

Klonerkennung

Klonerkennung

Lernziele

- Varianten der Klonerkennung (Erkennung duplizierten Codes)
- Bezug zu Abstraktionsebenen von Programmdarstellungen

Kontext

- Beseitigung von Redundanz auf Codierungsebene
- erleichtert nachfolgende Reengineering-Aktivitäten

Copy&Paste–Programmierung

Szenario I:

- Neue Funktionalität soll hinzugefügt, die ähnlich ist zu einer existierenden.
- Auswirkungen sind nicht bekannt:
 - Wer benutzt die existierende Funktionalität bereits?
 - Was benutzt die existierende Funktionalität alles?
- Häufige Folge: Funktion wird kopiert und leicht modifiziert.

Szenario II:

- Wegen unzureichender Versionskontrolle existieren mehrere Versionen derselben Funktion im selben Code oder in verschiedenen Systemen.

Szenario III:

- Programmiersprache unterstützt keine Vererbung und keine generischen Einheiten. Compiler bietet kein Inlining.
- Folge: Code wird dupliziert (Untersuchungen: 5–10% Code sind redundant).

Folgen der Copy&Paste–Programmierung

Die Folgen sind duplizierter Code und damit höherer Aufwand für Wartung, Verstehen und Test:

- Schwierigere Fehlerkorrektur
- Quelle subtiler Fehler
- Höherer Aufwand beim Verstehen
- Mehr Daten zu analysieren und visualisieren
- Änderbarkeit leidet

Arten von Klonen

Die Kopie eines Programmfragments wird Klon genannt.

- Typ I: Exakte Kopie
 - Keinerlei Veränderung an der Kopie (bis auf White Space und Kommentaren).
 - Z.B. Inlining von Hand.
- Typ II: Kopie mit Umbenennungen (parametrisierte Übereinstimmung)
 - Bezeichner werden in der Kopie umbenannt.
 - Z.B. “Wiederverwendung” einer Funktion, generische Funktion von Hand

Arten von Klonen

- Typ III: Kopie mit weiteren Modifikationen
 - Code der Kopie wird abgeändert, nicht nur Bezeichner.
 - Z.B. “Erweiterung” einer Funktion.
- (Typ IV: Semantische Klone
 - Verschiedene Implementierungen desselben Konzepts.)

Beispiele

i = length (l); if (i < 3) {...}	i = length (l); if (i < 3) {...}	Typ-I
i = length (l); if (i < 3) {...}	k = length (j); if (k < x) {...}	Typ-II
i = length (l); if (i < 3) {...}	k = length (j); if (j < 3) {...}	zwei Typ-II bzw.ein Typ-II mit inkonsistenter Umbenennung
i = length (l); ... if (i < 3) {...}	k = length (j); ... if (k < x) {...}	Typ-III
x = x + 1;	x ++;	primitiver Typ IV

Typ-II-Klone: Was ist eine Umbenennung?

Gegeben seien zwei strukturell gleiche Code-Fragmente C1 und C2.

Welche Eigenschaften muss die Umbenennung haben?

- Gleiche Bezeichner in C1
 - **(G1)**: müssen auf gleiche Bezeichner in C2 konsistent abgebildet werden

oder

- **(G2)**: dürfen auf verschiedene Bezeichner in C2 abgebildet werden

Typ-II-Klone: Was ist eine Umbenennung?

- Unterschiedliche Bezeichner in C1
 - **(U1)**: müssen auf unterschiedliche Bezeichner in C2 abgebildet werden
- oder
 - **(U2)**: dürfen auf gleiche Bezeichner in C2 abgebildet werden

Typ-II-Klone: Was ist eine Umbenennung?

	(U1)		(U2)	
(G1)	a x	a x	a x	a x
	b y	a x	b y	a x/y
	<i>symmetrisch</i>		<i>nicht symmetrisch</i>	
(G2)	a x	a x	a x	a x
	b x/y	a x	b x/y	a x/y
	<i>nicht symmetrisch</i>		<i>symmetrisch, aber Bezeichner ohne Relevanz</i>	

Granularität der Klone

Das Ziel der Klonererkennung bestimmt Granularität.
Je feiner die Granularität, desto höher der Aufwand.

- Sequenzen von Anweisungen
 - Beseitigung von manuellem Inlining.
 - Kapselung von Operationen in Unterprogrammen.
- Unterprogramme und Datenstrukturen
 - Identifikation von Kandidaten für die Einführung generischer Konzepte.
 - Vereinigung von Varianten.
 - “Objektifizierung”
- Pakete
 - Wie oben, jedoch im größeren Stil.
 - Säuberung nach gescheiterter Versionskontrolle.

Ansätze der Klonererkennung

Die Ansätze unterscheiden sich in

- Granularität des berücksichtigten Wissens
 - Anweisungssequenzen, Funktionen, Pakete
- und der Abstraktionsebene der Analyse
 - Text, Token–Ebene, Syntax, quantifizierbare Merkmale, Semantik

Existierende Ansätze

- Anzahl unterschiedlicher Zeilen des Unix-Diff-Programmes
- Pattern Matching auf Ebene der Lexeme (Baker)
- Matching auf dem abstrakten Syntaxbaum (Baxter et al.)
- Vergleich quantifizierter Merkmale (Metriken) des Codes (Mayrand et al.)

Probleme bei der Erkennung

- Jede Zeile/Funktion/Datei muss mit jeder anderen Zeile/Funktion/Datei verglichen werden:
 - Wie kann quadratischer Aufwand vermieden werden?
- Wie kann von Bezeichnern geeignet abstrahiert werden (ohne dabei willkürlich jeden Bezeichner mit jedem gleichzusetzen)?
 - Welche Arten von Typ-II-Klonen sollen erkannt werden (injektiv/bijektiv)?
 - Soll von der Position der Bezeichner abstrahiert werden?
- Typ-III-Klone: Geklonte Codefragmente können zu größeren Klonen zusammengefasst werden.
 - Codefragmente müssen nicht direkt zusammenhängend sein.

Verfahren nach Baker

- Verfahren basiert auf Lexemen.
- Vermeidung des quadratischen Aufwands:
 - quasi-parallele Suche in einer Programmrepräsentation, die jedes mögliche Anfangsstück des Programms enthält.
- Abstraktion von Bezeichnern:
 - Bezeichner werden auf relative Positionen abgebildet.
- Typ-III-Klone:
 - Separater Schritt am Ende

Verfahren von Baker

- Für jede Codezeile wird eine Zeichenkette aus Parametersymbolen und Nichtparametersymbolen generiert (sogenannter **Parameter-String** oder **P-String**):
 - Struktur der Zeile wird auf eindeutiges Nichtparametersymbol abgebildet (Funktork)
 - Bezeichner werden in Argumentliste erfasst
 - Ergebnis ist *Funktork Argumentliste*
 - *Beispiel: $x = x + y \rightarrow (P = P + P; x, x, y) \rightarrow \alpha x x y$*
- Die Konkatenation der P-Strings aller Codezeilen repräsentiert das Programm. Beispiel:

```
i = length (l) i
if (i < 3) {x = x + y}
→ α i length l β i x x y
```

Verfahren von Baker

- Kodierung *prev* (s) jedes P-Strings s:
 - Erstes Vorkommen eines Bezeichners erhält Nummer 0.
 - Jedes weitere Vorkommen erhält den relativen Abstand zum vorherigen Vorkommen (Funktoren mitgezählt).
 - Beispiel:
 $\alpha \ i \ length \ l \ \beta \ i \ x \ x \ y$
 $\rightarrow \alpha \ 0 \ 0 \ 0 \ \beta \ 4 \ 0 \ 1 \ 0$
 - Abstraktion der Bezeichner, jedoch nicht ihrer Reihenfolge.
 - S ist ein **P-Match** von T genau dann, wenn
 $prev(S) = prev(T)$
 Beispiel: $x = x + y$ und $a = a + b$ sind ein P-Match wegen
 $\alpha \ 0 \ 1 \ 0 = \alpha \ 0 \ 1 \ 0$

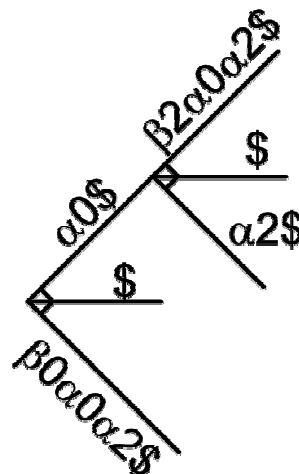
Verfahren von Baker

- Aufbau eines P-Suffix-Baums.

Sei $S_i = s_i s_{i+1} \dots s_n$ das i'te Suffix von S.

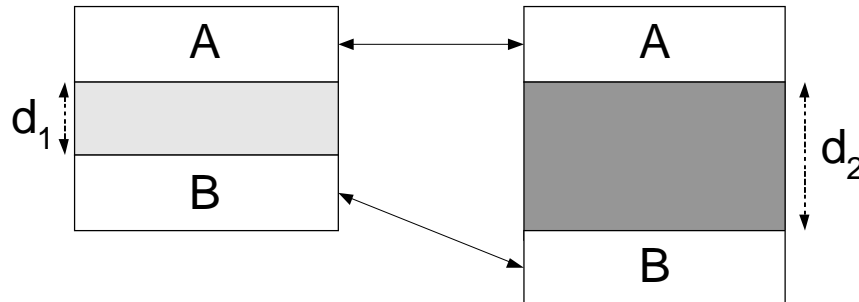
Der P-Suffix-Baum von S enthält alle P-Strings $prev(S_i)$ zu den Suffixen von S. Beispiel: $S = \alpha\beta\gamma\alpha x\alpha x\$$ (\$ ist das Endezeichen).

- $prev(S_1) = \alpha 0 \beta 2 \alpha 0 \alpha 2 \$$
- $prev(S_2) = 0 \beta 2 \alpha 0 \alpha 2 \$$
- $prev(S_3) = \beta 0 \alpha 0 \alpha 2 \$$
- $prev(S_4) = 0 \alpha 0 \alpha 2 \$$
- $prev(S_5) = \alpha 0 \alpha 2 \$$
- $prev(S_6) = 0 \alpha 2 \$$
- $prev(S_7) = \alpha 0 \$$
- $prev(S_8) = 0 \$$
- $prev(S_9) = \$$



Verfahren von Baker

- Klone lassen sich im Baum durch Verzweigungen identifizieren. (Benutzer legt die Mindestzeilenlänge der Klone fest; entspricht der Anzahl von Funktoren).
- Die gefundenen P-Matches werden, wenn möglich, zusammengefasst:



- Modi: (1) nur wenn $d_1 = d_2$
(2) wenn $\max(d_1, d_2) \leq \Theta$

Bewertung von Bakers Verfahren

- Bakers Klonerkennung ist lexem-basiert und damit sehr schnell:
 - Plattform: SGI IRIX 4.1, 40 Mhz R3000, 256 MB Hauptspeicher
 - System: 1,1 Mio LOC, mindestens zusammenhängende 30 LOC / Klon
 - nur 7 Minuten Analysezeit
- Der Ansatz ist invariant gegen Einfügung von Leerzeichen, Leerzeilen und Kommentaren.
- Der Ansatz ist weitgehend programmiersprachen-unabhängig (nur Regel für Bezeichner/Tokens notwendig).

Bewertung von Bakers Verfahren

Allerdings entgehen dem Ansatz:

- äquivalente Ausdrücke mit kommutativen Operatoren:

$x = x + y$	$x = y + x$
-------------	-------------

- gleiche Anweisungsfolgen, die verschieden umgebrochen wurden:

<code>if (a > 1) { x = 1 }</code>	<code>if (a > 1) { x = 1 }</code>
--------------------------------------	--

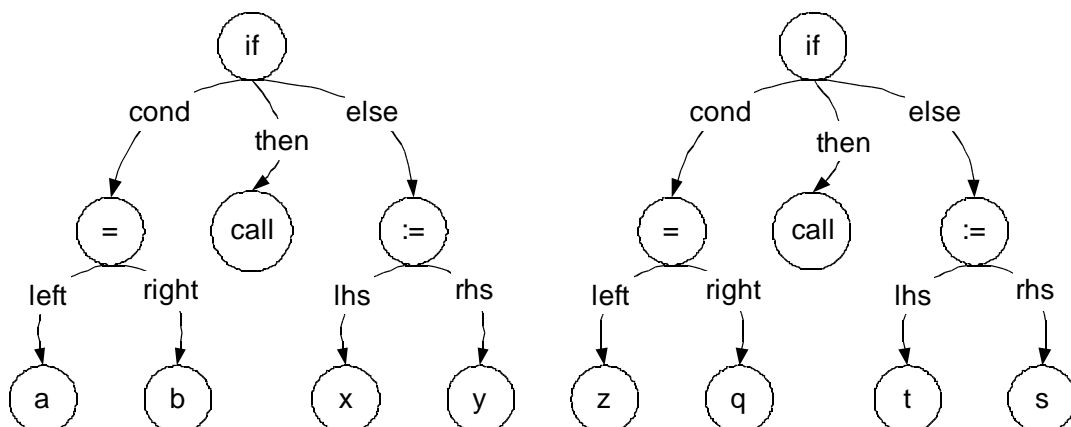
- gleiche Teilausdrücke:

<code>if (sp > 0)</code>	<code>if ((sp > 0) && (s [sp] != a))</code>
-----------------------------	--

AST-Matching

Ansatz basierend auf abstrakten Syntaxbäumen (ASTs):

- vergleiche jeden Teilbaum mit jedem anderen Teilbaum auf Gleichheit



Verfahren nach Baxter

- Verfahren basiert auf ASTs.
- Vermeidung des quadratischen Aufwands:
 - Partitionierung der zu vergleichenden Bäumen.
- Abstraktion von Bezeichnern:
 - Partitionierung und AST-Vergleich ignoriert Bezeichner.
- Typ-III-Klone:
 - Vergleich auf Ähnlichkeit statt Gleichheit.
 - Separater Schritt am Ende.

Skalierungsproblem und Typ-N-Klonerkennung

- Bäume werden durch Hash-Funktion partitioniert.
- Nur Bäume innerhalb einer gemeinsamen Partition werden verglichen.
- Hash-Funktion gleich wie bei Erkennung von gemeinsamen Teilausdrücken durch optimierende Compiler:
 $h : \text{node-type} \times \text{arg1} \times \text{arg2} \dots \times \text{argn} \rightarrow \text{Integer}$
 - Typ-I-Klonerkennung: h liefert genaue Partitionierung
 - Typ-II/III-Klonerkennung: h ignoriert Bezeichner

Skalierungsproblem und Typ-N-Klonerkennung

- Bäume werden auf Ähnlichkeit verglichen (nicht auf Gleichheit):

$$\text{Similarity}(T_1, T_2) = \frac{2 \times \text{Same}(T_1, T_2)}{2 \times \text{Same}(T_1, T_2) + \text{Different}(T_1, T_2)}$$

Similarity muss über benutzerdefiniertem Schwellwert Θ_S liegen: $\text{Similarity}(T_1, T_2) \geq \Theta_S$

- Nur Bäume T mit $\text{mass}(T) \geq \Theta_M$ werden betrachtet (mass = Anzahl von Knoten, Θ_M = benutzerdefinierter Schwellwert).
- Bei kommutativen Operatoren werden wechselseitig beide Teilbäume verglichen.

Probleme (und Lösungen) des AST-Matchings

- Skalierung
 - Vermeidung unnötiger Vergleiche
- Nicht nur Typ-I-Klone sollen erkannt werden.
 - Abstraktion von Bezeichnern.
 - Die Teilausdrücke müssen nicht gleich sein, sondern nur ähnlich genug.
- Klone sind Teil anderer Klone (geschachtelte Konstrukte).
 - Klone, die Teil eines anderen Klons sind, werden ignoriert, d.h. zusammengesetzte Klone subsumieren ihre Teile.

Basis-AST-Matching (Schritt 1)

Clones := \emptyset

```
for each subtree t loop
    if mass (t)  $\geq \Theta_m$  then
        hash t to bucket
    end if;
end loop;
for each subtree s and t in the same bucket loop
    if Similarity (s, t)  $\geq \Theta_s$  then
        for each subtree s' of s where Is_Member (s', Clones) loop
            Remove_Clone_Pair (Clones, s');
        end loop;
        for each subtree t' of t where Is_Member (t', Clones) loop
            Remove_Clone_Pair (Clones, t');
        end loop;
        Add_Clone_Pair (Clones, s, t);
    end if;
end loop;
```

AST-Matching von Sequenzen (Schritt 2)

- Sequenzen von Bäumen treten z.B. bei Anweisungsfolgen und Deklarationslisten auf:

<pre>void f () { x = 0; <u>a = a + 1;</u> <u>b = b + 2;</u> <u>c = c + 3;</u> w = g(); }</pre>	<pre>void g () { y = 2 * x; <u>a = a + 1;</u> <u>b = b + 2;</u> <u>c = c + 3;</u> i = h() * 3; }</pre>
--	--

- Der Basis-Matching-Algorithmus identifiziert nur die gleichen Zuweisungen, ignoriert aber die Gleichheit der beiden Anweisungsteilfolgen als Ganzes.
- Auch bei hinzugefügten bzw. gelöschten Anweisungen innerhalb von Anweisungsfolgen sollten wenigstens die gleichen Teilfolgen erkannt werden können.
- Der nächste Schritt identifiziert maximale gemeinsame Teilfolgen zweier Sequenzen, mit einer benutzerdefinierten Mindestlänge.

Erkennung von Klonsequenzen (Schritt 2)

```
for k in Min_Length .. Max_Length loop
  place all subsequences of length k into buckets;
  for each subsequence i and j in same bucket loop
    if Compare_Sequence (i, j, k) >  $\Theta_S$  then
      Remove_Sequence_Subclones_Of (Clones, i, j, k);
      Add_Sequence_Clone_Pair (Clones, i, j, k);
    end if;
  end loop;
end loop;
```

Erkennung zusammengesetzter Klone (Schritt 3)

- Gemeinsame Anweisungsfolgen wurden möglicherweise erst im zweiten Schritt nach dem Basis-AST-Matching erkannt.
- Durch die neu erkannten Klone könnten die Konstrukte, die diese Klone umfassen, nunmehr doch ähnlich sein.

<pre>s = 0; x = y; while (x > 0) { s = s + x; x = x - 1; ...ähnlich... }</pre>	<pre>p = 0; a = b; while (a > 0) { p = p * x; a = a - 1; ...ähnlich... }</pre>
---	---

D.h. die Vaterknoten der als Klone erkannten Teilbäume müssen noch einmal evaluiert werden.

Erkennung zusammengesetzter Klone (Schritt 3)

Clones_To_Generalize := Clones

while Clones_To_Generalize $\neq \emptyset$ **loop**

 Remove clone (s, t) from Clones_To_Generalize

if Compare_Clones (Parent (s), Parent (t)) $\geq \epsilon_s$ **then**

 -- *berücksichtige ähnliche Sequenzen*

 Remove_Clone_Pair (Clones, s, t);

 Add_Clone_Pair (Clones, Parent (s), Parent (t));

 Add_Clone_Pair (Clones_To_Generalize,
 Parent (s), Parent (t));

end if;

end loop;

Bewertung des AST-Matchings

- AST-Matching ist syntax-orientiert und deshalb aufwendiger als lexem-basierter Ansatz, da Parsing notwendig ist
 - beim lexem-basierten Ansatz müssen nur Schlüsselworte und Trennzeichen erkannt werden
 - was aber bei manchen Programmiersprachen eben doch Parsing voraussetzt, z.B. PL/1:
 IF IF = ELSE THEN ELSE := THEN ELSE IF := ELSE
- dafür aber genauer:
 - kommutative Operatoren werden berücksichtigt
 - syntaktische Einheiten werden verglichen, statt einzelner Code-Zeilen
 - übereinstimmende Teilausdrücke werden erkannt

Verfahren nach Mayrand et al.

- Verfahren basiert auf Metriken des Codes.
- Vermeidung des quadratischen Aufwands:
 - im Prinzip immer noch quadratisch, Vergleich ist aber relativ billig.
- Abstraktion von Bezeichnern:
 - Bezeichner werden von Metriken ignoriert.
- Typ-III-Klone:
 - Durch Toleranz der Metriken bzw. ihrer Zusammenfassung.

Vergleich auf Basis von Kennzahlen

- Hoffnung:
 $Code_1 = Code_2$
 $\Leftrightarrow \text{Kennzahlen}(Code_1) = \text{Kennzahlen}(Code_2)$
- Granularität üblicherweise Funktionsebene, da hierfür viele Metriken existieren
- Aspekte (nach Mayrand et al.):
 - Namen
 - Layout
 - Anweisungen
 - Kontrollfluss

Vergleichsmetriken (Mayrand et al.)

- Name
 - relative Anzahl gemeinsamer Zeichen
- Layout
 - Anzahl Zeichen von Kommentaren (Deklarationsteil, Implementierungsteil)
 - Anzahl mehrzeiliger Kommentare
 - Anzahl nicht-leerer Zeilen (inklusive Kommentare)
 - durchschnittliche Länge der Bezeichner

Vergleichsmetriken (Mayrand et al.)

Anweisungen

- gesamte Anzahl von Funktionsaufrufen
- Anzahl verschiedener Aufrufe
- durchschnittliche Komplexität der Entscheidungen in der Funktion
- Anzahl der Deklarationen
- Anzahl ausführbarer Anweisungen

Vergleichsmetriken (Mayrand et al.)

Kontrollfluss

- Anzahl der Kanten im Kontrollflussgraphen (KFG)
- Anzahl der Knoten im KFG
- Anzahl der Bedingungen im KFG
- Anzahl der Pfade im KFG
- Komplexitätsmetrik über dem KFG

Zusammenfassung der Metriken

Zwei Funktionen f_1 und f_2 sind in Bezug auf einen Aspekt:

- **gleich**: gleiche Metrikwerte
- **ähnlich**: alle Metrikwerte liegen in einer gewissen Bandbreite (spezifisch für jede individuelle Kennzahl definiert), sind aber nicht gleich
- **verschieden**: mindestens ein Metrikwert liegt außerhalb einer Bandbreite

Klassifikation

- **Exakte Kopie:** Funktionen sind in jedem Aspekt gleich (Typ-I-Klon)
- **Ähnliches Layout:** Ähnliches Layout und ähnliche Namen, gleiche Anweisungen und Kontrollfluss (\approx Typ-II-Klon)
- **Ähnliche Ausdrücke:** Name und Layout sind verschieden, Anweisungen und Kontrollfluss sind gleich (\approx Typ-II-Klon)
- **Verschieden:** Alle Aspekte sind verschieden.

Bewertung des Ansatzes von Mayrand et al.

- Aspekte sind nicht unabhängig.
- Definition der Bandbreite ist notwendig.
- Klassifikation ist unvollständig.
- Präzision?

Präzision

Code1 = Code2

⇒ Kennzahlen (Code1) = Kennzahlen (Code2)
ist erfüllt.

Code1 ≈ Code2

⇒ Kennzahlen (Code1) ≈ Kennzahlen (Code2)
ist erfüllt.

Kennzahlen (Code1) = Kennzahlen (Code2)

⇒ Code1 = Code2 ?

Kennzahlen (Code1) ≈ Kennzahlen (Code2)

⇒ Code1 ≈ Code2 ???