Formalising Refactorings with Graph Transformations

Carsten Rösnick

9. Juli 2008

Sich in Entwicklung befindliche Softwaresysteme verlieren über die Zeit häufig an Struktur. Auch kann sich die Spezifikation des Systems von Zeit zu Zeit ändern, was eine Anpassung der Struktur bedingt. Refactorings, manuelle oder auch automatisierte Strukturänderungen, stellen dazu eine der Möglichkeiten dar, die wir in dieser Ausarbeitung genauer betrachten wollen. Als Übersicht über Refactorings, sowie deren konkrete Anwendung, sei [1] empfohlen.

In der Praxis wird zumeist anhand von Tests (Unit-Tests, Assertions) sichergestellt, dass sich Code vor Anwendung eines Refactorings R gleich dem Code, hervorgehend durch Anwendung von R, verhält. Von theoretischer Seite gesehen ist dies jedoch eine nur ungenügende Argumentation. Was können uns Testfälle schon über die Korrektheit eines Refactorings sagen, wenden wir sie doch immer nur auf konkrete Situationen an. In den folgenden Abschnitten wollen wir daher die Möglichkeit, die Korrektheit von Refactorings entsprechend einer geeigneten Formalisierung zu beweisen, aufzeigen. Die wesentlichen Ideen und große Teile der Notation wurden in [2] eingeführt. Einige Korrekturen resp. Detailänderungen wurden durch [3] und [4] motiviert.

1 Motivation

Definition 1 (Refactoring)**.** Unter dem Begriff <u>Refactoring</u> verstehen wir im folgenden eine manuelle oder automatisierte Strukturänderung an einem gegebenen Softwaresystem. △

So intuitiv stimmig obige Definition eines Refactorings sein mag, so schwer ist sie in Praxi jedoch auch handhabbar. Einige essentiell wichtige Punkte spart die Definition nämlich aus:

1. Gibt es eine mathematisch elegante Formalisierung für Refactorings?
2. Refactorings besitzen gewisse (implizite) Vorbedingungen. Wie können wir selbige formalisieren?
3. Wie kann die Erhaltung der Programmsemantik, des Verhaltens eines Programms nach „außen“, auch nach Durchführung eines Refactorings (beweisbar) garantiert werden?

Eine belastbare Grundlage zur Beantwortung obiger Fragen wird in Abschnitt 2 eingeführt. Mit der Klärung der ersten Frage beschäftigt sich im Anschluß Abschnitt 3. Entsprechend der Gliederung dieser Ausarbeitung widmet sich Abschnitt 4 der Frage 2, eine Antwort auf die letzte Frage wird abschließend in Abschnitt 5 gegeben.
Die Ausarbeitung abrundend werden in Abschnitt 5.2 alle vorig diskutierten Konzepte zum Korrektheitsbeweis eines konkreten Refactorings eingesetzt.

2 Programmgraphen

Zunächst werde die Definition eines Programmgraphen, der als Darstellung für Softwaresysteme dienen soll, eingeführt. Ein konkretes Beispiel zur Übersetzung von Code in einen Programmgraphen wird in Abschnitt 2.2 diskutiert.

2.1 Grundbegriffe

Um Refactorings mathematisch formalisieren zu können benötigen wir ein zunächst eine von der verwendeten Programmiersprache unabhängige Darstellung des betrachteten Softwaresystems. Ähnlich dem Bereich des Compilerbaus, respektive der Compilergenerierung, wollen wir uns zur Darstellung eines Softwaresystems in der Graphentheorie bedienen, wobei wir statt Bäumen (= gerichtete, azyklische Graphen) beliebige gerichtete Graphen zulassen wollen. Präziser gesprochen wollen wir gerichtete Graphen mit Knoten-, sowie Kantenmarkierungen verwenden. Dies motiviert entsprechend die folgende

Definition 2 (Programmgraph)**.** Seien $\Sigma, \Delta \neq \emptyset$ zwei Alphabete, V_G eine Knoten- und $E_G \subseteq V_G \times \Delta \times V_G$ eine Kantenmenge. Elemente aus Σ seien zur Knotenmarkierung verwendet, entsprechend Elemente aus Δ zur Kantenmarkierung.
Ein <u>Programmgraph</u> sei nun ein 3-Tupel der Form $G = (V_G, E_G, \mathrm{nlab}_G : V_G \to \Sigma)$. △

Die Granularität der Darstellung eines Softwaresystems als Programmgraph hängt unmittelbar von der Wahl von Σ und Δ ab. Anders ausgedrückt: Je größer die Alphabete Σ, Δ sind und je mehr Konstrukte der jeweilig betrachteten Programmiersprache P damit ausgedrückt werden können, de-

sto detaillierter kann ein Softwaresystem, geschrieben in P, als Graph dargestellt werden. Wir stellen nun zwei natürliche Forderungen an die Wahl von Σ, Δ, um die Menge der Alphabete etwas einzugrenzen:

1. Die Alphabete Σ und Δ sollten „so klein wie möglich“ sein, da andernfalls die Darstellung größerer Programmabschnitte sehr unhandlich wird.
2. Auf der anderen Seite sollen die wesentlichen Aspekte jeder Programmiersprache mittels Σ und Δ ausgedrückt werden können.

Die zweite Forderung ist in dieser Form jedoch nicht zu erfüllen, denn es ist nicht evident, welche „wesentlichen Aspekte“ beispielsweise die Klassen der objektorientierten, funktionalen und logischen Programmiersprachen gemein haben, ganz abgesehen von der Frage, was wir als wesentlich auffassen wollen. Aus praktischen Gründen beschränken wir uns hier auf die Klasse der objektorientierten, sowie der prozeduralen Programmiersprachen.
Eine beispielhafte Belegung von Σ und Δ wird im folgendem Abschnitt vorgestellt.

2.2 Konkrete Belegung der Alphabete Σ, Δ

Betrachten wir zunächst das Codebeispiel (1). Ein Ziel soll es sein die Methode setValue(int value) aus der Kindklasse DFSVertex in die Vaterklasse Vertex zu verschieben, allgemein auch als *pull-up method*-Refactoring bezeichnet.

Listing 1: Beispielcode

```
1  public class Vertex {
2      protected int value;
3  }
4  public class DFSVertex extends Vertex {
5      public int discovered, finished;
6      public DFSVertex(
7          int value, int dt, int ft
8      ) {
9          super.value=value;
10         discovered=dt;
11         finished=ft;
12     }
13     public void setValue(int value) {
14         super.value=value;
15     }
16 }
```

Die Alphabete Σ, Δ sollen es uns in natürlicher Weise ermöglichen die Struktur eines jeden Programms in einen Programmgraph zu überführen. Konstrukte wie Klassen (***C**lass*), Methoden (***M**ethod* und ***M**ethod **D**efinition*), Variablen (***V**ariable* und ***V**ariable **D**efinition*), Parameter (***P**arameter*) wollen wir als Knoten auffassen, Beziehungen zwischen Klassen (***i**nheritance*), Typen von Variablen und Parametern, Funktionsaufrufe usw. hingegen als Kanten.

Verwenden wollen wir folgende Alphabete:

$$\Sigma = \{C, M, MD, V, VD, P, E\}$$
$$\Delta = \{l, i, m, t, p, e, c, a, u\}$$

Konkret verwenden wir das Kantenlabel l um den ***l**ookup* einer Methode zu dessen entsprechender Methodendefinition zu beschreiben. Auf den Sinn der Trennung von Methode und Methodendefinition geht Bemerkung (2) näher ein. Desweiteren interessieren wir uns für die Zugehörigkeit von Methoden und Variablen zu Klassen, kodiert als ***m**embership*-Kanten. ***t**ype*-Kanten zeigen an, welchem Typ eine Variable entspricht, wohingegen ***p**arameter*-Kanten den Typ eines Parameters oder auch die Zugehörigkeit eines Parameters zu einer Methodendefinition beschreiben.

Um die Funktion der Kantenlabel e,c,a,u verstehen zu können ist es zunächst notwendig das „Geheimnis“ um Knoten vom Typ E zu lüften: Eine Methodendefinition, also die Implementierung eines Methodenrumpfes, operiert möglicherweise auf nichtlokalen Variablen, sprich auf Variablen, die auch außerhalb selbiger Methodendefinition bekannt sind. Ebenso wollen wir den Aufruf anderer Methoden repräsentieren können. Ersteres ist bspw. bei Kapselung von Variablen elementar wichtig, letzteres bei Verschiebung von Methoden in andere Klassen. ***E**xpression*-Knoten repräsentieren einen (Teil-)Ausdruck in einer gegebenen Methodendefinition, die Kantenlabel e,c,a,u hingegen dienen der Beschreibung der Beziehung zwischen Ausdrücken (***e**xpressions*, Aufsplittung von Ausdrücken in Teilausdrücke), der Nachhaltung von Methodenaufrufen (***c**all*), sowie des Variablenzugriffs (***a**ccess*) und -updates (***u**pdate*).

Die genaue Festlegung, welche Kanten zwischen welchen Knoten zulässig sind, ist in [2] einzusehen.

Mit obiger Festsetzung der Alphabete Σ, Δ ergibt sich der zu Listing (1) assoziierte Programmgraph wie in Abbildung (1) zu sehen.

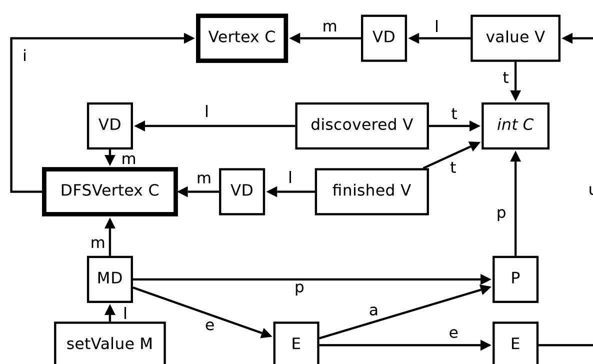

Abbildung 1: Programmgraph zu Listing 1

Bemerkung 1. Der primitive Datentyp `int` ist im Programmgraph ebenfalls als Knoten vom Typ C, sprich als Klasse aufgeführt. Jedoch gibt es in

Java die Unterscheidung zwischen primitiven Datentypen und Referenzen, anders als es in einer reinen objektorientierten Programmiersprache wie bspw. Smalltalk der Fall ist. Die Darstellung ist somit eigentlich inkorrekt, geht man von der Java-Spezifikation aus.
Dies akzeptieren wir jedoch, da es uns um Strukturaussagen geht. Zudem könnte ohne diese Ungenauigkeit in obigem Modell nicht ausgedrückt werden, dass ein Parameter vom Typ `int` ist. ◊

Bemerkung 2. Die Unterscheidung zwischen einer Methode und deren Definition scheint übertrieben detailliert, ist jedoch essentiell um bspw. die Überladung von Methoden beschreiben zu können. Selbiges gilt für die Überlagerung von Variablen. ◊

3 Refactorings

Die Vorarbeit des vorigen Abschnittes diente allein der formalen Beschreibung von Programmcode als Programmgraph. Ziel dieses Abschnittes soll es nun sein auch Refactorings formal zu beschreiben und darüber hinaus deren korrekte Arbeitsweise beweisen zu können, beides in der eingeführten Sprechweise von Programmgraphen.

3.1 Vorkommen

Stellen wir uns zunächst ein Refactoring als die Überführung eines gegebenen Programmgraphs L in einen neuen Programmgraph R vor. Diese Sprechweise impliziert, dass unser betrachtetes Refactoring auf dem ganzen Graphen L wirkt. In den meisten Fällen ist das jedoch eine viel zu grobe Annahme, da Refactorings meist nur auf bestimmten Teilen des Codes und damit auch nur auf den korrespondierenden Teilgraphen operieren. Das Stichwort „Teilgraph" motiviert die folgende

Definition 3.

1. Sei $G = (V_G, E_G, \text{nlab}_G)$ ein Programmgraph und $W \subseteq V_G$ eine Teilmenge der Knoten von G. Der vollständige Teilgraph über W in G ist definiert durch $G_W = (W, E_G \cap (W \times \Delta \times W), \text{nlab})$ mit $\text{nlab} := \text{nlab}_G|_W$.
2. Sei $G = (V_G, E_G, \text{nlab}_G)$ ein Programmgraph, W eine beliebige Menge und $f : V_G \to W$ eine injektive Funktion. Dann bezeichnen wir mit $f(G)$ den Programmgraph definiert durch $f(G) := \big(f(V_G), E_{f(G)}, \text{nlab}_G \circ f^{-1}\big)$ mit $E_{f(G)} := \{(f(v), \delta, f(w)) \mid (v, \delta, w) \in E_G\}$.
3. Seien $K = (V_K, E_K, \text{nlab}_K)$ und G zwei Programmgraphen. Ein Vorkommen von K in G ist eine injektive Funktion $\text{oc} : K \to G$ derart, dass $\text{oc}(K)$ den vollständigen Teilgraph über $\text{oc}(V_K)$ in G darstellt.

△

Sind nun obig neu eingeführte Definitonen und Konstruktionen ausdrucksstark genug, um Produktionen und deren Durchführung auch formal beschreiben zu können? Die Antwort ist erstaunlich und ernüchternd zugleich: Im Prinzip reicht diese Notation um Refactorings auszudrücken. Dann allerdings ergibt sich ein wesentliches Problem: Es können sogenannte **dangling edges**, zu deutsch „hängende Kanten", entstehen.

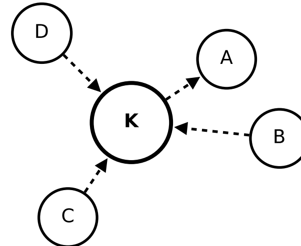

Abbildung 2: Entstehung hängender Kanten

Abbildung (2) zeigt die Grundsituation, in der hängende Kanten auftreten können. Bedingt ein Refactoring die Löschung des Teilgraphen K oder die Verschiebung von K innerhalb des umgebenden Programmgraphen, so müssen auch die ein-, sowie ausgehenden Kanten bei derlei Update an K gelöscht respektive verschoben werden.

Das unterligende Konzept, um derlei Problematiken aufzulösen, sei im Folgenden als Embedding bezeichnet.

3.2 Embeddings

Es seien $L \subseteq G$ und $R \subseteq H$ Teilgraphen, oc_1, oc_2 Vorkommen von L in G respektive R in H und p, sowie darauf aufbauend p^*, Produktionen. Diagramm (3) fasst nun das Konzept der Darstellung eines Refactorings als Produktion p, die Bestimmung von Vorkommen der linken Seite von p in G, sowie der rechten Seite in H nebst der Behandlung hängender Kanten durch Embeddings zusammen. p^* sei demnach aufzufassen als Produktion p + Embedding.

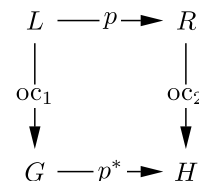

Abbildung 3: Abstrakter Zusammenhang zwischen Produktionen und Vorkommen

Nun treffen aber Produktionen selbst noch keine Aussage über die Behandlung ein- und ausgehender Kanten außerhalb ihres Definitionsbereichs. Diesem Umstand begegnen wir durch Betrachtung zweier Mengen von Umformungen, jeweils eine für ein- und ausgehende Kanten.

Definition 4.

$$\mathrm{Emb_{in}}, \mathrm{Emb_{out}} \subseteq (\Delta \times V_L) \times (\Delta \times V_R)\,. \qquad (1)$$

Ein Element $((\alpha, u), (\beta, v)) \in \mathrm{Emb_{in}}$ beschreibe nun die Umformung $(w \xrightarrow{\alpha} \mathrm{oc}_1(u) \in E_G) \mapsto (w \xrightarrow{\beta} \mathrm{oc}_2(v) \in E_H)$, $w \notin \mathrm{oc}_1(V_L) \cup \mathrm{oc}_2(V_R)$, d.h. allen in $\mathrm{oc}_1(u)$ mündenden Kanten mit Label α im Programmgraph G wird im Programmgraph H der neue Zielknoten $\mathrm{oc}_2(v)$, sowie das neue Label β zugewiesen. Analog werden Umformungen für ausgehende Kanten mit Hilfe der Menge $\mathrm{Emb_{out}}$ formalisiert. △

Schauen wir uns dazu nun ein etwas umfangreicheres Beispiel unter Verwendung von Vorkommen und Produktionen an.

Beispiel 1 (pull-up method)**.** Wie einleitend erwähnt soll es (exemplarisch) unser Ziel sein die Methode setValue(int value) in die Vaterklasse von DFSVertex zu verschieben, kurzum das *pull-up method*-Refactoring anzuwenden. Dieses setzt sich dabei aus zwei Schritten zusammen:

1. Methodendefinition von setValue im Programmgraph aus der Kindklasse in die Vaterklasse verschieben;
2. Alle *isomorphen Kopien* (= Methoden identischen Verhaltens) von setValue aus Geschwisterklassen löschen (dieser Schritt ist u.U. manuell vom Programmierer abzusegnen, da es äußerst schwierig ist automatisiert entscheiden zu lassen, ob zwei Methoden sich äquivalent verhalten).

In Abbildung (4) sind beide Schritte noch einmal mit der gewohnten Syntax von Programmgraphen veranschaulicht. Kombinieren wir nun die abstrakte

(a) Produktion p_1 (b) Produktion p_2

Abbildung 4: Bestandteile der Produktion zum Refactoring *pull-up method*

Notation aus Abbildung (4(a)) mit dem konkreten Programmgraphen aus Abbildung (1). Zur Vereinfachung ist der Teilgraph $\mathrm{oc}_1(L)$ in Abbildung (5) lediglich grau eingefärbt mit fein gestrichelten Kanten, hingegen die in $\mathrm{Emb_{in}}$, sowie in $\mathrm{Emb_{out}}$ Beachtung findenden Kanten gröber gestrichelt dargestellt. ◁

In der Tat haben wir nun alle Werkzeuge zusammen um Embeddings respektierende Produktionen definieren zu können. Die nötigen Details betrachtet

Abbildung 5: Teilgraph $\mathrm{oc}_1(L) \subseteq G$

Definition 5 (Embeddings respektierende Produktionen)**.** Seien $G = (V_G, E_G, \mathrm{nlab}_G), H = (V_H, E_H, \mathrm{nlab}_H)$ Programmgraphen, $\mathrm{oc}_1 : L \to G, \mathrm{oc}_2 : R \to H$ Vorkommen, so dass gilt $\mathrm{oc}_1(V_R) \cap V_G = \emptyset$. Seien überdies $\mathrm{Emb_{in}}, \mathrm{Emb_{out}}$ wie in Definition (4). Dann bezeichne $p^* = (L, R, \mathrm{Emb_{in}}, \mathrm{Emb_{out}})$ eine Embeddings respektierende Produktion. Außerdem sagen wir <u>G gehe mittels p^* direkt in H über</u>, falls

1. alle Knoten in $V_G \setminus \mathrm{oc}_1(V_L) \subseteq V_H$ erhalten bleiben und Änderungen ausschließlich an Knoten in $\mathrm{oc}_2(V_R)$ stattfinden. Formal:

$$V_H = \underbrace{V_G \setminus \mathrm{oc}_1(V_L)}_{\text{unveränderte Knoten}} \;\dot\cup\; \underbrace{\mathrm{oc}_2(V_R)}_{\text{veränderte Knoten}}$$

   Die Disjunktheit der Vereinigung folgt direkt anhand der eingangs gestellten Forderung $\mathrm{oc}_1(V_R) \cap V_G = \emptyset$.

2. die Labeling-Funktion von H in Einklang mit der disjunkten Vereinigung von V_H aus Punkt 1 ist:

$$\mathrm{nlab}_H(v) = \begin{cases} \mathrm{nlab}_G(v), & v \in V_G \setminus \mathrm{oc}_1(V_L) \\ \mathrm{nlab}_{\mathrm{oc}_2(V_R)}(v), & v \in \mathrm{oc}_2(V_R) \end{cases}$$

3. die Endknoten aller Kanten die Produktion respektieren, d.h. für alle Kanten $(u, \beta, w) \in V_H \times \Delta \times V_H$ gilt genau eine der folgenden Bedingungen:

   (a) $(u, \beta, w) \in E_G$

   (b) $(u, \beta, w) \in E_{\mathrm{oc}_2(V_R)}$

   (c)
   - $\exists\, \tilde{w} \in V_R : \mathrm{oc}_2(\tilde{w}) = w$
   - $\exists\, \tilde{v} \in V_L : (p^* \circ \mathrm{oc}_1)(\tilde{v}) = w$
   - $(u, \alpha, \mathrm{oc}_1(\tilde{v})) \in E_G$
   - $((\alpha, \tilde{v}), (\beta, \tilde{w})) \in \mathrm{Emb_{in}}$

   (d)
   - $\exists\, \tilde{u} \in V_R : \mathrm{oc}_2(\tilde{u}) = u$
   - $\exists\, \tilde{v} \in V_L : (p^* \circ \mathrm{oc}_1)(\tilde{v}) = u$
   - $(\mathrm{oc}_1(\tilde{v}), \alpha, w)) \in E_G$
   - $((\alpha, \tilde{v}), (\beta, \tilde{u})) \in \mathrm{Emb_{out}}$

△

So eingängig die Bedingungen 3.a und 3.b seien mögen, so unanschaulich sind hingegen 3.c und 3.d im ersten Moment. Abbildung (6) verdeutlicht die Sinnhaftigkeit und Idee hinter Bedingung 3.d, basierend auf der abstrakten Grundlage des kommutativen Diagramms (3). Die beiden gestrichelten

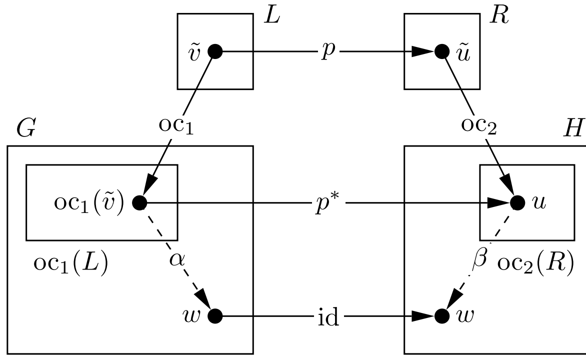

Abbildung 6: Konkrete Ausprägung des kommutativen Diagramms (3)

Kanten sind als konkret in E_G respektive in E_H auftretend mit den angegebenen Kantenlabeln zu verstehen. Hingegen stellen die durchgezogen gezeichneten Kanten Zuordnungen anhand der als Kantenlabel gegebenen Abbildungen dar. Beispiel: Zuordnung (Kante) $\tilde{u} \xrightarrow{\text{oc}_2} u$ ist zu verstehen als $u = \text{oc}_2(\tilde{u})$.

Bedingung 3.c kann analog veranschaulich werden, wird hier also entsprechend nicht explizit angeführt.

4 Well-formedness

Die im Abschnitt 2.1 vorgestellte Notation von Programmgraphen als Abstrahierung weg vom konkreten Programmcode liefert uns eine Möglichkeit nun auch formal und sprachenunabhängig über die Struktur und bedingt auch über das Verhalten von Programmen zu reden. Der Clou ist hierbei, dass ein der Definition entsprechender Programmgraph zu syntaktisch korrektem Code korrespondiert.

Wie in Abschnitt 3 gesehen bilden Programmgraphen das Fundament zur Formalisierung von Refactorings, kurzum der Beschreibung von Produktionen. Gehen wir gedanklich den nächsten Schritt, hin zum Beweis der Korrektheit eines gegebenen Refactorings, so fällt sehr schnell auf, dass die Korrespondenz von Programmgraphen zu syntaktischn korrektem Code eine mathematische Voraussetzung darstellt, ohne die wir nicht auskommen werden. Vielmehr noch soll die Korrespondenz zu syntaktisch korrektem Code von Refactorings erhalten werden.

Dies wirft eine Reihe von Fragen seitens der Formalisierung auf, die sukzessive von den folgenden drei Abschnitten beantwortet werden.

4.1 Type Graphs

Bevor wir „high-level“ Anforderungen an Programmgraphen formulieren können, müssen wir etwas viel grundlegenderes sicherstellen: Ein gegebener Graph G muss die in Definition (2) eingeführte Notation eines Programmgraphen erfüllen. So grundlegend die Forderung, so einfach auch die Lösung. Wie Abbildung (7) aufzeigt beschreiben wir ganz allgemein die Struktur der Menge aller der Notation entsprechenden Programmgraphen. Existiert nun für einen gegebenen Graphen G ein Graph-Morphismus (Abbildung von Knoten und Kanten; nicht mit Vorkommen zu verwechseln, da wir die Forderung nach Injektivität hier nicht stellen wollen und können) in den Type Graph, der Quell- und Zielknoten einer jeden Kante, sowie dessen Kantenlabel erhält, so bezeichnen wir G als einen wohlgeformten Programmgraphen.

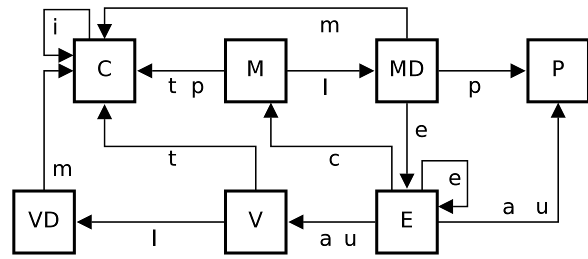

Abbildung 7: Type Graph

4.2 Graph Expressions

Wie in Abschnitt 4.1 erwähnt handelt es sich um eine sehr rudimentäre Anforderung, die ein gegebener Graph G erfüllen muss, um als wohlgeformter Programmgraph bezeichnet werden zu dürfen. Eine ganze Reihe von Anforderungen an G wird damit jedoch nicht abgedeckt. Ein paar Beispiele:

WF-1 Zwei oder mehr Variablen mit dem selben Bezeichner dürfen nicht innerhalb einer Klasse vorkommen.

WF-2 Zwei oder mehr Methoden mit identischer Signatur dürfen nicht innerhalb einer Klasse vorkommen.

WF-3 Methoden dürfen nicht auf Variablen zugreifen, die erst in Kindklassen bekannt sind.

WF-4 Methoden dürfen nicht auf Parameter anderer Methoden zugreifen.

Nehmen wir uns Beispiel **WF-3** heraus: Das Problem an dieser Forderung liegt buchstäblich in und an der Einordnung von Klassen in eine Klassenhierarchie. Eine Variable v, auf die von einer Methode m zugegriffen wird, kann beliebig tief in der Klassenhierarchie liegen. Zur Auflösung dieser Problematik nutzen wir ein wohlbekanntes Konstrukt aus der Informatik, und zwar reguläre Ausdrücke.

Bisher war es uns nur möglich einzelne Literale des zugrundeliegenden Alphabets als Kantenlabel zu verwenden. Lassen wir nun also reguläre Ausdrücke $\tilde{w}$ mit

$$\text{words}(\tilde{w}) \subseteq \Delta^* = \{\varepsilon\} \cup \bigcup_{n \in \mathbb{N}} \underbrace{\Delta \times \Delta \times \cdots \times \Delta}_{n\text{-mal}}$$

als Kantenlabel zu, so können wir Graphen angeben, deren Vorkommen wir in G verbieten.

Fassen wir dies nun also zu einer Definition zusammen, gefolgt von einem konkreten Beispiel.

Definition 6 (Graph Expressions)**.**

1. Eine Graph Expression GE ist ein Graph (V_{GE}, E_{GE}) über Σ und der Menge regulärer Ausdrücke über Δ.
2. Sei G eine Programmgraph. Ein Vorkommen von GE in G ist eine injektive Abbildung $\text{oc} : V_{GE} \to V_G$, so dass
   (a) für jeden Knoten $v \in V_{GE}$ gilt: $\text{nlab}_{GE}(v) \in \Sigma$ ist der Typ des Knotens $\text{oc}(v)$.
   (b) für jede Kante $u \xrightarrow{\tilde{w}} v \in E_{GE}, \text{words}(\tilde{w}) \subseteq \Delta^*$, ein Pfad $\text{oc}(u) =: x_0 \xrightarrow{w_1} x_1 \xrightarrow{w_2} \ldots \xrightarrow{w_k} x_k := \text{oc}(v)$ in E_G existiert mit $w_i \in \Delta, x_i \in E_G, i = 1, \ldots, k, k \in \mathbb{N}$ und $\text{words}(\tilde{w}) \ni (w_1, w_2, \ldots, w_k)$.

$\triangle$

Nun das angekündigte

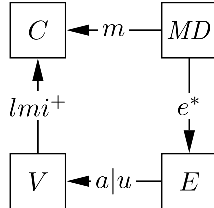

Abbildung 8: GE um Vorkommen von WF-3 in Programmgraph G zu verbieten

Beispiel 2 (GE für WF-3)**.** Entscheidend in Abbildung (8) ist die Kante $V \xrightarrow{lmi^+} C$, genauer der Teilausdruck i^+ des Kantenlabels. Dieser bringt zum Ausdruck, dass die zugegriffene Variable nicht in der gleichen Klasse C und auch nicht in einer Vaterklasse von C, sondern in der Tat in einer echten Kindklasse definiert sein muss. Wird diese Situtation nun entsprechend in einem Programmgraphen G gefunden, so bezeichnen wir selbigen nicht mehr als wohlgeformt. $\triangleleft$

4.3 Vorbedingungen

Schlussendlich stellt sich noch die Frage, ob und wann ein Refactoring resp. eine Produktion auf einen Programmgraphen angewendet werden darf. Betrachten wir dazu wieder den Code aus Listing (1). Angenommen in der Klasse Vertex befindet sich ebenfalls eine Methode setValue(int value) mit beliebiger Implementierung. Darf unter dieser Voraussetzung die korrespondierende Methode von DFSVertex in Vertex verschoben werden? Offensichtlich nicht, sofern es sich nicht um zwei äquivalente Implementierungen handelt.

An diesem Punkt angelangt müssen wir jedoch ein Tribut an die Semantik zollen, denn es ist überhaupt nicht klar, wann zwei gegebene Methoden sich äquivalent verhalten, entsprechend also die eine durch die andere Methode austauschbar ist. Daher stellen wir im Vorfeld, praktisch vor Anwendung der Produktion *pull-up method*, die Forderung, dass eine Methode mit gleicher Signatur nicht bereits in der Vaterklasse vorhanden sein darf.

Bemerkung 3. In der Tat ist es essentiell vor Durchführung eines Refactorings resp. einer Produktion zu prüfen, ob alle Vorbedingungen erfüllt sind. Verlagerten wir diese Überprüfung auf einen Zeitpunkt nach Anwendung eines Refactorings, so müssten die bereits durchgeführten Schritte rückgängig gemacht werden, falls eine der geprüften Bedingungen nicht erfüllt würde. $\diamondsuit$

Abbildung 9: GE_{pre1}, um Methodendefinition in Vaterklasse verbieten

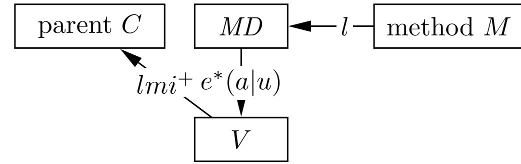

Abbildung 10: GE_{pre2}, um Zugriff auf Variablen außerhalb des Sichtbarkeitsbereichs von method nach pull-up zu verbieten

Abbildungen (9) und (10) zeigen Vorbedingungen zur in Abbildung (4(a)) eingeführten Produktion p_1, die Bestandteil der Produktion p^* und damit des *pull-up method* Refactorings ist.

5 Preservation of Behaviour

Das Augenmerk lag bisher auf der Formalisierung von Refactorings, ohne jedoch Argumente für die Erhaltung der korrekten Funktionalität zu erhalten. Diese noch ausstehende Lücke in unserem Versuch, eine formale Grundlage, die ebenso Korrektheitsbeweise auf elegante Art zulässt, sei nun zu schließen.

Dazu stellen wir zunächst die Frage, was wir unter dem Begriff „Verhalten“ überhaupt verstehen wollen. Ohne Klärung dieser Frage ist es insbesondere nicht möglich den Verhaltenserhalt zu fordern bzw. eine Menge von Regeln dafür zu formulieren.

Charakterisieren wollen wir das Verhalten eines Programms (genauer: einer Methode, wobei die Menge der Methoden im wesentlichen ein Programm ergeben) anhand der Menge der durchgeführten Methodenaufrufe, Variablenzugriffe und -änderungen. Damit motivieren wir die

Definition 7 (Erhaltung von Verhalten)**.** Seien G_1, G_2 zwei Programmgraphen, wobei G_2 nach Durchführung eines Refactorings aus G_1 hervorging. Seien desweiteren $m_1 \subseteq G_1, m_2 \subseteq G_2$ zwei beliebige aber feste Teilgraphen, die zu jeweils einer Methode korrespondieren, wobei m_2 ebenfalls anhand des Refactorings aus m_1 hervorging. Dann sagen wir, $\underline{m_1 \text{ verhält sich identisch zu } m_2}$, wenn folgende 3 Bedingungen erfüllt sind:

1. **Methodenaufrufe:** m_2 führt mindestens die gleichen Methodenaufrufe durch wie m_1. Hier zählen auch transitive Methodenaufrufe, daher fordern wir mindestens die gleichen Methodenaufrufe.
2. **Variablenzugriffe:** m_2 greift auf die gleichen Variablen zu wie m_1, wobei auch transitive Zugriffe zählen. Ein kanonisches Beispiel für einen transitiven Variablenzugriff stellt sicherlich der Zugriff auf eine Variable über eine getter-Methode dar.
3. **Variablenänderungen:** m_2 führt Änderungen an den gleichen Variablen durch wie m_1.

△

5.1 Tracking Function

Unmittelbar anschließend an Definition (7) stellt sich die Frage nach einer konkreten Formalisierung der angeführten 3 Forderungen. Die dazu erforderliche Idee ist sehr eingängig: Methodenaufrufe, Variablenzugriffe und -änderungen einer Methode sind sehr leicht mittels Graph Expressions $GE_i, i \in \mathbb{N}$ allgemein formulierbar. Da nach Anwendung einer Produktion die Menge der Operationen auf Methoden und Variablen niemals echt kleiner werden kann (nach Definition (7)) reicht es für jede Graph Expression $GE_i, i \in \mathbb{N}$ ein Vorkommen im neuen Programmgraphen H zu finden. Gelingt uns das für jede Graph Expression, die bereits ein Vorkommen in G besaß, so erhält die gegebene Produktion das Verhalten des Programms (entsprechend unserer Festlegung, was wir unter dem Begriff „Verhalten“ verstehen wollen).

Die Formalisierung der beschriebenen Idee zeigen wir auf in

Definition 8. Sei GE eine Graph Expression, G, H Programmgraphen und $\mathrm{tr} : V_G \to V_H$ eine Abbildung. Wir sagen, $\underline{\mathrm{tr} \text{ erhält } GE}$, wenn für jedes Vorkommen oc von GE in G ein Vorkommen $\mathrm{tr} \circ \mathrm{oc}$ von GE in H existiert. △

Wie wir bereits anhand Abbildung (4) gesehen haben ist es sinnvoll ein Refactoring in mehrere Produktionen aufzuspalten. Im Zuge dieser Erkenntnis versehen wir jede einzelne Produktion p^* mit einer Abbildung tr_{p^*}, die wir folgend als Tracking Function bezeichnen wollen.

Definition 9 (Tracking Function)**.** Geht G mittels $p^* = (L, R, \mathrm{Emb}_{\mathrm{in}}, \mathrm{Emb}_{\mathrm{out}})$ direkt in H über (siehe Definition (5)), so sei die Tracking Function $\mathrm{tr}_{p^*} : V_G \to V_H$ definiert als

$$\mathrm{tr}_{p^*} = \begin{cases} \left(\mathrm{oc}_2 \circ \mathrm{tr}_p \circ \mathrm{oc}_1^{-1}\right)(v), & v \in \mathrm{oc}_1(V_L) \\ v, & \text{sonst} \end{cases}$$

mit $\mathrm{tr}_p : V_L \to V_R$ wie in Definition (8). Wir sagen die Produktion $\underline{p^* \text{ erhält } GE}$, wenn die zugehörige Tracking Function tr_{p^*} die Graph Expression GE erhält. △

5.2 Beweis

All die eingeführten Formalisierungen dienten und dienen dazu die Korrektheit von Refactorings auch formal beweisen zu können. Genau einen solchen Beweis wollen wir nun skizzierend aufzeigen. Wir betrachten zwei wichtige Aspekte, unter denen wir das Refactoring *pull-up method* untersuchen und gegen selbige wir die Korrektheit zeigen wollen: Die Erhaltung der Wohlgeformtheit, sowie des Verhaltens.

5.2.1 Preservation of Well-formedness

Wie in Abschnitt 4 erwähnt muss ein jedes Refactoring resp. dessen korrespondierende Produktionen die Wohlgeformtheit von Programmgraphen erhalten. Exemplarisch wollen wir den Erhalt dieser Invariante anhand der Forderung WF-3 beweisen.

Wir betrachten nun also die Produktionen p_1, p_2, eingeführt in Abbildung (4), und zeigen, dass nach Anwendung von p_1 und p_2 keine Vorkommen der Graph Expression $GE_{\text{WF-3}}$ aus Abbildung (8) in H existieren, sofern keine Vorkommen im G existierten.

Sei G ein beliebiger Programmgraph, der dem Type Graph (Abbildung (7)), sowie den Graph Expressions GE_{pre1} (Abbildung (9)) und GE_{pre2} (Abbildung (10)) genügt.

Produktion p_1: Nehmen wir an, nach Anwendung von p_1 auf G lässt sich ein Vorkommen von $GE_{\text{WF-3}}$ im entstehenden Programmgraphen finden. Zunächst gehe H' aus G durch Anwendung von p_1 hervor. H' enthält also nun, nach Annahme, ein Vorkommen von $GE_{\text{WF-3}}$. Es reicht den Teilgraph $A \subseteq H'$ zu betrachten, der sich gegenüber G verändert hat, sprich in dem Vorkommen der linken Seite von p_1 in Vorkommen der rechten Seite von p_1 überführt wurden. Transformationen in A gegenüber G sind (nach p_1) von der Form ($MD \overset{m}{\to} childC \in$

$E_G) \mapsto (MD \xrightarrow{m} parentC \in E_A)$. Damit existiert weiterhin **kein** Vorkommen von GE_{pre2} in A. Da nach Annahme ein Vorkommen von $GE_{\text{WF-3}}$ in A existiert, muss es ebenfalls eine Kante $V \xrightarrow{lmi^+} C \in E_A^*$ (T^* bezeichne allgemein die transitive Hülle eines Graphen T) geben. Selbiges Argument gilt für die Kante $E \xrightarrow{a|u} V$. Dies jedoch steht im Widerspruch zur Folgerung, dass A ebenfalls kein Vorkommen von GE_{pre2} aufweist. Demnach wird durch Anwendung von p_1 auf G kein Vorkommen von $GE_{\text{WF-3}}$ in H' erzeugt.

Produktion p_2: Dieser Fall ist weitaus schneller behandelt. H' enthalte vor Anwendung von p_2 kein Vorkommen von $GE_{\text{WF-3}}$. Durch die Löschung isomorpher Kopien der verschobenen Methodendefinition werden keine neuen Kanten erzeugt. Da vorher kein Vorkommen von $GE_{\text{WF-3}}$ existierte, wird entsprechend nach Anwendung von p_2 ebenfalls ein solches nicht existieren.

5.2.2 Preservation of Behaviour

Ein sehr schöner Beweis wird in [2] aufgezeigt, einen äquivalenten Beweis wollen wir daher an dieser Stelle nicht skizzieren.

6 Schlußwort

Fassen wir nun einmal alle eingeführten Strukturen und deren Zusammenhang zusammen. Zunächst haben wir eine Abstraktion, weg vom Programmcode und hin zu Programmgraphen vorgestellt mit dem Ziel, die Korrektheit von Refactorings sprachunabhängig beweisen zu können. An Programmgraphen haben wir Anforderungen gestellt, um eine Korrespondenz zu syntaktisch korrektem Code zu erhalten. Zum einen bezeichneten wir einen Graphen G als Programmgraphen, sofern eine Beziehung zum eingeführten Type Graph existierte; zum anderen haben wir gewisse Strukturen mittels Graph Expressions verboten.

Der wesentliche Teil der Arbeit entfiel auf die Formalisierung von Refactorings als Produktionen. Eingeführt wurden Vorbedingungen, die eine Anwendung von Produktionen auf Programmgraphen steuerten. Ebenso besprochen haben wir die Frage, wie sich Teilgraphen in einem gegebenen Programmgraphen beschreiben lassen, auf die eine beliebige, aber feste Produktion anwendbar ist, nebst der Formalisierung von Embeddings, motiviert durch die Problematik hängender Kanten.

Den Schlußpunkt stellte die Diskussion um die Erhaltung des Programmverhaltens dar, in die sowohl eine Antwort auf die Frage, was wir überhaupt unter dem Begriff „Verhalten“ verstehen wollen gegeben wurde, als auch eine formale Möglichkeit gewisse Strukturen, die Verhalten beschreiben, über die Anwendung einer Produktion hinaus zu erhalten.

Nicht im Detail beleuchtet, obgleich von zentraler Bedeutung, wurden die sich durch die Formalismen ergebenden Herausforderungen. Als da wären die korrekte und vollständige Bestimmung von Vorbedingungen für ein gegebenes Refactoring, die Formulierung von Well-formedness-Bedingungen (immerhin haben wir lediglich 4 Beispiele genannt), sowie die vollständige Angabe verbotener (Sub-)Strukturen mittels Graph Expressions. Von praktischer Seite aus betrachtet sind vor allem minimale Mengen von Vorbedingungen interessant, die idealerweise auch von „besonders einfacher Gestalt“ seien mögen - sehr wichtig im Hinblick auf die konkrete Implementierung von Vorbedingungen, um automatisiert die zuvor als korrekt bewiesenen Refactorings ausführen lassen zu können. Nebst der Begegnung der aufgezählten Herausforderungen kann demnach die praktische Umsetzung der hier gewonnenen theoretischen Erkenntnisse den nächsten zu gehenden Schritt darstellen.

Literatur

[1] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.

[2] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *J. Softw. Maint. Evol.*, 17(4):247–276, 2005.

[3] Grzegorz Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: volume I. foundations.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[4] Lanlan Zhang and Shu Yang. Double pushout approach, September 2004. Seminar Graph Grammars, RWTH Aachen, `http://www.se.rwth-aachen.de/tikiwiki/tiki-download_file.php?fileId=219`.