# 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, kniesel}@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. JTransfomer 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 (‚type constraints') basierende Verfahren von Tip, Kiezun und Bäumer zur Entscheidung der 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 auch der Obertyp A genügen würde? Der Aufruf der Methode aus dem Typ A auf der Variablen x steht dem nicht im Wege. Tatsächlich erzwingt jedoch die Methode n

```
1   package mini;
2
3   class A {
4       void p() {}
5   }                              Zuweisungen,
6   class B extends A {            Methodenaufrufe und
7       void q() {}                Rückgabeanweisungen
8   }                              stellen Bedingungen an
9                                  die Typen [E] der betei-
10  class C {                      ligten Ausdrücke E:
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**

indirekt die Verwendung des Typs B in Methode m, da dort auf 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 Typbeschränkung. 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] ≤ [C.m()] ≤ [y] ≤ 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 Faktum dargestellt [1]. Das Prädikatensymbol 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[2].

```
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**

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

Der Begriff des ‚declaration element' bezeichnet in [3] die Deklaration des statischen Typs von Methoden, Parametern, Feldern und lokalen Vari-

[1] Die Einrückung in Abb. 2 hat keine Bedeutung, sie verdeutlicht lediglich die Schachtelung der Elemente.

[2] So ist zum Beispiel das zweite Argument eines jeden Faktes eine Referenz auf den Elternknoten.

ablen, was sich in Prolog durch das folgende Prädikat decl_element ausdrücken lässt[3]:

```
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(clIM, clsB):   NO
?- decl_element(Elem, clsB):  {ctrB, mthM, varX, und 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,_).
```

## 4. 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 Ergebnisrückgaben 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[4]:

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

Das folgende Prädikat constrained_by_type(E,T) setzt diese Folgerung direkt um. root_definition implementiert dabei die Funktion RootDef(M):

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

[3] Großschreibung in Regeln oder Fakten bezeichnet logische Variablen, Kleinschreibung Konstanten. Die ‚anonyme Variable' _ steht 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 Rückgabetyp Type ist, dann ist Elem ein typdeklarierendes Element mit deklariertem Typ Type." Mehrere Regeln für das gleiche Prädikat sind disjunktiv verknüpft.

[4] Der Kürze der Darstellung halber ignorieren wir hier multiple Subtypbeziehungen.

Der Typ des Ausdrucks auf der rechten Seite einer Zuweisung muss ein Subtyp dessen auf der linken Seite sein:

(Assign) $E_1 = E_2 \Rightarrow [E_2] \leq [E_1]$

Das Prädikat constrained_by(L,U) stellt die Beschränkung des Typs des Elements L durch den Typ des Elements U dar:

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

Die Menge der *nicht generalisierbaren* Elemente, deren deklarierter Typ *C* sich nicht durch einen allgemeineren Typ *T* ersetzen lässt, umfasst alle Elemente die laut anwendbarer Typbeschränkungen Subtypen einer Klasse *C'* sein müssten, die *nicht* Obertyp von *T* ist. Hinzu kommen alle Elemente die Subtypen eines nicht generalisierbaren Elements sein müssten. Der einfach unterstrichenen Ausdruck entspricht constrained_by_type, der doppelt Unterstrichene constrained_by:

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

Das Prädikat not_generalizable(E,T) setzt dies um:

```
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 JTransfomer realisierte logikbasierte Umgebung eine deutliche Vereinfachung der Implementierung komplexer Programmalysen ermöglicht. Die Umsetzung formaler Systeme ist direkt rückverfolgbar zur theoretischen Grundlage.

Die Effizienz und Skalierbarkeit des Ansatzes wurde in [2] demonstriert.

## Literatur


[1] JTransformer-Project, http://roots.iai.uni-bonn.de/research/jtransformer/.

[2] G. Kniesel, 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 Bäumer. Refactoring for Generalization Using Type Constraints. In *Proceedings of the 18th OOPSLA*, pages 13–26. ACM Press, 2003.