbeit wurden Grundlagen für die Formalisierung von Programmmanipulationen untersucht, auf denen in zukünftigen Arbeiten aufgebaut werden kann.

Programmmanipulationen

Im Rahmen dieser Arbeit wurden die Programmanipulationen Diff, Refactoring und Interaktionsanalyse untersucht (siehe Kapitel 2), aber nicht formalisiert. Für Diff liegt bereits in [Tae+10] eine geeignete Formalisierung vor, wo es als *minimal graph rule* bezeichnet wird. Darüber hinaus könnten auch andere Programmmanipulationen formalisiert werden, wie etwa die Delta-Komposition [Sch+10; SD10] und Aspektweben [Kic+97] aus dem Bereich der generativen Programmierung.

Für einen Teil der Programmmanipulationen sind möglicherweise neben den hier verwendeten Limites und Kolimites weitere kategorielle Ausdrucksmittel zur Beschreibung von Programmeigenschaften und -beziehungen notwendig. Zum einen könnte man als Erweiterung zu den Limites und Kolimites allgemeinere Arten von universellen Konstruktionen benutzen (vgl. Abschnitt 4.1). Zum anderen könnten beispielsweise Funktoren für die Beschreibung von Refactorings (siehe Abschnitt 2.5) nützlich sein, um Verhaltensäquivalenz auszudrücken.

Programmrepräsentation

In Abschnitt 3.1 wurde EMFText/Ecore als Grundlage für graphbasierte Programmrepräsentationen untersucht. Eine Schwäche von Ecore ist die Repräsentation impliziter Informationen und Zusammenhänge in Modellen. Die in Unterabschnitt 3.2.7 untersuchten Constraints decken die Repräsentation impliziter Zusammenhänge teils ab, eignen sich in der bisherigen Form aber nicht für die Berechnung impliziter Informationen. Man könnte deshalb beispielsweise Graphanfragetechniken wie GReQL [KW99] untersuchen, um diese Schwäche auszugleichen.

Graphfeatures

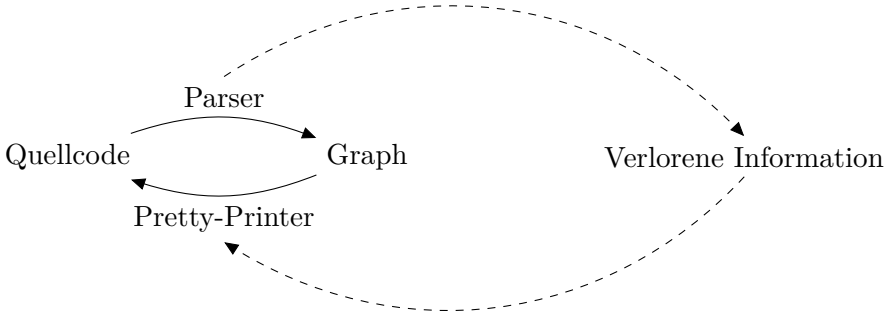
Als Ergänzung könnte man die in Abschnitt 3.2 vorgestellten Graphfeatures Attributierung, Ordnung, Vererbung und Constraints formalisieren. Die in Unterabschnitt 4.3.3 vorgestellte Formalisierung von Containment hat gute mathematische Eigenschaften, ist aber unflexibel, da Homomorphismen Knoten unterschiedlicher Containment-Tiefe nicht aufeinander abbilden können. Dies ließe sich möglicherweise durch eine Art von Pfadtypen für Knoten und Containmentkanten verbessern, die XPath-ähnlichen Pfadausdrücken [CD99] entsprechen würden. Zur Modellierung der Binderegeln von Programmiersprachen wurden hier Constraints vorgeschlagen. Man könnte aber auch versuchen, diese unmittelbar als Graphfeature umzusetzen. Zur Formalisierung zusätzlicher Graphfeatures könnten Erweiterungen der in Abschnitt 4.2 vorgestellten Sketcharten notwendig sein, insbesondere bei Constraints. Es existieren bereits einige Verallgemeinerungen von Sketches, die hierfür nützlich sein könnten, etwa [BW08; Dis03; Rut10; Lai87; Mak97]. Um weitere Programmrepräsentationen zu untersuchen, wäre es sinnvoll, formale Kriterien für die Eignung von Kategorien zur Programmrepräsentation zu finden. Möglicherweise können hier Kriterien aus dem Bereich des High-Level-Rewriting (HLR) [Ehr+06] nützlich sein.

Mapping zwischen Programmrepräsentationen

In dieser Arbeit wurde nur jeweils eine Programmrepräsentation für sich betrachtet. Es wäre aber durchaus wünschenswert, dass man auf ein Programm, das in einer Programmrepräsentation dargestellt ist, eine Programmmanipulation anzuwenden, die eine andere Programmrepräsentation benutzt. Hierfür wäre eine Art Mapping zwischen verschiedenen Programmrepräsentationen nötig. Formal könnte ein solches Mapping vielleicht durch Funktoren und Adjunktionen beschrieben werden [BW12]. Bei Adjunktionen handelt es sich um eine Verallgemeinerung von Galois-Verbindungen, die auch im Bereich der abstrakten Interpretation eine Rolle spielen [CC92].

Programmiersprachen einbinden

Mit dieser Fragestellung hängt auch der Übergang von Quelltext eines Programmes in eine Instanz eines Sketches zur Programmrepräsentation zusammen. Durch die Verwendung typisierter Sketches zur Programmrepräsentation ist der Formalismus sprachunabhängig, da man für jede Programmiersprache eine eigene Typinstanz des verwendeten typisierten Sketches definieren kann (vgl. Abschnitt 4.2). Das Einbinden von Programmiersprachen wurde aber nicht formalisiert. Hierzu müsste die Beziehung zwischen der Grammatik einer Programmiersprache und Typfragmenten von typisierten Sketches untersucht werden. Dadurch würde sich eine Art Formalisierung von EMFText [Hei+09] ergeben. In diesem Zusammenhang könnte man auch Techniken untersuchen, um Informationen zu erhalten, die durch die Abstraktion beim Parsen in der Programmrepräsentation nicht mehr vorhanden sind, wie etwa die Formatierung oder Kommentare, wie die folgende Abbildung illustriert:



Softwaretechnische Umsetzung des Formalismus

Ein wichtiger zukünftiger Schritt wäre eine softwaretechnische Umsetzung des Formalismus: die Implementierung eines Frameworks auf Basis des Formalismus zur Beschreibung und Ausführung von Programmanipulationen bzw. Generierung von Implementierungen davon. Dazu müssten die vier Merkmale von Programmmanipulationen umgesetzt werden:

- Für die Repräsentation müsste man eine Möglichkeit zur Definition von Programmrepräsentationen durch typisierte Sketches schaffen. Besondere Bedeutung kommt dabei der Kombination von Sketches durch Pushouts zu.
- Eigenschaften und Beziehungen könnte man mit einer deklarativen visuelle Sprache zur Beschreibung kategorientheoretischer Konstruktionen umsetzen.
- Die Operationen, also die Ausführung der Programmmanipulationen, würden durch Algorithmen zur Berechnung der so beschriebenen kategorientheoretischen Konstruktionen implementiert.

Da die Operationen in dieser Arbeit nicht betrachtet wurden, wäre eine Untersuchung zu bestehenden Ergebnissen über die Berechenbarkeit und Komplexität von kategorientheoretischen Konstruktionen eine wichtige Voraussetzung für die softwaretechnische Umsetzung. Weiterhin müsste ein Mechanismus zum Einbinden von Programmiersprachen in das Framework implementiert werden, um auf darin geschriebenen Programmen Programmmanipulationen ausführen zu können. Das Ziel wäre, ausgehend von der Grammatik einer Programmiersprache, der sketchbasierten Definition einer Programmrepräsentation und einer Zuordnung zwischen Grammatik und Repräsentation eine Typinstanz zu generieren. Ebenso könnten die weiter oben beschriebenen Mappings zwischen Programmrepräsentationen implementiert werden, um das Framework allgemeiner einsetzbar zu machen.

Literatur

[Aho+06] Alfred V. Aho u. a. *Compilers: principles, techniques, and tools*. Addison-Wesley series in computer science. Prentice Hall, 2006. ISBN: 9780321486813. URL: http://www.pearsonhighered.com/educator/product/Compilers-Principles-Techniques-and-Tools/9780321486813.page (siehe S. 37).

[AHS90] Jiří Adámek, Horst Herrlich und George E. Strecker. *Abstract and concrete categories: the joy of cats*. Wiley, 1990. URL: http://www.getcited.org/pub/102780944 (siehe S. 153).

[AKL12] Sven Apel, Christian Kästner und Christian Lengauer. „Language-Independent and Automated Software Composition: The FEATUREHOUSE Experience“. In: *Database* 1 (2012), S. 1–16 (siehe S. 10).

[AL08] Sven Apel und Christian Lengauer. „Superimposition: A Language-Independent Approach to Software Composition“. In: *Software Composition*. Hrsg. von Cesare Pautasso und Éric Tanter. Bd. 4954. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, S. 20–35. ISBN: 978-3-540-78788-4. DOI: 10.1007/978-3-540-78789-1_2 (siehe S. 18).

[Ape+10] Sven Apel u. a. „Language-independent reference checking in software product lines“. In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development - FOSD '10*. New York, New York, USA: ACM Press, 2010, S. 65–71. ISBN: 9781450302081. DOI: 10.1145/1868688.1868698. URL: http://portal.acm.org/citation.cfm?doid=1868688.1868698 (siehe S. 9, 19–21).

[Ape+11] Sven Apel u. a. „Semistructured merge: rethinking merge in revision control systems“. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*. New York, New York, USA: ACM Press, 2011, S. 190. ISBN: 9781450304436. DOI: 10.1145/2025113.2025141. URL: http://dl.acm.org/citation.cfm?doid=2025113.2025141 (siehe S. 16).

[BET08] Enrico Biermann, Claudia Ermel und Gabriele Taentzer. „Precise Semantics of EMF Model Transformations by Graph Transformation“. In: *Model Driven Engineering Languages and Systems*. Hrsg. von Krzysztof Czarnecki u. a. Bd. 5301. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, S. 53–67. ISBN: 978-3-540-87874-2. DOI: 10.1007/978-3-540-87875-9_4 (siehe S. 38).

[BSR04] Don Batory, Jacob Neal Sarvela und Axel Rauschmayer. „Scaling stepwise refinement“. In: *IEEE Transactions on Software Engineering* 30.6 (Juni 2004), S. 355–371. ISSN: 0098-5589. DOI: 10.1109/TSE.2004.23. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1321059 (siehe S. 9, 18, 135).

[Buf95] Jim Buffenbarger. „Syntactic software merging“. In: *Software Configuration Management*. Hrsg. von Jacky Estublier. Bd. 1005. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1995, S. 153–172. ISBN: 978-3-540-60578-2. DOI: 10.1007/3-540-60578-9_14 (siehe S. 9, 16).

[BW08] Atish Bagchi und Charles Wells. *Graph-based Logic and Sketches*. Sep. 2008. arXiv: 0809.3023. URL: http://arxiv.org/abs/0809.3023 (siehe S. 143).

[BW12] Michael Barr und Charles Wells. „Category Theory for Computing Science“. In: *Reprints in Theory and Applications of Categories* 22 (2012), S. 1–538. URL: http://www.tac.mta.ca/tac/reprints/articles/22/tr22.pdf (siehe S. 10, 25, 36, 45, 67, 69, 87, 95, 129, 143).

[CC77] Patrick Cousot und Radhia Cousot. „Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints“. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '77*. New York, New York, USA: ACM Press, 1977, S. 238–252. DOI: 10.1145/512950.512973. URL: http://portal.acm.org/citation.cfm?doid=512950.512973 (siehe S. 25).

[CC92] Patrick Cousot und Radhia Cousot. „Abstract Interpretation Frameworks“. In: *Journal of Logic and Computation* 2.4 (1992), S. 511–547. ISSN: 0955-792X. DOI: 10.1093/logcom/2.4.511. URL: http://logcom.oxfordjournals.org/cgi/doi/10.1093/logcom/2.4.511 (siehe S. 25, 143).

[CD99] James Clark und Steve DeRose. *XML Path Language (XPath) Version 1.0*. Nov. 1999. URL: http://www.w3.org/TR/1999/REC-xpath-19991116/ (siehe S. 143).

[CNL79] Roderic G.G. Cattell, Joseph M. Newcomer und Bruce W. Leverett. „Code generation in a machine-independent compiler“. In: *ACM SIGPLAN Notices* 14.8 (Aug. 1979), S. 65–75. ISSN: 03621340. DOI: 10.1145/872732.806955. URL: http://portal.acm.org/citation.cfm?doid=872732.806955 (siehe S. 20).

[Dis03] Zinovy Diskin. „Variable set semantics for keyed generalized sketches: formal semantics for object identity and abstract syntax for conceptual modeling“. In: *Data and Knowledge Engineering* 47.1 (Okt. 2003), S. 1–59. ISSN: 0169023X. DOI: 10.1016/S0169-023X(03)00047-8. URL: http://linkinghub.elsevier.com/retrieve/pii/S0169023X03000478 (siehe S. 143).

[EET11] Hartmut Ehrig, Claudia Ermel und Gabriele Taentzer. „A formal resolution strategy for operation-based conflicts in model versioning using graph modifications“. In: *Fundamental Approaches to Software Engineering* (2011), S. 202–216. DOI: 10.1007/978-3-642-19811-3_15. URL: http://www.springerlink.com/index/Y722H2QP020L4217.pdf (siehe S. 131, 142).

[Ehr+01] Hartmut Ehrig u. a. *Mathematisch-strukturelle Grundlagen der Informatik*. 2. Aufl. Springer, 2001. ISBN: 3-540-41923-3 (siehe S. 36).

[Ehr+06] Hartmut Ehrig u. a. *Fundamentals of Algebraic Graph Transformation*. 1. Aufl. Monographs in Theoretical Computer Science. An EATCS Series. Springer, März 2006, S. 5–20. ISBN: 3540311874. DOI: 10.1007/3-540-31188-2_1 (siehe S. 25, 43, 129, 143).

[EM85] Hartmut Ehrig und Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer-Verlag, 1985. ISBN: 0387137181. URL: http://www.amazon.com/exec/obidos/redirect?path=ASIN/0387137181 (siehe S. 36).

[EPT04] Hartmut Ehrig, Ulrike Prange und Gabriele Taentzer. „Fundamental Theory for Typed Attributed Graph Transformation“. In: *Graph Transformations*. Hrsg. von Hartmut Ehrig u. a. Bd. 3256. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, S. 161–177. DOI: 10.1007/978-3-540-30203-2_13 (siehe S. 129).

[FB99] Martin Fowler und Kent Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999 (siehe S. 9, 21).

[Fia05] José Luiz Fiadeiro. *Categories for Software Engineering*. Berlin/Heidelberg: Springer-Verlag, 2005. ISBN: 978-3-540-26891-8. DOI: 10.1007/b138249. URL: http://www.springerlink.com/index/10.1007/b138249 (siehe S. 45).

[Hei+09] Florian Heidenreich u. a. „Derivation and refinement of textual syntax for models“. In: *Model Driven Architecture-Foundations and Applications*. Hrsg. von Richard Paige, Alan Hartman und Arend Rensink. Bd. 5562. Lecture Notes in Computer Science. Springer, 2009, S. 114–129. ISBN: 978-3-642-02673-7. DOI: 10.1007/978-3-642-02674-4_9. URL: http://www.springerlink.com/content/w6r72r6631761206 (siehe S. 25, 26, 28, 144).

[Hei+10] Florian Heidenreich u. a. „Closing the Gap between Modelling and Java“. In: *Software Language Engineering*. Hrsg. von Mark van den Brand, Dragan Gaševic und Jeff Gray. Bd. 5969. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, S. 374–383. DOI: 10.1007/978-3-642-12107-4_25. URL: http://reuseware.org/publications/2009_SLE_JaMoPP.pdf (siehe S. 26, 28).

[Jac09] Judah Jacobson. *A formalization of Darcs patch theory using inverse semigroups*. Techn. Ber. UCLA, 2009 (siehe S. 15).

[Kan+90] Kyo C. Kang u. a. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Techn. Ber. Software Engineering Institute, Carnegie Mellon University, 1990. URL: http://www.sei.cmu.edu/library/abstracts/reports/90tr021.cfm (siehe S. 18).

[Kic+97] Gregor Kiczales u. a. „Aspect-oriented programming“. In: *ECOOP'97 — Object-Oriented Programming*. Hrsg. von Mehmet Aksit und Satoshi Matsuoka. Bd. 1241. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1997, S. 220–242. ISBN: 978-3-540-63089-0. URL: http://dx.doi.org/10.1007/BFb0053381 (siehe S. 143).

[KKP07] Sanjeev Khanna, Keshav Kunal und Benjamin C. Pierce. „A Formal Investigation of Diff3“. In: *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*. Hrsg. von V Arvind und Sanjiva Prasad. Bd. 4855. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007, S. 485–496. DOI: 10.1007/978-3-540-77050-3_40 (siehe S. 15).

[KLT07] Christian Koehler, Holger Lewin und Gabriele Taentzer. „Ensuring Containment Constraints in Graph-based Model Transformation Approaches“. In: *ECEASST*. Hrsg. von Andrew Fish und Harald Störrle. Bd. 6. Electronic Communications of the EASST. 2007. URL: http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/165/166 (siehe S. 38).

[KW99] B. Kullbach und A. Winter. „Querying as an enabling technology in software reengineering“. In: *Proceedings of the Third European Conference on Software Maintenance and Reengineering (Cat. No. PR00090)*. IEEE Comput. Soc, 1999, S. 42–50. ISBN: 0-7695-0090-0. DOI: 10.1109/CSMR.1999.756681. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=756681 (siehe S. 143).

[LA04] C. Lattner und V. Adve. „LLVM: A compilation framework for lifelong program analysis & transformation“. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, S. 75–86. ISBN: 0-7695-2102-9. DOI: 10.1109/CGO.2004.1281665. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1281665 (siehe S. 9).

[Lai87] C. Lair. „Trames et sémantiques catégoriques des systèmes de trames“. In: *Diagrammes* 18.1 (1987) (siehe S. 143).

[Lar+05] Juan de Lara u. a. *Attributed Graph Transformation with Node Type Inheritance: Long Version*. Techn. Ber. TU Berlin, 2005. URL: http://iv.tu-berlin.de/TechnBerichte/2005/2005-3.pdf (siehe S. 41).

[Lar+07] Juan de Lara u. a. „Attributed graph transformation with node type inheritance“. In: *Theoretical Computer Science* 376.3 (2007), S. 139–163. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2007.02.001. URL: http://www.sciencedirect.com/science/article/B6V1G-4N02J7J-3/2/e67ac0ee0f4125fa04aae385ba913f9b (siehe S. 36).

[Leß12] Olaf Leßenich. „Adjustable Syntactic Merge of Java Programs“. Master's thesis. University of Passau, 2012. URL: http://www.infosun.fim.uni-passau.de/spl/apel/theses/OlafLessenichMA.pdf (siehe S. 10, 16, 39).

[Lév59] Azriel Lévy. „On Ackermann's Set Theory“. In: *The Journal of Symbolic Logic* 24.2 (1959), S. 154–166 (siehe S. 153).

[LV61] Azriel Lévy und Robert Vaught. „Principles of partial reflection in the set theories of Zermelo and Ackermann“. In: *Pacific Journal of Mathematics* 11.3 (1961), S. 1045–1062 (siehe S. 153).

[Lyn10] Ian Lynagh. „Camp Patch Theory“. 2010. URL: http://projects.haskell.org/camp/files/theory.pdf (siehe S. 15).

[Mak97] M. Makkai. „Generalized sketches as a framework for completeness theorems. Part I“. In: *Journal of Pure and Applied Algebra* 115.1 (Feb. 1997), S. 49–79. ISSN: 00224049. DOI: 10.1016/S0022-4049(96)00007-2. URL: http://linkinghub.elsevier.com/retrieve/pii/S0022404996000072 (siehe S. 143).

[MM85] Webb Miller und Eugene W. Myers. „A file comparison program“. In: *Software: Practice and Experience* 15.11 (Nov. 1985), S. 1025–1040. ISSN: 00380644. DOI: 10.1002/spe.4380151102. URL: http://doi.wiley.com/10.1002/spe.4380151102 (siehe S. 14).

[MR10] Maarten de Mol und Arend Rensink. „On A Graph Formalism for Ordered Edges“. In: *Proceedings of the Ninth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2010), Paphos, Cyprus*. Hrsg. von J Küster und E Tuosto. Bd. 29. Electronic Communications of the EASST. The European Association for the Study of Science und Technology, Aug. 2010 (siehe S. 40).

[Mul01] F. A. Muller. „Sets, Classes, and Categories“. In: *Brit. J. Phil. Sci.* 52 (2001), S. 539–573 (siehe S. 52, 153).

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002. ISBN: 978-0262162098 (siehe S. 9).

[Pre97] Christian Prehofer. „Feature-oriented programming: A fresh look at objects“. In: *ECOOP'97 — Object-Oriented Programming*. Hrsg. von Mehmet Aksit und Satoshi Matsuoka. Bd. 1241. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1997, S. 419–443. ISBN: 978-3-540-63089-0. DOI: 10.1007/BFb0053389 (siehe S. 18).

[Rag+04] Shruti Raghavan u. a. „Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases“. In: *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, S. 188–197. ISBN: 0-7695-2213-0. DOI: 10.1109/ICSM.2004.1357803. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1357803 (siehe S. 25, 37, 38).

[Rag03] Shruti Raghavan. „Finding the Differences in Program Code Through Abstract Semantic Graph Comparison“. Master's Thesis. Case Western Reserve University, 2003. ISBN: 978-0-7695-3793-1. URL: http://web.archive.org/web/20050210004901/http://softlab4.cwru.edu/dex/RaghavanDEXReport.pdf (siehe S. 25, 37–39).

[Rut+09] Adrian Rutle u. a. „A Category-Theoretical Approach to the Formalisation of Version Control in MDE“. In: *Fundamental Approaches to Software Engineering*. Hrsg. von Marsha Chechik und Martin Wirsing. Bd. 5503. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, S. 64–78. DOI: 10.1007/978-3-642-00593-0_5 (siehe S. 10, 15, 25, 131).

[Rut10] Adrian Rutle. „Diagram predicate framework: A formal approach to MDE“. Diss. The University of Bergen, 2010 (siehe S. 143).

[RW91] David S. Rosenblum und Alexander L. Wolf. „Representing Semantically Analyzed C++ Code with REPRISE“. In: *Proc. Third USENIX C++ Conference*. 1991, S. 119–134 (siehe S. 25, 36).

[RWL08] Adrian Rutle, Uwe Egbert Wolter und Yngve Lamo. „Generalized Sketches and Model Driven Architecture“. In: *Reports in Informatics* 367 (2008), S. 45–59. ISSN: 0333-3590 (siehe S. 88).

[Sch+10] Ina Schaefer u. a. „Delta-Oriented Programming of Software Product Lines“. In: *Software Product Lines: Going Beyond*. Hrsg. von Jan Bosch und Jaejoon Lee. Bd. 6287. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, S. 77–91. DOI: 10.1007/978-3-642-15579-6_6. URL: http://dx.doi.org/10.1007/978-3-642-15579-6_6 (siehe S. 143).

[SD10] Ina Schaefer und Ferruccio Damiani. „Pure delta-oriented programming“. In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development - FOSD '10*. New York, New York, USA: ACM Press, 2010, S. 49–56. ISBN: 9781450302081. DOI: 10.1145/1868688.1868696. URL: http://portal.acm.org/citation.cfm?doid=1868688.1868696 (siehe S. 143).

[SKP11] Friedrich Steimann, Christian Kollee und Jens von Pilgrim. „A Refactoring Constraint Language and its Application to Eiffel“. In: *ECOOP 2009 ObjectOriented Programming*. Section 3. FU Hagen. Springer, 2011. DOI: 10.1007/978-3-642-22655-7_13. URL: http://www.fernuni-hagen.de/ps/veroeffentlichungen/ECOOP2011.shtml (siehe S. 10).

[SP11] Friedrich Steimann und Jens von Pilgrim. *Constraint-Based Refactoring with Foresight*. Techn. Ber. Fernuniversität in Hagen, 2011 (siehe S. 10).

[ST09] Friedrich Steimann und Andreas Thies. „From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility“. In: *ECOOP 2009 ObjectOriented Programming*. Lecture Notes in Computer Science 5653 (2009). Hrsg. von Sophia Drossopoulou, S. 419–443. DOI: 10.1007/978-3-642-03013-0. URL: http://www.springerlink.com/content/8pk578k101un8k04/ (siehe S. 10, 21, 22).

[Ste+08] Dave Steinberg u. a. *EMF: Eclipse Modeling Framework (2nd Edition)*. 2. Aufl. Addison-Wesley Professional, Dez. 2008. ISBN: 0321331885. URL: http://www.worldcat.org/isbn/0321331885 (siehe S. 9, 25, 26).

[Tae+10] Gabriele Taentzer u. a. „Conflict Detection for Model Versioning Based on Graph Modifications“. In: *Graph Transformations*. Hrsg. von Hartmut Ehrig u. a. Bd. 6372. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, S. 171–186. ISBN: 978-3-642-15927-5. DOI: 10.1007/978-3-642-15928-2_12. URL: http://www.springerlink.com/index/VU72W565867K3716.pdf (siehe S. 10, 15, 25, 131, 142, 143).

[Tae+12] Gabriele Taentzer u. a. „A fundamental approach to model versioning based on graph modifications: from theory to implementation“. In: *Software and Systems Modeling* (2012). ISSN: 1619-1366. DOI: 10.1007/s10270-012-0248-x. URL: http://www.springerlink.com/index/10.1007/s10270-012-0248-x (siehe S. 15).

[TR05] Gabriele Taentzer und Arend Rensink. „Ensuring Structural Constraints in Graph-Based Models with Type Inheritance“. In: *Fundamental Approaches to Software Engineering*. Hrsg. von Maura Cerioli. Bd. 3442. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, S. 64–79. DOI: 10.1007/978-3-540-31984-9_6 (siehe S. 43).

[Tsa93] Edward P. K. Tsang. *Foundations of Constraint Satisfaction*. 1993. URL: http://en.scientificcommons.org/43256499 (siehe S. 22).

[Wes91] Bernhard Westfechtel. „Structure-oriented merging of revisions of software documents“. In: *Proceedings of the 3rd international workshop on Software configuration management* -. New York, New York, USA: ACM Press, 1991, S. 68–79. ISBN: 08979144295. DOI: 10.1145/111062.111071. URL: http://portal.acm.org/citation.cfm?doid=111062.111071 (siehe S. 16).

Anhang A

Grunddefinitionen

In diesem Anhang werden die mathematischen Notationen und Grundbegriffe festgelegt, die zur Beschreibung des Formalismus verwendet werden.

Anforderungen an die Mengenlehre

Um die verschiedenen Graphvarianten des Formalismus zu beschreiben, werden Kategorien wie etwa die Kategorie **Set** der Mengen oder die Kategorie **Graphs** der Graphen (siehe Unterabschnitt 3.3.1) verwendet. Um beispielsweise **Set** definieren zu können, muss man die Gesamtheit aller Mengen benennen können, da die Mengen die Objekte dieser Kategorie sind. Anders formuliert: Wenn man nicht "*die Mengen*" sagen kann, kann man auch nicht sagen: "Die Objekte von **Set** sind *die Mengen*".

Eine weitere Anforderung für saubere formale Definitionen ist es, von möglichst allen betrachteten Gesamtheiten auch endliche Tupel und Funktionen bilden zu können, etwa um Kategorien formal korrekt als mathematisches Objekt benennen zu können. Eine detailliertere, allgemeinere Auseinandersetzung mit den Anforderungen an eine Mengentheorie, die als Grundlage für die Kategorientheorie im Speziellen und die Mathematik im Allgemeinen dient, liefert [Mul01].

Klassen und Mengen

Die übliche axiomatische Mengenlehre (ZFC) erfüllt beide Anforderungen nicht, denn sie kennt als Gesamtheiten ausschließlich Mengen: Da es keine "Menge aller Mengen" gibt, ist es nicht möglich, die Gesamtheit aller Mengen zu benennen, weshalb man die Kategorie der Mengen in ZFC nicht definieren kann. Ebenso existiert keine "Menge aller Graphen", weshalb man die Kategorie der Graphen in ZFC nicht definieren kann. Man kann dieses Problem umgehen, indem man in einer erweiterten Mengenlehre zwischen zwei Arten von Gesamtheiten unterscheidet. Die eine Art nennt man Mengen, die andere nennt man Klassen [AHS90, Kap. 0.1][Mul01].

NBG-Mengenlehre und MK-Mengenlehre

Es gibt zwei bekannte Erweiterungen von ZFC um Klassen, nämlich die Neumann-Bernays-Gödel-Mengenlehre (NBG) und die Morse-Kelley-Mengenlehre (MK). Bei NBG handelt sich um eine konservative Erweiterung von ZFC: Jeder Formel der Sprache von ZFC gilt genau dann in NBG, wenn die in ZFC gilt. MK hingegen ist nicht konservativ, d.h. man kann dort Formeln der Sprache von ZFC beweisen, die in ZFC nicht gelten. Man kann darin Tupel von echten Klassen definieren, aber echte Klassen können nicht in andern Klassen enthalten sein.

Ein Nachteil der NBG- und der MK-Mengenlehre ist, dass echte Klassen nie Elemente anderer Klassen sein können. Man kann zwar eine Klasse $\mathfrak{Set}$ aller Mengen definieren, die alle Teilmengen des mathematischen Universums enthält. Eine Klasse, die alle Teilklassen von $\mathfrak{Set}$ enthält, existiert aber nicht.

Ackermann-Mengenlehre und ARC-Mengenlehre

Die Ackermann-Mengenlehre erlaubt genau dies: Man kann zwar eine Klasse $\mathfrak{Set}$ aller Teilmengen des Universums ebenso definieren, wie eine Klasse $\mathcal{P}(\mathfrak{Set})$ aller Teilklassen von $\mathfrak{Set}$, ebenso $\mathcal{P}(\mathcal{P}(\mathfrak{Set}))$ und so weiter [Mul01; Lév59; LV61]. Tupel, Funktionen und Potenzklassen beliebiger Klassen lassen sich definieren. Dies wird dadurch erreicht, dass

Klassen und Mengen freier definiert werden können als in ZFC, NBG oder MK. Für genauere Ausführungen sei auf die Literatur verwiesen.

Mengen und Klassen

Auf Grundlage der ARC-Mengenlehre kann man die Klasse der Mengen also wie folgt definieren:

Definition 96 (Klasse aller Mengen)

Klasse aller Mengen

$\mathfrak{Set}$ ist die **Klasse aller Mengen**.

Im folgenden werden als Grundbegriffe die (binären) Relationen, davon abgeleitete Relationen, Eigenschaften von Relationen und Funktionen definiert, weil es teils mehrere Defintionsvarianten gibt. Da es in der mathematischen Literatur keine einheitliche Schreibweise zur Definition von Relationen gibt, wird hier eine eigene Schreibweise eingeführt, um die späteren Definitionen kürzer und lesbarer zu machen. Sie ist an die Schreibweise zur Definition von Funktionen $f : M \to N : m \mapsto f(m)$ angelehnt.

Definition 97 (Relation)

Relation

$\mathfrak{Rel} = \{(M, N, G) \mid M, N \in \mathfrak{Set} \wedge G \subseteq M \times N\}$ ist die Klasse der (kleinen) Relationen.

$R = (M, N, G) \in \mathfrak{Rel}$ heißt **Relation** von M nach N mit **Relationsgraph** G.
Man bezeichnet M als die Domäne $src(R)$, N als die Kodomäne $tar(R)$ und G als den Relationsgraph $graph(R)$ von R.

Schreibe dafür

$$R : M \nrightarrow N : G$$

Falls G nicht relevant ist, schreibe dafür

$$R : M \nrightarrow N$$

Schreibe

$$(m, n) \leftrightarrow F_{m,n}$$

für die Menge $\{(m, n) \in M \times N \mid F_{m,n}\}$ mit einer Formel $F_{m,n}$ mit freien Variablen m und n.

Dadurch lässt sich eine Relation definieren durch

$$R : M \nrightarrow N : (m, n) \leftrightarrow F$$

Üblicherweise schreibt man mRn anstatt $(m, n) \in graph(R)$. □

In der Mathematikliteratur umschreibt man eine Definition $R : M \nrightarrow N : (m, n) \leftrightarrow P(m, n)$ üblicherweise mit Worten, wie z.B.: "R ist eine Relation zwischen M und N. Für $m \in M$ und $n \in N$ gilt mRn genau dann, wenn $P(m, n)$."

Definition 98 (Abgeleitete Relationen)
Abgeleitete Relationen

Seien $R, R' : M \nrightarrow N$ Relationen.
Sei $S : N \nrightarrow L$ eine Relation.

$R \bullet S : M \nrightarrow L : (m, l) \leftrightarrow \exists n \in N\colon (mRn \wedge nRl)$ ist die **Komposition** von R und S.
$R \cap R' : M \nrightarrow N : (m, n) \leftrightarrow mRn \wedge mR'n$ ist der **Schnitt** von R und S.
$R \cup R' : M \nrightarrow N : (m, n) \leftrightarrow mRn \vee mR'n$ ist die **Vereinigung** von R und S.
$R^- : N \nrightarrow M : (n, m) \leftrightarrow mRn$ ist die Umkehrrelation von R.
Für Mengen M' und N' ist $R|^{N'}_{M'} : M' \nrightarrow N' : (graph(R) \cap M' \times N')$ die **Einschränkung** von R auf $M^pr \times N'$.
Falls $N' = N$, kann auch $R|_{M'}$ geschrieben werden. □

Für Relationen lassen sich einige Eigenschaften definieren und andere Relationen ableiten.

Definition 99 (Eigenschaften von Relationen)
Eigenschaften von Relationen

Sei $R : M \nrightarrow N$ eine Relation.

R ist **linkstotal** gdw. $\forall m \in M\colon \exists n \in N\colon mRn$.
R ist **rechtstotal** gdw. $\forall n \in N\colon \exists m \in M\colon mRn$.
R ist **linkseindeutig** gdw. $\forall m_1, m_2 \in M\colon \forall n \in N\colon m_1Rn \wedge m_2Rn \Rightarrow m_1 = m_2$.
R ist **rechtseindeutig** gdw. $\forall m \in M\colon \forall n_1, n_2 \in N\colon mRn_1 \wedge mRn_2 \Rightarrow n_1 = n_2$.
R ist **funktional** gdw. $\forall m \in M\colon \exists n \in N\colon mRn \wedge \forall n' \in N\colon mRn' \Rightarrow n = n'$. □

Die Funktionen sind ein Spezialfall der Relationen:

Definition 100 (Funktion)
Funktion

Sei $f : M \nrightarrow N$ eine Relation.

Dann ist f eine **Funktion** genau dann, wenn f funktional ist.

Für $f : M \nrightarrow N$ wird dann $f : M \to N$, und für $f : M \nrightarrow N : (m, n) \leftrightarrow n = t(m)$ mit einen Term t wird $f : M \to N : m \mapsto t(m)$ geschrieben. □

Definition 101 (Abgeleitete homogene Relationen)
Abgeleitete homogene Relationen

Sei $R : M \nrightarrow M$ eine Relation auf M.

$I_M : M \nrightarrow M(a, b) \leftrightarrow a = b$ ist die **Diagonale** auf M.
$R^0 = I_M$ ist die **0-fache Iteration** von R.
Für $n \in \mathbb{N}\backslash\{0\}$ ist $R^n : M \nrightarrow M(a, c) \leftrightarrow \exists b \in M\colon ((a, b) \in R^{n-1} \wedge (b, c) \in R)$ die **n-fache Iteration** von R.
$R^n_m : M \nrightarrow M(a, b) \leftrightarrow \exists n \in \mathbb{N}\colon R^i(a, b)$ ist die **m-bis-n-fache Iteration** von R.
$R^= = R \cup R^0$ ist die **reflexive Hülle** von R.
$R^+ = R^\infty_0$ ist die **transitive Hülle** von R.
$R^* = R^\infty_1$ ist die **transitiv-reflexive Hülle** von R.
$R^\sim = R \cup R^-$ ist die **symmetrische Hülle** von R.
$R^\equiv = (R^\sim)^*$ ist die **Äquivalenzhülle** von R. □

Anhang B

Beweise

In Definition 70 ist für einen linearen Sketch nur ein einziges kommutatives Diagramm vorgesehen. Dies wird dadurch gerechtfertigt, dass eine Menge $\mathcal{D}$ von Diagrammen genau dann kommutiert, wenn ihr Kotupel $L = \langle D \rangle_{D \in \mathcal{D}}$ kommutiert. Der Beweis hierfür ist der folgende:

Beweis 102 (Kommutativität von Kotupeln von Diagrammen)
Sei $\mathcal{D}$ eine Menge von kommutativen Diagrammen mit kleinen Formgraphen in einer Kategorie $\mathbf{C}$, d.h. $\forall D \in \mathcal{D}$: $src(D) \in \mathfrak{Graphs} \wedge tar(D) = \mathbf{C}$. Sei $L = \langle D \rangle_{D \in \mathcal{D}}$ das von $\mathcal{D}$.

Beweis 103 (Die Einschränkung eines allgemeinen Sketches ist eine Instanz)
Seien die Daten aus Definition 83 gegeben. Zu zeigen ist, dass $I' := h_I = (h_L)_I = h_S \bullet I$ eine Instanz von $\mathcal{S}_1$ ist, also dass die formalen Limites als Limites und die formalen Kolimites als Kolimites interpretiert werden.

Sei also $j \in \widehat{J}_1$ der Index eines formalen Limes von $\mathcal{S}_1$. Somit ist $h_{\widehat{J}}(j)$ der Index des entsprechenden formalen Limes von $\mathcal{S}_2$. Fixiere dieses j und $h_{\widehat{J}}(j)$ und schreibe von hier an D_1 anstatt $D_{1,j}$ und D_2 anstatt $D_{1,h_{\widehat{J}}(j)}$, G_1 anstatt $G_{1,j}$ und G_2 anstatt $G_{1,h_{\widehat{J}}(j)}$ und so weiter. Außerdem schreibe $I(x)$ anstatt x_I für die Interpretation von formalen Knoten, Pfeilen oder Kegeln x.

Die Interpretation $I'(V_1)$ des formalen Kegels ist auf jeden Fall ein Kegel in $\mathbf{C}$. Zu zeigen ist, dass für sie gilt (mit $I'(p_1) = (I'(p_{1,v}))_{v \in \widehat{G}_1}$):

$$(I'(p_1) : I'(\widehat{P}_1) \to \widehat{D}_1 \bullet I') \\ \wedge \forall X \in \mathbf{C}^0\text{: } \forall (q : X \to \widehat{D}_1 \bullet I')\text{: } \exists! u\text{: } (u : q \to I'(p_1))$$

Zunächst ist also zu zeigen, dass die Interpretation des formalen Kegels ein kommutativer Kegel ist, und anschließend, dass eindeutige Homomorphismen kommutativer Kegel existieren. Aufgrund der Homomorphismusdefinition wird der Scheitel des formalen Kegels durch denselben Knoten interpretiert wie der des formalen Kegels von $\mathcal{S}_2$, denn es gilt:

$$I'(\widehat{P}_1) = h_S(\widehat{P}_1) = \widehat{P}_2$$

Für die Interpretation des Diagrammes gilt:

$$\widehat{D}_1 \bullet I' = \widehat{D}_1 \bullet h_S \bullet I = h_{\widehat{G}} \bullet \widehat{D}_2 \bullet I$$

Da $h_{\widehat{G}}$ ein Isomorphismus ist, heißt das, dass die Interpretation im Wesentlichen dieselbe ist wie des Diagrammes des formalen Kegels von $\mathcal{S}_2$. Aus demselben Grund ist auch die Familie $I'(p_1) = (I'(p_{1,v}))_{v \in \widehat{G}_1}$ der Interpretationen der Projektionen des formalen Limes im Wesentlichen identisch mit der Familie des formalen Kegels von $\mathcal{S}_2$, denn für $v \in \widehat{G}_1$ gilt:

$$I'(p_{1,v}) = (h_S \bullet I)(p_{1,v}) = I(h_S(p_{1,v})) = I(p_{2,h_{\widehat{G}}(v)})$$

Wendet man diese Gleichheiten an, erhält man die folgende zu zeigende Behauptung:

$$\begin{aligned}(I'(p_1) : I(\widehat{P}_2) \to h_{\widehat{G}} \bullet \widehat{D}_2 \bullet I) \\ \wedge\ \forall X \in \mathbf{C}^0\colon \forall (q : X \to h_{\widehat{G}} \bullet \widehat{D}_2 \bullet I)\colon \exists! u\colon (u : q \to I'(p_1))\end{aligned}$$

$I'(V_1)$ ist ein kommutativer Kegel, denn für $v \xrightarrow{e} w \in \widehat{G}_1^1$ gilt:

$$\begin{aligned} I'(p_{1,v}) \bullet \widehat{D}_1(e) &= h_S \bullet I(p_{1,v}) \bullet \widehat{D}_1(e) \\ &= I(h_S(p_{1,v})) \bullet \widehat{D}_1(e) \\ &= I(p_{2,h_{\widehat{G}}(v)}) \bullet \widehat{D}(e) \\ &= I(p_{2,h_{\widehat{G}}(w)}) \\ &= I'(p_{1,w}) \end{aligned}$$

Sei nun $q : X \to \widehat{D}_1 \bullet I'$ ein weiterer kommutativer Kegel über dem Diagramm $\widehat{D}_1 \bullet I' = h_{\widehat{G}} \bullet \widehat{D}_2 \bullet I : \widehat{G}_1 \to \mathbf{C}$. Da $h_{\widehat{G}}$ ein Isomorphismus ist, hat er ein Inverses $r = h_{\widehat{G}}^{-1} : \widehat{G}_2 \to \widehat{G}_1$. Wenn man auf die Komponenten dieses kommutativen Kegels den Homomorphismus $h_{\widehat{G}}$ und auf die Indizes sein Inverses anwendet, erhält man den Kegel $r \bullet q : X \to \widehat{D}_2$ mit $r \bullet q = (q_{r(v)})_{v \in \widehat{G}_2}$ über $\widehat{D}_2 \bullet I$.

Dieser ist kommutativ, denn für einen beliebigen Pfeil $v \xrightarrow{e} w \in \widehat{G}_2^1$ von $\widehat{G}_2$ gilt.

$$\begin{aligned} (r \bullet q)_v \bullet (\widehat{D}_2 \bullet I(e)) &= q_{r(v)} \bullet (\widehat{D}_2 \bullet I(e)) \\ &= q_{r(v)} \bullet (r \bullet h_{\widehat{G}} \bullet \widehat{D}_2 \bullet I(e)) \\ &= q_{r(v)} \bullet (h_{\widehat{G}} \bullet \widehat{D}_2 \bullet I(r(e))) \\ &= q_{r(v)} \bullet (\widehat{D}_1 \bullet h_S \bullet I(r(e))) \\ &= q_{r(v)} \bullet (\widehat{D}_1 \bullet I'(r(e))) \\ &= q_{r(w)} \\ &= (r \bullet q)_w \end{aligned}$$

Daher gibt es einen eindeutigen Homomorphismus kommutativer Kegel $u : r \bullet q \to I(p_2)$, d.h. einen eindeutigen Pfeil $u : X \to I(\widehat{P}_2)$ so dass für alle $v \in \widehat{G}_2^0$ gilt: $u \bullet I(p_{2,v}) = (r \bullet q)_v$. Dies ist auch ein Homomorphismus kommutativer Kegel $u : q \to I'(p_1)$, denn für $v \in \widehat{P}_1$

gilt:

$$\begin{aligned} u \bullet I'(p_{1,v}) &= u \bullet (h_S \bullet I(p_{1,v})) \\ &= u \bullet (I(h_S(p_{1,v}))) \\ &= u \bullet (I(p_{2,h_{\widehat{G}}(v)}) \\ &= (r \bullet q)_{h_{\widehat{G}}(v)} \\ &= q_{r(h_{\widehat{G}}(v))} \\ &= q_v \end{aligned}$$

Ebenso ist es eindeutig, denn wenn $u' : q \to I'(p_1)$ ein weiterer Homomorphismus kommutativer Kegel ist, dann ist es auch ein Homomorphismus kommutativer Kegel $u' : r \bullet q \to I(p_2)$, denn für $v \in \widehat{P}_2$ gilt:

$$\begin{aligned} u' \bullet I(p_{2,v}) &= u' \bullet I(h_S(p_{1,r(v)})) \\ &= u' \bullet (h_S \bullet I(p_{1,r(v)})) \\ &= u' \bullet I'(p_{1,r(v)}) \\ &= q_{r(v)} \\ &= (r \bullet q)_v \end{aligned}$$

Da es nur einen solchen Homomorphismus von $r \bullet q$ nach $I(p_2)$ gibt, ist $u = u'$. Somit ist also $I'(V_1)$ tatsächlich ein Limes.

Analog ist auch jeder formale Kokegel der Einschränkung ein Kokegel. □

Anhang C

Ecore-Bestandteile

In Abschnitt 3.1 wurde EMF/Ecore als Grundlage für den Entwurf der formalen graphbasierten Programmrepräsentation verwendet. In Folgenden findet sich eine Tabelle mit einer vollständigen Auflistung der Elemente des Ecore-Metamodelles. Zu jedem Element sind angemerkt:

- die Entwurfsentscheidung, ob das Element für die graphbasierten Programmrepräsentation relevant ist,
- die Anforderung aus Abschnitt 3.1, die die Entwurfsentscheidung begründet,
- eine Anmerkung, in welcher Form das Element in Abschnitt 3.2 graphbasiert umgesetzt wird.

Name	Entsch.	Grund	Anmerkung
EAttribute	Ja	**IncProgRep**	Attributierung
id	Nein	**OnlyModelling**	-
eAttributeType	Ja	**IncProgRep**	Attributierung (nicht reflektiv)
EAnnotation	Nein	**OnlyModelling**	-
EClass	Ja	**IncProgRep**	Typisierung
abstract	Ja	**IncProgRep**	Vererbung
interface	Nein	**IncProgRep**	Kein Unterschied zu abstrakten Klassen
eSuperTypes	Ja	**IncProgRep**	Vererbung
eOperations	Nein	**IncProgRep**	-
eReferences	Ja	**IncProgRep**	Typisierung
eAttributes	Ja	**IncProgRep**	Attributierung
eIDAttribute	Nein	**OnlyModelling**	-
eStructuralFeatures	Nein	**IncProgRep**	-
eGenericSuperTypes	Nein	**IncProgRep**	-
eAllAttributes	Ja	**IncProgRep**	Attributierung / Vererbung
eAllReferences	Ja	**IncProgRep**	Typisierung / Vererbung
eAllContainments	Ja	**IncProgRep**	Containment / Vererbung
eAllOperations	Nein	**IncProgRep**	-
eAllStructuralFeatures	Ja	**IncProgRep**	Keine explizite Modellierung
eAllSuperTypes	Ja	**IncProgRep**	Vererbung

Fortsetzung nächste Seite

Name	Entsch.	Grund	Anmerkung
eAllGenericSuperTypes	Nein	**IncProgRep**	-
isSuperTypeOf()	Ja	**IncProgRep**	Vererbung
getFeatureCount()	-	**OnlyModelling**	-
getEStructuralFeature()	-	**OnlyModelling**	-
getFeatureID()	-	**OnlyModelling**	-
getOperationCount()	-	**OnlyModelling**	-
getEOperation()	Nein	**IncProgRep**	-
getEOperationID()	-	**OnlyModelling**	-
getOverride()	Nein	**IncProgRep**	-
EClassifier	Ja	**IncProgRep**	Keine separate Modellierung nötig
isInstance()	Ja	**IncProgRep**	Typisierung
getClassifierID()	-	**OnlyModelling**	-
instanceClassName	Nein	**OnlyModelling**	-
instanceClass	Nein	**OnlyModelling**	-
defaultValue	Nein	Unnötig	-
instanceTypeName	Nein	**OnlyModelling**	-
ePackage	Nein	**OnlyModelling**	-
eTypeParameters	Nein	**IncProgRep**	-
EDataType	Ja	**IncProgRep**	Attributierung
serializable	Nein	**OnlyModelling**	-
EEnum	Ja	**IncProgRep**	Attributierung (keine separate Modellierung)
getEEnumLiteral()	Nein	**OnlyModelling**	-
eLiterals	Z.D.	**IncProgRep**	(Nicht reflektiv!)
EEnumLiteral	Z.D.	**IncProgRep**	-
value	Z.D.	**IncProgRep**	-
instance	Z.D.	**IncProgRep**	-
literal	Z.D.	**IncProgRep**	-
eEnum	Z.D.	**IncProgRep**	-
EFactory	Nein	**OnlyModelling**	-
		OnlyModelling	-
create()	Nein	**OnlyModelling**	-
		OnlyModelling	-
createFromString()	Nein	**OnlyModelling**	-
convertToString()	Nein	**OnlyModelling**	-
ePackage	Nein	**OnlyModelling**	-
EModelElement	Nein	Unnötig	-
getEAnnotation()	Nein	**OnlyModelling**	-
eAnnotations()	Nein	**OnlyModelling**	-
ENamedElement	Nein	Unnötig	-
name	Nein	Unnötig	-
EObject	Ja	**IncProgRep**	
eClass()	Ja	**IncProgRep**	Typisierung

Fortsetzung nächste Seite

Name	Entsch.	Grund	Anmerkung
eIsProxy()	Nein	**OnlyModelling**	-
eResource()	Nein	**OnlyModelling**	-
eContainer()	Ja	**IncProgRep**	-
eContainingFeature()	Nein	Unnötig	-
eContainmentFeature()	Nein	Unnötig	-
eContents()	Ja	**IncProgRep**	-
eAllContents()	Ja	**IncProgRep**	-
eCrossReferences()	Ja	**IncProgRep**	-
eGet()	Nein	**OnlyModelling**	-
eSet()	Nein	**OnlyModelling**	-
eIsSet()	Nein	**OnlyModelling**	-
eUnset()	Nein	**OnlyModelling**	-
eInvoke()	Nein	**OnlyModelling**	-
EOperation	Nein	**IncProgRep**	-
getOperationID()	Nein	**IncProgRep**	-
isOverrideOf()	Nein	**IncProgRep**	-
eContainingClass	Ja	**IncProgRep**	-
typeParameters	Nein	**IncProgRep**	-
eParameters	Nein	**IncProgRep**	Geordnet? Namen?
eExceptions	Nein	**IncProgRep**	-
eGenericExceptions	Nein	**IncProgRep**	-
EPackage	Nein	**OnlyModelling**	-
getEClassifier()	-	-	-
nsURI	-	-	-
eFactoryInstance	-	-	-
eClassifiers	-	-	-
eSubPackages	-	-	-
eSuperPackage	-	-	-
EParameter	Nein	**IncProgRep**	-
eOperation	Nein	**IncProgRep**	-
EReference	Ja	**IncProgRep**	-
containment	Ja	**IncProgRep**	-
container	Ja	**IncProgRep**	-
resolveProxies	Nein	**OnlyModelling**	-
eOpposite	Ja	**IncProgRep**	Constraints?
eReferenceType	Ja	**IncProgRep**	Typgraph
eKeys	Ja	**IncProgRep**	Constraints?
EStructuralFeature	Ja	**IncProgRep**	Keine explizite Modellierung
changeable	Nein	**OnlyModelling**	-
volatile	Nein	**OnlyModelling**	-
transient	Nein	**OnlyModelling**	-
defaultValueLiteral	Nein	**OnlyModelling**	-
defaultValue	Nein	**OnlyModelling**	-
unsettable	Nein	**OnlyModelling**	-

Fortsetzung nächste Seite

Name	Entsch.	Grund	Anmerkung
derived	Ja	**IncProgRep**	Constraints?
eContainingClass	Ja	**IncProgRep**	Typgraph
ETypedElement	Ja	**IncProgRep**	Keine explizite Modellierung
ordered	Ja	**IncProgRep**	Geordnete Pfeile
unique	Ja	**IncProgRep**	Constraints?
lowerBound	Ja	**IncProgRep**	Constraints?
upperBound	Ja	**IncProgRep**	Constraints?
many	Ja	**IncProgRep**	Constraints?
required	Ja	**IncProgRep**	Constraints?
eType	Ja	**IncProgRep**	Typhomomorphismus
eGenericType	Ja	**IncProgRep**	Typhomomorphismus
EGenericType	Nein	**IncProgRep**	-
eUpperBound	Nein	**IncProgRep**	-
eTypeArguments	Nein	**IncProgRep**	-
eRawType	Nein	**IncProgRep**	-
eLowerBound	Nein	**IncProgRep**	-
eTypeParameter	Nein	**IncProgRep**	-
eClassifier	Nein	**IncProgRep**	-
ETypeParameter	Nein	**IncProgRep**	-
eBounds	Nein	**IncProgRep**	-

Tabelle 2: Ecore-Elemente

Anhang D

Erklärung zur Masterarbeit

Erklärung nach §12 Abs. 8 Satz 6 Prüfungs- und Studienordnung für den Master-Studiengang Informatik an der Universität Passau vom 2. März 2006:

Hiermit erkläre ich, dass ich diese Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen als Hilfsmittel benutzt habe.

Datum Unterschrift