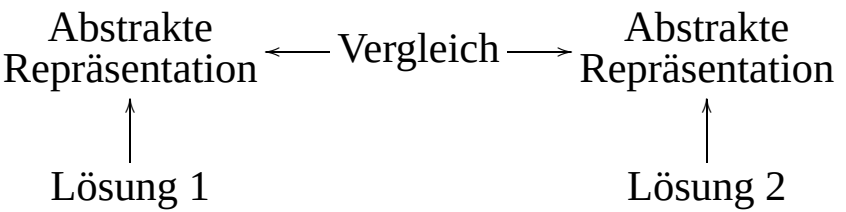
Das Problem

- Es ist nicht immer möglich, die partielle Korrektheit eines Programms zu überprüfen (analog zu Halteproblem)
- Rein syntaktischer Vergleich reicht nicht aus: Variation sowohl auf der syntaktischen als auch auf der algorithmischen Ebene

Die Idee


Abstrakte Repräsentation
Vergleich
Abstrakte Repräsentation
Lösung 1
Lösung 2

Fehler

- Spezifikationsunabhängige Fehler:
  (z.B. Endlosschleifen, nicht initialisierte Variablen)
- Spezifikationsabhängige Fehler:
  (z.B. implizite Grenzen für Anzahl und Grösse der zu verarbeitenden Elemente, falsche Reihenfolge)

Programmbeschreibungen

- **Konzepte** als *Semantische Bausteine* des Programms
  Beispiel: Finden eines Maximums
- **Konzeptuelle Datensequenzen (CDS)**
  Beispiel: die Zweierpotenzen bis n, Eingaben von der Tastatur
- **Klischees** als Implementationsart für Konzepte und CDSen
  Beispiel: das Horner-Schema als Implementierung des Polynomauswertungs-Konzepts
  Ein Klischee kann wieder Referenzen auf Konzepte/CDSen besitzen
- **Semantisch annotierte Programmprimitive (SAPPs)**, die die Verbindung zwischen Code und Klischees darstellen.
  Beispiel: Summieren (a:=a+x), Zählen (a:=a+1), aber auch: laufender Durchschnitt (x:=(x*c+i)/(c+1))
  SAPP$\neq$Codezeile

Konzepte

- Beispiele: Finden eines Minimums/Maximums, eine Aktion n-mal wiederholen, eine Matrix multiplizieren, einen Mittelwert bilden
- Eine oder mehrere Standard-Implementationsweisen sind bekannt. Dies sind die *Klischees*

CDSen

- **Eingabe-CDS** für eine Reihe von Eingabewerten
- **Aufzählungs-CDS** für eine Reihe von konsekutiven Werten (1,2,3,...)
- **Berechnungs-CDS** für eine Reihe von berechneten Werten
- **Kombinations-CDS** für Reihen von Elementen, die aus anderen CDS zusammengesetzt werden

Beispiel

Ein imperatives Programm zur Berechnung eines Polynoms (Horner-Schema)

```
VAR x,n,a,i,res : INTEGER;

BEGIN
  READ(x);
  READ(n);
  res:=0;
  FOR i:=0 TO n DO
  BEGIN
    READ(a);
    res:=(res*x)+a;
  END;
  WRITE(res)
END.
```

Die Programmbeschreibung

```
INPUTS:
  input(1): value
  input(2): value
  input(3): CDS_ref1
OUTPUTS:
  output(1): value
CONCEPT PE_ref1
  type = polynomial_evaluation
  eval_value-source = input(1)
  degree-known? = yes
        -source = input(2)
  coefficients-order  = descending
              -source = CDS_ref1
  results-pol_value = output(1)
CDS CDS_ref1
  type = input
  control-type = fixed_number
  properties-exact_number = input(2)+1
```

Implementation

- Analyse der SAPPs

$$x_{var} := x_{var} * a_{var} \longrightarrow \begin{bmatrix} sapp & \\ \text{type} & \text{accumulating\_sum} \\ \text{instructions} & \text{(Programmzeile)} \\ \text{target} & x \\ \text{sum} - \text{what} & y \end{bmatrix}$$

- Kontrollfluss
- Datenfluss (beides als gerichtete Pfeile zwischen SAPPs)
- Analyse der Klischees
  Bestimmte SAPPs werden als Indiz für das Vorhandensein eines Klischees gedeutet (Beacon).
  Fehlende zusätzliche Teile des Klischees können als spezifikationsunabhängige Fehler gedeutet werden oder das Klischee wird mit einer

Warnung markiert, die u.U. als spezifikationsabhängiger Fehler gedeutet werden kann.

- aus den erkannten Klischees werden Konzeptbeschreibungen erstellt.
- Code (SAPPs), dessen Funktion nicht erkannt wird, führt zu einer Warnung.

Verlässlichkeit

- Der Programmanalysierer sollte keine Fehler in korrekten Programmen finden und auch keine fehlerhaften Implementationen als fehlerfrei beurteilen.
- Studenten sind i.A. sehr kreativ, was das Ausdenken unerwarteter (aber möglicherweise trotzdem richtiger) Lösungen

$\Rightarrow$ Der Programmanalysierer sollte seine eigenen Grenzen kennen

Nutzen der Programmanalyse

- Paraphrasieren des Programms
- Geben von verständlichen Hinweisen (Annahme: Konzepte entsprechen der Denkweise des Studenten)

Programmbeschreibung für Prolog

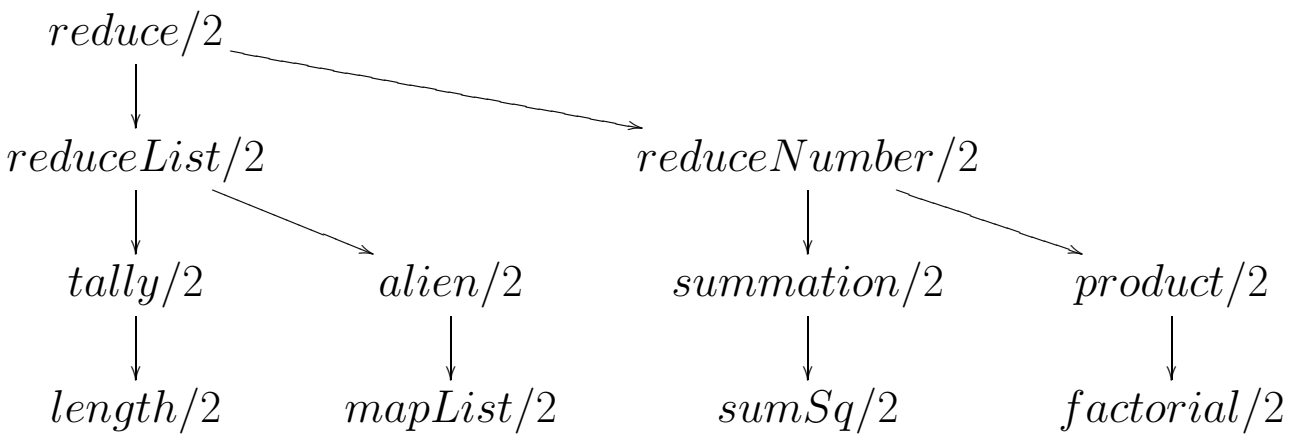
- Schemata (Tim Gegg-Harrison)
- Techniques (Paul Brna et al)

Schemata

Prädikate mit “Löchern”

```
pred([],___).
pred([H|T],___):-
  ___,
  pred(T,___),
  ___.
```

Schema-Hierarchie



Techniques

beschreiben die Beziehung zwischen Variablen im Klauselkopf und denen in der Argumentposition des Rekursionsaufrufes.

- **same** `p(...X...):-p(...X...)`
- **list subgoal** `p(...T...):-p(...[H|T]...)`
- **after** `p(...,X):-p(...,Y),g(...,X,Y)`
- **accumulator pair** als Kombination von *same* und *before*
  `p(X,Z):-g(X,Y),p(Y,Z)`

Darstellung als Grammatik

(Jun Hong / ITS 1998)

$$
\begin{aligned}
\langle \text{technique\_A\_program} \rangle &::= \langle \text{base case} \rangle_{Pred}\ \langle \text{recursive case} \rangle_{Pred} \\
\langle \text{base case} \rangle_{Pred} &::= \langle \text{pred} \rangle_{Pred} \textbf{([ ],}\langle \text{optional args} \rangle\textbf{)} \\
\langle \text{recursive case} \rangle_{Pred} &::= \langle \text{rule head} \rangle_{Pred,H,T}\ \textbf{:-}\ \langle \text{rule body} \rangle_{Pred,H,T} \\
\langle \text{rule head} \rangle_{Pred,H,T} &::= \langle \text{pred} \rangle_{Pred} \textbf{(}\langle \text{list split} \rangle_{H,T} \textbf{,}\langle \text{optional args} \rangle\textbf{)} \\
\langle \text{list split} \rangle_{H,T} &::= \textbf{[}\langle \text{list head} \rangle_{H} \textbf{|}\langle \text{list tail} \rangle_{T}\textbf{]} \\
\langle \text{rule body} \rangle_{Pred,H,T} &::= \ldots
\end{aligned}
$$

...

Zusätzlich: Klassifikation in einer Hierarchie von Schemata