Automatisierte Logik und Programmierung

Lektion 17

Synthese im Kleinen
Paradigmen & Strategien

1. Grundsätzliche Ansätze

2. Beweise als Programme

3. Synthese durch Transformationen

Ansätze zur Programmsynthese

- **Künstliche Intelligenz: Automatisches Programmieren**
  - Intelligenter Agent ersetzt menschlichen Programmierer
  - Ziel ist Vollautomatische Erzeugung von Programmen aus Spezifikationen
  - Strategien konzentrieren sich auf logisch-deduktive Verfahren
  - Forschungen lieferten wichtige theoretische Grundlagen
  - Verfahren erzeugen meist nur einfache Algorithmen $\mapsto$ "Synthese im Kleinen"
- **Software-Engineering: Programmierunterstützung**
  - Synthesewerkzeug unterstützt den menschlichen Programmierer
  - Benutzergesteuerte Erzeugung von Programmen mit Korrektheitsgarantie
  - Strategien verwenden symbolisch-algebraische Techniken
  - Infrastruktur benötigt theoretische Grundlagen aus der Deduktion
  - Forschung liefert Formalisierung von Programmierwissen und -methoden
  - Verfahren liefern praktisch wertvolle Algorithmen
  - Noch zu wenig Unterstützung für modulare und verteilte Systeme

Wie kann man an das Syntheseproblem herangehen?

- **Unterscheidungsmerkmale**
  - Repräsentation des Syntheseproblems als interne Aufgabenstellung
  - Interne Lösungsmethode: Art der Inferenzen
  - Konstruktion des Algorithmus aus interner Lösung
- **Beweise als Programme**
  - Extraktion aus konstruktivem Beweis eines Theorems
- **Synthese durch Transformationen**
  - Äquivalenzumformungen in ausführbare, meist rekursive Form
- **Gegenseitige Simulation prinzipiell möglich**

Synthese–Strategien

- **Konkrete Verfahren zur Programmsynthese**
  - Steuern Anwendung eines Formalismus (Beweis / Transformation)
  - Strukturieren Lösungsweg durch interne Teilziele
  - Lösen Teilziele durch heuristisch kontrollierte Suche
  - Codieren Wissen über Programme und Programmstrukturen
  - Verwenden ggf. Rückfragen an Benutzer
- **Methoden nicht gebunden an Paradigma**
  - Semi-Formale Methoden: Mehr eine Arbeitsvorschrift
  - Automatische Beweiser: Logik erster Stufe, Induktionsbeweisen, ...
  - Rewrite-Techniken: Für Transformationen oder Beweisführung

⋮

Historisch relevante Verfahren

- **Informale Methoden**
  – Polya, Dijkstra, Gries, ... (Lehrbücher)
- **Beweiser mit Skolemisierung und Resolution**
  – Green, 1969 (Stanford/Kestrel Institute)
  – Manna/Waldinger, 1971,75 (SRI International)
  Transformations- und Formationsregeln
  – Manna/Waldinger, 1972–1980
  Beweiskalkül gekoppelt mit Transformationen
  – Manna/Waldinger, 1980–
- **Fold/Unfold Techniken**
  – Burstall/Darlington (Edinburgh) 1975–81
- **Modifizierte Knuth-Bendix Vervollständigung**
  – Dershowitz (Illinois) 1985–
- **Synthese von Logikprogrammen**
  – Clark, Hogger (London) 1980 –

PARADIGMA: BEWEISE ALS PROGRAMME

- **Beweise Erfüllbarkeit einer Spezifikation**
  - Konstruktiver Beweis zeigt wie Ausgabe aus Eingabe bestimmt wird
  - Funktionales Programm implizit im Beweis enthalten
  - Korrektheit des Programms kann garantiert werden
  - Effizienz des Programms hängt von Art des Beweises ab

- **Grundsätzliche Vorgehensweise**
  - Gegeben sei die Spezifikation
    - $\texttt{FUNCTION}\ \boldsymbol{f}\texttt{(}\boldsymbol{x}\texttt{:}\boldsymbol{D}\texttt{):}\boldsymbol{R}\ \texttt{WHERE}\ \boldsymbol{I}[\boldsymbol{x}]\ \texttt{RETURNS}\ \boldsymbol{y}\ \texttt{SUCH THAT}\ \boldsymbol{O}[\boldsymbol{x},\boldsymbol{y}]$
  - Erzeuge **Spezifikationstheorem**: $\forall \boldsymbol{x}{:}\boldsymbol{D}.\exists \boldsymbol{y}{:}\boldsymbol{R}.\ \boldsymbol{I}[\boldsymbol{x}] \Rightarrow \boldsymbol{O}[\boldsymbol{x},\boldsymbol{y}]$
  - Suche formalen Beweis in konstruktivem logischen Kalkül
  - Extrahiere aus Beweis einen Algorithmus zur Berechnung von y aus x

- **Forschungsschwerpunkte**
  - Ausdrucksstarke Kalküle
  - Effiziente Beweisstrategien und Beweisplaner (Induktion)
  - Effiziente Extraktionsmechanismen (gute Algorithmen ohne Beweisballast)

Formaler Beweis: Integerquadratwurzel

```
⊢ ∀n:ℕ. ∃r:ℕ. r² ≤ n < (r+1)²
BY allR

  n:ℕ
  ⊢ ∃r:ℕ. r² ≤ n < (r+1)²
  BY NatInd 1

  .....basecase.....
       ⊢ ∃r:ℕ. r² ≤ 0 < (r+1)²
  √    BY existsR ⌈0⌉ THEN Auto

  .....upcase.....
       i:ℕ⁺,  r:ℕ,  r² ≤ i-1 < (r+1)²
       ⊢ ∃r:ℕ. r² ≤ i < (r+1)²
       BY Decide ⌈(r+1)² ≤ i⌉ THEN Auto

       .....Case 1.....
            i:ℕ⁺,  r:ℕ,  r² ≤ i-1 < (r+1)²,  (r+1)² ≤ i
            ⊢ ∃r:ℕ. r² ≤ i < (r+1)²
       √    BY existsR ⌈r+1⌉ THEN Auto'

       .....Case 2.....
            i:ℕ⁺,  r:ℕ,  r² ≤ i-1 < (r+1)²,  ¬((r+1)² ≤ i)
            ⊢ ∃r:ℕ. r² ≤ i < (r+1)²
       √    BY existsR ⌈r⌉ THEN Auto
```

Algorithmenkonstruktion aus Beweisen

- **Formale Grundlage der Methode**
  - **Ist t ein Beweisterm für $\vdash \forall x{:}D.\exists y{:}R.\ I[x] \Rightarrow O[x,y]$, so ist das folgende Program korrekt**

    `FUNCTION` f`(`x`:`D`):`R `WHERE` $I[x]$

    `RETURNS` y `SUCH THAT` $O[x,y]$

    $\equiv$ `fst(`t`(x))`
  - Algorithmus entsteht durch Unterdrückung des Korrektheitsbeweisanteils
  - Theoretisches Fundament: Curry Howard Isomorphismus

- **Konstruktionsmethode**
  - Extrahiere Beweisterm t aus vollständigem Beweis für

    $\forall x{:}D.\exists y{:}R.\ I[x] \Rightarrow O[x,y]$
  - Konstruiere den Programmkörper λ`x.fst(`t`(x))`
  - Optimiere durch Reduktion und Elimination überflüssiger Information

Beweisterm: Integerquadratwurzel

- **Mit Korrektheitsbeweisinformation**

```
let rec sqrt n
= if n=0 then <0,<Ax,Ax>>
  else let <r,%1>  = sqrt (n-1)
       in  if (r+1)²≤n  then <r+1,<Ax,Ax>>
           else <r,<Ax,Ax>>
```

- **Nach Projektion und Optimierung**

```
let rec sqrt n
= if n=0 then 0
  else let r = sqrt (n-1)
       in  if (r+1)²≤n  then r+1
           else r
```

$\lfloor\sqrt{n}\rfloor$ – Synthese eines effektiveren Algorithmus

```
⊢ ∀n:ℕ. ∃r:ℕ. r² ≤ n < (r+1)²
BY allR
  n:ℕ
  ⊢ ∃r:ℕ. r² ≤ n < (r+1)²
  BY NatInd4 1
  .....basecase.....
      ⊢ ∃r:ℕ. r² ≤ 0 < (r+1)²
  √  BY existsR ⌈0⌉ THEN Auto
  .....upcase.....
      i:ℕ,  r:ℕ,  r² ≤ i÷4 < (r+1)²
      ⊢ ∃r:ℕ. r² ≤ i < (r+1)²
      BY Decide ⌈((2*r)+1)² ≤ i⌉ THEN Auto
      .....Case 1.....
          i:ℕ,  r:ℕ,  r² ≤ i÷4 < (r+1)²,  ((2*r)+1)² ≤ i
          ⊢ ∃r:ℕ. r² ≤ i < (r+1)²
      √  BY existsR ⌈(2*r)+1⌉ THEN Auto'
      .....Case 2.....
          i:ℕ,  r:ℕ,  r² ≤ i÷4 < (r+1)²,  ¬(((2*r)+1)² ≤ i)
          ⊢ ∃r:ℕ. r² ≤ i < (r+1)²
      √  BY existsR ⌈2*r⌉ THEN Auto
```

```
let rec sqrt n
= if n=0 then 0
  else let r = sqrt (n÷4)
       in  if (2*r+1)²≤n  then 2*r+1
           else 2*r
```

Maximale Segmentsumme einer Liste von Zahlen

Gegeben eine Folge $a_1, a_2, \ldots, a_n \in \mathbb{Z}$ bestimme die Summe $\Sigma_{i=p}^{q} a_i$ eines Segmentes, die maximal bezüglich aller möglicher Segmentsummen ist

2	**3**	**-6**	**4**	**5**	**-3**	**8**	**-2**	**-1**	**5**	**-9**	**2**	**3**	16

- **Direkte Lösung leicht zu finden, aber ineffizient**
  - Aufsummieren aller Segmente und Vergleich

- **Lösungsansatz für eleganteren, effizienten Algorithmus**
  - Betrachte Eigenschaften von $M_n \equiv max\{\Sigma_{i=p}^{q} a_i \mid 1 \le p \le q \le n\}$
  - Induktive Analyse liefert
    - $M_1 = a_1, \quad M_{n+1} = max(M_n, max\{\Sigma_{i=p}^{n+1} a_i \mid 1 \le p \le n\})$
  - Definiere $L_n \equiv max\{\Sigma_{i=p}^{n} a_i \mid 1 \le p \le n\}$
    - $L_1 = a_1, \quad L_{n+1} = max(L_n + a_{n+1},\ a_{n+1})$

- **Umsetzung in formalen Beweis erforderlich**

Maximale Segmentsumme: Objekttheorie

- **Formale Begriffe**

$\mathsf{max}(i,j) \equiv \mathsf{if}\ i<j\ \mathsf{then}\ \mathtt{j}\ \mathsf{else}\ i$

$a_i \equiv a[i]$

$\Sigma_{i=p}^{q} a_i \equiv \mathsf{ind}(q-p; a_p;\ \mathtt{i,sum.\ sum}+a_{p+i+1})$

$M = \mathtt{maxseg}(a) \equiv \exists 1\leq \mathtt{k} \leq \mathtt{j} \leq |a|.\ M=\Sigma_{i=k}^{j} a_i\ \wedge\ \forall 1\leq \mathtt{p}\leq \mathtt{q}\leq |a|.\ M\geq \Sigma_{i=p}^{q} a_i$

$L = \mathtt{maxbeg}(a) \equiv \exists 1\leq \mathtt{j} \leq |a|.\ L=\Sigma_{i=1}^{j} a_i\ \wedge\ \forall 1\leq \mathtt{q}\leq |a|.\ L\geq \Sigma_{i=1}^{q} a_i$

- **Mathematische Gesetze**

M1: $\mathtt{a_1=maxbeg([a_1])\ =\ maxbeg(a_1.[])}$

M2: $\mathtt{a_1=maxseg([a_1])\ =\ maxseg(a_1.[])}$

M3: $\mathtt{L=maxbeg(a)}\ \Rightarrow\ \mathtt{max(L+a_1,a_1)\ =\ maxbeg(a_1.a)}$

M4: $\mathtt{M=maxseg(a)}\ \wedge\ \mathtt{L=maxbeg(a_1.a)}\ \Rightarrow\ \mathtt{max(M,L)\ =\ maxseg(a_1.a)}$

Maximale Segmentsumme: formale Synthese

- **Voraussetzung: Folge a ist nicht leer:**
  - Leichter darzustellen durch "a hat die Form $\texttt{a}_1\texttt{.a}$ für ein $a_1 \in \mathbb{Z}$"

- **Spezifikation der Berechnung von L und M**

$$\texttt{FUNCTION MAXSEG}(\texttt{a}_1{:}\mathbb{Z},\texttt{a}{:}\texttt{Seq}(\mathbb{Z})){:}\mathbb{Z}\times\mathbb{Z}\ \texttt{WHERE true}$$
$$\texttt{RETURNS L,M SUCH THAT L=maxbeg}(\texttt{a}_1\texttt{.a}) \wedge \texttt{M=maxseg}(\texttt{a}_1\texttt{.a})$$

- **Spezifikationstheorem**

$$\forall \texttt{a}{:}\texttt{Seq}(\mathbb{Z}).\forall \texttt{a}_1{:}\mathbb{Z}.\exists \texttt{L}{:}\mathbb{Z}.\exists \texttt{M}{:}\mathbb{Z}.\ \texttt{L=maxbeg}(\texttt{a}_1\texttt{.a}) \wedge \texttt{M=maxseg}(\texttt{a}_1\texttt{.a})$$

- **Struktur des Beweisterms**

$$\lambda \texttt{a}.\lambda \texttt{a}_1.\ \textsf{seqind}(\texttt{a};\ \langle \texttt{a}_1,\texttt{a}_1,pf_{base}\rangle;\ \ \texttt{x,l,v}.\lambda \texttt{a}_1.\textbf{let}\ \langle \texttt{L},\langle \texttt{M},\texttt{v}_2,\texttt{v}_3\rangle\rangle = \texttt{v x}$$
$$\textbf{in}\ \langle \textsf{max}(\texttt{L+a}_1,\texttt{a}_1),\textsf{max}(\texttt{M},\ \textsf{max}(\texttt{L+a}_1,\texttt{a}_1)),pf_{ind}\rangle)$$

- **Algorithmus nach Optimierung**

```
let rec MAXSEG(a1,a) = if a=[] then (a1,a1)
                        else let x.l = a
                             let <L,M>= MAXSEG(x,l)
                              let new-L =max(L+a1,a1)
                               in (new-L, max(M,new-L))
```

Maximale Segmentsumme: formaler Beweis

```
⊢ ∀a:Seq(ℤ).∀a_1:ℤ.∃L:ℤ.∃M:ℤ. L=maxbeg(a_1.a)  ∧  M=maxseg(a_1.a)
BY all_i THEN seq_e 1 THEN all_i                                   (Induktion auf  a)
|\
| a:Seq(ℤ), a_1:ℤ ⊢ ∃L:ℤ.∃M:ℤ. L=maxbeg(a_1.[])  ∧  M=maxseg(a_1.[])
| BY ex_i a_1 THEN ex_i a_1 THEN and_i
| |\
| | a:Seq(ℤ), a_1:ℤ ⊢ a_1=maxbeg(a_1.[])
| | BY lemma M1
| \
|   a:Seq(ℤ), a_1:ℤ ⊢ a_1=maxseg(a_1.[])
|   BY lemma M2
 \
  ... x:ℤ, l:Seq(ℤ), v:∀a_1:ℤ.∃L:ℤ.∃M:ℤ. L=maxbeg(a_1.l) ∧ M=maxseg(a_1.l)
  ⊢ ∃L:ℤ.∃M:ℤ. L=maxbeg(a_1.(x.l)) ∧ M=maxseg(a_1.(x.l))
  BY all_e 4 x THEN thin 4
   \
    ... v_1: ∃L:ℤ.∃M:ℤ. L=maxbeg(x.l) ∧ M=maxseg(x.l)
    ⊢ ∃L:ℤ.∃M:ℤ. L=maxbeg(a_1.(x.l)) ∧ M=maxseg(a_1.(x.l))
    BY ex_e 5 THEN ex_e 6 THEN and_e 7
     \
      ... L:ℤ, M:ℤ, v_2: L=maxbeg(x.l), v_3: M=maxseg(x.l)
      ⊢ ∃L:ℤ.∃M:ℤ. L=maxbeg(a_1.(x.l)) ∧ M=maxseg(a_1.(x.l))
      BY ex_i  max(L+a_1,a_1)  THEN  ex_i  max(M, max(L+a_1,a_1))  THEN and_i
      |\
      | ... v_2: L=maxbeg(x.l), v_3: M=maxseg(x.l)
      | ⊢  max(L+a_1,a_1) =maxbeg(a_1.(x.l))
      | BY lemma M3
       \
        ... v_2: L=maxbeg(x.l), v_3: M=maxseg(x.l)
        ⊢   max(M, max(L+a_1,a_1)) =maxseg(a_1.(x.l))
        BY ... lemma M3 ... lemma M4
```

Strategien: OYSTER/CLAM Beweisplaner

- **Planer simuliert Kalkül — Beweiser führt Plan aus**
  – Verzögerung von Entscheidungen durch Skolemvariablen und Unifikation

- **Induktionsbeweise**
  – **Induktionsschema** $\Leftarrow$ Analyse der Rekursionsstruktur + globales Schema
  – **Rippling**: Verschiebe Unterschiede zwischen Induktionshypothese und -schluß bis Funktion der Hypothese entsteht

---

Literatur:

– A. Bundy, A. Smaill, G. Wiggins. *The synthesis of logic programs from inductive proofs.* Computer Logic Proceedings, Springer Verlag, 1990.

– A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill. *Experiments with proof plans for induction.* JAR 7:303–324, 1991.

– A. Bundy. *The use of explicit plans to guide inductive proofs.* CADE-9, Springer LNCS 310, 111–120, 1988.

– A. Bundy. *Automatic guidance of program synthesis proofs.* Workshop on Automating Software Design, IJCAI-89, 57–59, 1989.

– A. Bundy, F. van Harmelen, A. Smaill, A. Ireland. *Extensions to the rippling-out tactic for guiding inductive proofs.* CADE-10, Springer LNCS 449, 132–146, 1990.

Beweise als Programme: Eigenschaften

- **Korrektheit des Programms ist garantiert**
- **Verschiedene Kalküle sind theoretisch äquivalent**
  – Große praktische Unterschiede
  – Sequenzenkalküle für Mensch und Maschine geeignet
- **Verschiedene Extraktionsverfahren**
  – Beeinflussen Effizienz erzeugter Programme
- **Beweisschritte zu atomar (Assemblerniveau)**
  – Beweise für komplexe Programme interaktiv kaum durchführbar
  – Automatisierung schwierig:
    · Analyse der Formel, Unifikation, Suche nach geeigneten Induktionen
  – Nur praktikabel in Kombination mit Definitionen und Spezialtaktiken

Paradigma: Synthese durch Transformation

- **Transformiere in effektiv ausführbare Formel**
  - Spezifikation ist ineffektive Formel
  - Transformationen verbessern algorithmisches Verhalten der Formel
  - Vorwärtsschließen ohne konkret vorgegebenes Ziel

- **Grundsätzliche Vorgehensweise**
  - Gegeben sei die Spezifikation
    - `FUNCTION` $f(x{:}D){:}R$ `WHERE` $I[x]$ `RETURNS` y `SUCH THAT` $O[x,y]$
  - Definiere neues Prädikat P über $D{\times}R$ durch:
    - $\forall x{:}D.\forall y{:}R.\ I[x] \;\Rightarrow\; (P(x,y) \Leftrightarrow O[x,y])$
  - Transformiere in äquivalente Formel der Gestalt:
    - $\forall x{:}D.\forall y{:}R.\ I[x] \;\Rightarrow\; (P(x,y) \Leftrightarrow O_f[x,y,P])$

    $O_f[x,y,P]$ darf nur aus erfüllbaren Prädikaten bestehen
  - Extrahiere Programm aus Formel oder interpretiere als Logik-Programm

- **Forschungsschwerpunkte**
  - Leistungsfähige Transformationsregeln
  - Effiziente Rewrite Techniken und Heuristiken für Vorwärtsinferenz

Programmtransformation

- **Historisch: Optimierung von Programmen**
  - Erzeuge abstraktes, verifiziertes Prototyp-Programm
  - Transformiere in äquivalentes effizienteres Programm
- **Synthese: Optimierung nichtausführbarer Programme**
  - Spezifikation $\equiv$ nichtausführbares Programm
  - Transformiere in äquivalente, ausführbare Formel

  Individuelle Formalismen variieren sehr stark

Literatur:

- R. M. Burstall, J. Darlington: *A Transformation System for Developing Recursive Programs*, JACM 24:44-67, 1977.
- C. J. Hogger: *Derivation of Logic Programs*, JACM 28:372–392, 1981.
- Z. Manna, R. Waldinger: *Synthesis: Dreams $\Rightarrow$ Programs*, IEEE.SE SE-5 (4):294–328, 1979.

Transformationen, präzisiert

- **Anwendung bedingter Ersetzungsregeln der Form**

  $\forall z{:}T.\;\; B[z] \;\Rightarrow\; (\; Q[z] \;\Leftrightarrow\; Q'[z] \;)$

  (Ersetze Vorkommen von $Q[z]$ durch $Q'[z]$, falls Bedingung $B[z]$ erfüllt)

  – Regeln sind Äquivalenzen oder Verfeinerungen (Implikationen)

- **Regeln ergeben sich aus**

  – Lemmata der Wissensbasis

  – Dynamisch erzeugte Definitionen

  – Elementare Tautologien und Abstraktionen

  – Dynamisch erzeugte Kombinationen

- **Mechanismus basiert auf Vorwärtsinferenz**

  – Ziel ist bestimmte Struktur der Formel zu erreichen

  – Starke heuristische Steuerung notwendig

Maximale Segmentsumme im Transformationsansatz

Bestimme $max\{\Sigma_{i=p}^{q} a_i \mid 1{\leq}p{\leq}q{\leq}n\}$ für Zahlen $a_1,\ldots,a_n \in \mathbb{Z}$

1. **Spezifikation der Berechnung von $max\{\Sigma_{i=p}^{q} a_i \mid 1{\leq}p{\leq}q{\leq}n\}$**

   ```
   FUNCTION MAXSEG(a:Seq(ℤ)):ℤ WHERE a≠[]
      RETURNS m SUCH THAT m=maxseg(a)
   ```

2. **Definiere neues Prädikat `MAXSEG`**

   ```
   ∀a:Seq(ℤ).∀m:ℤ. a≠[] ⇒ (MAXSEG(a,m) ⇔ m=maxseg(a))
   ```

3. **Generalisiere: Einführung eines Prädikats `MAX_AUX`:**

   ```
   ∀a:Seq(ℤ).∀m:ℤ. a≠[] ⇒ (MAXSEG(a,m) ⇔ ∃l:ℤ. MAX_AUX(a,m,l))

   ∀a:Seq(ℤ).∀m:ℤ. a≠[] ⇒ (MAX_AUX(a,m,l) ⇔ m=maxseg(a) ∧ l=maxbeg(a))
   ```

4. **Transformation durch Anwendung von Lemmata**

   ```
   ∀a:Seq(ℤ).∀m:ℤ. a≠[] ⇒ (MAXSEG(a,m) ⇔ ∃l:ℤ. MAX_AUX(a,m,l))

   ∀a:Seq(ℤ).∀m:ℤ. a≠[] ⇒ MAX_AUX(a,m,l)
           ⇔ |a|=1 ∧ m=a₁ ∧ l=a₁
            ∨ |a|>1 ∧ ∃m',l':ℤ. l'=maxbeg(tl(a)) ∧ l=max(a₁,l'+a₁)
                                ∧ m'=maxseg(tl(a)) ∧ m=max(l,m')
   ```

Maximale Segmentsumme: Transformationssynthese (2)

$\forall$a:Seq($\mathbb{Z}$).$\forall$m:$\mathbb{Z}$. a$\neq$[] $\Rightarrow$ (MAXSEG(a,m) $\Leftrightarrow$ $\exists$l:$\mathbb{Z}$. MAX_AUX(a,m,l))

$\forall$a:Seq($\mathbb{Z}$).$\forall$m:$\mathbb{Z}$. a$\neq$[] $\Rightarrow$ MAX_AUX(a,m,l)
$\Leftrightarrow$ |a|=1 $\wedge$ m=a_1 $\wedge$ l=a_1
$\vee$ |a|>1 $\wedge$ $\exists$m',l':$\mathbb{Z}$. l'=maxbeg(tl(a)) $\wedge$ l=max(a_1,l'+a_1)
$\wedge$ m'=maxseg(tl(a)) $\wedge$ m=max(l,m')

5. Einsetzen der Definition von MAX_AUX:

$\forall$a:Seq($\mathbb{Z}$).$\forall$m:$\mathbb{Z}$. a$\neq$[] $\Rightarrow$ (MAXSEG(a,m) $\Leftrightarrow$ $\exists$l:$\mathbb{Z}$. MAX_AUX(a,m,l))

$\forall$a:Seq($\mathbb{Z}$).$\forall$m:$\mathbb{Z}$. a$\neq$[] $\Rightarrow$ MAX_AUX(a,m,l)
$\Leftrightarrow$ |a|=1 $\wedge$ m=a_1 $\wedge$ l=a_1
$\vee$ tl(a)$\neq$[] $\wedge$ $\exists$m',l':$\mathbb{Z}$. MAX_AUX(tl(a),m',l')
$\wedge$ l=max(a_1,l'+a_1) $\wedge$ m=max(l,m')

6a. Umwandlung in Logik-Programm

```
MAXSEG(a,m) :- MAX_AUX(a,l,m).
MAX_AUX([x],x,x).
MAX_AUX(x.a',l,m):- MAX_AUX(a',m',l'), max(x,l'+a,l), max(l,m',m).
```

6b. Umwandlung in Funktionales Programm

```
let MAXSEG(a) =
     let rec MAX_AUX(a) = if |a|=1 then (a1,a1)
                          else let (m',l') = MAX_AUX(tl(a)) in
                                  let l=max(a1,l'+a1) in (max(l,m'),l)
in snd(MAX_AUX(a))
```

Transformationen: Programmerzeugung

- **Anwendung von Programmformierungsregeln**
  - Sprachspezifische Umwandlung einer Formelmenge in Programm
- **Erzeugung von Logikprogrammen**
  - Deklaratives Auslesen der Formeln mit impliziten Quantoren
  - Allquantoren und Eingabebedingung entfallen
  - Existenzquantoren rechts entfallen (freie Variablen werden instantiiert)
  - Disjunktionen rechts ergeben zwei Klauseln (nach Normalisierung)
  - Funktionsaufrufe `y=g(x)` werden zu Prädikaten `G(x,y)`
  - Destruktoren (`hd(a)`, `tl(a)`) werden zu Konstruktoren `x.a` im Kopf
  - Gleichheiten (`m=a`$_1$) werden direkt im Kopf eingesetzt
- **Erzeugung funktionaler Programme**
  - Disjunktion werden zu Fallunterscheidungen
  - Konjunktionen werden zu zusätzlichen Eingabebedingungen
  - Existenzquantoren werden zu Generalisierung + kaskadischer Aufruf
  - Rekursionen erfordern Terminierungsbeweis

Rechtfertigung von Programmformierungsregeln

- **Disjunktion $\longmapsto$ Fallunterscheidung**

  Sind `(D,R,I,O,body)` und `(D,R,I',O',body')` korrekt, dann auch

  `FUNCTION` f`(x:D):R WHERE I(x)` $\vee$ `I'(x) RETURNS y SUCH THAT O(x,y)` $\vee$ `O'(x,y)`

  $\equiv$ `if I(x) then body(x) else body'(x)`

- **Existenzquantor $\longmapsto$ Generalisierung**

  Sind `(D,R,I,O,body)` und `(R,R',J,O',body')` korrekt, dann auch

  `FUNCTION` f`(x:D):R' WHERE I(x)` $\wedge$ `J(body(x))`

  `RETURNS z SUCH THAT` $\exists$`y:R. O(x,y)` $\wedge$ `O'(y,z)`

  $\equiv$ `body'(body(x))`

- **Rekursive Formel $\longmapsto$ Rekursion**

  $\mathtt{f}_d\mathtt{:D}\not\to\mathtt{D}$ wohlfundierte 'Reduktionsfunktion' und

  1. $\forall$`x:D. I(x)` $\Rightarrow$ $\mathtt{I(f}_d\mathtt{(x))}$
  2. $\forall$`x:D.`$\forall$`y:R. I(x)` $\Rightarrow$ `O(x,y)` $\Leftrightarrow$ $\exists\mathtt{y}_r\mathtt{:R.\ O(f}_d\mathtt{(x),y}_r\mathtt{)} \wedge \mathtt{O}_C\mathtt{(x,f}_d\mathtt{(x),y}_r\mathtt{,y)}$
  3. `FUNCTION` $f_C\mathtt{(x,x}_r\mathtt{,y}_r\mathtt{:D}\times\mathtt{D}\times\mathtt{R):R\ WHERE\ I(x)\ RETURNS\ y\ SUCH\ THAT\ O}_C\mathtt{(x,x}_r\mathtt{,y}_r\mathtt{,y)}$

     $\equiv$ $\mathtt{body(x,x}_r\mathtt{,y}_r\mathtt{)}$ ist korrekt

  dann ist das folgende rekursive Programm korrekt

  `FUNCTION` f`(x:D):R WHERE I(x) RETURNS y SUCH THAT O(x,y)`

  $\equiv$ $\mathtt{body(x,\ f}_d\mathtt{(x),\ }f\mathtt{(f}_d\mathtt{(x)))}$

Strategien: LOPS – LOgische ProgrammSynthese

- **KI-Orientierter Ansatz:**
  - Syntaktische Transformationen logischer Formeln kontrolliert durch semantische Informationen
- **Kombination einer kleinen Menge von Teilstrategien**
  - **GUESS/DOMAIN**: Raten einer Teillösung
  - **GET-REC**: Rekursionseinführung
  - Vereinfachung
  - Erzeugung von Unterproblemen
  - Test auf Auswertbarkeit von Teilformeln

  ⋮
- **Algorithmenkonstruktion:**
  - Umformung rekursiver Formeln in logische/funktionale Programme

---

Literatur:

– Wolfgang Bibel: *Syntax–directed, Semantics–supported Program Synthesis* AI Journal 14:243–261, 1980

Ansatz hat sich nicht bewährt, da außer in Spezialfällen nicht formalisierbar

Synthese im Kleinen – Rückblick

- **Pradigmen sind i.w. gleichwertig**
  - Transformationen sind durch Beweise mit Gleichheitslemmata simulierbar
  - Beweisregeln können als Rewrite-Regeln beschrieben werden
- **Unterschiede liegen in Methodik**
  - Proofs-as-Programs: Analytischer Beweis eines Spezifikationstheorems
    - Korrektheitsgarantien stehen im Vordergrund
  - Transformationen: Vorwärtsinferenzen mit meist unverifiziertem Wissen
    - Effizienz steht im Vordergrund
- **Inferenzniveau zu niedrig**
  - Elementare Beweisregeln oder Transformationen
  - Ignoriert bekanntes Programmierwissen
  - Nicht skalierend: Suchraum explodiert bei nichttrivialen Problemen

$$\Downarrow$$

Wissenbasierter Ansatz erforderlich