Programmieren durch analoges Schließen

Ute Schmid*

Institut für Angewandte Informatik

Fachgebiet "Methoden der Künstlichen Intelligenz"

Technische Universität Berlin

Franklinstr. 28-29, D-10587 Berlin

email: schmid@cs.tu-berlin.de

Zusammenfassung

Es wird ein Ansatz zum Programmieren durch analoges Schließen dargestellt, der auf dem Konzept rekursiver Programmschemata basiert. Ist kein hinreichend ähnliches Beispiel zu einem aktuellen Problem vorhanden, so werden solche Schemata induktiv aus Ein-Ausgabe-Beispielen aufgebaut. Das Gedächtnis ist hierarchisch organisiert, indem ausgehend von konkreten Programmschemata generalisierte Strukturen gebildet und gespeichert wer-

*Ich bedanke mich bei Studierenden der Informatik der Technischen Universität Berlin, die in zwei studentischen Projekten große Teile der Implementation durchgeführt haben, insbesondere bei Eric Heymann, Dirk Matzke und Rene Mercy, bei meiner Kollegin Kristina Schädler, die die studentischen Arbeiten mitbetreut hat und bei Mark Müller, der die Implementationen zu einem Gesamtsystem integriert hat. An Heike Pisch - wie fast immer - vielen Dank für die kritische Durchsicht des Manuskripts. Spezieller Dank gebührt Prof. Fritz Wysotzki, der mir in vielen fruchtbaren Diskussionen wichtige Anregungen zu dieser Arbeit gegeben hat.

den. Beim analogen Schließen werden folgende Teilprozesse modelliert: Die Ähnlichkeit zwischen einem aktuellen Problem und bereits gelösten Beispielen wird über ein strukturelles Distanzmaß auf (Term-)Bäumen bestimmt. Dieses Ähnlichkeitsmaß definiert gleichzeitig die Modifikationen, die notwendig sind, um ein Beispielprogramm an die neuen Anforderungen anzupassen. Bei erfolgreichem analogem Transfer wird das neu inferierte Schema sowie eine über Beispiel und aktuelles Problem generalisierende Struktur in das Gedächtnis eingefügt. Das Problem des analogen Transfers bei nicht-isomorphen Strukturen wird diskutiert und es werden Lösungsideen aufgezeigt. Der Ansatz ist als erster Prototyp in Lisp implementiert.

Abstract: Programming by Analogy

We propose an approach to programming by analogy which is based on the notion of recursive program schemes. If there are no examples with sufficient structural similarity to a new problem, the program scheme is inferred from the scratch using induction over input-output examples. Program schemes are stored in a hierarchically organized knowledge structure by constructing generalized schemes from concrete programs. Our approach handles the following processes of analogical reasoning: Similarity between new problems and example solutions are calculated by a structural measure defined on trees. In addition, this similarity measure defines which modifications have to be performed to adapt the example solution to the new problem. In case of succesful transfer the new solution together with a generalized structure are stored in memory. The problem of handling non-isomorphic structures is discussed and possibilities for handling transfer in such cases are proposed. Our approach is implemented as a first prototype in Lisp.

1 Einführung

Analoges Schließen ist ein Inferenzprinzip, das in vielen Bereichen menschlicher und maschineller Informationsverarbeitung nutzbar ist. Der analoge Transfer von bereits vorhandenen Wissensstrukturen auf neue Anforderungen ermöglicht eine effiziente Informationsverarbeitung, da zumindest Teilstrukturen eines aktuellen Problems nicht gänzlich neu inferiert werden müssen.

Typische Bereiche, in denen analoges Schließen in psychologischen Experimenten untersucht und für die Computermodelle entwickelt werden, sind Begriffsbildung, Kausalerklärungen und Problemlösen. Bei Begriffsanalogien wird eine Relation, die zwischen zwei Ausdrücken besteht, auf ein anderes Paar von Ausdrücken übertragen (siehe Klix, 1993): etwa die Inklusionsrelation (*Dackel zu Hund verhält sich wie Rose zu Blume*), semantische Relationen (z. B. Instrument: *Malen zu Pinsel verhält sich wie Reiten zu Sattel*), oder auch Relationen zwischen Begriffsklassen (z. B. Kausalität: *Zünden zu Brennen verhält sich wie Erhitzen zu Schmelzen*). Das Erschließen von Kausalbeziehungen durch Nutzung bereits bekannter Strukturen wird beispielsweise für die Konstruktion physikalischer Erklärungsmodelle eingesetzt (siehe Falkenhainer, Forbus & Gentner, 1989). So kann die Struktur des Sonnensystems auf die Struktur eines Atoms abgebildet werden, um auf diese Weise Wirkkräfte im Atom analog zu den (bereits bekannten) Wirkkräften des Sonnensystems zu erklären (Rutherford-Analogie). Beim analogen Problemlösen können Lösungen von bereits erfolgreich bearbeiteten Problemen auf strukturähnliche neue Probleme übertragen werden (Anderson & Thompson, 1989, Novick & Holyoak, 1991). Während bei Begriffsanalogien und bei der Übertragung von Kausalmodellen Beispiel- und Zielproblem aus zwei verschiedenen Wissensbereichen stammen (sogenannte *between domain analogies*), wird beim

Problemlösen innerhalb eines Bereichs gearbeitet (sogenannte *within domain analogies*, siehe Vosniadou & Ortony 1989). So kann eine bereits erfolgreich in Lisp implementierte Funktion als Beispiel dienen, um die Lösung einer ähnlichen Funktion zu konstruieren.

Allen Analogieschlüssen liegt das Prinzip des Strukturvergleichs und der Strukturübertragung zugrunde (siehe Abbildung 1, siehe Klix, 1993; Holland, Holyoak, Nisbett & Thagard, 1986, Kapitel 2.2). Eine Beispielstruktur (A, A'; Urbildbereich) ist durch Elementarobjekte (wie Begriffe, physikalische Körper, Grundfunktionen einer Programmiersprache) und Relationen auf diesen Elementarobjekten repräsentiert. Bei Begriffsanalogien wird genau eine Relation (R_1^*), die zwischen zwei Begriffen besteht, betrachtet. Bei Kausalstrukturen und Problemen bzw. Problemlösungen sind die Elementarobjekte dagegen durch verschiedene Relationen verknüpft. Elementarobjekte können zusätzlich durch Attribute (einstellige Relationen) charakterisiert sein. Die Ähnlichkeit zwischen Beispiel- und Zielbereich wird durch einen Vergleich (Abbildung $S : Beispielstruktur \rightarrow Zielstruktur$) der relationalen Strukturen ermittelt (Gentner 1983). Im Zielbereich (B, B'; Bildbereich) fehlende Relationen (R_2^*) werden vom Beispielbereich übernommen. Analogieschlüsse entsprechen im Idealfall homomorphen Abbildungen: Die Struktur B' wird via A' als auch via B erreicht ($R_2^*(S(A)) = B' = S(R_1^*(A))$). Diese strenge Beziehung gilt jedoch für viele praktisch relevanten Bereiche, in denen Analogieschlüsse angewendet werden, nicht. Für Fälle, in denen die Funktion S nicht eindeutig festgelegt ist, führen Holland und andere (Holland, Holyoak, Nisbett & Thagard, 1986) die Bezeichnung Quasi-Homomorphismen ein.

————————-

Abbildung 1 ungefähr hier

————————-

Im Gegensatz zu Oberflächenähnlichkeiten (*literal similarity*) sind Analogien nicht durch die Übertragung von Attributen, sondern von Strukturen, also von mehrstelligen Relationen, charakterisiert. Entsprechend werden Beispiel- und Zielbereich als strukturierte Objekte repräsentiert. Formal handelt es sich dabei um Graphen. Als Format der Wissensrepräsentation bieten sich *frames* (im folgenden Schemata) an. Die *slots* eines Schemas repräsentieren Basisobjekte und Attribute. Mehrstellige Relationen können über die Slots definiert werden.

Im folgenden konzentrieren wir uns auf Analogien beim Problemlösen. Durch die Verwendung von *within analogies* bestehen hier Beziehungen zum fallbasierten Schließen (Riesbeck & Schank, 1989). Im Unterschied zum fallbasierten Schließen sind Ähnlichkeitsvergleiche hier, wie bei Analogien allgemein, strukturbasiert und nicht merkmalsbasiert. Abbildung 1 kann als Rahmen für den Bereich Problemlösen betrachtet werden, indem A als Beispielproblem, A' als Beispiellösung, B als Zielproblem und B' als zu konstruierende Ziellösung interpretiert werden.

Holyoak (Novick & Holyoak, 1991) identifiziert vier Teilprozesse beim analogen Problemlösen:

- ***Retrieval***: Abruf eines dem Zielproblem strukturähnlichen Beispielproblems, für das bereits eine Lösung vorliegt
- ***Mapping***: Identifikation von strukturellen Gemeinsamkeiten zwischen Beispiel- und Zielproblem
- ***Adaptation***: Anpassung der Beispiellösung an das Zielproblem
- ***Learning***: Generalisierung über die gemeinsame Struktur von Beispiel- und Zielproblem.

Der Vergleichsprozeß (*mapping*) ist sowohl für den Abruf eines möglichst ähnlichen Beispiels als auch für die Anpassung (*adaptation*) der Beispiellösung an das Zielproblem zentral. Der Vergleich wird üblicherweise mit einem Ähnlichkeits- oder Distanzmaß realisiert. Dieses Maß soll insbesondere das Ausmaß der strukturellen Ähnlichkeit erfassen (z. B. Falkenhainer, Forbus & Gentner, 1989; Holyoak & Thagard, 1989). Beispielsweise identifizieren Falkenhainer und andere (Falkenhainer, Forbus & Gentner, 1989) mit einem *constraint*-basierten Ansatz den größten gemeinsamen Teilgraph von Beispiel- und Zielproblem. Wurde die Beispiellösung erfolgreich an das Zielproblem angepaßt, so kann eine Erweiterung der Wissensbasis erfolgen, indem ein generalisiertes Schema aufgebaut wird, das die gemeinsame Struktur beider Probleme repräsentiert.

Analoges Problemlösen wird vor allem in den Bereichen Mathematik (Novick & Holyoak, 1991; Reed & Bolstad, 1991; Ross & Kennedy, 1990) und Programmierung (Anderson & Thompson, 1989; Weber, 1994) untersucht. Empirische Untersuchungen zeigen insbesondere, daß die Ähnlichkeit zwischen Beispiel- und Zielproblem sowohl den analogen Transfer als auch die Übertragbarkeit auf neue Probleme (Generalisierungserfolg) beeinflußt (Novick & Holyoak 1991; Reed & Bolstad, 1991; Ross & Kennedy 1990; Schmid & Kaup, 1995). Implementierte kognitionswissenschaftlich orientierte Modelle zum analogen Schließen im Bereich Programmierung wurden von Anderson (Anderson & Thompson, 1989) und von Weber (1994) entwickelt. Anderson nutzt Mechanismen des analogen Problemlösens in der Produktionssystemarchitektur ACT-R (Anderson, 1993, Kapitel 4) und demonstriert deren Anwendbarkeit unter anderem beim Erstellen von Lisp-Funktionen. Weber integriert analoges Schließen in ein intelligentes tutorielles System zum Erlernen von Lisp. Programmieranfänger werden durch Erinnerung an eigene bereits implementierte Funktionen beim Erwerb von Programmierfertigkeiten unterstützt (Weber, 1996).

Im Bereich Automatisches Programmieren (Biermann, Guiho & Kodratoff, 1984) schlug Dershowitz (1986) eine Methode vor, bei der Analogien zwischen der Spezifikation eines bereits implementierten Programms und eines neuen Programmierproblems gezogen werden. Der Code des bereits vorliegenden Programms wird dann auf Basis der Gemeinsamkeiten und Unterschiede der Spezifikationen so modifiziert, daß er der neuen Spezifikation genügt. Ein weiterer Ansatz, bei dem Programmierprobleme als Spezifikationen repräsentiert werden, stammt von Williams (1988). Während Dershowitz mit einer imperativen Zielsprache arbeitet, verwendet Williams die funktionale Sprache Lisp. Dershowitz versucht, ein vollautomatisches System zu entwickeln und benötigt entsprechend eine Menge von Regeln und Gleichungen zur Bestimmung von Invarianten und zur Programmtransformation als Hintergrundwissen. Williams System ist dagegen auf Interaktion mit einem Benutzer ausgelegt. Das System sucht in einem Gedächtnis abgelegte Code-Fragmente und läßt den Benutzer entscheiden, ob solche Fragmente zur Lösung des neuen Problems herangezogen werden sollen.

Die Idee, Systeme zur Automatischen Programmierung (AP) mit einem Gedächtnis und Techniken zur Wiederverwendbarkeit von bereits implementierten Programmen auszustatten, wurde bereits in den 70er Jahren geäußert (Manna & Waldinger 1975). Obwohl die Nutzung von Techniken des analogen Problemlösens eine Performanzsteigerung von AP-Systemen erwarten läßt, existieren nur wenige ausgearbeitete Ansätze. Auch im derzeit aktuellen Bereich der Induktiven Logischen Programmierung wird diese Idee kaum genutzt (Sadohara & Haraguchi, 1995). Im Gegensatz zu den oben genannten spezifikationsbasierten Ansätzen, wird im Bereich der Induktiven Logischen Programmierung ein Programm auf Basis von Ein-Ausgabe-Beispielen konstruiert. Während die Transformation von Spezifikationen in lauffähige Programme deduktiv, mithilfe von Termersetzungsregeln, erfolgt, ist die Programmkonstruktion aus Beispielen ein induktiver Prozeß, der darauf basiert, eine kleinste gemeinsame Generalisierung über die Beispiele zu konstruieren. Dieses induktive Vorgehen ist eher als Modell menschlicher Programmiertätigkeit geeignet als die Verwendung von vollständigen Spezifikationen.

Auch der im folgenden dargestellte Ansatz geht von Beispielen aus. Im Gegensatz zur Induktiven Logischen Programmierung wird als Zielsprache eine abstrakte funktionale Sprache verwendet. Zunächst werden rekursive Programmschemata (RPS) eingeführt, die abstrakten funktionalen Programmen entsprechen. Es wird dargestellt, wie solche RPSs induktiv aus Ein-Ausgabe-Beispielen inferiert werden können. Dieser, durch Konzepte der theoretischen Informatik fundierte Ansatz, liefert den Ausgangspunkt für die Methode zum analogen Programmieren, die dann in den nächsten beiden Abschnitten dargestellt wird.

2 Rekursive Programmschemata

Im folgenden stellen wir dar, wie rekursive Programme aus wenigen Ein-Ausgabe-Beispielen inferiert werden können, wenn keine Beispiellösungen vorliegen, die zum analogen Transfer genutzt werden können. Anstelle konkreter - etwa in Lisp implementierter - Programme, verwenden wir rekursive Programmschemata (Engelfriet, 1974). Solche RPSs werden auf Basis von Funktionssymbolen einer Termalgebra definiert. Diese Symbole können durch Funktionen mit unterschiedlicher Semantik interpretiert werden. Damit repräsentiert ein RPS die Klasse aller strukturgleichen Programme (siehe auch Vorberg & Göbel, 1991). Gleichzeitig sind solche RPSs durch die Betonung von Programmstrukturen ein geeignetes

Repräsentationsformat für analoges Schließen. Sie erfüllen alle Charakteristika, die im Bereich der Wissensrepräsentation für Schemata gefordert werden (siehe oben, vgl. Anderson & Thompson 1989; Novick & Holyoak, 1991; Weber 1996).

Wir gehen davon aus, daß ein aktuelles Problem zunächst exploriert wird, indem für verschiedene Eingaben, die mit dem gesuchten Programm bearbeitet werden sollen, die notwendigen Operatoren zur Berechnung der Ausgabe durch allgemeine Problemlösestrategien (etwa heuristische Suchverfahren, siehe Nilsson, 1980) ermittelt werden. Beispielsweise kann ein Programmieranfänger, dem bereits einige primitive Operationen einer Programmiersprache bekannt sind, nach solchen Operatorfolgen suchen, die für eine Liste von einem bis zu drei Elementen jeweils das letzte Element liefern (siehe Tabelle 1). Sein Ziel ist es, in den Beispielen Regelmäßigkeiten zu identifizieren, die eine rekursive Verallgemeinerung auf Listen mit beliebig vielen Elementen erlauben.

————————

Tabelle 1 ungefähr hier

————————

Die Ausdrücke in Tabelle 1 können als bedingte Anweisungen gelesen werden, etwa "Wenn *empty(tail(l))* Dann *head(l)*". Diese bedingten Anweisungen können zu einer verschachtelten bedingten Anweisung, im folgenden als Anfangsprogramm bezeichnet, zusammengefaßt werden:

```
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 ].
```

Auf die verwendeten Verfahren der heuristischen Suche, sowie auf die Methode zur Verknüpfung der Operatorfolgen zu einem Anfangsprogramm gehen wir hier nicht weiter ein (siehe hierzu Schmid & Wysotzki, 1996b). Wir gehen im folgenden direkt von solchen Anfangsprogrammen aus. Zur Notation der bedingten Anweisung verwenden wir das McCarthy-Konditional: *if x then y else z* $=_{Def}$ *g(x,y,z)*. Damit sind alle Terme einheitlich in Präfix-Notation dargestellt.

Um aus dem konstruierten Anfangsprogramm ein RPSs zu inferieren, verwenden wir eine Methode zur induktiven Programmsynthese, die Wysotzki (Wysotzki, 1983) vorgeschlagen hat (siehe auch Le Blanc, 1994). Grundgedanke dieser Methode ist es, in einem Anfangsprogramm einen Term tr zu identifizieren, der wiederholt mit unterschiedlichen Parameterbelegungen $\Theta = [v/t]$ vorkommt und mit dem schrittweise immer komplexere Anfangsstücke des gesuchten Programms konstruierbar sind:

$$G^{(i)} = tr(G^{(i-1)}_{[v/t]}).$$

Um tr und Θ zu ermitteln, wird das Anfangsprogramm stückweise betrachtet (Ω bedeuted "nicht definiert"):

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

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

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

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

Wir identifizieren tr mit $g(empty(tail(l)), head(l), \Omega)$ und Θ mit $[l/tail(l)]$, damit gilt

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

Wir nehmen nun an, daß die durch einen einfachen Mustervergleichsprozeß identifizierte Struktur der gesuchten rekursiven Struktur entspricht und "Falten" das Anfangsprogramm zu

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

Die Funktion *last* ist hier durch ein RPS repräsentiert, das durch geeignete Interpretation der Funktionssymbole in eine konkrete Programmiersprache überführbar ist. Allgemein gilt, daß die Funktionssymbole eines RPS beliebig semantisch interpretierbare Funktionen repräsentieren. Betrachten wir etwa das RPS

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

Dieses RPS repräsentiert nicht nur die Fakultätsfunktion, sondern auch die Summenfunktion (mit der Interpretation $1 \rightarrow 0$, $mult \rightarrow plus$).

Wenn wir im folgenden auf Programmieren durch analoges Schließen eingehen, kann Abbildung 1 so interpretiert werden, daß A ein Anfangsprogramm und A' ein zugehöriges RPS ist. Zum Anfangsprogramm B wird ein RPS B' gesucht, das durch analogen Transfer konstruiert werden soll. Dieses RPS entspricht bei geeigneter Interpretation der Funktionssymbole dem gesuchten funktionalen Programm. Gleichzeitig entspricht es einem abstrakten Schema, das die Struktur der gesuchten rekursiven Funktion repräsentiert. Die Schemastruktur legt die funktionale Bedeutung von Termen fest. Beispielsweise definiert das erste Argument des McCarthy-

Funktionals die Abbruchbedingung der Rekursion, entsprechend steht hier eine boolsche Funktion. Anders als in vielen Ansätzen zum analogen Problemlösen müssen hier die Schemata nicht vorgegeben werden, sondern sie können direkt aus durch Problemlöseerfahrung gebildeten sequentiellen Lösungsstrukturen inferiert werden.

3 Gedächtnisorganisation und Abruf von Beispiellösungen

Um Programme durch analoges Schließen zu konstruieren, müssen bereits gelöste Programmierprobleme als Beispiele zur Verfügung gestellt werden. In unserem Ansatz speichern wir bereits erstellte RPSs in einem hierarchischen Gedächtnis (siehe Abbildung 2). Wird ein RPS zum analogen Problemlösen verwendet, so wird das durch analogen Transfer neu erzeugte RPS in das Gedächtnis eingefügt und ein abstraktes RPS erzeugt, das über die gemeinsame Struktur von Beispiel-RPS und neuem RPS generalisiert. Bisher werden alle nicht übereinstimmenden Symbole der beiden RPS durch Variablen ersetzt. Die Generalisierung kann zusätzlich über Typinformation (Datentypen der Ein- und Ausgabeparameter der Funktionen) gesteuert werden. In Abweichung zu gängigen Ansätzen zum analogen Problemlösen, können nicht nur die Beispiele selbst, sondern auch die generalisierten RPS zum analogen Transfer herangezogen werden.

——————————-

Abbildung 2 ungefähr hier

——————————-

Anfangs besteht das Gedächtnis aus mehreren unverbundenen RPS-Bäumen, zum Teil sogar aus isolierten RPS-Knoten. Mit zunehmender

Lernerfahrung sollte sich eine immer stärker verbundene Struktur aufbauen. Die Entwicklung dieser Gedächtnisstruktur wird durch die Art der Lernerfahrung bedingt (Pirolli & Anderson, 1985): werden z. B. Programmierprobleme gestellt, die große strukturelle Unterschiede aufweisen, kommt es häufig vor, daß ein RPS durch die im vorangegangenen Abschnitt dargestellte Inferenzmethode - ohne Zuhilfenahme eines Beispiels - konstruiert werden muß. Das Gedächtnis besteht dann aus vielen unverbundenen RPSs.

Die Auswahl eines möglichst strukturähnlichen Beispiels erfolgt über ein Maß der Baumdistanz von Lu (1979). Das aktuelle Problem liegt als Anfangsbaum vor. Um ein möglichst strukturähnliches Beispielproblem im Gedächtnis zu identifizieren, müssen die dort gespeicherten RPSs zu Anfangsprogrammen expandiert werden. Dies geschieht durch mehrmaliges Auffalten des RPS (die Definition des RPS wird auf dessen rechten Seite anstelle des rekursiven Aufrufs mit der entsprechenden Subsitution der Parameter eingefügt), bis es die Tiefe des aktuellen Anfangsprogramms erreicht hat. Das Anfangsprogramm entspricht einem Term, der entsprechend als Termbaum notiert werden kann (siehe Abbildung 3).

————————-

Abbildung 3 ungefähr hier

————————-

Die Ermittlung der strukturellen Ähnlichkeit des aktuellen Anfangsbaums mit einem expandierten Anfangsbaum aus dem RPS-Gedächtnis erfolgt nun, indem alle Operationen gezählt werden, die notwendig sind, um den Beispiel-Baum in den aktuellen Baum zu transformieren. Operationen sind die Abbildung eines Knotenlabels auf ein anderes (s), das Löschen eines Knotens (d) und das 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. Für s, d und i können gleiche oder verschiedene Wertigkeiten angegeben werden. Das Maß kann zusätzlich an der durchschnittlichen Knotenzahl der Bäume relativiert werden. Zwei Bäume sind isomorph, wenn nur eineindeutige Substitutionen durchgeführt werden müssen, um einen Baum in den anderen zu transformieren. Wir explorieren zur Zeit verschiedene Möglichkeiten der konkreten Umsetzung dieses Ähnlichkeitsmaßes. Wie bei der Generalisierung kann auch bei der Ermittlung der Strukturähnlichkeit Typinformation herangezogen werden. Für jedes Funktionssymbol (inklusive Konstanten) wird der Datentyp ermittelt. Der Parameter s hat einen geringen Wert für typgleiche Abbildungen (etwa *mult* auf *plus*) und einen höheren Wert für Abbildungen von Funktionen eines Typs auf einen anderen Typ (etwa *head* auf *pred*).

Alternativ zu der auf Bäumen definierten Distanzmetrik explorieren wir eine Methode des Graphmatching, bei der der größte gemeinsame Untergraph der beiden Termbäume identifiziert wird. Der aufwendige Suchprozeß wird durch den Einsatz einer speziellen Hopfield-Netz Variante effizienter gestaltet (Schädler & Wysotzki, 1996; vgl. Holyoak & Thagard, 1989).

4 Adaptation: Das Problem nicht-isomorpher Strukturen

Ist ein möglichst strukturähnliches Beispiel zu einem aktuellen Problem gefunden, wird im nächsten Schritt versucht, die Beispiellösung so zu modifizieren, daß sie eine Lösung des aktuellen Problems darstellt. Im Bereich der Programmerstellung wird häufig mit strukturgleichen (isomorphen) Problemen gearbeitet. Die Standardanwendung ist die Verwendung der Fakultätsfunktion zur Lösung der Summenfunktion (siehe vorangegangener Abschnitt, siehe Anderson & Thompson, 1989). Die Adaptation beschränkt sich hier auf eine eineindeutige Substitution von Funktionssymbolen. In diesem Fall liefert die Baumdistanz eine Menge von eineindeutigen Abbildungen von Knotenlabeln, die direkt zur Modfikation des Beispiel-RPS verwendet werden können. Einfügungen und Löschungen von Knoten werden nicht benötigt.

Problematisch wird die Adaptation der Beispiellösung an das aktuelle Problem dann, wenn die beiden Strukturen nur strukturähnlich, aber nicht strukturidentisch sind. Dieser Fall wird in der Praxis jedoch sicherlich häufiger vorkommen als die vollständige Übereinstimmung von Strukturen. Nehmen wir an, es soll ein anschauliches Problem, das Abbauen eines Turmes aus n Blöcken in Analogie zur Fakultätsfunktion, also einem abstrakt/mathematischen Problem, gelöst werden (siehe hierzu auch Schmid & Wysotzki 1996). Der Anfangsbaum für das *clearblock*-Problem ist in Abbildung 4 dargestellt. Ein Vergleich der Baumstrukturen aus Abbildung 3 und Abbildung 4 führt zu den in Tabelle 2 aufgeführten Beziehungen.

————————-

Abbildung 4 ungefähr hier

————————-

————————

Tabelle 2 ungefähr hier

————————

Zusätzlich zu Ersetzungen (s) von Symbolen müssen zwei Einfügungen (i) von $topof$-Knoten durchgeführt werden. Die Probleme sind nicht isomorph. Die Transformation des Fakultätsproblems in das *clearblock*-Problem entspricht einem Epimorphismus (einer surjektiven Abbildung). Entsprechend ist die Adaptation hier nicht trivial zu definieren. Es muß identifiziert werden, an welcher Stelle im Fakultät-RPS das Symbol *pred* durch das Symbol *topof* zu ersetzen ist und an welchen Stellen *topof* zustätzlich eingefügt werden muß. Die beiden nicht isomorphen RPSs lauten:

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

Das von uns gewählte Verfahren der Baumtransformation zur Bestimmung der Strukturähnlichkeit kann verwendet werden, um die Stellen im RPS zu identifizieren, an denen Einfügungen beziehungsweise Löschungen vorgenommen werden müssen. So kann zum Beispiel festgestellt werden, daß bei der Transformation des Fakultät-Baums in den *clearblock*-Baum die Einfügungen immer im ersten Argument der zweistelligen Operation (*mult* in der Beispielstruktur, *puttable* in der Zielstruktur) vorgenommen werden. Entsprechend wird *topof* an dieser Stelle im RPS eingefügt.

Eine alternative Strategie zur direkten Adaptation bei nicht-isomorphen Strukturen ist es, zu versuchen, eines der beiden Probleme so zu transformieren, daß Isomorphie erreicht wird. Die Transformation kann beispielsweise durch Hinzunahme von Hintergrundwissen in Form von Termersetzungsregeln erfolgen. Verwenden wir die Regel

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

so erreichen wir zumindest eine partielle Isomorphie zum Fakultätsproblem (siehe Abbildung 5). 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:

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

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 *clear-block*-Problem und der Fakultätsberechnung konstruiert. Diese Modifikation war jedoch nur durch Hinzunahme von semantischer Information über den Problembereich möglich. Diese Strategie ist also im Vergleich zur erstgenannten wissensabhängig und damit weniger allgemein. Es scheint uns jedoch lohnend, beide Strategien bezüglich ihrer formalen Eigenschaften zu vergleichen und in psychologischen Untersuchungen zu prüfen, ob eine oder beide der vorgeschlagenen Strategien beim analogen Transfer angewendet werden.

——————————-

Abbildung 5 ungefähr hier

——————————-

5 Bewertung und Ausblick

Der vorgeschlagene Ansatz zum analogen Programmieren ermöglicht eine integrierte Behandlung von *learning by doing* und *learning by analo-*

gy. Mit einer allgemeinen Methode zur induktiven Programmsynthese kann modelliert werden, wie konkrete Problemlöseerfahrungen zu rekursiven Programmschemata generalisiert werden können. Die induktive Programmsynthese liefert also einen in der theoretischen Informatik fundierten Ansatz zur Beschreibung von Prozessen des Schemaerwerbs (Rumelhart & Norman, 1978). Rekursive Programmschemata erfüllen zudem die in der Wissensrepräsentation für Schemata formulierten Eigenschaften (siehe oben) und stellen somit einen geeigneten Repräsentationsformalismus für Prozesse des analogen Schließens bereit.

Unser Ansatz ist in dem System IPAL realisiert, das als erster Prototyp in Lisp implementiert ist (Müller & Schmid, 1996). In IPAL nehmen wir eine allgemeinere Sicht auf Problemlösen, induktive Inferenz und analoges Schließen, als in diesem Artikel dargestellt (Schmid & Wysotzki, 1996a; 1996b). Die zentrale Wissensstruktur der rekursiven Programmschemata kann nicht nur als abstrakte Repräsentation von funktionalen Programmen aufgefaßt werden, sondern allgemein als Repräsentation von Problemlösefertigkeiten. Diese Idee läßt sich etwa an dem im vorangegangenen Abschnitt dargestellten *clearblock*-Problem veranschaulichen. Das *clearblock*-RPS kann einmal als Wissen über die Programmstruktur gesehen werden, mit der das Abbauen eines Turmes gesteuert werden kann. Zum zweiten kann es als mentale Repräsentation der Fertigkeit, einen Turm aus beliebig vielen Blöcken abzubauen, aufgefaßt werden, die aus der Erfahrung mit konkreten Problemen, etwa einem Turm aus drei Blöcken, generalisiert wurde. Damit liefert IPAL einen allgemeinen Rahmen für die Beschreibung des Erwerbs von Problemlösefertigkeiten, bei dem Problemlösen, induktive Generalisierung und analoges Schließen und Lernen in Beziehung gebracht werden.

Literatur

Anderson, J. R. (1993). *Rules of the Mind.* Lawrence Erlbaum: Hillsdale, NJ.

Anderson, J. R. & Thompson, R. (1989). Use of analogy in a production system architecture. In: S. Vosniadou and A. Ortony (Eds.), *Similarity and Analogical Reasoning* (pp. 267–297). Cambridge University Press.

Biermann, A. W., Guiho, G., & Kodratoff, Y. (Eds.) (1984). *Automatic Program Construction Techniques.* Collier Macmillan.

Le Blanc, G. (1994). BMWk revisited: Generalization and formalization of an algorithm for detecting recursive relations in term sequences. In: F. Bergadano and L. de Raedt (Eds.), *Machine Learning, Proc. of ECML-94* (pp. 183–197).

Dershowitz, N. (1986). Programming by analogy. In: J. G. Carbonell R. S. Michalski and T. M. Mitchell (Eds.), *Machine Learning - An Artificial Intelligence Approach* (Vol. 2, pp. 393–422). Morgan Kaufmann: Los Altos, CA.

Engelfriet, J. (1974). *Simple Program Schemes and Formal Languages.* Springer.

Falkenhainer, B., Forbus, K. D., & Gentner, D. (1989). The structure mapping engine: Algorithm and example. *Artificial Intelligence*, *41*, 1–63.

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

Holland, K. J., Holyoak, J. H., Nisbett, R. E. & Thagard, P. R. (1986). *Induction - Processes of Inference, Learning, and Discovery.* MIT Press: Cambridge, MA.

Holyoak, K. J. & Thagard, P.R. (1989). Analogical mapping by constraint satisfaction. *Cognitive Science*, *13*, 295–355.

Klix, F. (1993). Analytische Betrachtungen über Struktur und Funktion von Inferenzen. *Zeitschrift für Psychologie*, *201*, 393–414.

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

Müller, M. & Schmid, U. (1996). IPAL - a system that integrates problem solving, skill acquisition, and learning by analogy. In: U. Schmid, J. Krems, and F. Wysotzki (Eds.), *Proceedings of the First European Workshop on Cognitive Modelling (14.-16.11.96, TU Berlin)* (pp. 246–247).

Manna, Z. & Waldinger, R. (1975). Knowledge and reasoning in program synthesis. *Artificial Intelligence*, *6*, 175–208.

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

Nilsson, N. J. (1980). *Principles of Artificial Intelligence.* Springer.

Pirolli, P. L. & Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology*, *39*, 240–272.

Reed, S. K. & Bolstad, C. A. (1991). Use of examples and procedures in problem solving. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 17(4), 753–766.*

Ross, B. H. & Kennedy, P. T. (1990). Generalizing from the use of earlier examples in problem solving. Journal of Experimental Psychology: Learning, Memory, and Cognition*,* 16(1), 42–55.

Rumelhart, D. E. & Norman, D. A. (1978). Accretion, tuning and restructuring: Three modes of learning. In: J. W. Cotton and R. L. Klatzky (Eds.), *Semantic Factors in Cognition* (pp. 37–53). Erlbaum: Hillsdale, NJ.

Riesbeck, C. K. & Schank, R. C. (1989). *Inside Case-based Reasoning.* Lawrence Erlbaum: Hillsdale, NJ.

Sadohara, K. & Haraguchi, M. (1995). Analogical logic program synthesis from examples. In: N. Lavrac and S. Wrobel (Eds.), *Proceedings of the Eighth European Conference in Machine Learning (ECML-95)* (pp. 232–244). Springer.

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

Schädler, K. & Wysotzki, F. (1996). Klassifizierungslernen mit Hilfe spezieller Hopfield-Netze. In W. Dilger, M. Schlosser, J. Zeidler, and A. Ittner (Eds.), *Beiträge zum 9.Fachgruppentreffen Maschinelles Lernen der GI-Fachgruppe 1.1.3. August 1996*, Chemnitzer Informatik-Berichte, CSR-96-06, Technische Universität Chemnitz-Zwickau.

Schmid, U. & Wysotzki, F. (1996a). Skill acquisition can be regarded as program synthesis. In: U. Schmid, J. Krems, and F. Wysotzki

(Eds.), *Proceedings of the First European Workshop on Cognitive Modelling (14.-16.11.96, TU Berlin)* (39–45).

Schmid, U. & Wysotzki, F.. Induktion von rekursiven Programmschemata und Analoges Lernen. In: R. H. Kluwe (Ed.), *Kognitionswissenschaft: Strukturen und Prozesse intelligenter Systeme.* Westdeutscher Universitätsverlag: Wiesbaden, in Druck.

Vorberg, D. & Göbel, R. (1991). Das Lösen rekursiver Programmierprobleme: Rekursionsschemata. *Kognitionswissenschaft*, *1*, 83–95.

Vosniadou, S. & Ortony, A.. Similarity and analogical reasoning: A synthesis. In S. Vosniadou and A. Ortony (Eds.), *Similarity and Analogical Reasoning* (pp. 1–17). Cambridge University Press.

Weber, G. (1994). *Fallbasiertes Lernen und Analogien - Unterstützung von Problemlöse- und Lernprozessen in einem adaptiven Lernsystem.* PVU: Weinheim.

Weber, G. (1996). Episodic learner modeling. *Cognitive Science*, *20*, 195–236.

Williams, R. S. (1988). Learning to program by examining and modifying cases. In: J. Laird (Ed.), *Proceedings of the fifth international conference on Machine Learning, Univ. of. Michigan* (pp. 318–324). Morgan Kaufmann: San Mateo, CA.

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

Tabelle 1: Operatorfolgen zur Ausgabe des letzten Elements einer Liste von einem bis drei Elementen (*empty(l)* liefert wahr, wenn die Liste leer ist, sonst falsch; *head(l)* liefert das erste Element einer nichtleeren Liste; *tail(l)* liefert eine nichtleere Liste ohne ihr erstes Element)

Nr	Anzahl von Listenelementen		Operatorfolge
1	einelementige Liste	empty(tail(l))	head(l)
2	zweielementige Liste	empty(tail(tail(l)))	head(tail(l))
3	dreielementige Liste	empty(tail(tail(tail(l))))	head(tail(tail(l)))

Tabelle 2: Substitutionen zur Transformation des Fakultät-Problems in das *clearblock*-Problem (Die Abbildung der Identitätsfunktion $id(x) = x$ auf *topof* entspricht der Einfügung der Operation $topof$)

Fakultät	*clearblock*
x	x
1	s
equal0	cleartop
mult	puttable
pred	topof
id	topof

$\rightarrow$ surjektiv

$\leftarrow$ nicht rechtseindeutig

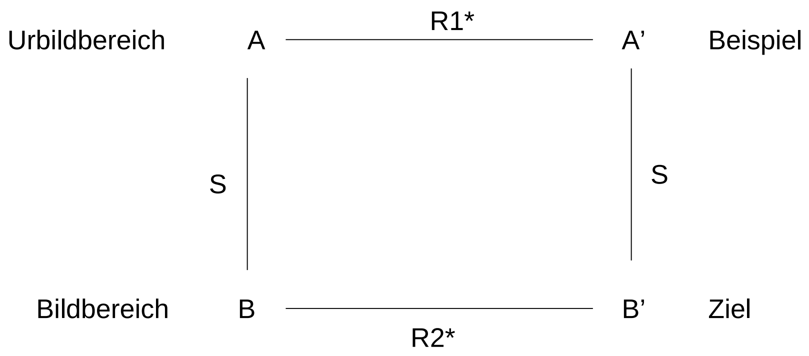
Abbildung 1: Analogieschlüsse als (verallgemeinerte) Homomorphismen

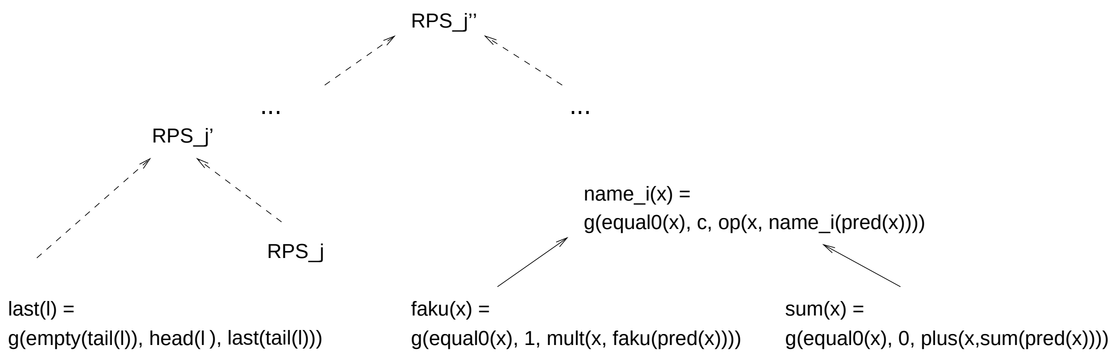
Abbildung 2: Struktur des RPS-Gedächtnisses

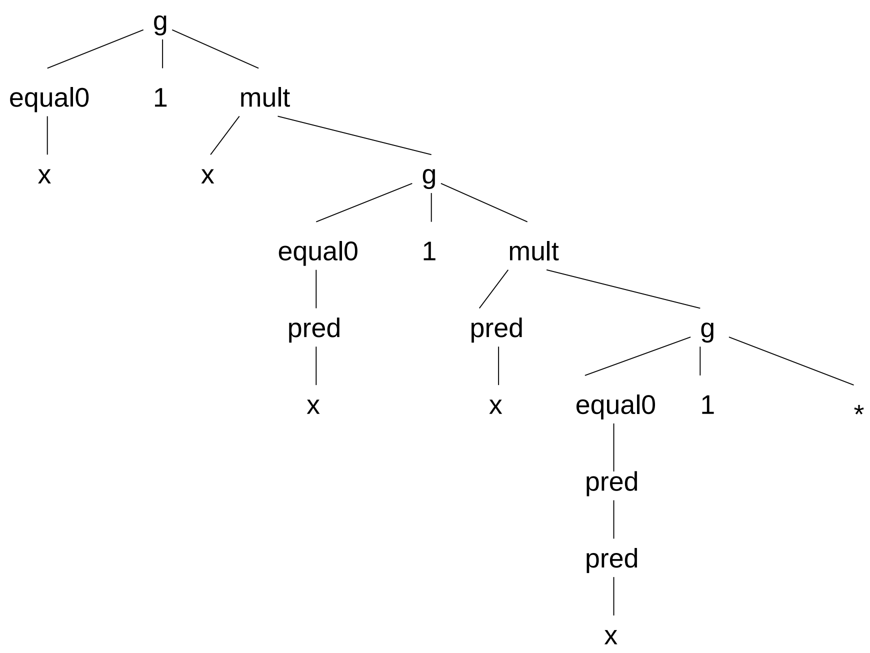
Abbildung 3: Das RPS "Fakultät" als Termbaum

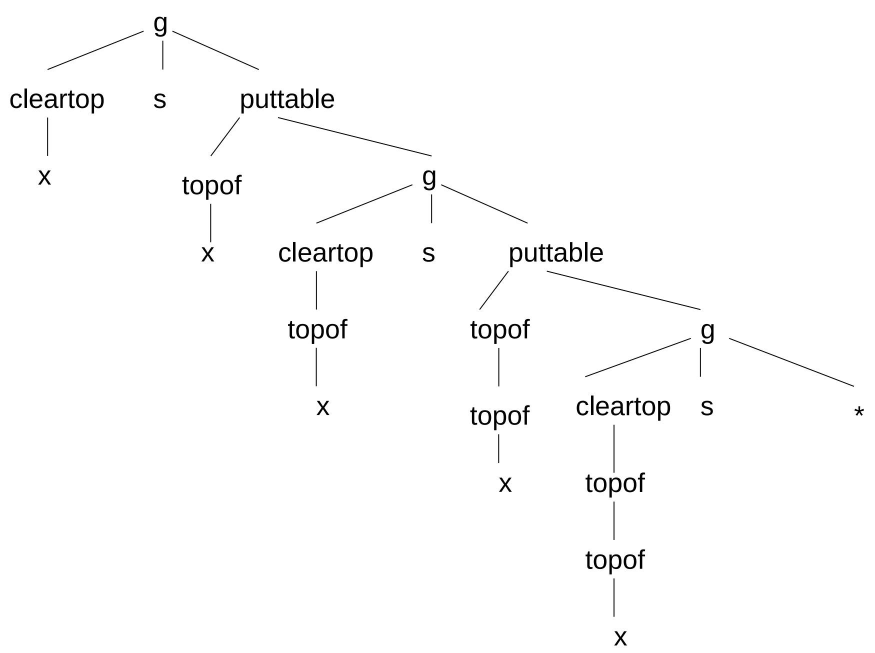
Abbildung 4: Anfangsbaum für das *clearblock*-Problem (*cleartop(x)* liefert wahr, wenn kein Block auf Block x liegt, sonst falsch; *puttable(x)* ist eine Operation, um Block x auf den Tisch zu legen; *topof(x)* liefert den Block, der auf einem Block x liegt; s isteine Situationsvariable, in der der aktuelle Zustand des Problems repräsentiert wird)

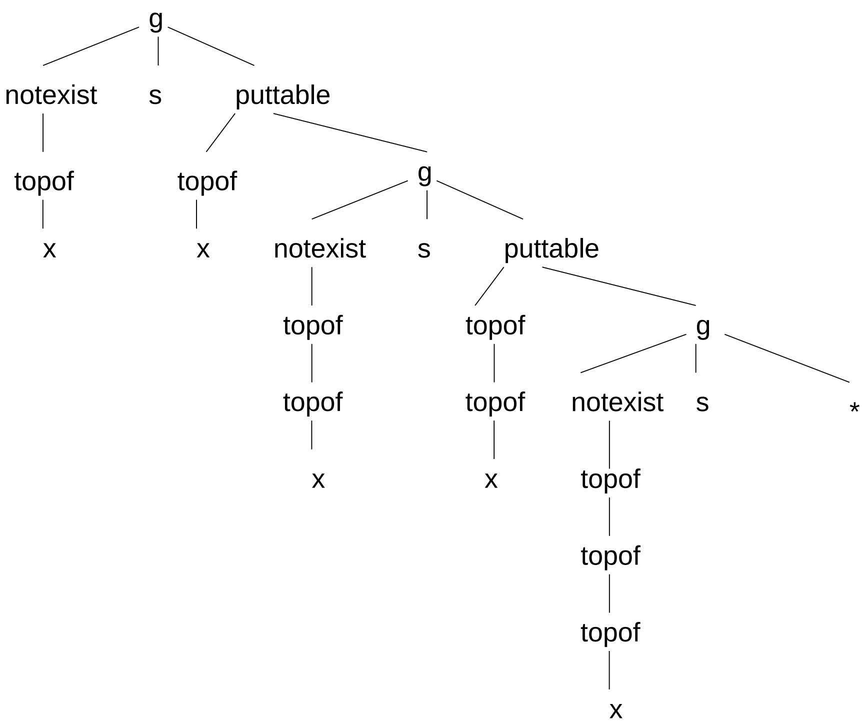
Abbildung 5: Durch Hintergrundwissen transformierter Anfangsbaum für das *clearblock*-Problem


R1*
Urbildbereich
A
A’
Beispiel
S
S
Bildbereich
B
B’
Ziel
R2*


RPS_j''
...
...
RPS_j'
RPS_j
name_i(x) =
g(equal0(x), c, op(x, name_i(pred(x))))
last(l) =
g(empty(tail(l)), head(l), last(tail(l)))
faku(x) =
g(equal0(x), 1, mult(x, faku(pred(x))))
sum(x) =
g(equal0(x), 0, plus(x,sum(pred(x))))


g
equal0
1
mult
x
x
g
equal0
1
mult
pred
pred
g
x
x
equal0
1
*
pred
pred
x


g
cleartop
s
puttable
x
topof
g
x
cleartop
s
puttable
topof
topof
g
x
topof
cleartop
s
*
x
topof
topof
x


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