

Code-Transformationen

⑥ Codetransformation und Refactoring

Vorteile (halb-)automatischer Transformationen

Definition

Transformationssystem

Schritte

Bestandteile

Implementierung

Eigenschaften

Beispiele konkreter Transformationssysteme

Code-Transformationen

- Lernziele
 - Grundsätzliche Bestandteile und Vorgehen kennen
 - Transformationen anwenden können
 - Implementierung kennen
- Kontext
 - Code-/Entwurfsebene
 - Automatische Code-Transformationen beinhalten Mustersuche
 - Aspekte des Reengineering, nicht nur des Reverse Engineerings

(Halb-)Automatische Transformationen

- (halb-)automatisch und beliebig wiederholbar
- dokumentarisch
- Transformation ist Spezifikation der Änderung
- kontrollierbar
- Transformationen repräsentieren Implementierungswissen
- Vor- und Nachbedingungen der Transformationen sind prüfbar
- Erhalt der Semantik bei einer Transformation lässt sich zeigen

Transformation

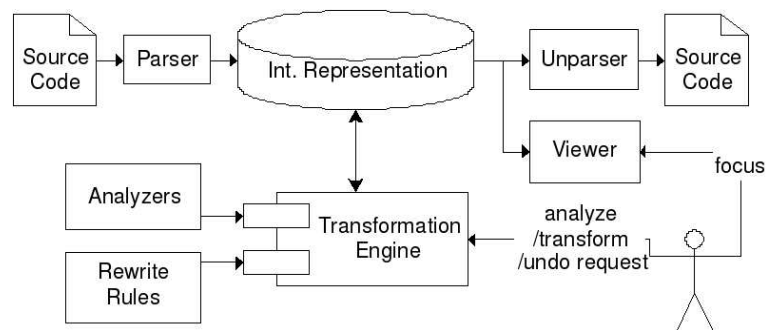
- Eine **Transformation** ist eine partielle Funktion t :
 - t : Spezifikation/Programm \mapsto Spezifikation/Programm
 - Beispiele: Compiler, YACC, Programmrestrukturierer
- Transformationen werden oft als Rewrite-Regeln mit Pattern-Variablen repräsentiert:

```
rule eliminate_additive_identity
  replace [expression]
    T [expression] + 0
  by
    T
end rule
```

Syntax von TXL; siehe www.txl.ca

Transformationssystem

Ein Transformationssystem ist ein System, das semantisch wohl-definierte (teil-)mechanisierte Programmmodifikationen ermöglicht.



Schritte einer Transformation

- ① Lokation: Identifikation der Stelle, an der Transformation angewandt werden soll

Schritte einer Transformation

- ① Lokation: Identifikation der Stelle, an der Transformation angewandt werden soll
 - ① Mustersuche

Schritte einer Transformation

- ① Lokation: Identifikation der Stelle, an der Transformation angewandt werden soll
 - ① Mustersuche
 - ② Lokation durch Benutzer

Schritte einer Transformation

- ① Lokation: Identifikation der Stelle, an der Transformation angewandt werden soll
 - ① Mustersuche
 - ② Lokation durch Benutzer
- ② Prüfung der Vorbedingungen der Transformation

Schritte einer Transformation

- ① Lokation: Identifikation der Stelle, an der Transformation angewandt werden soll
 - ① Mustersuche
 - ② Lokation durch Benutzer
- ② Prüfung der Vorbedingungen der Transformation
- ③ Anwendung, falls Vorbedingungen erfüllt sind

Vorbedingungen von Transformationen

Die Anwendbarkeit von Transformationen ist oft an Bedingungen geknüpft:

```
rule eliminate_additive_identity
  replace [expression]
    T1 [expression] + T2 [expression]
  where
    value (T2) = 0
  by
    T1
end rule
```

Umsetzung von Transformationen

Ersetzung wird als Prozedur auf dem abstrakten Syntaxbaum implementiert:

```
procedure eliminate_additive_identity (n : in out node) is
begin
  if type (n) = plus then  — Mustersuche
    if value (n.right) = 0 — Bedingung
      then
        — Ersetzung
        replace_child (parent (n), from => n, to => n.left);
        n.left.parent := n.parent;
        delete_tree (n.right); delete (n);
      end if;
    end if;
  end;
```

Eigenschaften von Transformationen

- Bestimmte Eigenschaften müssen bei der Transformation erhalten bleiben, andere Eigenschaften sollen sich ändern:
 - Performanz
 - Strukturiertheit
 - Änderbarkeit
 - ...
- In vielen (aber nicht allen) Fällen soll die Semantik erhalten bleiben.

Semantikerhaltende Transformationen I

Formale Betrachtung:

- Objekt o hat Attribute und ist definiert in einem Kalkül
 - grundsätzliche Wahrheiten (Axiome) A
 - Menge von Inferenzregeln i
- die Eigenschaften von Objekt o sind dessen Attributen sowie alle Fakten, die sich aus den Axiomen und Attributen mittels Inferenzregeln herleiten lassen
- Transformation t ist **semantikerhaltend** genau dann, wenn für alle Objekte o gilt:
 - die Eigenschaften von o bleiben durch die Transformation erhalten:
 $P(o) \subseteq P(t(o))$
 - $P(t(o))$ ist konsistent (enthält keine Widersprüche)(alte Eigenschaften bleiben erhalten, neue dürfen hinzukommen, es ergeben sich keine Widersprüche)

Semantikerhaltende Transformationen II

- Die Erhaltung der Semantik ist in der Praxis nur sehr schwer nachweisbar:
 - Beweis muss nicht nur die Semantik der Programmiersprache, sondern auch die aller aufgerufenen Betriebssystemfunktionen u.ä. einbeziehen, und ist im Allgemeinen schwierig.
- Nach Transformationen stets Regressionstests durchführen!

Beispiel eines Transformationssystems

TXL¹ (Queens University, Canada): generisches Transformationssystem;
ausgeprägt für C, C++, Java, Javascript, Modula, Object Pascal, XML

- abstrakter Syntaxbaum wird generiert
- funktionale Programmiersprache mit Transformationsregeln mit „native Patterns“:
- automatische Traversierung des abstrakten Syntaxbaums und Anwendung der Transformationen, solange dies möglich ist

¹<http://www.txl.ca>

TXL

```
rule eliminate_redundant_declarations
  replace [repeat statement]
    var X [id] : T [type_spec]
    Rest_of_Scope [repeat statement]
  where not
    Rest_of_Scope [references X]
  by
    Rest_of_Scope
end rule

function references X [id]
  match * [id] X
end function
```

Domain Maintenance System (DMS)³

- Hersteller: Semantic Designs, Austin, Texas, USA
- generisches Transformationssystem
- parametrisierbar durch so genannte Domains (Grammatiken)
- konkrete Instanzen für C, C++, Java, Cobol, Jovial, ...
- abstrakte Regeln für Parse-Baum
- Clone-Doctor² erkennt Klone (mit dem Verfahren von Baxter et al.) und beseitigt sie

²Eingetragenes Warenzeichen von Semantic Designs

³<http://www.semdesigns.com>

DMS-Beispiel

```
default base domain Java ;

rule merge-ifs(\condition1 ,
              \condition2 ,
              \then-statements)
=
  " if (\condition1)
    if (\condition2)
      { \then-statements }"
rewrites to
  " if (\condition1 && \condition2)
    { \then-statements }"
;
```

Raincode⁴

- Hersteller: Raincode, Brüssel, Belgien
- unterstützte Sprachen: Ada, APS, C, C++, COBOL, CSP, Delphi, Ideal, Informix 4GL, Java, Natural, PL/1
- Skript-Sprache, um Transformationen zu programmieren
- Transformation findet auf dem Text selbst statt

⁴<http://www.raincode.com>

Raincode-Beispiel I

```
PROCEDURE TERMINATE
VAR value;
BEGIN
  — scan all nodes
  FOR exp IN ROOT.SubNodes DO
    — if it is an expression that is not a simple literal
    — and it has not been modified yet
    IF exp IS NonCommaExpression
    AND exp IS NOT Literal
    AND exp["Patched"]<>TRUE THEN
      — do evaluation
      value := Eval(exp);
      — if evaluation succeeds
      IF value <> VOID THEN
        — replace the expression by its value
        PATCH.ReplaceNt(exp, value);
        — report something on console
        exp.WriteError(LIST.ToString(PATCH.NtImage(exp))
```

Raincode-Beispiel II

```
    || STR.Trim(X," ")," " &value);  
    — add annotation so we do not process subnodes  
    FOR IN exp.SubNodes DO  
        X.Patched := TRUE;  
    END;  
END;  
END;  
— save the processed source  
PATCH.Save(ROOT.SourceName & ".patched");  
END;
```