Induktion von Rekursiven Programmschemata und Analoges Lernen

Ute Schmid & Fritz Wysotzki

FG Methoden der Künstlichen Intelligenz, Sekr. FR 5-8

Institut für Angewandte Informatik

Technische Universität Berlin

Franklinstr. 28/29, D-10587 Berlin

schmid@cs.tu-berlin.de, wysotzki@cs.tu-berlin.de

Running Head: Induktion von RPS und Analoges Lernen

Zusammenfassung

Wir stellen einen Ansatz zum Erwerb von Problemlösefertigkeiten vor, der auf dem im Bereich "Automatisches Programmieren" entwickelten Prinzip der Induktion rekursiver Programmschemata basiert. Dieser Ansatz ermöglicht es, drei Ebenen der Generalisierung zu modellieren: In einem ersten Schritt wird aufgrund konkreter Problemlöseerfahrungen ein bedingtes (initiales) Programm aufgebaut, das über die Problemlösungen für bereits explorierte Anfangszustände eines Problems generalisiert (*learning by doing*). Im zweiten Schritt wird aus diesem initialen Programm ein rekursives Programmschema inferiert. Dies entspricht einer Generalisierung über rekursiv aufzählbare Problemräume. In einem dritten Schritt kann von der konkreten Bedeutung der Operationssymbole eines Programmschemas abstrahiert werden. Die Struktur des rekursiven Programmschemas generalisiert über die Klasse strukturgleicher Probleme (*learning by analogy*). Mit diesem Ansatz ist es möglich, *learning by doing* und *learning by analogy* in einem einheitlichen Rahmen zu beschreiben. Dabei wird nicht nur die Nutzung, sondern auch der Aufbau von (Programm-) Schemata modelliert. Schließlich liefert der Ansatz der induktiven Programmsynthese eine theoretisch fundierte formale Basis für den Erwerb von Problemlösefertigkeiten.

Einleitung

Sowohl in der kognitiven Psychologie, als auch in der Künstlichen Intelligenz wird unter dem Erwerb von Problemlösefertigkeiten im Allgemeinen der Aufbau von Regeln verstanden, die es ermöglichen, nachfolgende Probleme ohne aufwendige Such- und Inferenzprozesse ("automatisch") zu lösen ([And83]). Üblicherweise wird davon ausgegangen, daß Problemlösefertigkeiten im Problemlöseprozeß - also durch learning by doing - erworben werden ([ACC89]). Anderson [And83]) bezeichnet die Automatismen, die Problemlösefertigkeiten zugrundeliegen, auch als prozedurales Wissen (*know how*), das er mit deklarativem Wissen (*know that*) kontrastiert. Prozedurales Wissen wird üblicherweise durch Produktionsregeln repräsentiert. Wissen über die Struktur von Problemen kann deklarativ, zum Beispiel in Form von Schemata (*frames*) repräsentiert werden.

Seit den 60er Jahren wird Problemlösen als heuristische Suche in einem Problemraum modelliert ([NS72]). Es werden sowohl Verfahren zur Vorwärtssuche (z. B. A*, [Nil80]) als auch zielgesteuerte Produktionssysteme (ACT-R, [And93]; Soar, [New91]) eingesetzt. In beiden Fällen ist die Angabe von Problemlöseoperatoren und deren Anwendungsbedingungen in Form von Produktionsregeln zentral. Ein Interpreter steuert die Auswahl und Anwendung von Produktionsregeln: Es wird zyklisch solange eine Produktionsregel auf Daten im Arbeitsspeicher angewendet, bis das Problemlöseziel erreicht oder keine Regel mehr anwendbar ist. Produktionsregeln entsprechen also bedingten Programmen ohne Schleifenkonstrukte. Lernen wird in diesem Kontext als Aufbau, insbesondere als Verknüpfung, solcher Produktionsregeln modelliert. Der Erwerb von Wissen über zyklische Anwendung von Operatoren, wie es etwa beim Turm von Hanoi ([BL80]) benötigt wird, kann in diesem Rahmen nicht behandelt werden. Abarbeitungszyklen sind fest als Interpreterstrategie implementiert.

Wird beim Problemlösen zusätzlich zu einem aktuellen Problem ein bereits gelöstes, strukturähnliches Problem herangezogen, so spricht man von analogem Problemlösen (sogenannte "*within domain analogies*", siehe [VO89]) oder fallbasiertem Schließen. Das aktuelle Problem wird hier nicht durch Suche nach geeigneten Problemlöseoperatoren bearbeitet, sondern durch die Übertragung der Lösungsstruktur des bereits gelösten auf das aktuelle Problem. Zentral beim analogen Problemlösen ist der Strukturvergleich (*mapping*) von Beispiel- und Zielproblem (siehe [Gen83], [HT89]). Wird nach erfolgreichem analogem Transfer über die gemeinsame Struktur von Beispiel- und Zielproblem generalisiert, so spricht man von analogem Lernen. Bei Modellen zum analogen Lernen wird jedoch meist bereits eine Grundstruktur als Problemschema vorgegeben ([AT89]). Es existieren nur wenige Arbeiten, die den Aufbau von Problemlöseschemata behandeln [RN81]).

Die Bereiche, in denen Problemlöseprozesse untersucht werden, reichen von Spielproblemen (Kryptoarithmetik, Turm von Hanoi, Blockweltprobleme; [NS72]), über mathematisches Problemlösen ([NH91])sowie das Erstellen von Computerprogrammen ([AT89], [Web94b]), bis hin zum Aufbau von komplexen Handlungsplänen ([Car83]).

Unser Ansatz basiert auf einer Methode zur induktiven Synthese rekursiver Programmschemata, die im Bereich Automatisches Programmieren ([BGK84], [BB93]) entwickelt wurde. Im engeren Sinn handelt es sich also um einen Beitrag zur Modellierung des Erwerbs von Programmierfertigkeiten. Wir werden jedoch aufzeigen, daß diese Methode prinzipiell zur Modellierung des Fertigkeitserwerbs in verschiedenen Problembereichen geeignet ist. Im Gegensatz zu den oben genannten Ansätzen konzentrieren wir uns nicht auf den Aufbau sequentieller sondern zyklischer (rekursiver) Handlungsprogramme, die als Programmschemata repräsentiert werden.

Diese Programmschemata werden ohne Vorgabe einer Grundstruktur aus konkreten Sequenzen von Problemlöseoperatoren inferiert. Die Schemata werden gespeichert und können zum analogem Problemlösen eingesetzt werden. Neben der Möglichkeit, learning by doing und analoges Schließen in einen einheitlichen Rahmen zu behandeln, liefert der Ansatz der induktiven Programmsynthese eine Basis zur formalen Behandlung des Fertigkeitserwerbs auf der Basis von Konzepten der theoretischen Informatik.

Im folgenden stellen wir zunächst unseren Ansatz im Überblick dar. Danach gehen wir auf die Modellierung des Erwerbs von Problemlösefertigkeiten durch induktive Programmsynthese und auf die Verwendung von rekursivem Programmschemata beim analogen Schließen ein.

Integration von learning by doing und learning by analogy

Um unseren Ansatz zu veranschaulichen, betrachten wir ein einfaches Blockweltproblem: Gegeben seien drei Blöcke A, B und C, die alphabetisch geordnet zu einem Turm gestapelt sind. Es steht ein Problemlöseoperator - $puttable(x)$ - zur Verfügung. Problemlöseziel sei es, den untersten Block (C) frei zu legen. Der Problemraum besteht entsprechend aus drei Zuständen (siehe Abbildung 1). Nehmen wir an, daß ein Problemlöser das Problem vollständig exploriert. Er erwirbt also beim Problemlösen Wissen darüber, mit welchen Operatorsequenzen er jeden der Zustände in den Zielzustand transformieren kann. Dieses Wissen kann in ein bedingtes Programm integriert werden, die Konstanten (*labels* der Blöcke) werden dabei variabilisiert. Dies entspricht dem *chunking* bzw. der *compilation* von Produktionsregeln sowie der Generalisierung bei produktionssystembasierten Ansätzen zum Fertigkeitserwerb ([RN86]; [And86]).

———————

Abb. 1 ungefähr hier

———————

Wird der Problemlöser erneut mit dem Turm-Abbau-Problem konfrontiert, so kann er die entsprechende Operatorsequenz unmittelbar (automatisch) anwenden, also die Suche nach einem Weg zum Ziel umgehen. Wir gehen davon aus, daß Erfahrung mit einem konkreten Problem nicht nur zum Erwerb von Fertigkeiten für dieses spezielle Problem führt, sondern, daß gleichzeitig eine generalisierte Wissensstruktur inferiert wird. Hat ein Problemlöser etwa die Fertigkeit erworben, Türme aus drei Blöcken abzubauen, so sollte es ihm nicht schwer fallen, diese Fertigkeit auch auf Türme aus vier oder mehr Blöcken anzuwenden. Das heißt, der Problemlöser generalisiert über rekursiv aufzählbare Problemräume. Diese Generalisierung entspricht der Inferenz eines rekursiven Programmschemas (RPS) aus einem Anfangsprogramm (siehe Abbildung 1).

Rekursive Programmschemata sind programmiersprachenunabhängig. Die Operationssymbole können durch verschiedene semantische Operationen interpretiert werden. Damit repräsentiert eine Programmschema eine Klasse strukturgleicher rekursiver Programme. Diese Eigenschaft kann für analogen Tranfer von Problemlösungen ausgenutzt werden (siehe Abbildung 1). Ist bereits ein Handlungsprogramm (RPS) mit einer bestimmten rekursiven Struktur erworben, so sollten neue, strukturisomporphe (oder allgemein strukturähnliche) Probleme durch analogen Transfer lösbar sein. Faßt man beide Probleme als Programmierprobleme auf, handelt es sich um *within domain* Analogien. Faßt man die RPSs jedoch allgemein als Handlungsprogramme zum Problemlösen auf, so kann es sich auch um *between domain* Analogien handeln ([VO89]). Dies ist etwa bei der Lösung eines mathematischen Problems (wie Fakultätsberechnung) durch analogen Transfer eines anschaulichen Problems (wie das Abbauen eines Bauklötzchenturmes) der Fall. Durch analogen Transfer kann der aufwendige Inferenzprozeß eines RPS umgangen werden. In empirischen Untersuchungen zeigt sich immer wieder, daß menschliche Problemlöser Analogieschlüsse anwenden ([Web94a]; [SK95]). Bei Anfängern ist es jedoch notwendig, diesen Prozeß durch Hinweise zu unterstützen, die die Strukturübertragung erleichtern (z.B. Oberflächenähnlichkeit, [Nov88]).

Unseren Ansatz haben wir in der Architektur des Systems IPAL realisiert, das als erster Prototyp in LISP implementiert ist[1] (siehe Abbildung 2). Eingabe in das System ist eine Menge von Problemzuständen, Problemlöseoperatoren und ein Zielzustand. Durch ein Problemlöseverfahren wird für jeden eingegebenen Problemzustand die (optimale) Operatorfolge zur Transformation in den Zielzustand ermittelt. Die Verknüpfung der Operatorfolgen zu einem Anfangsprogramm realisiert dann die erste Ebene der Generalisierung. Im nächsten Schritt wird geprüft, ob ein zum aktuellen Anfangsprogramm (*act*) hinreichend strukturähnliches Problem im Gedächtnis vorhanden ist. Im Gedächtnis sind RPSs abgelegt. Zur Durchführung des Ähnlichkeitsvergleichs werden diese zu Anfangsprogrammen expandiert. Liegt keine der vorhandenen Lösungen über einer Ähnlichkeitsschwelle S, so wird aus dem aktuellen Anfangsprogramm mit einer Methode zur Induktiven Programmsynthese ein RPS extrapoliert. Anderenfalls wird die Struktur des ähnlichsten im Speicher befindlichen RPS auf das aktuelle Problem übertragen. Das neue RPS wird in den Speicher eingefügt.

———————

Abb. 2 ungefähr hier

———————

Aufbau von initialen Handlungsprogrammen beim Problemlösen

Den Aufbau von Anfangsprogrammen aus Problemlöseerfahrungen stellen wir beispielhaft am Turm-Abbau-Problem aus Abbildung 1 dar. Die drei Problemzustände liegen als prädikatenlogische Beschreibung vor:

(1) $cleartop(A), cleartop(B), cleartop(C)$

(2) $on(B, C), cleartop(A), cleartop(B)$

(3) $on(A, B), on(B, C), cleartop(A)$.

Im Zielzustand soll gelten $cleartop(C)$.

Ferner ist der Operator $puttable(x)$ zusammen mit seinen Anwendungsbedingungen als Produktionsregel gegeben:

$on(x, y), cleartop(x) \rightarrow puttable(x)$.

Die Auswirkung der Anwendung des Operators $puttable(x)$ wird beschrieben durch

ADD $cleartop(y)$ DEL $on(x, y)$.

Zudem führen wir die Selektor-Funktion $topof(x) = y \Leftrightarrow on(y, x)$ ein.

Nun wird versucht, für jeden vorgegebenen Zustand eine Problemlösung zu erzeugen. Zustand (1) erfüllt das Zielprädikat $cleartop(C)$ bereits, es ist also keine Transformation erforderlich. Für Zustand (2) sind die Anwendungsbesdingungen der Produktionsregel durch $on(B, C), cleartop(B)$ erfüllt. In der Produktionsregel wird also die Variable x durch die Konstante B und y durch C substituiert. Die Anwendung von $puttable(B)$ erzeugt Zustand (1), das Problemlöseziel ist erreicht. In Zustand (3) wird zunächst $puttable(A)$, dann $puttable(B)$ ausgeführt.

Zur Erzeugung des Anfangsprogramms führen wir nun folgende Schritte durch:

- Alle Konstanten werden konstruktiv, in Abhängigkeit vom Zielblock C formuliert. In Zustand (2) gilt $B = topof(C)$. In Zustand (3) gilt zusätzlich $A = topof(topof(C))$.

- Um Operatorsequenzen als einem funktionalen Term zu formulieren, verwenden wir das Situationskalkül ([Gre69]). Alle Prädikate und Operationssymbole werden um ein Argument - die Situationsvariable s - erweitert. Wir notieren im folgenden nur die Operation $puttable(x, s)$ unter Berücksichtigung der Situationsvariable, um verschachtelte funktionale Terme zu erzeugen. Bei allen anderen Ausdrücken kann s implizit dazu gedacht werden.

- Die Konstanten werden variabilisiert. Dabei wird über den vorgegebenen konkreten Problemraum (Abbildung 1) generalisiert. Es werden beispielsweise alle Situationen behandelt, in denen ein fester Block x frei ist, also auch solche, wo x nicht auf dem Tisch, sondern auf anderen Blöcken liegt.

Die so umformulierten Zustände und Operatoren sind in Tabelle 1 aufgeführt. Die Konstruktion des Anfangsprogramms wird folgendermaßen fortgestzt:

- Es wird eine minimale Konjunktion von Prädikaten identifiziert, die ausreicht, um die Zustände zu diskriminieren. Dabei handelt es sich um ein typisches Begriffsbildungsproblem, das etwa mit einem Entscheidungsbaumverfahren (z. B. CAL2, [UW81]) bearbeitet werden kann. In unserem Beispiel genügt das Prädikat $cleartop(x)$.

- Die Zustände werden, geordnet nach der Komplexität des funktionalen Ausdrucks, zu einer bedingten Anweisung zusammengefaßt.

———————

Tab. 1 ungefähr hier

———————

Die durch obige fünf Schritte konstruierte bedingte Anweisung entspricht dem in Abbildung 1 dargestellten Anfangsprogramm.

Alternativ zu diesem Vorgehen, verwenden wir einen Ansatz zur generalisierten Planung, den Wysotzki ([Wys87]) entwickelt hat. Anstelle der Erzeugung von Operatorsequenzen durch Vorwärtssuche wird ausgehend vom Zielzustand ein Planbaum aufgebaut. Die linke Verzweigung behandelt jeweils den Fall, in dem das aktuelle Zielprädikat erfüllt ist, die rechte Verzweigung den Fall, in dem das Zielprädikat nicht erfüllt ist. Anstelle der Produktionsregeln verwenden wir sogenannte Basisprogramme. Für das Turm-Abbau-Problem gilt:

$$G_{cleartop(x)} = cleartop(x)(s|\,|, puttable(topof(x))(s|cleartop(topof(x))|)).$$

Das Basisprogramm erhält den Namen des behandelten Prädikats als Index. Auf der rechten Seite wird geprüft, ob das Prädikat erfüllt ist. Der erste Term in der Klammer gibt an, welcher Operator ausgeführt werden soll, wenn das Prädikat erfüllt ist (in unserem Beispiel wird hier die Situation unverändert beibehalten). Der zweite Term gibt den Operator an, der ausgeführt werden soll, um das Prädikat zu erfüllen. In Klammern werden dann Anwendungsbedingungen (hier $cleartop(topof(x))$) für diesen Operator angegeben. Diese Anwendungsbedingungen werden im Plan als neue, zu erfüllende Teilziele aufgenommen.

Der Plan für das Turm-Abbau-Problem ist Abbildung 3 zu entnehmen. Der Plan ist zu lesen als "Wenn $cleartop(x)$ dann s sonst $puttable(topof(x))$ in einer Situation, in der gilt $cleartop(topof(x))$, ...".

———————

Abb. 3 ungefähr hier

———————

Wir werden im nächsten Abschnitt zeigen, daß diese Repräsentation für bestimmte Probleme besser geeignet ist, um die Verallgemeinerung zu einem RPS zu inferieren. Zudem ist diese Methode der Planung besonders gut dafür geeignet, Zielzustände, die aus mehreren (eventuell abhängigen) Teilzielen bestehen, zu behandeln. Eine ausführliche Darstellung des Algorithmus findet sich in [Wys87]. Eine Weiterentwicklung des Ansatzes ist in [SW96] zu finden.

Erwerb von Problemlösefertigkeiten als induktive Programmsynthese

Unser Ansatz zur Induktion rekursiver Programmschemata basiert auf einer von Wysotzki ([Wys83]) vorgeschlagenen Methode, die eine Verallgemeinerung des Syntheseansatzes von [Sum77] darstellt. Wir stellen die Methode an einem Beispiel dar. Der theoretische Hintergrund ist ausführlich in [Wys83]) beschrieben.

Wir verfolgen zunächst nicht das Turm-Abbau-Problem weiter, sondern betrachten ein "einfacheres" Problem: das Ausgeben des letzten Elements einer Liste, die aus einem bis zu drei Elementen bestehen kann. Auf das Turm-Abbau-Problem kommen wir im Anschluß daran wieder zurück. Das Anfangsprogramm für das Listen-Problem lautet:

```
IF empty(tail(l))
THEN head(l)
ELSE IF empty(tail(tail(l)))
     THEN head(tail(l))
```

```
ELSE IF empty(tail(tail(tail(l))))
     THEN head(tail(tail(l)))
     ELSE undefined.
```

Im folgenden verwenden wir anstelle des If-then-else Konstruktes das McCarthy-Konditional (als nicht strikte Funktion höherer Ordnung): if x then y else $z =_{Def} g(x,\, y,\, z)$. Damit haben wir eine einheitliche Präfixnotation für alle Funktionen (bzw. Funktionale).

Das Anfangsprogramm repräsentiert die Erfahrung, die ein Problemlöser mit Listen bis zur Länge drei gemacht hat. Er hat jeweils nur direkten Zugriff auf das erste Listenelement ($head(l)$) und er kann das erste Element aus der Liste entfernen ($tail(l)$). Nach der Lösung dieses Problems sollte der Problemlöser die Fertigkeit erworben haben, bei Listen beliebiger Länge das letzte Element auszugeben. Die Regel, die er zu diesem Zweck erworben haben sollte, lautet "Solange bis die Liste nur noch ein Element erhält, verkürze die Liste. Wenn die Liste genau ein Element enthält, dann gib dieses aus."

Diese Regel soll nun als RPS aus dem Anfangsprogramm inferiert werden. Die Grundidee des Ansatzes von [Wys83] ist es, im Anfangsprogramm einen Term tr zu identifizieren, der sich rekursiv mit einer eventuellen Substitution Θ der Variablen v durch einen Term t (in jedem Schritt!) wiederholt. Zu diesem Zweck wird das Anfangsprogramm in eine Sequenz zunehmend komplexerer Teilterme zerlegt, die als Anfangsstück einer Kleene-Folge interpretiert werden können:

$G^{(0)} = \Omega$

$G^{(1)} = g(empty(tail(l)),\; head(l),\; \Omega)$

$G^{(2)} = g(empty(tail(l)),\; head(l),\; g(empty(tail(tail(l))),\; head(tail(l)),\; \Omega))$

$$G^{(3)} = g(empty(tail(l)),\; head(l),\; g(empty(tail(tail(l))),\; head(tail(l)),\\ g(empty(tail(tail(tail(l)))),\; head(tail(tail(l))),\; \Omega))).$$

Das Symbol Ω steht für "*undefined*". Der Term tr, für den obige Bedingung gilt, lautet hier

$$tr = g(empty(tail(l)), head(l), m) \text{ mit } \Theta = \{l/tail(l)\}.$$

Das Symbol m gibt die Stelle im Term an, an der ein bestimmter Teilterm beginnt. Für $G^{(2)}$ gilt beispielsweise

$$G^{(2)} = g(empty(tail(l)),\ head(l),\ G^{(1)}_{[l/tail(l)]}).$$

Ist ein fester Term tr identifiziert, so kann das Anfangsprogramm zu einem RPS gefaltet werden:

$$last(l) = g(empty(tail(l)),\ head(l),\ last(tail(l))).$$

Die Methode der induktiven Programmsynthese liefert uns also einen formalen Ansatz, um in einer gegebenen Struktur Regularitäten zu identifizieren und über diese zu generalisieren.

Für das eingangs dargestellte Turm-Abbau-Problem ist die Inferenz eines RPS etwas komplizierter. Im Gegensatz zu der endrekursiven Struktur von "*last*", handelt es sich hier um ein teil-rest-rekursives Problem ([VG91]). Verwendet man zur Generierung des Anfangsprogramm ein Vorwärtssuchverfahren, so ergibt sich das in Abbildung 1 dargestellte Programm, aus dem die teil-rest-rekursive Struktur nicht unmittelbar inferierbar ist. Das Problem liegt in der Tatsache, daß für den Teilterm $tr = g(cleartop(x), s, m)$ keine einheitliche Substitution für x und s gefunden werden kann.

Verwendet man zum Problemlösen einen Ansatz zur generalisierten Planung ([Wys87], [SW96]), so erhält man als Anfangsprogramm den in Abbildung 3 dargestellten Plan:

```
IF cleartop(x)
THEN s
ELSE puttable(topof(x),
```

```
IF cleartop(topof(x))
THEN s
ELSE puttable(topof(topof(x)),
         IF cleartop(topof(topof(x)))
         THEN s
         ELSE undefined)).
```

Hieraus läßt sich analog zum obigen Beispiel das RPS aus Abbildung 1 extrapolieren.

Unterschiedliche Problemlösestrategien bedingen also den Aufbau unterschiedlicher Repräsentationen der Problemlösung. Die Repräsentation einer Problemlösung ist maßgeblich für den Lernerfolg, nämlich für die Generalisierung über rekursiv aufzählbare Problemräume. Es ist ein bekannter Befund der Problemlöseforschung, daß die Art der gewählten Problemrepräsentation den Problemlöseerfolg maßgeblich beeinflußt ([Sim70], [JK76]). Unserer Meinung nach wäre es lohnend, auch den durch die aufgebaute Repräsentation bedingten Einfluß von Problemlösestrategien auf die Generalisierungsleistung in empirischen Untersuchungen und entsprechenden formalen Analysen genauer zu explorieren.

Programmieren durch analoges Schließen

Analoges Problemlösen kann durch folgende vier Teilprozesse beschrieben werden (siehe [NH91]):

- *Retrieval*: Zu einem aktuellen Problem (Zielproblem) wird ein möglichst strukturähnliches Beispielproblem dessen Lösung bereits bekannt ist gesucht.
- *Mapping*: Struktur von Beispiel- und Zielproblem werden verglichen (erfolgt das Retrieval bereits mit einem Maß der strukturellen Ähnlichkeit, so wird das Mapping bereits beim Retrieval durchgeführt).

- *Adaptation*: Aufgrund der identifizierten strukturellen Ähnlichkeiten wird die Beispiellösung an das Zielproblem adaptiert. Wir werden zeigen, daß die Adaptation nur bei isomorphen (strukturidentischen) Problemen rein syntaktisch durchgeführt werden kann. Für nicht isomorphe Probleme muß unter Umständen semantische Information aus den entsprechenden Problembereichen hinzugezogen werden.

- *Learning*: Es wird ein Schema aufgebaut, das über die strukturellen Gemeinsamkeiten von Beispiel- und Zielproblem generalisiert.

In unserem Ansatz realisieren wir diese Prozesse wie folgt: Gegeben ist ein Anfangsprogramm als Zielproblem. Gesucht wird die Verallgemeinerung des Anfangsprogramms zu einem RPS. Im Gedächtnis sind bereits gelernte RPS in einer (zumindest partiell) hierarchischen Struktur abgelegt. Wie diese hierarchische Struktur aufgebaut wird, wird weiter unten beschrieben. Für den Ähnlichkeitsvergleich des Zielproblems mit den bereits vorhandenen RPSs, müssen die RPSs wieder zu Anfangsprogrammen (Beispielprobleme) expandiert werden. Dieses "*unfolding*" ([BD77]) wird so lange durchgeführt, bis die Tiefe des aktuellen Anfangsprogramms erreicht ist. Die Ähnlichkeit zwischen dem Zielproblem und den Beispielproblemen wird mit einem Baumdistanzmaß ([Lu79]) bestimmt. Dieses rein syntaktische Maß des Strukturvergleichs ist genau auf die Struktur von Anfangsprogrammen zugeschnitten: Programme entsprechen Termen und Terme können als Bäume dargestellt werden ([HU90], Kap. 1)[2]. Das Ähnlichkeitsmaß ergibt sich als Summe der Operationen, die notwendig sind, um einen Baum in einen anderen zu transformieren. Es existieren drei Operationstypen, die unterschiedlich stark gewichtet werden können: Abbildung eines Knotenlabels auf ein anderes (s), Löschen eines Knotens (d) und Einfügen eines Knotens (i). Die Ähnlichkeit zwischen zwei Bäumen x und y errechnet sich als

$$ds(x, y) = s * n_1 + d * n_2 + i * n_3,$$

wobei n_i die Häufigkeit angibt, mit der eine Operation ausgeführt wurde.

Werden nur eineindeutige Abbildungen durchgeführt, so handelt es sich um isomorphe Probleme. Dies ist zum Beispiel bei den Problemen Fakultät und Summe (siehe Abbildung 4 (a) und (b)) der Fall. Die durchzuführenden Ersetzungen (von Funktionssymbolen) lauten $[1/0, mult/plus]$. Man beachte, daß die Ersetzung zwar rein syntaktisch (auf Symbolen) durchgeführt wird, daß aber die Funktionssymbole über eine (zumindest denotationale) semantische Interpretation verfügen müssen.

Ist die Ähnlichkeit zwischen zwei Anfangsprogrammen ermittelt, ist damit auch gleichzeitig der Strukturvergleich abgeschlossen. Die Adaptation ist für isomorphe Probleme einfach: die beim *mapping* ermittelten Substitutionen werden auf das RPS dessen Anfangsbaum untersucht wurde angewendet. Das RPS für summe kann aus dem RPS für Fakultät

faku(x) = g(equal0(x), 1, mult(x, faku(pred(x))))

erzeugt werden, indem 1 durch 0 und mult durch plus ersetzt wird.

Das so durch analoges Schließen generierte RPS wird ins Gedächtnis aufgenommen. Das Beispiel- und Ziel-RPS werden generalisiert, indem alle ungleichen Symbole durch Variablen ersetzt werden. Das generalisierte RPS bildet einen Elternknoten für die beiden konreten RPS. Auf diese Art wird schrittweise eine hierarchische Struktur aufgebaut (siehe Abbildung 1). Im Gedächtnis entsteht also ein Baum (bzw. ein Wald) von partiell geordneten RPSs. Beim *Retrieval* wird der Strukturvergleich von den Wurzeln dieser Bäume ausgehend durchgeführt. Es muß dann nur noch der Baum durchsucht werden, dessen Wurzel die höchste Ähnlichkeit zum aktuellen Problem aufweist. In einem Baum muß entsprechend auch nur ein Pfad bis zu den Blättern verfolgt werden. Dadurch ergibt sich, daß das Retrieval mit zunehmender Anzahl von Elementen im Gedächtnis nicht ineffizienter wird.

Abschließend wollen wir als Beispiel betrachten, wie das RPS für die Berechnung der Fakultät (Abbildung 4a) durch Analogieschluß aus dem RPS für das Turm-Abbau-Problem (Abbildung 4c) erzeugt werden kann. Hier handelt es sich um einen Analogieschluß von einem anschaulichen Problembereich (Turm-Abbau) auf ein mathematisches Problem (Fakultätsberechnung), also um eine *between domain* Analogie. Die notwendigen Transformationen sind in Tabelle 2 dargestellt. Zusätzlich zu Ersetzungen (s) von Symbolen müssen zwei Löschungen (d) von $topof$ Knoten durchgeführt werden. Die Probleme sind nicht isomorph. Die Transformation des Turm-Problems in das Fakultätsproblem entspricht nicht einmal den Anforderungen an eine Abbildung (Rechtseindeutigkeit). Entsprechend ist die Adaptation hier nicht trivial zu definieren. Es müßte identifiziert werden, an welcher Stelle im RPS "*clearblock*" das Symbol $topof$ durch $pred$ zu ersetzen ist und an welcher Stelle es gelöscht werden muß.

———————

Tab. 2 ungefähr hier

———————

Die beiden nicht isomorphen RPSs lauten:

$$
\begin{array}{llll}
\mathrm{G(x, s)} & = \mathrm{g(cleartop(x),} & \mathrm{s,\ puttable(topof(x),} & \mathrm{G(topof(x), s)))} \\
\mathrm{F(x)} & = \mathrm{g(equal0(x),} & \mathrm{1,\ mult(x,} & \mathrm{F(pred(x)))).}
\end{array}
$$

Wir explorieren zur Zeit einen anderen Weg: Es wird versucht, durch Hinzunahme von Hintergrundwissen in Form von Termersetzungsregeln, das Beispielproblem so zu transformieren, das es zum Zielproblem isomorph wird. Verwenden wir die Regel

$$cleartop(x) \rightarrow notexist(topof(x))$$

so erreichen wir zumindest eine partielle Isomorphie zum Fakultätsproblem (siehe Abbildung 4a und 4d). Wird beim Fakultätsproblem der Sonderfall $x = 0$ nicht berücksichtigt (und damit nur die Fakultät von $x - 1$ berechnet) und läßt man eine Konstante als Funktionsparameter zu, so sind die Strukturen identisch:

G(x, s)	= g(notexist(topof(x)),	s,	puttable(topof(x),	G(topof(x), s)))
F(x, 1)	= g(equal0(pred(x)),	1,	mult(pred(x),	F(pred(x), 1))).

Diese (unbefriedingende Lösung) kann durch eine Umformung des RPS für das Fakultätsproblem repariert werden:

F(x)	= F'(x, x)			
F'(x, y)	= g(equal0(pred(x)),	y,	mult(pred(x),	F'(pred(x), y))).

Durch das RPS $F'(x, y)$ wird bei Aufruf mit $F'(x, x)$ die Zahl x mit der Fakultät der Zahl $x - 1$ multipliziert, also die gewünschte Fakultätsfunktion berechnet. Damit ist der gesuchte Isomorphismus zwischen dem Turm-Abbau-Problem und der Fakultätsberechnung konstruiert. Diese Modifikation ist jedoch nur durch Hinzunahme von semantischer Information über den Problembereich möglich. Hierzu stehen weiterführende Untersuchungen an.

Diskussion

Der dargestellte Ansatz zur einheitlichen Behandlung von *learning by doing* und *learning by analogy* auf der Grundlage einer Methode zur induktiven Programsynthese scheint uns sowohl für die kognitive Psychologie als auch für die Künstliche Intelligenzforschung fruchtbar zu sein. Für die Psychologie liefert der Ansatz einen durch Mittel der theoretischen Informatik fundierten formalen Hintergrund zur Beschreibung grundlegender Prozesse des Fertigkeitserwerbs. Die Behandlung von Problemlösen und dem Erwerb von Problemlösefertigkeiten mit einem Ansatz der Autmatischen Programmierung ist eine Alternative zu den bestehenden, meist als Produktionssysteme realisierten Ansätzen.

Betrachtet man speziell die Generierung von Programmen aus Problemspezifikationen, so liefert unsere Arbeit einen Beitrag zum Bereich der wissensbasierten Softwareentwicklung ([LD89]). Die Kombination von Autmatischem Programmieren und Maschinellem Lernen kann zu einer Performanzsteigerung solcher Systeme beitragen. Durch die Wiederverwendung bereits inferierter Module und deren Anpassung an neue Problemstellungen kann der aufwendige Prozeß der Programmkonstruktion abgekürzt werden. Ein weiterer Anwendungsbereich sind intelligente Tutor-Systeme für Programmieranfänger. Diesen können bereits bearbeitete Probleme als Lösungshilfe angeboten werden ([Web94b]).

Für die drei Generalisierungsarten, die unser Ansatz integriert, existieren bisher nur prototypisch implementierte Algorithmen. Auf allen drei Ebenen sind Weiterentwicklungen und theoretische Arbeiten notwendig. Offene Fragen, bei denen es sinnvoll wäre, auch psychologische Untersuchungen heranzuziehen, sind etwa der Einfluß der gewählten Problemlösestrategie auf die Generalisierung, wie es in Abschnitt 3 angesprochen wurde. Im Bereich analoges Schließen wäre es für uns von Interesse, zwei Adaptationsstrategien zu kontrastieren: Den - aufwendigen und unsicheren - Transfer bei nicht isomorphen Problemen versus die Umformung des Beispielproblems in eine zum Zielproblem isomorphe Struktur (falls diese existiert) auf der Grundlage von Hintergrundwissen.

Literatur

[ACC89] J. R. Anderson, F. G. Conrad, and A. T. Corbett. Skill acquisition and the LISP tutor. *Cognitive Science*, 13:467–505, 1989.

[And83] J. R. Anderson. *The Architecture of Cognition.* Havard University Press, Cambridge, MA, 1983.

[And86] J. R. Anderson. Knowledge compilation: A general learning mechanism. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning - An Artificial Intelligence Approach*, volume 2, pages 289–310. Tioga, 1986.

[And93] J.R. Anderson. *Rules of the Mind.* Lawrence Erlbaum, Hillsdale, NJ, 1993.

[AT89] J.R. Anderson and R. Thompson. Use of analogy in a production system architecture. In S. Vosniadou and A. Ortony, editors, *Similarity and Analogical Reasoning*, pages 267–297. Cambridge University Press, 1989.

[BB93] W. Bibel and A. W. Biermann. Special issue: Automatic programming, foreword of the guest editors. *Journal of Symbolic Computation*, 15(5, 6):463–465, 1993.

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24(1):44–67, 1977.

[BGK84] A. W. Biermann, G. Guiho, and Y. Kodratoff, editors. *Automatic Program Construction Techniques.* Collier Macmillan, 1984.

[BL80] P. Buneman and L. Levy. The towers of hanoi problem. *Information Processing Letters*, 10(4, 5):243–244, July 1980.

[Car83] J. G. Carbonell. Learning by analogy: Formulating and generalizing plans from past experience. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning - An Artificial Intelligence Approach*, volume 1, pages 137–162. Springer, New York, 1983.

[Gen83] D. Gentner. Structure-mapping: a theoretical framework for analogy. *Cognitive Science*, 7:155–170, 1983.

[Gre69] C. Green. Application of theorem proving to problem solving. Technical report, IJCAI 1, 1969.

[HT89] K. J. Holyoak and P. Thagard. Analogical mapping by constraint satisfaction. *Cognitive Science*, 13:295–355, 1989.

[HU90] J.E. Hopcroft and J.D. Ullman. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie.* Addison-Wesley, Bonn, 1990.

[JK76] B. Jülisch and W. Krause. Semantischer kontext und problemlöseprozesse. In F. Klix, editor, *Psychologische Beiträge zur Analyse kognitiver Prozesse*, pages 274–301. Deutscher Verlag der Wissenschaften, Berlin, 1976.

[LD89] M. Lowry and R. Duran. Knowledge-based software engineering. In P. R. Cohen A. Barr and E. A. Feigenbaum, editors, *Handbook of Artificial Intelligence*, volume IV, pages 241–322. Addison-Wesely, Reading, MA, 1989.

[Lu79] S. Lu. A tree-to-tree distance and its application to cluster analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):219–224, April 1979.

[New91] A. Newell. *Unifyied Theories of Cognition*. Cambridge University Press, Cambridge, MA, 1991.

[NH91] L. R. Novick and K. J. Holyoak. Mathematical problem solving by analogy. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 14:510–520, 1991.

[Nil80] N. J. Nilsson. *Principles of Artificial Intelligence*. Springer, 1980.

[Nov88] L. R. Novick. Analogical transfer, problem similarity, and expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 14(510-520), 1988.

[NS72] A. Newell and H. A. Simon. *Human Problem Solving*. Prentice Hall, Englewood Cliffs, NJ, 1972.

[RN81] D. E. Rumelhart and D. A. Norman. Analogical processes in learning. In J. R. Anderson, editor, *Cognitive Skills and Their Acquisition*, pages 335–360. Lawrence Erlbaum, Hillsdale, NJ, 1981.

[RN86] P. S. Rosenbloom and A. Newell. The chunking of goal hierarchies: A generalized model of practice. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning - An Artificial Intelligence Approach*, volume 2, pages 247–288. Morgan Kaufmann, 1986.

[Sim70] R. J. Simon. Encoding effects on complex problem solving. *Journal of Experimental Psychology*, 83:227–231, 1970.

[SK95] U. Schmid and B. Kaup. Analoges Lernen beim rekursiven Programmieren (Analogical learning in recursive programming). *Kognitionswissenschaft*, 5:31–41, 1995.

[Sum77] P. D. Summers. A methodology for LISP program construction from examples. *Journal ACM*, 24(1):162–175, 1977.

[SW96] U. Schmid and F. Wysotzki. Fertigkeitserwerb durch induktive Programmsynthese und generalisiertes Planen. In W. Dilger, M. Schlosser, J. Zeidel, and A. Ittner, editors, *Proceedings of FGML-96 (19.-21.8.96, TU Chemnitz)*, pages 106–111, 1996.

[UW81] S. Unger and F. Wysotzki. *Lernfähige Klassifizierungssysteme.* Akademie-Verlag, Berlin, 1981.

[VG91] D. Vorberg and R. Göbel. Das lösen rekursiver Programmierprobleme: Rekursionsschemata. *Kognitionswissenschaft*, 1:83–95, 1991.

[VO89] S. Vosniadou and A. Ortony. Similarity and analogical reasoning: A synthesis. In S. Vosniadou and A. Ortony, editors, *Similarity and Analogical Reasoning*, pages 1–17. Cambridge University Press, Cambridge, 1989.

[Web94a] G. Weber. Examples and remindings in a case-based help system. In *Proc. of the 2nd European Workshop on Case-Based Reasoning (Chantilly, France, Nov. 94)*, 1994.

[Web94b] G. Weber. *Fallbasiertes Lernen und Analogien - Unterstützung von Problemlöse- und Lernprozessen in einem adaptiven Lernsystem.* PVU, Weinheim, 1994.

[Wys83] F. Wysotzki. Representation and induction of infinite concepts and recursive action sequences. In *Proceedings of the 8th IJCAI, Karlsruhe*, 1983.

[Wys87] F. Wysotzki. Program synthesis by hierarchical planning. In P. Jorrand and V. Sgurev, editors, *Artificial Intelligence: Methodology, Systems, Applications*, pages 3–11. Elsevier Science, Amsterdam, 1987.

Fußnoten

[1]Die Implementationsarbeiten wurden insbesondere in zwei studentischen Projekten realisiert und von Mark Müller zu einem Gesamtsystem integriert.

[2]Wir arbeiten derzeit an einer Erweiterung dieses Maßes, bei der Information über Stelligkeit und Typen von Operatoren bzw. Funktionen beim Strukturvergleich berücksichtigt wird.

Tabelle 1: Problemzustände und zugehörige Operatorsequenzen für das Turm-Abbau-Problem

(1)	cleartop(x)	s
(2)	on(topof(x), x), cleartop(topof(x))	puttable(topof(x),s)
(3)	on(topof(topof(x)), topof(x)), on(topof(x), x), cleartop(topof(topof(x)))	puttable(topof(x), puttable(topof(topof(x)), s))

Tabelle 2: Substitutionen zur Transformation des Turm-Abbau-Problems in das Fakultät-Problem (Die Abbildung von $topof$ auf die Identitätsfunktion $id(x) = x$ entspricht der Löschung der Operation $topof$)

Turm	Fakultät
x	x
s	1
cleartop	equal0
puttable	mult
topof	pred
topof	id

$\rightarrow$ nicht rechtseindeutig

$\leftarrow$ surjektiv

Abbildungsüberschriften

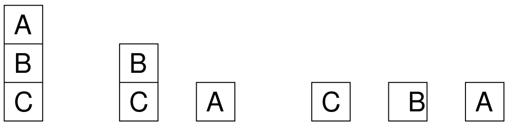
Abbildung 1: Drei Ebenen der Generalisierung beim Erwerb von Problemlösefertigkeiten

Abbildung 2: Das System IPAL

Abbildung 3: Plan für das Turm-Abbau-Problem (* heißt undefiniert)

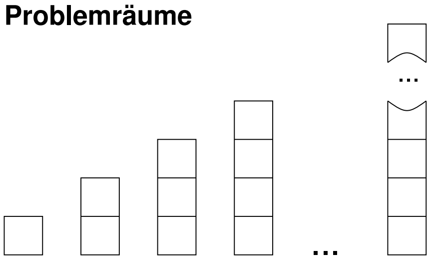
Abbildung 4: Termbäume für Fakultät (a), Summe (b), Turm-Abbau (c) und transformiertes Turm-Abbau-Problem (d)

Generalisierung über Problemzustände → **Anfangsprogramm**




```
IF cleartop(x)
THEN s
ELSE IF cleartop(topof(x))
     THEN puttable(topof(x),s)
     ELSE IF cleartop(topof(topof(x)))
          THEN puttable(topof(x),
               puttable(topof(topof(x)), s)
          ELSE undefined
```

Generalisierung über rekursiv aufzählbare Problemräume → **Rekursives Programmschema**



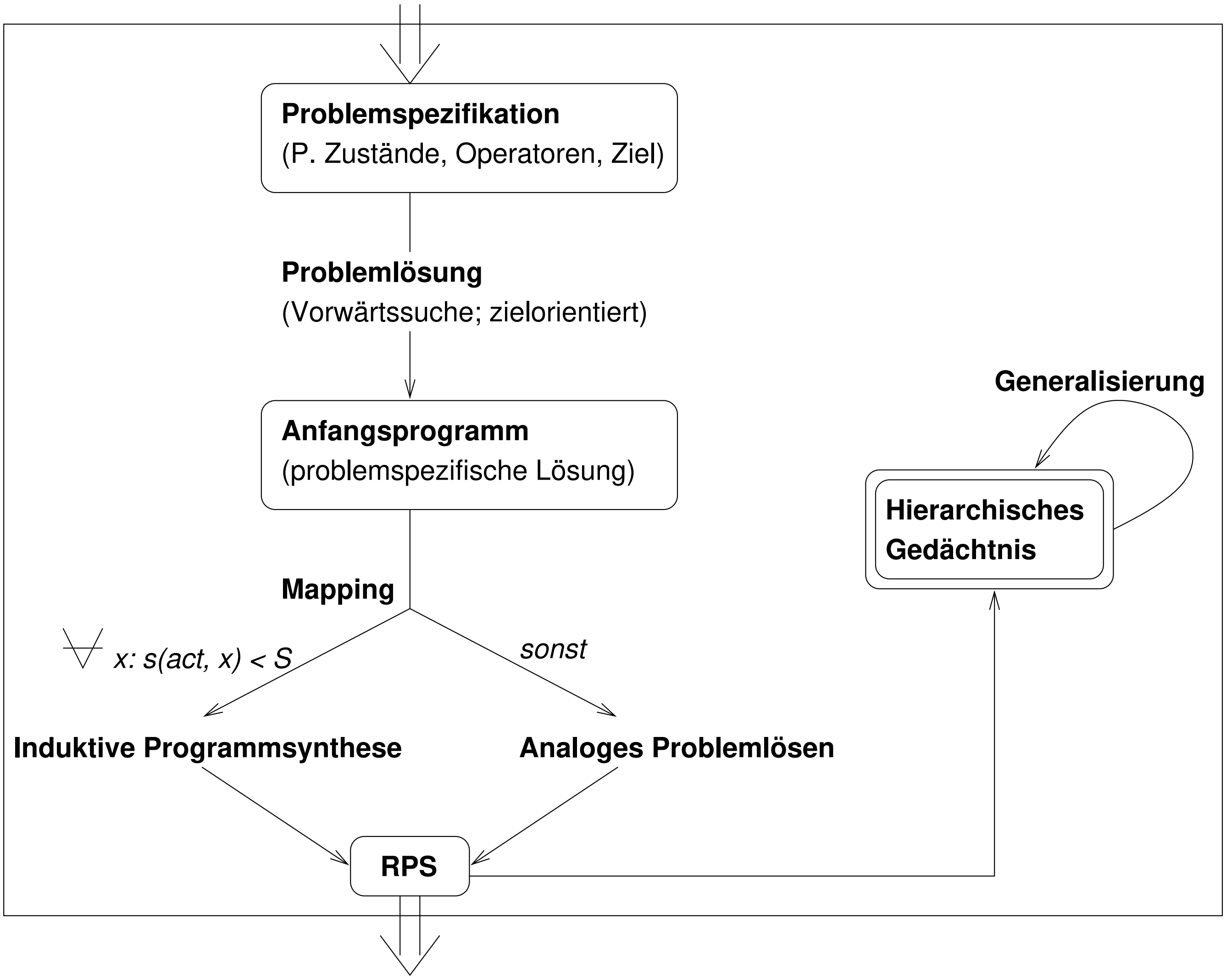
```
clearblock(x, s) =
IF cleartop(x)
THEN s
ELSE puttable(topof(x), clearblock(topof(x),s
```

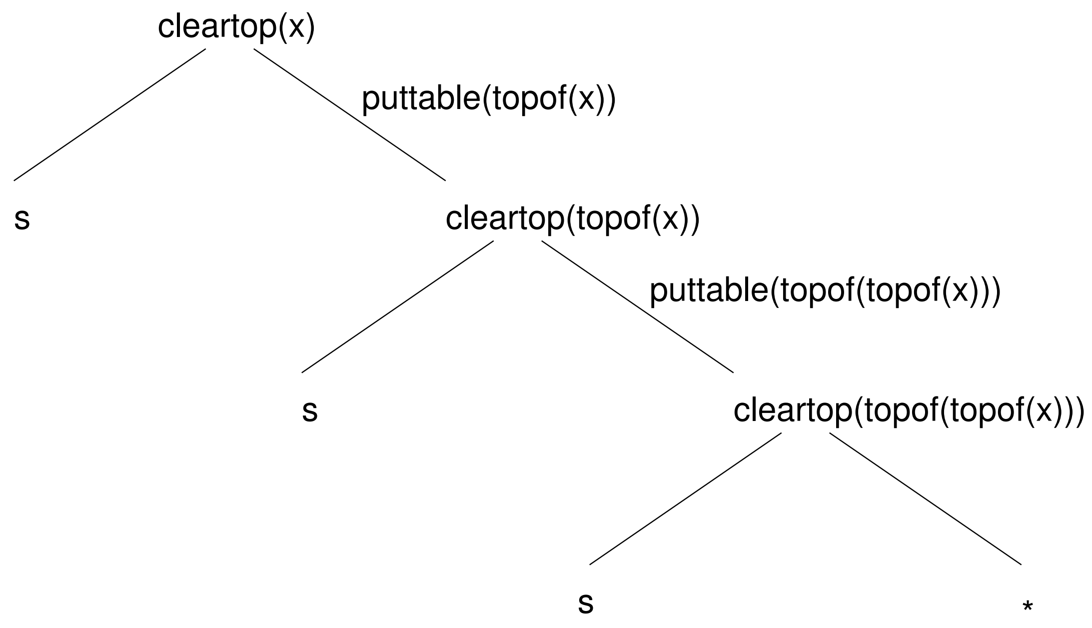
Generalisierung über Problemklassen → **Schemahierarchie**

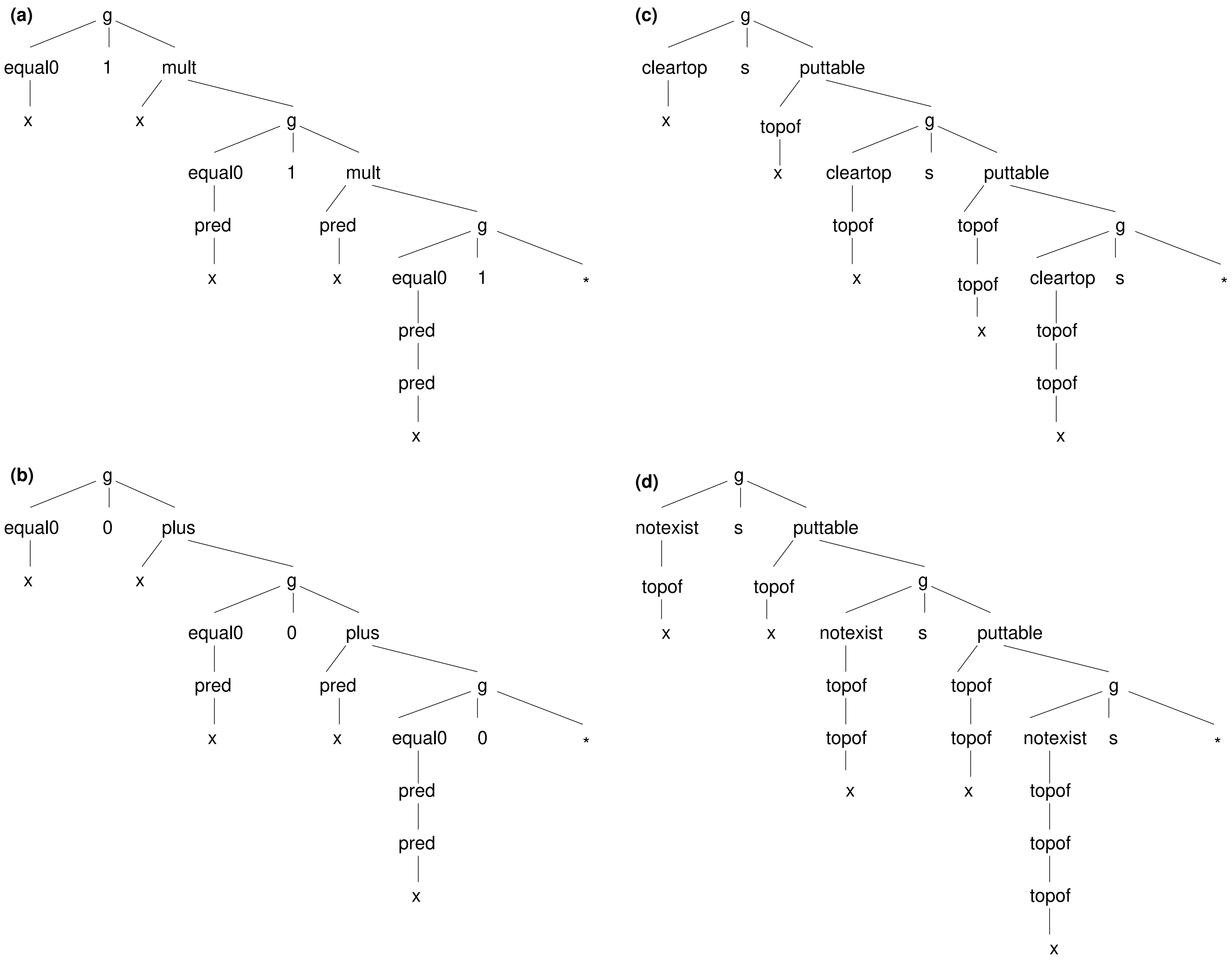
```
r(p) =
IF bo(p)
THEN c1
ELSE op1(op2(p), r(op3(p), c2))
```

```
clearblock(x, s) =
IF cleartop(x)
THEN s
ELSE puttable(topof(x), clearblock(topof(x),s))
```

```
faku(x) =
IF equal0(x)
THEN 1
ELSE mult(x, faku(pred(x)))
```


Problemspezifikation
(P. Zustände, Operatoren, Ziel)
Problemlösung
(Vorwärtssuche; zielorientiert)
Anfangsprogramm
(problemspezifische Lösung)
Mapping
∀ x: s(act, x) < S
sonst
Induktive Programmsynthese
Analoges Problemlösen
RPS
Generalisierung
Hierarchisches
Gedächtnis


cleartop(x)
puttable(topof(x))
s
cleartop(topof(x))
puttable(topof(topof(x)))
s
cleartop(topof(topof(x)))
s
*


(a)
g
equal0 1 mult
x x g
equal0 1 mult
pred pred g
x x equal0 1 *
pred
pred
x
(b)
g
equal0 0 plus
x x g
equal0 0 plus
pred pred g
x x equal0 0 *
pred
pred
x
(c)
g
cleartop s puttable
x topof g
x cleartop s puttable
topof topof g
x topof cleartop s *
x topof
topof
x
(d)
g
notexist s puttable
topof topof g
x x notexist s puttable
topof topof g
topof topof notexist s *
x x topof
topof
topof
x