

# JTransformer – Eine logikbasierte Infrastruktur zur Codeanalyse

Daniel Speicher, Tobias Rho, Günter Kniesel  
Institut für Informatik III, Universität Bonn, Römerstrasse 164, 53117 Bonn

Email: {dsp, rho, gk}@iai.uni-bonn.de

## Kurzfassung

Reengineering erfordert oft komplexe Analysen mit solider formaler Fundierung. Je weiter das Abstraktionsniveau des Formalismus dabei von dem der Implementierung entfernt ist, umso schwieriger ist die Implementierung und Wartung der Analyse. JTransformer bietet daher eine logikbasierte Infrastruktur zur Codeanalyse, die eine angemessen direkte Umsetzung der formalen Grundlagen in ausführbaren Code erlaubt. Dies demonstrieren wir am Beispiel der Vorbedingungen für generalisierende Refactorings.

## 1. For Example Type Constraints

Als Fallstudie betrachten wir das auf Typbeschränkungen basierende Verfahren von Tip, Kiezun und Bäumler zur Entscheidung über die Durchführbarkeit generalisierender Refactorings [3].

Zur Illustration der Grundgedanken untersuchen wir im Beispiel in Abb. 1. die Frage, ob in der Methode *m* wirklich der konkrete Typ *B* verwendet werden muss, oder ob auch der Obertyp *A* genügen würde. Der Aufruf der Methode *p* aus dem Typ *A* auf der Variablen *x* steht dem nicht im Wege. Tatsächlich erzwingt jedoch die Methode *n* indirekt die Verwendung des Typs *B* in Methode *m*, da dort auf

|    |                     |                               |
|----|---------------------|-------------------------------|
| 1  | package mini;       |                               |
| 2  |                     |                               |
| 3  | class A {           | Zuweisungen,                  |
| 4  | void p() {}         | Methodenaufrufe und           |
| 5  | }                   | Rückgabeanweisungen           |
| 6  | class B extends A { | stellen Bedingungen an        |
| 7  | void q() {}         | die Typen [E] der beteiligten |
| 8  | }                   | Ausdrücke E:                  |
| 9  |                     |                               |
| 10 | class C {           |                               |
| 11 | B m() {             |                               |
| 12 | B x = new B();      | [new B()] ≤ [x]               |
| 13 | x.p();              | [x] ≤ A                       |
| 14 | return x; }         | [x] ≤ [C.m()]                 |
| 15 | void n() {          |                               |
| 16 | B y = m();          | [C.m()] ≤ [y]                 |
| 17 | y.q(); }            | [y] ≤ B                       |
| 18 | }                   |                               |

Abb. 1. Java Code mit Typbeschränkungen

dem Ergebnis von *m* eine Methode aus *B* aufgerufen wird. Der Ansatz von [3] erlaubt es diesen Zusammenhang formal abzuleiten:

Abb. 1 zeigt neben den unteren Codezeilen eine aus der jeweiligen Zeile abgeleitete Typbeschrän-

kung. Z.B. folgt aus der Zuweisung des Rückgabewertes von *m* an *y* (Zeile 16), dass der Rückgabewert von *m* ein Subtyp des Typs von *y* sein muss. Aus den aufgeführten Ungleichungen ergibt sich die Ungleichungskette  $[x] \leq [C.m()] \leq [y] \leq B$  und damit die behauptete Notwendigkeit. Die statischen Deklarationen des Typs von *x* oder *y* werden dabei nicht mit einbezogen, weil diese ja gerade zur Disposition stehen.

## 2. Logische Repräsentation und Analyse von Programmen

JTransformer [1] transformiert ein Java Programm in eine logische Faktenbasis, die die Grundlage für Programmanalysen darstellt. Wie in Abb. 2 illustriert, wird dabei jeder Knoten des abstrakten Syntaxbaumes eines Programms als ein logisches Fak-

```
classDef(clsC, pkgMini, 'C').
methodDef(mthM, clsC, 'm', clsB).
localDef(varX, mthM, clsB, 'x', newB).
newClass(newB, varX, ctrB, refB).
ident(refB, newB, clsB).
call(cllP, mthM, refX1, mthP).
ident(refX1, cllP, varX).
return(retX, mthM, refX2).
ident(refX2, retX, varX).
methodDef(mthN, clsC, 'n', void).
localDef(varY, mthN, clsB, 'y', cllM).
call(cllM, varY, 'this', mthM).
call(cllQ, mthN, refY, mthQ).
ident(refY, cllQ, varY).
```

Abb. 2. Fakten für C

tum dargestellt<sup>1</sup>. Das Prädikatsymbol repräsentiert den Typ des Knotens. Der erste Parameter wird als eine eindeutige Identität dieses Knotens verwendet. Die anderen Parameter

sind entweder primitive Werte (Zahlen, Namen) oder Identitäten anderer Knoten, die Referenzen auf diese Knoten darstellen<sup>2</sup>.

Prädikate können auch ganz oder teilweise über mitunter rekursive Ableitungsregeln definiert werden. Somit ist der Übergang von der Programmdarstellung zu Analysen dieser Darstellung fließend.

Der Begriff des ‚declaration element‘ bezeichnet in [3] die Deklaration des statischen Typs von Methoden, Parametern, Feldern und lokalen Variablen, was sich in Prolog durch das folgende Prädikat `decl_element` ausdrücken lässt<sup>3</sup>:

<sup>1</sup> Die Einrückung in Abb. 2 hat keine Bedeutung, sie verdeutlicht lediglich die Schachtelung der Elemente. Die gezeigte Faktendarstellung ist vereinfacht.

<sup>2</sup> So ist zum Beispiel das zweite Argument eines jeden Faktens eine Referenz auf den Elternknoten.

<sup>3</sup> Großschreibung in Regeln oder Fakten bezeichnet logische Variablen, Kleinschreibung Konstanten. Die ‚anonyme Variable‘ `_` steht

```

decl_element(Elem, Type) :- methodDef(Elem, _, _, Type).
decl_element(Elem, Type) :- paramDef(Elem, _, Type, _, _).
decl_element(Elem, Type) :- fieldDef(Elem, _, Type, _, _).
decl_element(Elem, Type) :- localDef(Elem, _, Type, _, _).

```

Dies erlaubt erste Abfragen über unser Beispiel:

```

?- decl_element(mthM, clsB): YES
?- decl_element(cllM, clsB): NO
?- decl_element(Elem, clsB): {ctrB, mthM, varX, varY}

```

Die Abbildung eines Ausdrucks auf die Deklaration des davon referenzierten Elements wird durch das Prädikat `referenced_decl` umgesetzt:

```

referenced_decl(Ident, RefElem) :- ident(Ident, _, RefElem).
referenced_decl(Call, CalledM) :- call(Call, _, _ CalledM).
referenced_decl(Call, CalledC) :- newClass(Call, _, CalledC, _).

```

### 3. Logikbasierte Umsetzung von 'Type Constraints'

Im Beispiel müssen aufgrund der Verwendung der Methoden `p` und `q` die Variablen `x` und `y` mindestens den Typ `A` bzw. `B` haben. Die Zuweisungen und Ergebnissrückgaben implizieren dann Typbeschränkungen zwischen typdeklarierenden Elementen. Daraus ergibt sich eine Ungleichungskette, die zeigt, dass auch `x` den Typ `B` haben muss. Im Folgenden zitieren wir die entsprechenden formalen Definitionen und Implikationen aus [3] und geben dazu unsere Implementierung an.

Der Typ eines Ausdrucks auf dem die Methode `M` aufgerufen wird muss ein Subtyp sein von `Decl(RootDef(M))`, d.h. des allgemeinsten Typs der eine Deklaration von `M` enthält<sup>4</sup>:

$$\text{(Call)} \quad \text{call } E.m() \text{ to a virtual method } M \Rightarrow [E] \leq \text{Decl}(\text{RootDef}(M))$$

Gegeben eine Implementierung von `RootDef`, ist das folgende Prädikat eine direkte Umsetzung:

```

constrained_by_type(Elem, Type) :-
  call(_, _, CalledOn, CalledMethod),
  referenced_decl(CalledOn, Elem),
  root_definition(CalledMethod, RootMethod),
  methodDef(RootMethod, Type, _, _).

```

Zuweisungen implizieren eine Subtyprelation zwischen den Typen der Ausdrücke rechts und links:

$$\text{(Assign)} \quad E_1 = E_2 \Rightarrow [E_2] \leq [E_1]$$

Auch diese Implikation lässt sich umsetzen:

```

constrained_by(Lower, Upper) :-
  assign(_, _, LHS, RHS),
  referenced_decl(LHS, Upper),
  referenced_decl(RHS, Lower).
constrained_by(Lower, Upper) :-
  localDef(Upper, _, _, InitExpression),
  referenced_decl(InitExpression, Lower).

```

für beliebige Werte. Jede Regel ist als Implikation von rechts nach links zu lesen. So bedeutet z.B. die erste Regel „Wenn `Elem` eine Methodendefinition mit Rückgabebetyp `Type` ist, dann ist `Elem` ein typdeklarierendes Element mit deklariertem Typ `Type`.“ Mehrere Regeln für das gleiche Prädikat sind disjunktiv verknüpft.

<sup>4</sup>Der Kürze der Darstellung halber ignorieren wir im Folgenden multiple Subtypbeziehungen und verzichten auf die Angabe der Prädikate `root_definition` und `subtype`.

Der Typ des Rückgabeausdrucks einer Methode muss ein Subtyp des deklarierten Rückgabetyps sein:

$$\text{(Ret)} \quad \text{return } E \text{ in method } M \Rightarrow [E] \leq [M]$$

```

constrained_by(Lower, Upper) :-
  return(_, Upper, Expression),
  referenced_decl(Expression, Lower).

```

Die Menge der typdeklarierenden Elemente, deren deklarierter Typ `C` sich nicht durch einen allgemeineren Typ `T` ersetzen lässt, umfasst alle Elemente die Subtypen einer Klasse `C` sein müssen, die *nicht* Obertyp von `T` ist. Hinzu kommen alle Elemente die Subtypen eines solchen nicht generalisierbaren Elements sein müssen:

$$\begin{aligned} \text{(Gen)} \quad \text{Bad}(P, C, T) = & \\ & \{ E \mid E \in \text{All}(P, C) \wedge [E] \leq C' \in \text{TC}_{\text{fixed}}(P) \wedge \\ & \neg T \leq C' \} \cup \\ & \{ E \mid E \in \text{All}(P, C) \wedge [E] \leq [E'] \in \text{TC}_{\text{fixed}}(P) \\ & \wedge E' \in \text{Bad}(P, C, T) \} \end{aligned}$$

Das Prädikat `not_generalizable` setzt diese Definition um. Die unterstrichenen Ausdrücke entsprechen dabei den jeweiligen Constraint-Prädikaten:

```

not_generalizable(Element, GeneralizedType) :-
  constrained_by_type(Element, Type),
  not( subtype(GeneralizedType, Type) ).
not_generalizable(Element, GeneralizedType) :-
  constrained_by(Element, Upper),
  not_generalizable(Upper, GeneralizedType).

```

### Fazit

Dieser Beitrag gibt eine Idee davon, dass die in JTransformer realisierte logikbasierte Umgebung die Implementierung komplexer Programmanalysen vereinfacht und die Rückverfolgbarkeit zur jeweiligen theoretischen Grundlage fördert. Die Erweiterbarkeit gewinnt ebenso. So ist es ein Leichtes auch die Hindernisse einer Typgeneralisierung zu finden:

```

constrained_by(Lower, Type, Reason, Rule) :-
  Reason = Call, Rule = '(Call)',
  call(Call, _, CalledOn, CalledMethod),
  ... weiter wie vorhin ...

```

Die Effizienz und Skalierbarkeit des Ansatzes wurde in [2] anhand von Design Pattern-Detection demonstriert. Das System unterstützt auch Programmtransformationen und wurde erfolgreich eingesetzt für die Implementierung von Refactorings, Aspektorientierung, Designmetriken, Pattern-Mining, sowie Bad Smell-, Clone- und Refactoring-Detection.

### Literatur

- [1] JTransformer-Projekt, <http://roots.iai.uni-bonn.de/research/jtransformer/>.
- [2] G. Kiesel, J. Hannemann, T. Rho: A Comparison of Logic-Based Infrastructures for Concern Detection and Extraction, LATE'07, March 12-16, 2007, Vancouver, BC
- [3] Frank Tip, Adam Kiezun, Dirk Bäumler. Refactoring for Generalization Using Type Constraints. In Proceedings of the 18th OOPSLA, pages 13–26. ACM Press, 2003.