Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany

SEKI-PROJEKT

SEKI MEMO

Elimination von Rekursionen

Ulrike Petersen

Memo SEKI-83-10

Überblick

Operationen sind in algorithmischen Spezifikationen oft rekursiv beschrieben. Um bei der Implementierung solcher Spezifikationen effiziente Programme zu erhalten, ist es notwendig, Rekursionen in eine repetitive Form zu überführen. Ausgehend von einem Klassifikationsschema für rekursive Funktionen werden zwei Techniken zur Entrekursivierung vorgestellt, die auf der abstrakten Ebene arbeiten.
Zunächst werden einige Transformationsschemata von Bauer und Wössner [BaWö 81] übernommen, modifiziert und erweitert, und in einem deterministischen Transformationsregelsystem zusammengefaßt, wobei es wesentlich ist, die Semantik der verwendeten Operationen zu berücksichtigen. Anschließend wird die heuristische unfold/fold-Methode von Darlington und Burstall [DaBu 77] beschrieben und einige ihrer Anwendungsmöglichkeiten dargestellt.

Abstract

Operations in algorithmic specifications are often described recursively. To obtain efficient programs for implementing such specifications, it is necessary to transform recursive functions into iterative ones.
Starting with a classification scheme for recursive functions, two technics for recursion removal are proposed, which work at abstract levels. At first some transformation schemes from Bauer and Wössner [BaWö 81] are modified, augmented and combined in a rule based transformation system, where it is essential to consider the semantics of the operations used. Subsequently the heuristic unfold/fold method of Darlington and Burstall [DaBu 77] is described and illustrated by examples.

Wenn nicht mehr Zahlen und Figuren
Sind Schlüssel aller Kreaturen,
Wenn die so singen oder küssen
Mehr als die Tiefgelehrten wissen,
Wenn sich die Welt ins freie Leben
Und in die Welt wird zurückbegeben,
Wenn dann sich wieder Licht und Schatten
Zu echter Klarheit werden gatten
Und man in Märchen und Gedichten
Erkennt die ewgen Weltgeschichten,
Dann fliegt von Einem geheimen Wort
Das ganze verkehrte Wesen fort.

Novalis
(aus: Geistliche Lieder)

Inhalt

1. Einführung

2. Erkennung von Rekursionen

3. Bestimmung der Art einer Rekursion

   3.1 Allgemeines
   3.2 Erkennungsregeln
   3.2 Beispiele

4. Transformationsschemata und Regeln

   4.1 Die lineare Rekursion
       4.1.1 Um-Klammerung
       4.1.2 Operandenvertauschung
       4.1.3 Funktionsumkehr
       4.1.4 Funktionsumkehr unter Verwendung eines stacks

   4.2 Die geschachtelte Rekursion

   4.3 Die kaskadenartige Rekursion - Entflechtung
       4.3.1 Allgemeines
       4.3.2 Entflechtung von Kaskaden
       4.3.3 Entflechtung einer geschachtelten Rekursion
       4.3.4 Transformation einer entflochtenen Form in eine repetitive

   4.4 Beispiele für die Transformation der verschiedenen Rekursionsarten
       4.4.1 Transformation der Fakultätsfunktion
       4.4.2 Transformation der '91-Funktion' von Manna
       4.4.3 Transformation der Fibonacci-Funktion

   4.5 Schlußbemerkung zu den Transformationen

5. Beweise für die Korrektheit der verwendeten Transformationsschemata

   5.1 Beweise für das linear rekursive Muster
       5.1.1 Um-Klammerung
       5.1.2 Operandenvertauschung
       5.1.3 Funktionsumkehr
       5.1.4 Funktionsumkehr unter Verwendung eines stacks

5.2 Beweise für die geschachtelt rekursiven Muster
5.2.1 einfach geschachtelt, Schachtelungstiefe n=2
5.2.1.1 Grundauflösung, ohne Nebenbedingung
5.2.1.2 Auflösung unter Eigenschaft 0
Überführung in eine lineare Rekursion
5.2.1.3 Auflösung unter Eigenschaft 0 und 1
5.2.1.4 Auflösung unter Eigenschaft 1,2 und 3
5.2.2 einfach geschachtelt, Schachtelungstiefe n≥2
5.2.3 verschränkt geschachtelt,Schachtelungstiefe n=2
5.2.4 verschränkt geschachtelt,Schachtelungstiefe n>2
5.2.5 linear verschränkt geschachtelt, Schachtelungstiefe n≥2

5.3 Beweise für die kaskadenartigen Muster
5.3.1 Entflechtung, Faktor n=2
5.3.2 Entflechtung, Faktor n>2

6. Die unfold/fold-Methode

6.1 Allgemeine Beschreibung

6.2 Betrachtungen und Beispiele nach Martin S. Feather
6.2.1 Das Telegrammproblem
6.2.2 Das ZAP-System
6.2.3 Die ZAP-Sprache
6.2.4 Default-Generatoren des ZAP-Systems
6.2.5 Zwei weitere Beispiele nach Martin S. Feather

6.3 Eine Synthese verschiedener Sortieralgorithmen nach John Darlington

6.4 Algorithmen zur Berechnung der transitiven Hülle einer Relation nach Lothar Schmitz

6.5 Abschließende Bemerkungen zur unfold/fold-Methode

7. Zusammenfassung und abschließende Bemerkungen

Anhang 1 Regelbasis für das Erkennen der Rekursionsart

Anhang 2 Regelbasis für die Transformationen

Anhang 3 Muster M1 - M11

Literaturangaben

Stichwortverzeichnis

1. Einführung

Diese Arbeit umfaßt theoretische Betrachtungen von Methoden, die rekursiv definierte Funktionen durch Programmtransformation in repetitive Formen übertragen.

Unter Programmtransformation verstehen wir jegliche Veränderung eines gegebenen Programms unter Verwendung syntaktischer und/oder semantischer Informationen über dieses Programm, wobei Korrektheit und Termination erhalten bleiben.
Ziel der hier betrachteten Programmtransformationen ist die Effizienzsteigerung von Algorithmen. Denkbar wäre auch Programmtransformation zur Vereinfachung von Programmstrukturen, z.B. um die Voraussetzungen zu schaffen, von einer Programmiersprache in eine andere übersetzen zu können, oder um eine Verifikationshilfe zu geben.

In der vorliegenden Arbeit soll ausschließlich die Elimination rekursiver Funktionsdefinitionen durch Programmtransformationen behandelt werden, wobei eine Funktion rekursiv heißt, wenn sie sich direkt oder indirekt auf sich selbst stützt (s. Kapitel 2).

Alle μ-rekursive Funktionen können in äquivalente nichtrekursive transformiert werden. In Programmiersprachen (wie ALGOL, PASCAL, SIMULA), in denen das Formulieren von Rekursionen erlaubt ist, wird das Problem mit Hilfe eines Kellers (stack) gelöst, in dem Parameter und Protokollvariablen abgespeichert werden. Damit wird jedoch das eigentliche Problem nur verlagert von der Programm- auf die Datenstruktur. Das Ziel dieser Arbeit ist es, zu zeigen, daß unter gewissen Bedingungen auf die oben beschriebene Problemverlagerung verzichtet werden kann, und bestimmte μ-rekursive Funktionen in iterative transformiert werden können ohne Unterstützung durch einen Keller. Um diese Möglichkeiten abzugrenzen, werden auch Transformationsschemata angegeben, bei denen auf einen Keller nicht verzichtet werden kann. Unter einem Transformationsschema verstehen wir zwei Funktionsmuster, die unter gewissen Bedingungen nachweisbar ineinander überführt werden können (siehe Kapitel 4).

Theoretische Betrachtungen von Strong [Stro 70] ergaben, daß bestimmte μ-rekursive Funktionen beschreibbar sind durch Flußdiagramme (flowchartable). Strong stellt ebenfalls fest, daß kaskadenartige rekursive Funktionen nicht in Form von Flußdiagrammen beschrieben werden können, und Paterson und Hewitt weisen in ihrem Artikel 'Comparative Schematology' [PaHe 70] nach, daß bei diesen Funktionen im allgemeinen weder auf einen Parameter- noch auf einen Protokollkeller verzichtet werden kann. Ein weiteres theoretisches Ergebnis von Strong ist, daß die Übertragbarkeit eines Programms in ein Flußdiagramm nicht entscheidbar ist.

Das hier zu entwerfende Transformationssystem wird syntaktische und semantische Informationen über gegebenen μ-rekursive Funktionen benutzen, um sie in eine effizientere repetitive Form zu übertragen. Angestrebt wird also ein auf Wissen basierendes System (knowledge-based system) , das automatisch die Elimination von Rekursionen durchführt.

In welchem Maße ein solches Transformationssystem tatsächlich die Effizienz von Programmen steigert, kann jedoch erst eine Analyse von Testläufen mit konkreten Funktionen zeigen. Hierbei ist jedoch zu berücksichtigen, daß ein Compiler ein gegebenes Programm bei der Übersetzung eventuell optimiert, sodaß die Effektivität der transformierten Programme nicht nur von den durchgeführten Transformationen abhängt.
Da das zu entwerfende System im Hinblick auf Effizienzsteigerung entwickelt wird, sollte die Eingabe aus einfachen, gut lesbaren rekursiven Definitionen von Funktionen besteht, die in kompliziertere, umfangreichere und gegebenenfalls unübersichtlichere, aber effizientere Programme transformiert werden.

Um aus einem gegebenen Programm die Teile herauszufiltern, die eventuell optimiert werden können, muß zunächst ein Algorithmus entwickelt werden, der Rekursionen erkennt und die entsprechenden Funktionen kennzeichnet. Dieser Vorgang wird in Kapitel 2 untersucht, wobei entscheidende Einschränkungen gemacht werden und auf einige Schwierigkeiten hingewiesen wird. Nachdem die rekursiven Funktionsdefinitionen, die von unserem Transformationssystem bearbeitet werden sollen, erkannt worden sind, muß in einem zweiten Schritt die Art jeder einzelnen Rekursion festgestellt werden. Mit dem Problem der Artbestimmung einer gegebenen rekursiv definierten Funktion beschäftigt sich Kapitel 3. Hierbei wird eine gegebene Funktion mit den Mustern verglichen, die für alle Funktionsformen gegeben sind, die wir verarbeiten können. Das Matchen der Muster, das in dieser Arbeit nicht in allen Einzelheiten behandelt werden soll, wird zu einer schwierigen Aufgabe, da die Zuordnung der Argumente einer konkreten Funktion bezüglich der abstrakten Muster der Transformationsschemata nicht eindeutig sein kann.

Sind nun alle potentiell für eine Verbesserung in Frage kommenden Funktionen registriert und ist ihre Art bestimmt, dann kann der eigentliche Transformationsprozeß beginnen.

In Kapitel 4 werden Transformationsschemata für drei verschiedene Rekursionsarten vorgestellt und anschließend mit Beispielen illustriert. Dieser Teil des Systems ist als erweiterbares Regelsystem vorgesehen. In Kapitel 5 wird die Korrektheit dieser Transformationsschemata bewiesen.

Da nicht alle denkbaren Rekursionsmuster ohne Schwierigkeiten in repetitive Formen aufgelöst werden können, wird in Kapitel 6 kurz eine heuristische Methode vorgestellt, die ursprünglich zur Programmsynthese verwendet wurde, deren Weiterentwicklung aber möglicherweise auch die Leistungsfähigkeit des vorgeschlagenen Regelsystems zur Effizienzsteigerung von Programmen erreichen könnte.

Offen bleibt die Implementierung des hier entworfenen Systems, für das nur kurz in der abschließenden Zusammenfassung ein Aufbauvorschlag gegeben wird.

2. Erkennung von Rekursionen

Eine Funktion F heißt rekursiv, wenn ihre Definition den Funktionsnamen F direkt oder indirekt enthält.
Eine Funktion F, die in ihrer Definition ihren eigenen Namen enthält, heißt direkt oder explizit rekursiv.
Eine Funktion F, die andere Funktionsnamen in ihrer Definition enthält, in deren Definitionen der Funktionsname F direkt oder indirekt auftritt, heißt indirekt oder implizit rekursiv.
Eine Funktion, die nicht rekursiv ist, heißt einfach.

Rekursive Funktionsdefinitionen können auf verschiedene Art und Weise beschrieben werden.
Z.B. als

```
Funktion F if   Bedingung
           then Operation 1
           else Operation 2
           fi
```

wobei Operation 1 und/oder Operation 2 wiederum if-then-else - Konstrukte enthalten können und einen oder mehrere Aufrufe der Funktion F,
oder als

```
Funktion F case Bedingung 1      Aktion 1
                Bedingung 2      Aktion 2
                ....................
                Bedingung n-1    Aktion n-1
                                 Aktion n
```

wobei Aktion 1 ... Aktion n wiederum case - Konstrukte enthalten können und einen oder mehrere Aufrufe der Funktion F,

oder als eine Mischform aus den beiden oben beschriebenen Mustern.

Der Übersichtlichkeit halber und aus programmtechnischen Gründen wollen wir von einer einheitlichen Darstellung ausgehen, und wählen daher willkürlich für die weiteren Ausführungen die ALGOL-ähnliche if-then-else - Notation. Wichtig ist diese Einschränkung auch für die Notwendigkeit, die Regelbasis möglichst klein zu halten, denn ein Kriterium für die Nützlichkeit eines Systems zur Elimination von Rekursionen sollte sein, daß dieser Transformationsprozeß nicht mehr Zeit erfordert als ein Programmierer für das Umschreiben seiner Programme in eine effizientere Fassung per Hand brauchen würde.

Wir gehen davon aus, daß die Eingabe für unser System ein Programm ist, das in der internen Darstellung der von Beierle und Voss [BeVo 83] entwickelten ASPIK-Sprache geschrieben ist. Da in dieser Sprache nur case - und if-then-else - Konstrukte zur Beschreibung rekursiver Funktionsdefinitionen erlaubt sind, beschränken wir uns auf die oben angegebenen Darstellungsmöglichkeiten. Hierzu ist zu bemerken, daß jedes einfache oder geschachtelte case -Konstrukt in ein entsprechendes einfaches oder geschachteltes if-then-else - Konstrukt überführt werden kann. Damit wird eine gesonderte Betrachtung und zusätzliche Regelbasis für case - Muster umgangen. Ein entsprechender Umformungsalgorithmus [UMFORM1] muß dann dem eigentlichen Transformationsprogramm vorangestellt werden.

Die Erkennung expliziter oder impliziter Rekursivität kann als graphentheoretisches Problem behandelt werden. Eine detaillierte Untersuchung würde jedoch den Rahmen dieser Arbeit sprengen. Unser Systementwurf unterliegt daher der Einschränkung, daß nur explizit rekursive Funktionen verarbeitet werden.

Die Erkennung auflösbarer Rekursionen kann in diesem Fall wie folgt ablaufen:

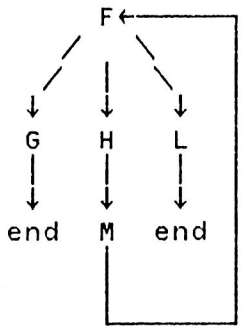
a) Durchsuchen der Funktionsdefinition nach erneutem Auftreten des Funktionsnamens.
   Falls der Funktionsname ein- oder mehrfach in der Definition auftritt, dann müßte noch untersucht werden, ob die Definition auch implizit rekursiv ist (siehe (b)). Falls die Definition nicht implizit rekursiv ist, übergeben wir sie an ein noch zu beschreibendes Umformungsprogramm, das die ursprüngliche Funktionsdefinition in eine Standardform überführen soll.

b) Tritt der Funktionsname in der Definition selber nicht auf, so ist die Funktion entweder nicht rekursiv oder implizit rekursiv.
   Eine entsprechende Entscheidung kann auch hier nur getroffen werden, wenn alle weiteren in der Definition vorkommenden Funktionen untersucht werden. Aufgrund der Spezifikationshierarchie können in ASPIK implizite Rekursionen nur innerhalb einer Spezifikation auftreten. Es brauchen daher nur die entsprechenden Funktionen betrachtet zu werden.

Zwei Beispiele, die mögliche Schwierigkeiten bei der Erkennung von Rekursionen und bei der Einschränkung auf von unserem System verarbeitbare Funktionen andeuten sollen:

1. F = ...(F....).... explizit rekursiv

2. F = ...(...G...H...L)... nicht explizit rekursiv
   Untersuchung von G,H und L
   a) G ist eine einfache Funktion
   b) H = ...(...G...M...)... M muß untersucht werden
      M = ...(...G...F...)... implizite Rekursion von F
   c) L ist eine einfache Funktion

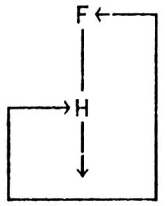


graphentheoretische Lösung

Zyklus über mehrere Stufen

3. F = ...(...H...)...
   H = ...(...F...)... implizit rekursiv

   Dieses Beispiel stellt einen wechselseitigen Aufruf dar, dessen Termination nur datenabhängig untersucht werden kann.



graphentheoretische Lösung

Zyklus über eine Stufe

   Falls H selbst rekursiv ist und der rekursive Aufruf F als Argument enthält, ist dieses Problem für unser System nicht lösbar, sonst wird die Definition von H in F 'hineingezogen', d.h. an der Stelle eingesetzt, an der H auftritt.

Es soll an dieser Stelle nochmals darauf hingewiesen werden, daß nicht die Erkennung solcher 'Verschachtelungen' die Schwierigkeit ist, sondern die Auflösung, da semantische Informationen berücksichtigt werden müssen.

Gehen wir aber zunächst davon aus, daß wir nur direkt rekursive Funktionen in if-then-else - Darstellung vorliegen haben.

Wir haben also: Funktion F
```
if   Bed
then Op1
else Op2
fi
```

Nehmen wir an, daß Op1 und Op2 keine weiteren if-then-else - Konstrukte enthalten. Dann können direkte Rekursionen in Op1 und in Op2 auftreten.

Enthält sowohl Op1 als auch Op2 einen rekursiven Aufruf, so muß eine Fehlermeldung erfolgen, da die Berechnung dieser Funktion nicht terminiert.

Enthält nur Op1 einen rekursiven Aufruf, so kann die Funktion an den Programmteil des Systems übergeben werden, der die Art der Rekursion bestimmt [REKART].

Ist nur Op2 rekursiv, so wird die Funktion an ein zusätzliches Umformungsprogramm [UMFORM2] übergeben, das durch Negation der Bedingung die Voraussetzung dafür schafft, daß der Rekursionsteil nach vorne gezogen wird, d.h. Op1 und Op2 vertauscht werden. Diese zweite Umformung ist notwendig, um die Eingabe für den Systemteil, der die Art der Rekursion bestimmt, zu vereinfachen. Dabei ist zu berücksichtigen, ob die Auswertung der Bedingung Seiteneffekte verursacht. Dies ist z.B. in LISP ein Problem bei der Berechnung logischer Ausdrücke, die nur soweit berechnet werden bis der Wert festliegt. Es ist in der Konzeption dieser Sprachen vorgesehen, daß Unterausdrücke gegebenenfalls nicht berechnet werden. Bei einer Umkehrung der Bedingung ist ein entsprechendes Verhalten nicht sichergestellt.

Sind Seiteneffekte ausgeschlossen, und ist eine Rekursion in die von uns gewählte Standardform überführt worden, so steht der rekursive Teil immer im ersten vorkommenden Operationsteil, und die Bedeutung der Funktion hat sich nicht geändert.

Da wir alle hier erwähnten Konstrukte ausschließlich auf syntaktischer Ebene betrachten, müssen wir zunächst die Bedingung, daß nur einer der Operationsteile rekursiv sein darf, ebenso für entsprechende Erweiterungen fordern. Bisher haben wir rekursive Funktionen mit zwei Operationsteilen betrachtet. Eine Erweiterung auf drei Operationsteile soll die vorläufige Einschränkung auf einen rekursiven Operationsteil begründen.

Dazu betrachte man das folgende Beispiel:

```
F(x) = if x=0
       then F(x+1)
       else if x=1
            then F(x-1)
            else x:=2
            fi
       fi
```

Rein syntaktisch betrachtet ist nicht entscheidbar, ob die Funktionsberechnung terminiert, da zumindest eine 'Abbruchzuweisung' existiert. Semantisch betrachtet ist jedoch offensichtlich, daß diese Funktion nie terminieren kann, wenn sie einmal rekursiv aufgerufen wurde, da der rekursive Aufruf im ersten then - Teil immer die Bedingung erzeugt, die den zweiten then - Teil als nächsten zur Ausführung kommen läßt, und der zweite then - Teil immer den Zustand herstellt, der den ersten then - Teil als nächsten zur Ausführung bringt, sodaß der 'Abbruch - else - Teil' nie erreicht wird.

Betrachtet man die Syntax der oben gegebenen Form mit genau einem else-if , dann können folgende Fälle auftreten:

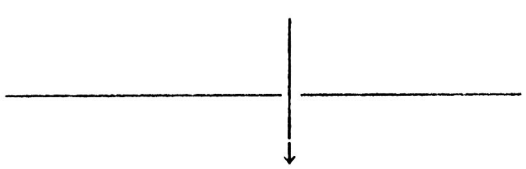
a) Nur einer der drei Operationsteile ist rekursiv. Dann läßt sich die Funktion auf jeden Fall so umstellen, daß diese Rekursion in den ersten Operationsteil geschoben wird, und anschließend die Funktion in Standardform an den Programmteil übergeben wird, der die Art der Rekursion bestimmt.

b) Alle drei Operationsteile sind rekursiv. Dann muß eine Fehlermeldung ausgegeben werden, da die Ausführung dieser Funktion auf jeden Fall in eine Endlosschleife führt, und das Programm somit nicht terminiert.

c) Zwei der drei Operationsteile sind rekursiv. Diese Form einer rekursiven Funktion kann unser System verarbeiten, wenn (i) beide Operationsteile identisch sind, und durch Kombination der Bedingungen zu einer einzigen Rekursion reduziert werden können, oder wenn (ii) durch Einführung einer Hilfsfunktion ein rekursiver Aufruf eingespart wird. Für die Einführung einer solchen Hilfsfunktion muß dem eigentlichen Transformationsprozeß ein weiteres Umformungsprogramm [UMFORM3] vorangestellt werden.

Beispiel zu (ii): Suchen in einem sortierten Baum

```
funct SEARCH(x, tree):
      if is-leaf(tree)
      then contents(tree)
      else if x ≤ contents(left(tree))
           then SEARCH(x, left(tree))
           else SEARCH(x, right(tree))
           fi
      fi
```



```
funct SEARCH(x, tree)
      if is-leaf(tree)
      then contents(tree)
      else SEARCH(x, h(x,tree))
      fi
      where
      funct h(x,tree):
            if x ≤ contents(left(tree))
            then left(tree)
            else right(tree)
            fi
```

Auch das Muster einer geschachtelten if-then-else - Form, in der genau ein then-if vorkommt, kann umgeformt werden:

```
Funktion F = if Bed.1                          if Bed.1 and Bed.2
             then if Bed.2                        then Op.1
                  then Op.1                       else if Bed.1
                  else Op.2        ===>                 then Op.2
                  fi                                    else Op.3
             else Op.3                                  fi
             fi                                    fi
```

Dies gilt wieder unter der Voraussetzung, daß keine ungewollten Seiteneffekte mit den Bedingungen verbunden sind.

Ausgedehnt werden kann diese Klasse von verarbeitbaren Funktionen auf eine beliebig tiefe if-then-else - Schachtelung, allerdings immer unter Berücksichtigung der oben beschriebenen Einschränkungen.

Unter diesen Bedingungen können die als explizit rekursiv erkannten und in die Standardform gebrachten Funktionen dem nächsten Teilprogramm unseres Systems [REKART] übergeben werden, das die Art der Rekursionen bestimmt.

3. Bestimmung der Art einer Rekursion

3.1 Allgemeines

Um einen Überblick darüber zu bekommen, wie die Umgebung eines rekursiven Aufrufs aussehen kann, werden zunächst die möglichen Rekursionsmuster zusammengestellt.

Dabei werden drei Grundformen unterschieden:

a) die lineare Rekursion
b) die geschachtelte Rekursion und
c) die kaskadenartige Rekursion.

Die Standardform einer repetitiven Funktion sieht wie folgt aus:

R(x) ≡ if B(x) then R(K(x)) else H(x) fi

wobei R der Name der rekursiven Funktion, x der Parameter, B die Bedingung ist und K und H einfache Funktionen sind.

Sie kann in einer applikativen Programmiersprache in ein nichtrekursives Programm übersetzt werden:

R(x) ≡ while B(x) do x:= K(x) od ; H(x)

Das erste Grundmuster bezeichnet die lineare Rekursion.

Sie hat die folgende Standardform:

L(x) ≡ if B(x) then phi(L(K(x)), E(x)) else H(x) fi

wobei L der Name der rekursiven Funktion, x der Parameter, B die Bedingung ist, und K, E, H und phi einfache Funktionen sind.

Daraus ist zu erkennen, daß eine lineare Rekursion in repetitiver Form ist, wenn phi die Identitätsfunktion , also der rekursive Aufruf die letzte Aktion im Ablauf der Berechnung eines Rekursionsschrittes ist.

Linear rekursiv ist auch die folgende Form:

L(x) ≡ if B(x) then H(x) else phi(L(K(x)), E(x)) fi

Hier muß das in Kapitel 2 erwähnte zweite Umformungsprogramm [UMFORM2] den rekursiven Teil nach vorne holen (unter Negation der Bedingung B(x) und Vertauschen der beiden Operationsteile), und somit bekommen wir die oben angegebene Standardform einer linear rekursiven Funktion.

Sieht die linear rekursive Rechenvorschrift wie folgt aus:

$$L(x) \equiv \underline{case} \quad \begin{array}{ll} W_1(x) & A_1(x) \\ W_2(x) & A_2(x) \\ \ldots\ldots\ldots\ldots & \\ W_n(x) & A_n(x) \\ & A_{n+1}(x) \end{array}$$

wobei $W_i(x)$ Werte sind, die die verschiedenen möglichen Fälle anzeigen, und $A_i(x)$ die entsprechenden Aktionen oder Anweisungen, und hat genau ein A_i die Form 'phi(L(K(x), E(x))', so muß zuerst das in Kapitel 2 erwähnte Umformungsprogramm [UMFORM1] durchlaufen werden, das case - Konstrukte in if-then-else - Schachtelungen überführt, und anschließend gegebenenfalls das zweite Umformungsprogramm [UMFORM2], das die Funktion in die Standardform bringt. Sind mehrere $A_i(x)$ rekursiv, so ist zwischen UMFORM1 und UMFORM2 das Vorprogramm UMFORM3 aufzurufen.

Im Folgenden wollen wir davon ausgehen, daß die notwendigen Umformungen bereits durchgeführt sind, und die Funktion daher in der if-then-else - Standardform vorliegt, da für die Artbestimmung nicht die Umgebung interessant ist, sondern lediglich der rekursive Operationsteil.

Das zweite Grundmuster ist das der geschachtelt rekursiven Funktionen.

Es sieht in einfacher Standardform mit der Schachtelungstiefe n=2 wie folgt aus:

F(x) ≡ if B(x) then F(F(f(x))) else g(x) fi

wobei F der Name der rekursiven Funktion, x der Parameter, B die Bedingung ist, und f und g einfache Funktionen sind.

Bemerkung: Im Folgenden soll die spitze Klammer > für alle an dieser Stelle noch zu schließenden runden Klammern stehen.

Modifikationen ergeben sich
a) bei größerer Schachtelungstiefe F(F(F(.....F(f(x>
b) beim Hinzufügen einer Funktion, die die geschachtelten rekursiven Funktionen als Argument enthält
phi(F(F(.....(f(x>
Diese Rekursion nennen wir linear geschachtelt.
c) beim Einfügen einfacher Funktionen zwischen den geschachtelten F(h(F(g(F(....F(f(x>
Diese Rekursion nennen wir verschränkt geschachtelt.
d) bei Kombination von (b) und (c)
phi(F(h(F(g(F....F(f(x>
Diese Rekursion nennen wir linear verschränkt geschachtelt. Insbesondere kann auch die unter (b) aufgeführte Form als linear verschränkt geschachtelt bezeichnet werden, wenn man für die fehlenden Zwischenfunktionen die Identitätsfunktion einsetzt.

Das dritte Grundmuster repräsentiert die kaskadenartige Rekursion.

```
C(x) ≡ if B(x)
       then phi(C(K1(x)), C(K2(x)), E(x))
       else H(x)
       fi
```

wobei C der Name der rekursiven Funktion, x der Parameter, B die Bedingung, phi eine einfache dreistellige (falls E(x)={} ==> zweistellige) Funktion ist, und K_1, K_2, E und H einfache Funktionen sind.

Auch hier ergibt sich eine Erweiterung, wenn mehr als zwei rekursive Aufrufe nebeneinander stehen.

Nicht betrachtet werden
a) Mischformen aus den oben beschriebenen Mustern, von denen allerdings auch anzunehmen ist, daß sie in der Praxis kaum vorkommen.
b) Zuweisungen vor der eigentlichen Rekursion an Variablen, die in der Rekursion wieder vorkommen, da diese Zuweisungen in die rekursive Definition 'hineingezogen' werden können (Vorsicht: Seiteneffekte!).
c) indirekte Rekursionen.

3.2 Erkennungsregeln

Gehen wir aber davon aus, daß auf den Teil der Funktion, der die Rekursion enthält, zugegriffen werden kann, dann können mit Hilfe des im folgenden skizzierten Regelsystems die verschiedenen Rekursionsarten unterschieden und bestimmt werden.

Die Regeln sind wie folgt aufgebaut:

```
Name der Regel      Bedingung 1
                    .........
                    Bedingung n

                ==> Aktion 1
                    ......
                    Aktion m
```

wobei der Aktionsteil hinter dem Symbol '==>' nur dann ausgeführt wird, wenn alle Voraussetzungen des Bedingungsteils erfüllt sind.

Der erste Schritt trennt repetitive und lineare von geschachtelten und kaskadenartigen Funktionen. Es werden folgende Regeln gebraucht, wobei die Kennzeichnung M angibt, daß es sich um Matchregeln (Erkennungsregeln) handelt. Das Punktpaar a.b bezeichnet die Nummer der Regel im geordneten Ableitungsbaum.

M 1.1 der Funktionsname tritt in der Rekursion genau einmal auf

==> einfache Rekursion

M 1.2 der Funktionsname tritt in der Rekursion mehrfach auf

==> mehrfache Rekursion

Verfolgen wir nun zunächst die einfache Rekursion weiter.

Die repetitive Form unterscheidet sich von der linearen dadurch, daß sie in 'reiner' Form auftritt, während bei der letzteren weitere Funktionen auf den rekursiven Aufruf angewendet werden. Die Unterscheidung erfordert hier also folgende Regeln:

M 2.1 einfach rekursiv
das erste Element des rekursiven Operationsteils ist der Funktionsname der zu untersuchenden Funktion (rekursiver Aufruf)

==> repetitiv
Art bestimmt

M 2.2 einfach rekursiv
¬(Art bestimmt)
das erste Element des rekursiven Operationsteils ist der Name einer Funktion, die den rekursiven Funktionsaufruf als ein Argument enthält

==> linear
Art bestimmt

Wird der Aktionsteil von Regel M 2.1 ausgeführt, d.h. die Eigenschaft 'repetitiv' an den Namen der zu untersuchenden Funktion gebunden, so braucht nicht transformiert zu werden, da bereits eine repetitive Form vorliegt.
Wird der Aktionsteil von Regel M 2.2 ausgeführt, d.h. die Eigenschaft 'linear' an den Namen der zu untersuchenden Funktion gebunden, so wird die Funktion mit der neugewonnenen Information an das Transformationsprogramm [TRANSFORM] weitergegeben.

Bemerkung:
Nicht berücksichtigt worden ist hier der Fall, daß vor dem rekursiven Aufruf eine Funktion oder Zuweisung steht, die sich auf eine Variable bezieht, die auch innerhalb des rekursiven Aufrufs auftritt. Eine solche Funktion oder Zuweisung wäre auch nur sinnvoll, wenn sie absichtlich Seiteneffekte produziert. Derartige 'Pseudofunktionen' werden daher nicht in Betracht gezogen, können auch bei funktionalen Programmiersprachen nicht auftreten.

Damit ist die Artbestimmung für einfache Rekursionen erfaßt, und es müssen noch Unterscheidungen bezüglich der Arten der hier mit 'mehrfach' bezeichneten Rekursionen getroffen werden.

Falls die nächsten Regeln zur Anwendung kommen, steht also fest, daß der Name der rekursiven Funktion in der Definition mindestens zweimal auftritt.
Zunächst wird der rekursive Operationsteil nach dem ersten Auftreten des entsprechenden Funktionsnamens durchsucht. Direkt hinter diesem Auftreten wird mit dem Zählen der öffnenden und schließenden Klammern begonnen, um ein Kriterium zu haben, nach dem geschachtelte und kaskadenartige Rekursionen zu unterscheiden sind. Tritt nämlich der nächste rekursive Funktionsaufruf auf, bevor zum ersten Mal die Zahl der schließenden Klammern gleich der Zahl der öffnenden ist, dann bettet der erste Funktionsaufruf den zweiten ein, und es handelt sich um eine geschachtelte Rekursion. In allen anderen Fällen ist der Argumentbereich des ersten Funktionsaufrufs abgeschlossen, bevor der nächste rekursive Aufruf erfolgt, d.h. die Rekursionen sind parallel bzw. kaskadenartig.

Berücksichtigt werden muß noch die Schachtelungstiefe (ST) bzw. die Anzahl paralleler rekursiver Aufrufe (FAKTOR). Da wir jedoch davon ausgehen, daß der Funktionsname in den seltensten Fällen öfter als zweimal in der Definition auftritt, verteilen wir die Unterscheidung auf vier Regeln, wobei die ersten beiden das zweimalige Auftreten behandeln und die letzten beiden das n-fache (mit n>2). Der Vorteil dieser Unterteilung ist, daß in den meisten Fällen eine der ersten beiden Regeln zur Anwendung kommt, und somit die Rechenzeit nicht unnötig verlängert wird, da bei zweifachem Auftreten des rekursiven Funktionsnamens der rekursive Operationsteil nicht rekursiv durchsucht werden muß.

M 3.1 mehrfach rekursiv
der rekursive Funktionsname tritt in der Definition genau zweimal auf
der zweite rekursive Funktionsaufruf tritt innerhalb des Argumentbereichs des ersten rekursiven Funktionsaufrufs auf

==> geschachtelt
Schachtelungstiefe ST = 2
Art bestimmt

M 3.3 mehrfach rekursiv
der rekursive Funktionsname tritt in der Definition genau zweimal auf
der zweite rekursive Funktionsaufruf tritt unabhängig (parallel) zum ersten rekursiven Funktionsaufruf, d.h. außerhalb des Argumentbereichs des ersten, auf

==> kaskadenartig
FAKTOR F = 2
Art bestimmt

M 3.2 mehrfach rekursiv
der rekursive Funktionsname tritt n-mal in der Definition auf (n>2)
der nächste rekursive Funktionsaufruf tritt jeweils im Argumentbereich des vorhergehenden auf

==> geschachtelt
Schachtelungstiefe ST = n
Art bestimmt

M 3.4 mehrfach rekursiv
der rekursive Funktionsname tritt n-mal in der Definition auf (n>2)
der nächste rekursive Funktionsaufruf tritt jeweils außerhalb des Argumentbereichs des vorhergehenden auf

==> kaskadenartig
FAKTOR F = n
Art bestimmt

Bei der Betrachtung der von Bauer und Wössner angegebenen Sonderfälle für geschachtelte Rekursionsmuster fällt auf, daß die Eigenschaft 'geschachtelt', die beim Matchen bisher erkannt werden kann, nicht ausreicht. Hinzugefügt werden müssen Regeln für eine genauere Bestimmung der Art einer Schachtelung.
Die Arten der Schachtelung unterscheiden sich nach den in Kapitel 3.1 bereits beschriebenen Modifikationen (b) - (d).
Die erste Regel muß daher prüfen, ob eine Verschränkung gegeben ist, die zweite, ob eine lineare Schachtelung vorliegt. Sollte weder die eine noch die andere Regel zur Anwendung kommen, so bleibt der Initialwert 'geschachtelt (einfach)' bestehen.

M 4.1 geschachtelt
∃ einfache Funktion h, sodaß mindestens ein rekursiver Aufruf im Argumentbereich von h liegt

==> verschränkt

M 4.2 geschachtelt
∃ einfache Funktion phi, sodaß der erste rekursive Aufruf im Argumentbereich von phi liegt

==> linear

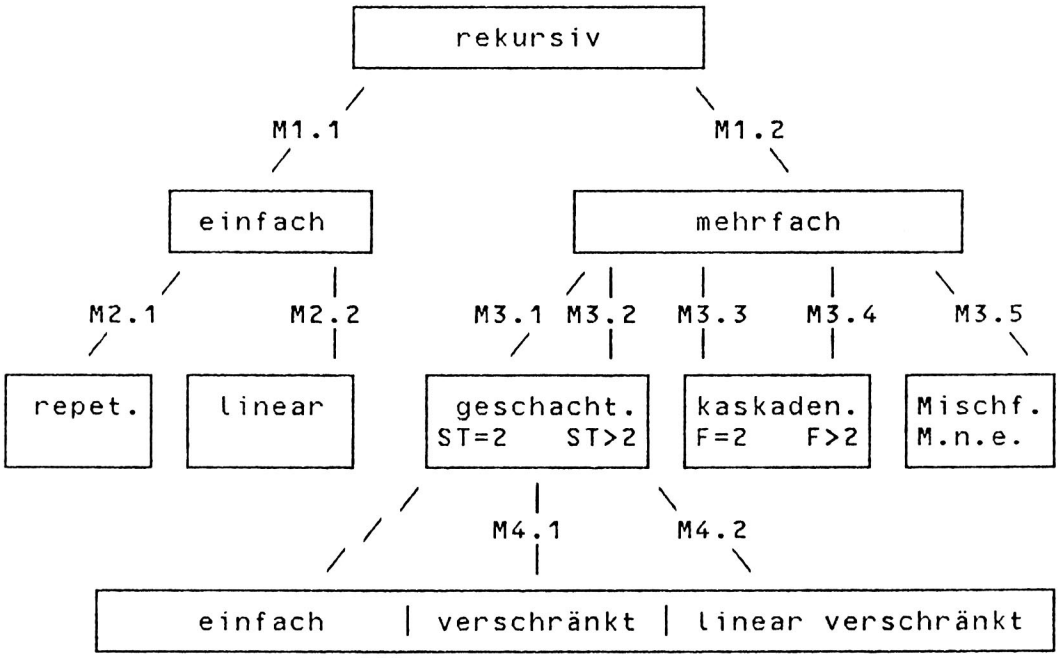
Sind nur die Bedingungen von Regel M 4.1 erfüllt, so heißt die Funktion 'verschränkt geschachtelt'. Gelten die Voraussetzungen für beide Regeln, so wird die gegebene Funktion als 'linear verschränkt geschachtelte' Rekursion behandelt.

Bemerkung:
Unberücksichtigt bleibt hier die Möglichkeit einer Mischform. Keine der Regeln sieht vor, daß z.B. der zweite Funktionsaufruf im Argumentbereich des ersten, aber der dritte außerhalb des Argumentbereichs des zweiten liegt. Derartige Fälle dürften auch in der Praxis kaum vorkommen. Sollten sie dennoch auftreten, wird man für diese Spezialfälle auf individuelle Überlegungen angewiesen sein. Eine entsprechende Fehlermeldung kann den Benutzer auf eine solche Mischform und deren Nichtauflösbarkeit aufmerksam machen. Gehen wir wieder davon aus, daß die Matchregeln in der Reihenfolge ihrer Nummern abgeprüft werden, so genügt die Voraussetzung, daß die Funktion nicht einem der gegebenen Muster zugeordnet werden konnte.

M 3.5 ¬(Art bestimmt)

==> Muster nicht erkannt (M.n.e.)

Der Entscheidungsbaum für die hier beschriebene Aufgliederung der Rekursionen in fünf Klassen (wobei die geschachtelte Rekursion weiter unterteilt ist) sieht dann wie folgt aus:



Damit ist für jede zu untersuchende rekursive Funktionsdefinition die Art bestimmt worden. Gehört eine Funktion zu einer der mittleren drei Klassen, so kann sie an das eigentliche Transformationsprogramm übergeben werden, repetitive Formen brauchen nicht transformiert zu werden, Mischformen können von unserem System vorerst nicht behandelt werden.

Die Implementierung eines derartigen Matchprogramms ist wie schon in der Einführung erwähnt mit einigen Schwierigkeiten verbunden. Hier müssen Operationen und Muster 2. Ordnung (second-order pattern) gematcht werden. Ein algorithmischer Lösungsansatz ist bei Huet und Lang [HuLa 78] zu finden, und wurde für die Eingabe von ASPIK-Programmen beliebiger Stelligkeit von Gerlach [Gerl 83] modifiziert und implementiert.

3.3 Beispiele

Das allgemeine Schema für die lineare Rekursion sieht wie folgt aus:

```
funct L ≡ (λ x) λ:
      if B(x)
      then phi(K(x), E(x))
      else H(x)
      fi
```

Durch Konkretisierung von

L	≡	FAK	K(x)	≡	x-1
λ	≡	nat	E(x)	≡	x
x	≡	x	phi(x,y)	≡	x * y
B(x)	≡	x≠0	H(x)	≡	1

ergibt sich z.B. die Berechnung der Fakultät einer natürlichen Zahl x:

```
funct FAK ≡ (nat x) nat:
      if x≠0
      then x * FAK(x-1)
      else 1
      fi
```

Das allgemeine Schema für die geschachtelte Rekursion mit der Schachtelungstiefe n=2 sieht wie folgt aus:

```
funct F ≡ (λ x) λ:
      if B(x)
      then F(F(f(x)))
      else g(x)
      fi
```

Durch Konkretisierung von

F	≡	F91	B(x)	≡	x≤100
λ	≡	nat	f(x)	≡	x+11
x	≡	x	g(x)	≡	x-10

ergibt sich z.B. die Berechnung der '91-Funktion' von Manna :

```
funct F91 ≡ (nat x) nat:
      if x≤100
      then F91(F91(x+11)
      else x-10
      fi
```

Diese Funktion liefert für alle x≤101 den Wert 91, und für alle x>101 den Wert x-10 (nat = Menge der natürlichen Zahlen).

Das allgemeine Schema für die kaskadenartige Rekursion mit dem FAKTOR n=2 sieht wie folgt aus:

```
funct C ≡ (λ x) ρ:
      if B(x)
      then phi(C(K1(x)), C(K2(x)))
      else H(x)
      fi
```

Durch Konkretisierung von

C	≡	FIBO	$K_1(x)$	≡	x-2
λ	≡	pnat	$K_2(x)$	≡	x-1
x	≡	x	phi(x,y)	≡	x+y
B(x)	≡	x>2	H(x)	≡	1

ergibt sich z.B. die Berechnung der Fibonaccizahlen:

```
funct FIBO ≡ (pnat x) pnat:
      if x>2
      then FIBO(x-2) + FIBO(x-1)
      else 1
      fi
```

(pnat = Menge der natürlichen Zahlen ohne 0)

Anhand dieser drei Beispiele wird in Kapitel 4 die Anwendung des Regelsystems für die Transformationen verdeutlicht.

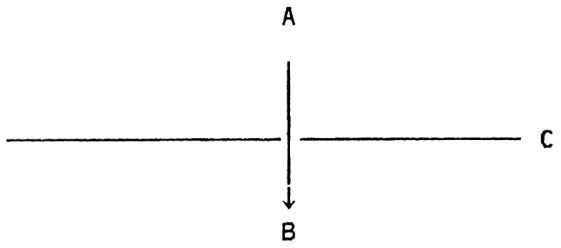
"...the transformation from recursion to iteration is one of the most fundamental concepts of computer science."

Knuth

4. Transformationsschemata und Regeln

Für die im vorhergehenden Kapitel als erkennbare rekursive Funktionen beschriebenen Muster wird versucht geeignete Transformationsregeln zu finden.
Aus Gründen der Übersichtlichkeit wird die Notation von Bauer und Wössner weiterhin beibehalten.

Ein Transformationsschema soll wie folgt beschrieben werden:



wobei A und B Funktionsmuster sind und C die für eine Transformation hinreichenden Bedingungen darstellt (in Kapitel 5: Seitenbedingungen).

4.1 Transformationsschemata und Regeln für die lineare Rekursion

Das allgemeine lineare Rekursionsmuster sieht nach Bauer/Wössner wie folgt aus:

```
funct L ≡ (λ x)ρ:
      if B(x)
      then phi(L(K(x)), E(x))
      else H(x)
      fi
```

wobei folgende Stelligkeiten gelten:

```
K   : λ → λ
phi: ρ × ν → ρ
E   : λ → ν
H   : λ → ρ
```

Wir setzen an dieser Stelle voraus, daß der Benutzer vor Beginn des Transformationsprogramms gefragt wird, ob er eventuell irgendwelche Zusatzinformationen über die von ihm eingegebenen rekursiven Funktionen geben kann. Falls ja, wird er in den entsprechenden Fällen gefragt, falls nein, sind wir gezwungen uns auf programminterne (z.B. in Assoziationslisten festgehaltene) Informationen zu stützen.

Zur Auflösung derartiger linearer Rekursionsschemata gibt es nach Bauer/Wössner verschiedene Verfahren:

a) Klammernverschiebung (oder Um-Klammerung)
b) Operandenvertauschung
c) Funktionsumkehr
d) Transformation von Paterson und Hewitt
e) Funktionsumkehr unter Einführung von Stapeln.

Die ersten drei Methoden, die auf Cooper [Coop 66] zurückgehen, unterliegen bestimmten Bedingungen, die eine aufzulösende Funktion erfüllen muß. Die vierte Methode soll hier nicht betrachtet werden, da sie zu uneffizienten Abläufen führt (siehe [BaWö 81], Kapitel 4.2.4) und die letzte Methode ist grundsätzlich und ohne einschränkende Bedingungen anwendbar, wobei zu berücksichtigen ist, daß es sich hierbei nur um eine Verschiebung des eigentlichen Problems von der Programm- auf die Datenstruktur handelt. Eine Hierarchie der Regelbasis könnte hier gewährleisten, daß diese letzte Methode nur dann zur Anwendung kommt, wenn keine der anderen Auflösungen möglich ist.

Bei der Betrachtung aller von Bauer und Wössner beschriebenen linearen Schemata scheint die erste Vereinfachung darin zu liegen, die Funktion phi auf Einstelligkeit zu überprüfen. Und zwar aus folgenden Gründen:
Wenn phi einstellig ist, dann wissen wir, daß phi auch rechtskommutativ ist, und somit durch Operandenvertauschung aufgelöst werden kann (siehe Kapitel 4.1.2) , wobei die Überprüfung, ob eine Um-Klammerung möglich ist, übersprungen werden kann.

Daraus ergibt sich die erste Regel:

Regel1 E(x) = { }

==> phi einstellig
(phi rechtskommutativ)

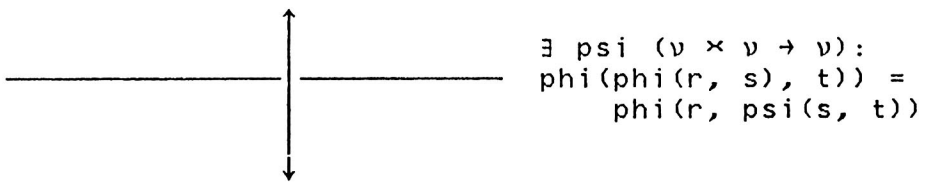
d.h. die Seitenbedingung bei der Operandenvertauschung braucht nicht mehr abgefragt zu werden, da sie trivialerweise nur Tautologien liefert (siehe Kapitel 4.1.2).

Anmerkung:
Bei einem Regelsystem sollte im Grunde gewährleistet sein, daß unabhängig von der Reihenfolge der Regeln eine Abarbeitung erfolgen kann. Da jedoch für die Elimination konkreter linear rekursiver Funktionen oft mehrere Möglichkeiten existieren, die von unterschiedlichen Bedingungen abhängig sind, wird zunächst willkürlich eine Reihenfolge gewählt. Ein Effizienzvergleich könnte später dazu führen diese Reihenfolge zu ändern (eventuell über ein entsprechendes Metaregelsystem - siehe Zusammenfassung, Kapitel 7).

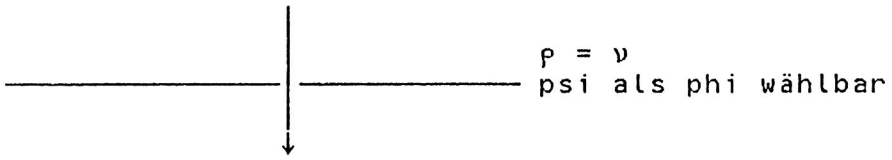
4.1.1 Um-Klammerung

Die von Bauer und Wössner gegebene Transformation mit Hilfe dieser Methode sieht wie folgt aus:

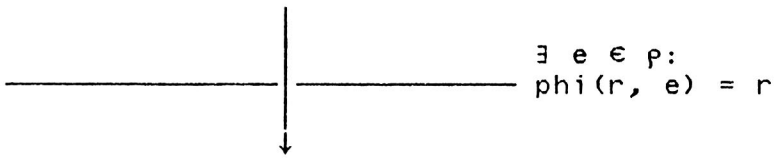
```
funct L ≡ (λ x)ρ:
      if B(x)
      then phi(L(K(x)), E(x))
      else H(x)
      fi
```




```
funct L ≡ (λ x)ρ:
      if B(x) then G(K(x),E(x)) else H(x) fi
      where
      funct G ≡ (λ x, ν z)ρ:                                    M1
            if B(x)
            then G(K(x), psi(E(x),z))
            else phi(H(x), z)
            fi
```




```
funct L ≡ (λ x)ρ:
      if B(x) then G(K(x), E(x)) else H(x) fi
      where
      funct G ≡ (λ x, ρ z)ρ:
            if B(x)
            then G(K(x), phi(E(x), z)
            else phi(H(x), z)
            fi
```




```
funct L ≡ (λ x)ρ:
      G(x,e)
      where
      funct G ≡ (λ x, ρ z)ρ:
            if B(x)
            then G(K(x), phi(E(x), z)
            else phi(H(x), z)
            fi
```

Dazu ist folgendes zu bemerken:
Es ist nicht einzusehen, warum die Assoziativität von phi zu einem Zwischenmuster führen soll, da dieses keine Vorteile gegenüber Muster M1 bietet.
Falls es möglich ist, psi = phi zu wählen (wobei auch zu prüfen ist, ob ρ = ν gilt), würde ein Benutzer auf Anfrage des Systems, ob es überhaupt ein psi gibt, das die Bedingung 'phi(phi(r, s), t) = phi(r, psi(s, t))' erfüllt, dieses sicher angeben, bzw. würde dies eventuell aus einer entsprechenden Tabelle entnommen werden können. Daher übergehen wir dieses Zwischenmuster und ziehen die Bedingungen der Assoziativität und der Existenz eines neutralen Elementes zusammen. Auch die Frage nach der Existenz eines neutralen Elementes erfordert eine Zusatzinformation, was in den Regeln mit call-user-or-tablesearch (CUOTS) angedeutet wird.

Es ergeben sich also für die Um-Klammerung folgende Regeln:

```
Regel2      ¬(phi einstellig)
            ∃ psi: ν × ν → ν                                  CUOTS
            phi(phi(r, s) t) = phi(r, psi(s, t))

      ==>   M1
            Schema aufgelöst
            Um-Klammerung
```

```
Regel3      ¬(phi einstellig)
            Um-Klammerung
            ρ = ν                                             CUOTS
            psi als phi wählbar
            phi(phi(r, s), t) = phi(r, phi(s, t))
            ∃ e ∈ ρ: phi(r e) = r

      ==>   ersetze in M1 die erste if-then-else - Clause
            durch G(x,e)
```

Damit ist die Um-Klammerung abgeschlossen und es kann zur nächsten Methode, der Operandenvertauschung übergegangen werden.

4.1.2 Operandenvertauschung

Die Methode der Operandenvertauschung beruht auf der Eigenschaft der (verallgemeinerten) Rechtskommutativität der Funktion phi, die die Ausführungsreihenfolge der Operationen umkehrt und damit 'hängende' Operationen beseitigt.

Eine Funktion phi heißt in diesem Zusammenhang rechtskommutativ, wenn gilt:

$$phi(phi(r, s), t) = phi(phi(r, t), s).$$

Allgemein soll auch eine Funktion $phi: \rho \times \nu \to \rho$ rechtskommutativ genannt werden, wenn eine weitere Funktion $psi: \rho \times \nu \to \rho$ existiert, sodaß gilt:

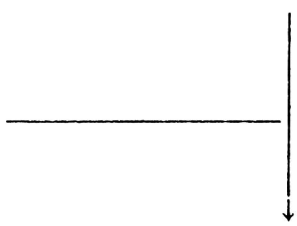
$$phi(psi(r, s), t) = psi(phi(r, t), s).$$

Im Gegensatz zur Um-Klammerung gehen die Vereinfachungen des Grundmusters bei der Operandenvertauschung nicht sukzessive auseinander hervor, sondern werden unter alternativen Bedingungen erstellt. Daher wird eine Regel gebraucht, die das Grundmuster (M2) erstellt, und weitere Regeln, die die entsprechenden 'Lücken', gekennzeichnet mit *n, wobei n eine laufende Nummer ist, füllen.

Zunächst muß jedoch abgeprüft werden, ob phi als einstellig erkannt wurde (Regel1), da dann die Funktion psi nicht vom Benutzer erfragt werden muß.

Die Überführung des linearen Rekursionsmusters in eine repetitive Form mit Hilfe der Operandenvertauschung sieht dann wie folgt aus:

```
funct L ≡ (λ x)ρ:
      if B(x)
      then phi(L(K(x), E(x)))
      else H(x)
      fi
```



$\exists$ psi: $\rho \times \nu \to \rho$
phi(psi(r, s), t) =
    psi(phi(r, t), s)
psi(H(n, r)) = phi(H(n, r))
für $\forall$ n $\in \{K^i(x) \mid x \in \lambda$,
$i \in N_0$, $(\forall m=0,\ldots,i-1\ B(K^m(x)))$,
$\neg[B(K^i(x))]\}$

```
funct L ≡ (λ x)ρ:
      G(x,F)
      where *1                                              M2
      funct G ≡ (λ x,ρ z)ρ:
            if B(x)
            then G(K(x), psi(z, E(x)))
            else z
            fi
```

Wobei ein Wert für F noch gefunden werden muß. Vereinfachungen sind möglich, wenn nicht umständlich in Abhängigkeit von x ein Wert für F berechnet werden muß.

Daraus ergeben sich für die Operandenvertauschung folgende Regeln:

```
Regel4      ¬(Schema aufgelöst)
            phi einstellig

      ==>   psi := phi
            rechtskommutativ
            psi generiert
```

```
Regel5      ¬(Schema aufgelöst)
            ¬(phi einstellig)
            ∃ psi: ρ × ν → ρ                                CUOTS
            phi(psi(r, s), t) = psi(phi(r, t), s)

      ==>   psi := user-input
            rechtskommutativ
            psi generiert
```

```
Regel6      ¬(Schema aufgelöst)
            rechtskommutativ
            psi generiert

      ==>   M2
```

```
Regel7      ¬(Schema aufgelöst)
            rechtskommutativ
            psi generiert
            H ≡ const.

      ==>   ersetze F durch H(x) = const.
            lösche *1
            Schema aufgelöst
```

Regel8 ¬(Schema aufgelöst)
rechtskommutativ
psi generiert
$(B(x) = (x \neq x_1))$ CUOTS

==> ersetze F durch $H(x_1)$
lösche *1
Schema aufgelöst

Regel9 ¬(Schema aufgelöst)
rechtskommutativ
psi generiert

==> ersetze F durch F(x)
ersetze *1 durch
funct F ≡ (λ x)p:
if B(x) then F(K(x)) else H(x) fi
Schema aufgelöst

Um sicherzustellen, daß Regel 9 nicht vor Regel 7 und 8 abgeprüft wird, was eine eventuell mögliche Vereinfachung verhindern könnte, müßte eine weitere Variable (Eigenschaft oder Schalter) eingeführt werden. Da jedoch davon ausgegangen wird, daß die Regeln in der aufgeführten Reihenfolge abgeprüft werden, wird auf diese Erweiterung an dieser Stelle verzichtet.

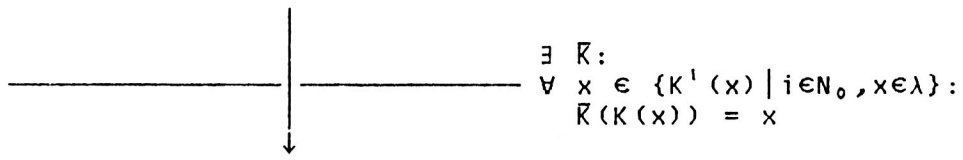
Damit ist auch die Operandenvertauschung abgeschlossen, und es kann nun die dritte Methode zur Auflösung linear rekursiver Funktionen , die Funktionsumkehr, betrachtet werden.

4.1.3 Funktionsumkehr

Die Funktionsumkehr ist ähnlich zu behandeln wie die Operandenvertauschung. Hierbei werden die 'hängenden' Operationen (siehe Kapitel 4.1.2) beseitigt, indem man von einem vorberechneten Endwert ausgeht, mit dessen Hilfe die Parameterwerte rekonstruiert werden bis zurück zum Anfangswert.
Unter der Bedingung, daß eine Umkehrfunktion $\overline{K}$ existiert, transformiert man zunächst in das Grundmuster M3. Falls die Bedingung B(x) von der Form $x \neq x_1$ ist, braucht $\lambda\, x_0$ nicht berechnet zu werden. Sonst wird die Berechnungsfunktion P eingesetzt.

Die Transformation sieht dann wie folgt aus:

```
funct L ≡ (λ x)ρ:
      if B(x) then phi(L(K(x)), E(x)) else H(x) fi
```




```
funct L ≡ (λ x)ρ:
      R(x0, H(x0), x)
      where *1                                              M3
      funct R ≡ (λ y, ρ z, λ x)ρ:
            if (y ≠ x)
            then R(K̄(y), phi(z, E(K̄(y))), x)
            else z
            fi
```

Es werden also die folgenden Regeln gebraucht:

```
Regel10     ¬(Schema aufgelöst)
            ¬(phi einstellig)
            ∃ K̄: ∀ x ∈ {K'(x) | i∈N0, x∈λ}: K̄(K(x))=x        CUOTS

      ==>   M3
            Umkehrfunktion

Regel11     Umkehrfunktion
            B(x) = (x ≠ x1)                                  CUOTS

      ==>   lösche *1
            ersetze x0 durch x1
            Schema aufgelöst

Regel12     Umkehrfunktion
            ¬(Schema aufgelöst)

      ==>   ersetze *1 durch
                λ x0 ≡ P(x)
                where
                funct P ≡ (λ x)λ:
                    if B(n) then P(K(n)) else n fi
```

Wieder wird davon ausgegangen, daß die Regeln in der hier aufgeführten Reihenfolge abgeprüft werden.

Damit sind alle Regeln für die Funktionsumkehr ohne Verwendung eines stacks aufgestellt, und es kann nun zur vierten, immer auf linear rekursive Funktionen anwendbare Auflösungsmethode übergegangen werden, der Funktionsumkehr unter Verwendung eines stacks.

4.1.4 Funktionsumkehr unter Verwendung eines stacks

Der Trick dieser allgemeingültigen Auflösungsmethode für lineare Rekursionsmuster liegt darin, die Funktion L so zu erweitern, daß sie selbst nicht mehr rekursiv ist, sondern das Problem auf die lokale Funktion L* verlagert wird, von der jedoch bekannt ist, daß sie durch Funktionsumkehr auflösbar ist (Eigenschaft der stack-Funktionen).

Es gibt nun zwei Möglichkeiten für die Behandlung dieser Methode:

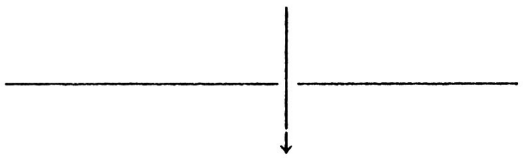
a) man transformiert die Funktion in das von Bauer und Wössner [BaWö 81] angegebene Schema,
b) man nutzt die bereits vorhandenen Regeln zur Funktionsumkehr und übergibt praktisch die Funktion L* an die entsprechenden Regeln.

Die zweite Möglichkeit scheint eleganter, hat jedoch auch Nachteile. Die Funktion L wird dabei zunächst in ein Zwischenmuster transformiert, und in dieser Regel muß entweder ein Hinweis gegeben werden darüber, daß die Regeln zur Funktionsumkehr nunmehr wieder zu den noch möglichen - oder besser gesagt zu den eventuell noch anwendbaren - Regeln gehören, obwohl eine Funktionsumkehr vorher als nicht möglich angenommen wurde (dies wäre eventuell ein Eingriff in eine Metaregelkomponente, die die Liste der noch möglichen Regeln verwaltet). Dann könnte L* durch Funktionsumkehr aufgelöst und an der entsprechenden Stelle in das Zwischenmuster eingesetzt werden. Oder, was einfacher erscheint, L* wird in eine Liste eingetragen, die noch zu bearbeitende Funktionen enthält. Hierbei kann ein erneuter Matchvorgang umgangen werden, indem an dieser Stelle die Eigenschaft 'rekursiv' mit dem Wert 'linear' an den Funktionsnamen gebunden wird.

Löst man jedoch die Funktion L nach Bauer/Wössner direkt auf (Fall a, Muster M4), ohne L* gesondert zu betrachten, so entsteht ein einfacheres Muster, in das mit nur einer Regel transformiert werden kann (im Gegensatz zur späteren Einsetzung der aufgelösten Form von L*).

Die Transformation in das einfache Muster sieht wie folgt aus:

```
funct L ≡ (λ x)ρ:
      if B(x)
      then phi(L(K(x)), E(x))
      else H(x)
      fi
```



```
funct L ≡ (λ x)ρ:
      R(P(x, empty))
      where
      funct P ≡ (λ x, stack λ sx) (stack λ, ρ):
            if B(x)
            then P(K(x), sx & x)
            else (sx, H(x))
            fi                                          M4
      funct R ≡ (stack λ sy, ρ z)ρ:
            if sy ≠ empty
            then R(rest sy, phi(z, E(top sy)))
            else z
            fi
```

wobei der Datentyp stack wie folgt definiert ist:
(nach Bauer/Wössner Kapitel 3.2.5.1 und 3.6.2.1)

```
type STACK ≡ (mode λ) stack λ:
    mode empty ≡ atomic { }
    mode stack λ ≡ empty | {λ item, stack λ trunk}
    funct empty ≡ stack λ:{ }
    funct isempty ≡ ( stack λ s) bool : empty :: s
    funct top ≡ ( stack λ s : ¬isempty(s)) λ: item of s
    funct rest ≡ ( stack λ s : ¬isempty(s)) stack λ : trunk of s
```

Die der oben gegebenen Transformation entsprechende Regel ist:

```
Regel13     ¬(Schema aufgelöst)
            ¬(phi einstellig)

      ==>   M4
            Schema aufgelöst
```

Damit sind die Betrachtungen zur Auflösung des linearen Rekursionsmusters abgeschlossen, und wir gehen zur Auflösung geschachtelt rekursiver Muster über.

"A lecture without stack
is like a day without sunshine."

Guttag

4.2 Transformationsschemata und Regeln für die geschachtelte Rekursion

Ausgegangen wird von dem einfachsten Fall einer geschachtelten Rekursion, für den grundsätzlich eine Auflösung in eine repetitive Form möglich ist, und der wie folgt aussieht:

```
funct F ≡ (λ x)λ:
      if B(x) then F(F(f(x))) else g(x) fi
```

Es gibt nun verschiedene Möglichkeiten, dieses Muster aufzulösen, wobei allein Gesichtspunkte der Effizienz von Bedeutung sind, um verschiedene Fälle zu unterscheiden.

Drei der zu betrachtenden Auflösungen führen direkt zu repetitiven Formen, eine bildet zunächst in ein lineares Rekursionsmuster ab.

Wichtig ist eine Reihenfolge für die Prüfungen, ob eine der Transformationen möglich ist, zu finden, um möglichst in das effizienteste Muster zu transformieren.
Bauer und Wössner beginnen mit dem umständlichsten und enden mit dem effektivsten Muster. Bei der Aufstellung entsprechender Regeln muß also diese Reihenfolge umgekehrt werden.

Die transformierten Muster sehen wie folgt aus:

A Grundauflösung ohne Nebenbedingungen (jedoch am uneffizientesten):

```
funct F ≡ (λ x)λ:
      Q(x,1)
      where
      funct Q ≡ (λ x, nat i)λ:
            if i≠0                                          M8
            then Q( if B(x)
                    then (f(x), i+1)
                    else (g(x), i-1)
                    fi )
            else x
            fi
```

B Gilt die Eigenschaft O: $\forall x \in \lambda$: {B(g(x)) => B(x)}, dann kann das einfach geschachtelte Muster wie folgt aufgelöst werden:

```
funct F ≡ (λ x)λ:
      if B(x) then g(F(f(x))) else g(x) fi                    M7
```

Dieses Schema ist zwar linear rekursiv, es ist aber sofort klar, daß es auflösbar ist, da es die Eigenschaft 'phi einstellig' besitzt, somit implizit rechtskommutativ ist, und mit Hilfe der Operandenvertauschung transformiert werden kann (siehe Kapitel 4.1.2).

C Gilt über die Eigenschaft O hinaus die Eigenschaft 1: $\forall x \in \lambda$: g(f(x)) = f(g(x)), d.h. f und g kommutieren, dann kann unmittelbar in das folgende Muster umgewandelt werden:

```
funct F ≡ (λ x)λ:
      K(x,x)
      where                                                   M6
      funct K ≡ (λ x, λ z)λ:
            if B(x)
            then K(f(x), g(f(z)))
            else g(z)
            fi
```

D Gelten die folgenden Bedingungen:

1: (s.o.)
2: $\forall x \in \lambda$: B(g(f(x))) => B(x)
3: $\forall x \in \lambda$: B(x) => B(g(x)),

dann gelangt man zu dem einfachsten zur Grundform äquivalenten Muster:

```
funct F ≡ (λ x)λ:
      if B(x) then F(g(f(x))) else g(x) fi                    M5
```

Die entsprechenden Regeln müssen demnach wie folgt aussehen und in der angegebenen Reihenfolge abgeprüft werden (wobei die Werte der Eigenschaften standardmäßig mit 'FALSE' initialisiert sind):

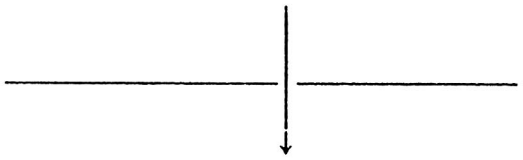
```
Regel14     geschachtelt (einfach)
            es gilt die Eigenschaft O                         CUOTS

      ==>   Wert(O):=TRUE
```

```
Regel15    geschachtelt (einfach)
           es gilt die Eigenschaft 1                    CUOTS

      ==>  Wert(1):=TRUE


Regel16    geschachtelt (einfach)
           Wert(0) = TRUE
           Wert(1) = FALSE

      ==>  M7
           phi einstellig
           ersetze 'geschachtelt (einfach)'
             durch 'linear'


Regel17    geschachtelt (einfach)
           Wert(1) = TRUE
           es gilt die Eigenschaft 2                    CUOTS

      ==>  Wert(2):=TRUE


Regel18    geschachtelt (einfach)
           Wert(1) = TRUE
           Wert(2) = TRUE
           es gilt die Eigenschaft 3                    CUOTS

      ==>  Wert(3):=TRUE
           M5
           ersetze 'geschachtelt (einfach)'
             durch 'Schema aufgelöst'


Regel19    ¬(Schema aufgelöst)
           geschachtelt (einfach)
           Wert(0) = TRUE
           Wert(1) = TRUE

      ==>  M6
           ersetze 'geschachtelt (einfach)'
             durch 'Schema aufgelöst'


Regel20    geschachtelt (einfach)
           ¬(Schema aufgelöst)

      ==>  M8
           ersetze 'geschachtelt (einfach)'
             durch 'Schema aufgelöst'
```

Damit sind alle Regeln für die Transformation der einfach geschachtelten Rekursion mit der Schachtelungstiefe 2 gegeben.

Es ist offensichtlich, daß bei größerer Schachtelungstiefe ohne zusätzliche Veränderungen eine zur Grundumwandlung analoge Auflösung erzielt werden kann.
Eine geschachtelte rekursive Funktion mit der Schachtelungstiefe 3 sieht wie folgt aus und kann analog zur Grundumwandlung der einfach geschachtelten rekursiven Funktion mit der Schachtelungstiefe 2 aufgelöst werden:

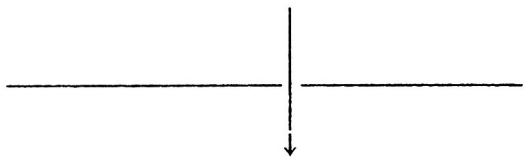
```
funct F ≡ (λ x)λ:
      if B(x) then F(F(F(f(x)))) else g(x) fi
```



```
funct F ≡ (λ x)λ:
      Q(x,1)
      where
      funct Q ≡ (λ x, nat i)λ:
            if i≠0
            then Q( if B(x)
                    then (f(x), i+2)
                    else (g(x), i-1)
                    fi )
            else x
            fi
```

Spätestens bei der Betrachtung einer geschachtelt rekursiven Funktion mit der Schachtelungstiefe 4 fällt auf, an welcher Stelle sich die Schachtelungstiefe auswirkt:

```
funct F ≡ (λ x)λ:
      if B(x) then F(F(F(F(f(x))))) else g(x) fi
```



```
funct F ≡ (λ x)λ:
      Q(x,1)
      where
      funct Q ≡ (λ x, nat i)λ:
            if i≠0
            then Q( if B(x)
                    then (f(x), i+3)
                    else (g(x), i-1)
                    fi )
            else x
            fi
```

An der Stelle, an der die Variable i hochgezählt wird, ist ein Unterschied zum Grundmuster M8, sonst nirgends. D.h. alle Schachtelungstiefen können mit einem Muster abgehandelt werden, wenn man an dieser Stelle (i + Schachtelungstiefe - 1) einsetzt. Regel20 muß entsprechend geändert werden. Ferner müssen Regel14 und Regel19 als Voraussetzung 'Schachtelungstiefe 2' enthalten.

Betrachtet werden soll nun eine weitere Möglichkeit geschachtelter Rekursionen und ihre Auflösung:

```
funct F ≡ (λ x)λ:
      if B(x) then F(h(F(f(x)))) else g(x) fi
```

Diese Rekursionsform wird 'geschachtelt (verschränkt)' genannt, und der Ausdruck F(h(F(f(x)))) soll kurz mit A bezeichnet werden.

Hierzu kann man sich folgendes überlegen:

1.) Wenn die Bedingung B(x) gilt, wird ein F durch A ersetzt, gilt die Bedingung B(x) nicht, so wird für F 'g(x)' eingesetzt.
2.) Ist einmal F(h(F(f(x)))) eingesetzt worden, dann beinhaltet der Ausdruck zwei F's, die noch ersetzt werden müssen. Hier kann eine Zählervariable eingeführt werden, die diese "offenen" F's zählt, also nach einmaligem Einsetzen von A auf 2 steht, nach zweimaligem auf 3 usw.
3.) Bei der Abarbeitung dieser F's gibt es nun folgende Situationen:
    a) i=0, d.h. kein offenes F mehr => x

b) B(x), d.h. die Bedingung ist erfüllt, A wird eingesetzt, es kommt ein F dazu => (f(x), i+1)
c) ¬B(x) und i=1, d.h. die Bedingung ist nicht mehr erfüllt und es gibt nur noch ein offenes F => (g(x), i-1=0)
d) ¬B(x) und i≠1, d.h. die Bedingung ist nicht mehr erfüllt und es gibt mehr als ein offenes F => (h(g(x)), i-1)

Die aufgelöste Form sieht demnach wie folgt aus:

```
funct F ≡ (λ x)λ:
      Q(x,1)
      where
      funct Q ≡ (λ x, nat i)λ:
            if i=0
            then x
            else if B(x)
                 then Q(f(x), i+1)
                 else if i=1
                      then g(x)
                      else Q(h(g(x)), i-1)
                      fi
                 fi
            fi
```

Schreibt man diese aufgelöste Form analog zu M8, so sind wieder Übereinstimmungen festzustellen:

```
funct F ≡ (λ x)λ:
      Q(x,1)
      where
      funct Q ≡ (λ x, nat i)λ:
            if i≠0
            then if ¬B(x) and i=1
                 then g(x)
                 else Q( if B(x)
                         then (f(x),i+1)
                         else (h(g(x)),i-1)
                         fi )
                 fi
            else x
            fi
```

Nur an zwei Stellen unterscheiden sich die Muster.

1.) Falls ¬B(x) gilt und i=1 ist, wird das letzte F durch g(x) ersetzt.

2.) Statt des einfachen Ersetzens eines F's durch g(x) wird h(g(x)) eingesetzt, falls die Bedingung B(x) nicht gilt und noch mindestens zwei offene F's existieren.

Auch dieses Muster kann in M8 (Regel20) integriert werden. Der Übersichtlichkeit halber wird diese Regel jedoch auseinandergezogen und auf drei Regeln verteilt. Die erste beinhaltet dann das Grundmuster, die zweite und dritte füllen in Abhängigkeit von der Rekursionsart die 'Lücken'.

Daraus entstehen Regel20 (neu), Regel21 und Regel22:

```
Regel20     geschachtelt
            ¬(Schema aufgelöst)

      ==>   M8 (neu)
```

```
Regel21     geschachtelt (einfach)
            Schachtelungstiefe y
            ¬(Schema aufgelöst)

      ==>   streiche *1
            ersetze *2 durch g(x)
            ersetze ST durch (i + Schachtelungtiefe - 1)
            ersetze 'geschachtelt (einfach)'
              durch 'Schema aufgelöst'
```

```
Regel22     geschachtelt (verschränkt)
            ¬(Schema aufgelöst)

      ==>   ersetze *1 durch
              if ¬B(x) and i=1 then g(x) else
            ersetze *2 durch h(g(x))
            ersetze 'geschachtelt (verschränkt)'
              durch 'Schema aufgelöst'
```

wobei das Muster M8 nun wie folgt aussieht:

```
funct F ≡ (λ x)λ:
      Q(x,1)
      where
      funct Q ≡ (λ x, nat i)λ:                          M8 (neu)
            if i≠0
            then *1
                 Q( if B(x)
                    then (f(x), ST)
                    else (*2, i-1)
                    fi )
            else x
            fi
```

Auch bei dieser verschränkt geschachtelten Form ist denkbar, daß die Reihe 'rek.Funk. - einf.Funk. - rek.Funk. - einf.Funk.' eine größere Schachtelungstiefe hat.

Z.B. Schachtelungstiefe 3:

```
funct F ≡ (λ x)λ:
      if B(x) then F(l(F(h(F(f(x)))))) else g(x) fi
```

wobei l, h, f und g einfache Funktionen sind.

Zunächst soll versucht werden die Funktion mit Hilfe von zwei Zählervariablen i und j aufzulösen. Die Funktion Q wäre dann in der aufgelösten Form 3-stellig.

Es können folgende Fälle auftreten:

1.) i=0, d.h. es existiert kein offenes F mehr => (x,i,j)
2.) ¬B(x)
   a) i=1, j=0, d.h. die Bedingung gilt nicht, und es existiert noch ein offenes F vor einem l => (g(l(x)), i-1, j)
   b) i=1, j=1, d.h. die Bedingung gilt nicht, und es existieren noch zwei offene F's, eines vor der Funktion l und eines vor der Funktion h. Da das letzte F immer vor einem l steht, ist auch dieser Fall eindeutig => (g(h(x)), i, j-1)
   c) i=2, j=... An dieser Stelle wird deutlich, daß einfache Zählervariablen nicht ausreichen, da aus der Anzahl der noch vorhandenen l's nicht hervorgeht, an welcher Stelle diese l's stehen.

Angenommen, es gilt i=2 und j=2.
Ein derartiger Ausdruck kann durch verschiedene Bedingungsfolgen aus F abgeleitet werden:

```
F — B(x)—> F(l(F(h(F(f(x))))))
  — B(x)—> F(l(F(h(F(l(F(h(F(f(f(x)))))))))))
  - ¬B(x)—> F(l(F(h(F(l(F(h(g(f(f(x)))))))))))
            2        2        y1
```

```
F — B(x)—> F(l(F(h(F(f(x))))))
  - ¬B(x)—> F(l(F(h(g(f(x))))))
  — B(x)—> F(l(F(l(F(h(F(f(h(g(f(x)))))))))))
            2      2       y2
```

Das gleiche Problem tritt auf, wenn man die Möglichkeiten durchspielt für den Fall, daß die Bedingung B(x) gilt:

3.) B(x)
   a) i=1, j=0, d.h. die Bedingung gilt, und es existiert ein offenes F vor einem l => (f(l(x)), i, j+1)
   b) i=1, j=1, d.h. die Bedingung gilt, und es existieren zwei offene F's, eines vor einem l und eines vor einem h. Da das letzte F immer vor einem l steht, ist auch dieser Fall eindeutig => (f(h(x)), i+1, j)
   c) i>1, j=... Auch hier ist nicht eindeutig, ob ein F direkt vor einer Funktion l oder vor einer Funktion h ersetzt werden soll.

Lösung dieses Problems: Die Funktion Q in der aufgelösten Form bleibt zweistellig. Sie enthält als zweites Argument jedoch keine Zählervariable, sondern einen stack.

Es gibt nun verschiedene Möglichkeiten diesen stack aufzubauen. Das Einfachste wäre die gesamte noch nicht verarbeitete Zeichenkette im stack unterzubringen, Schritt für Schritt abzuarbeiten, zu expandieren oder zu reduzieren. In unserem Fall kann jedoch Platz gespart werden, da gewisse Vorinformationen verfügbar sind.
h's können nicht in direkter Folge vorkommen, sondern werden immer von Fl^iF mit $i \geq 1$ getrennt. D.h. es reicht, die Potenz von l im stack zu speichern. Bei der Abarbeitung des stacks kann dann davon ausgegangen werden, daß jeweils zwischen zwei Funktionen l^i ein FhF steht. Ist der stack irgendwann leer, so ist das letzte l^i verarbeitet worden, und es muß die Grundform F(y) berechnet werden. Gilt nun im nächsten Schritt B(x), dann wird der stack neu aufgefüllt, gilt ¬B(x), dann wird F(y) durch g(y) ersetzt und die Funktion ist fertig berechnet.

Ein dritter Punkt ist noch zu berücksichtigen. Um zu wissen, wie die nächste Reduktion auszusehen hat, muß im stack vermerkt werden, ob h miteinbezogen werden muß oder nicht. Zu diesem Zweck notiert man eine 0 an oberster stack-Position, falls h direkt vor dem bereits berechneten Teil in der Zeichenkette steht.

Die aufgelöste Form der verschränkt geschachtelten Funktion F mit der Schachtelungstiefe 3 unter Verwendung eines stacks sieht dann wie folgt aus:

```
funct F ≡ (λ x)λ:
      Q(x, empty)
      where
      funct Q ≡ (λ x, stack nat s)λ:
            if ¬(¬B(x) and s=empty)
            then Q( if B(x)
                    then if top s = 0
                         then (f(x), (((rest s) & 1) & 0))
                         else (f(x),
                                   (((rest s) & (top s + 1) & 0))
                         fi
                    else if top s = 0
                         then ((h(g(x))), (rest s))
                         else (L((top s), g(x)), (rest s))
                         fi
                    fi )
            else g(x)
            fi
            where
            funct L ≡ (nat n, λ y)λ:
                  if n>0
                  then L(n-1, l(y))
                  else y
                  fi
```

Die Form der verschränkt geschachtelten rekursiven Funktion kann auch auf eine beliebige Schachtelungstiefe verallgemeinert werden:

funct F ≡ (λ x)λ:
if B(x) then $F(h_1(F(h_2(\ldots(F(h_n(F(f(x>$ else g(x) fi

Bei der Auflösung dieser allgemeineren Form scheint es jedoch sinnvoll, den stack vollständig aufzubauen, d.h. die gesamte Zeichenkette aufzunehmen, und dann bei jedem Schritt in Abhängigkeit vom top-Element die Verarbeitung fortzusetzen. Eine weitere Möglichkeit ist die ablauforientierte Lösung dieses Problems, die in Kapitel 4.3 unter dem Thema 'Transformation einer entflochtenen Form' im Einzelnen behandelt werden wird.

Die aufgelöste Form mit einem die Zeichenkette vollständig (bis auf die zwischen je zwei h_i's stehenden F's) enthaltenden stack hat dann folgendes Aussehen:

```
funct F ≡ (λ x)λ:
      Q(x, empty)
      funct Q ≡ (λ x, stack {h1,...,hn} s)λ:
            if ¬(¬B(x) and s=empty)
            then Q( if B(x)
                    then (f(x), s & h1 &...& hn)
                    else (top s(g(x)), rest s)
                    fi )
            else g(x)
            fi
```

Eine weitere Variante der verschränkt geschachtelten rekursiven Funktion ist die Form, bei der an erster Stelle kein rekursiver Funktionsaufruf steht, sondern eine einfache Funktion.

Man beachte, daß es sich hierbei bereits um eine Mischform aus linearer und verschränkt geschachtelter Rekursion handelt.

Für die Schachtelungstiefe 2 sieht die Funktion wie folgt aus:

```
funct F ≡ (λ x)λ:
      if B(x)
      then l(F(h(F(f(x)))))
      else g(x)
      fi
```

und die allgemeinere Form für die Schachtelungstiefe n:

```
funct F ≡ (λ x)λ:
      if B(x)
      then h1(F(h2(F(...(hn(F(f(x>
      else g(x)
      fi
```

Betrachtet man nun verschiedene Bedingungsfolgen, so wird deutlich, daß hinter jedem h_i $(1 \le i \le n)$ beliebig viele h_1's auftreten können, die von h_i nicht durch F getrennt sind. Der stack muß entsprechend aufgebaut werden. Man kann z.B. folgende Struktur wählen:

stack $\{h_1,...,h_n\} \times$ nat s,

wobei top_1 s 'h_i' angibt und top_2 s die Potenz i von h_1[1]. Ist top_2 s = 0, dann folgt kein h_1, sondern direkt F.

Die aufgelöste Form der allgemeinen linear verschränkt geschachtelten Rekursion sieht dann folgendermaßen aus:

```
funct F ≡ (λ x)λ:
      Q(x, empty, 0)
      where
      funct Q ≡ (λ x, stack {h1,...,hn} × nat s, {0,1} v) λ:
            if ¬((¬B(x) or v=1) and s=empty)
            then Q( if B(x)
                 then if s≠empty
                      then (f(x), (rest s)
                                  & ((top1 s), ((top2 s) + 1))
                                     & (h2,0) &...& (hn,0), 1)
                      else (f(x),s & (h1,0) &...& (hn,0), 1)
                      fi
                 else (top1 s(H(top2 s, g(x))), (rest s), 1)
                 fi )
            else if v=1 then x else g(x) fi
            fi
            where
            funct H ≡ (nat n, λ y)λ:
                  if n>0
                  then H(n-1, h1(y))
                  else y
                  fi
```

wobei die Variable v in der Definition von Q angibt, ob mindestens einmal expandiert wurde oder nicht. In Abhängigkeit davon wird als letzter Schritt der Rekursion entweder g(x) berechnet (letzter Schritt = erster Schritt, v=0), bzw. der berechnete Funktionswert x ausgegeben (v=1).

Enthalten in diesen Schachtelungsmöglichkeiten sind z.B. auch die folgenden rekursiven Operationsteile:

a) F(l(F(F(f(x)))))
b) h_1(F(F(F(f(x)))))

a) entspricht der Schachtelung F(l(F(h(F(f(x)))))) mit h = id (Identität)
b) entspricht der Schachtelung h_1(F(h_2(F(h_3(F(f(x))))))) mit h_2 = id und h_3 = id .

Allerdings würde das Einbeziehen dieser Fälle bei dem oben beschriebenen Transformationsschema zu fehlerhaften Ergebnissen führen, da die verwendete Zusatzinformation hierfür nicht mehr gilt.

Falls also derartige verschränkt geschachtelte und linear verschränkt geschachtelte Funktionen mitaufgenommen werden sollen, so muß dies bereits beim Matchen (einsetzen von Identitätsfunktionen) als auch beim Programmtransformieren bezüglich der Verwendung der stacks berücksichtigt werden. Meiner Meinung nach sind diese Formen jedoch mehr von theoretischem Interesse und dürften in der Praxis kaum Anwendung finden. Eine Erweiterung der Verwendung der stacks würde im Gegensatz zu der oben be-

schriebenen reduzierten Verwendung das Programm erheblich verlangsamen.

Daher sollen diese komplizierteren Fälle hier nicht weiter untersucht werden, sondern lediglich die verschränkt geschachtelten rekursiven Funktionen mit einer Schachtelungstiefe größer als 2 und die linear verschränkt geschachtelten berücksichtigt werden, sodaß die Regeln 20, 21 und 22 entsprechend geändert und vier neue Regeln hinzugefügt werden müssen.

```
Regel20(neu) rekursiv (geschachtelt)
             ¬(Schema aufgelöst)

         ==> M8
```

wobei sich das Muster M8 wiederum verändert hat. Es müssen mehr 'Lücken' gelassen werden, die bei der Anwendung weiterer Regeln ausgefüllt werden.

```
funct F ≡ (λ x)λ:
      Q(x,z)
      where
      funct Q ≡ (λ x, z-Sorte)λ:                    M8 (3. Fassung)
            if BED
            then *1
            Q( if B(x)
               then *3
               else *4
               fi )
            else *6
            fi
            *5
```

```
Regel21     geschachtelt (einfach)
            Schachtelungstiefe y>2

      ==>   fülle die Lücken in M8:
            z := 1
            z-Sorte := nat i
            BED := i≠0
            *1 := id
            *3 := (f(x), i + Schachtelungstiefe - 1)
            *4 := (g(x), i-1)
            *5 := { }
            *6 := x
            ersetze 'geschachtelt (einfach)'
              durch 'Schema aufgelöst'
```

```
Regel22     geschachtelt (verschränkt)
            Schachtelungstiefe y=2

      ==>   fülle die Lücken in M8:
            z := 1
            z-Sorte := nat i
            BED := i≠0
            *1 := if ¬B(x) and i=1 then g(x) else
            *3 := (f(x), i+1)
            *4 := (h(g(x)), i-1)
            *5 := { }
            *6 := x
            ersetze 'geschachtelt (verschränkt)'
              durch 'Schema aufgelöst'


Regel23     geschachtelt (verschränkt)
            Schachtelungstiefe y>2

      ==>   fülle die Lücken in M8:
            z := empty
            z-Sorte := stack *2 s
            BED := ¬(¬B(x) and s=empty)
            *1 := { }
            *6 := g(x)


Regel24     geschachtelt (verschränkt)
            Schachtelungstiefe y=3

      ==>   fülle die Lücken in M8:
            *2 := nat
            *3 := if top s = 0
                  then (f(x), ((rest s) & 1 & 0)))
                  else (f(x), ((rest s) & ((top s) + 1) & 0)))
                  fi
            *4 := if top s = 0
                  then ((h(g(x))), (rest s))
                  else (L((top s), g(x)), (rest s))
                  fi
            *5 := where
                  funct L ≡ (nat n, λ y)λ:
                        if n>0 then L(n-1, l(y)) else y fi
            ersetze 'geschachtelt (verschränkt)'
              durch 'Schema aufgelöst'
```

Regel25 geschachtelt (verschränkt)
Schachtelungstiefe y>3

==> fülle die Lücken im (erweiterten) Muster M8:
*2 := $\{h_1,...,h_n\}$
*3 := (f(x), s & h_1 &.....& h_n)
*4 := ((top s(g(x))), (rest s))
*5 := { }
ersetze 'geschachtelt (verschränkt)'
durch 'Schema aufgelöst'

Regel26 geschachtelt (linear verschränkt)

==> fülle die Lücken in M8:
z := empty,0
z-Sorte := stack ($\{h_1,..,h_n\}$ × nat) s, {0,1} v
BED := ¬((¬B(x) or v=1) and s=empty)
*1 := { }
*3 := if s≠empty
then (f(x), (rest s) & ((top_1 s), (top_2 s + 1))
& (h_2,0) &...& (h_n,0), 1)
else (f(x), s & (h_1,0) &...& (h_n,0), 1)
fi
*4 := ((top_1 s)(H((top_2 s), g(x)), (rest s), 1)
*5 := where
funct H ≡ (nat n, λ y)λ:
if n>0
then H(n-1, h_1(y))
else y
fi
*6 := if v=1 then x else g(x) fi
ersetze 'geschachtelt (linear verschränkt)'
durch 'Schema aufgelöst'

Damit seien die Betrachtungen der geschachtelten rekursiven Funktionen vorerst abgeschlossen (Ergänzungen siehe Kapitel 4.3 über Entflechtung).

"I find it difficult to believe that
whenever I see a tree
I am really seeing a string of symbols"

McCarthy

4.3 Transformationsschemata und Regeln für die kaskadenartige Rekursion

4.3.1 Allgemeines

Wie man am einfachsten Muster für eine kaskadenartige Rekursion bereits erkennen kann, sind derartige Funktionen im allgemeinen nicht ohne Einführung von stacks in repetitive Form überführbar:

funct $C \equiv (\lambda\ x)p$:
if $B(x)$ then $(C(K_1(x)), C(K_2(x)))$ else $H(x)$ fi

Paterson und Hewitt [PaHe 70] haben durch spezielle Konkretisierung des oben angegebenen Musters gezeigt, daß man nicht einmal mit einem Keller auskommt, sondern sowohl einen Parameter- als auch einen Protokollkeller für die Überführung in repetitive Form braucht. Das heißt nicht, daß ein Keller nicht sowohl Parameter als auch Adressen enthalten kann, sondern bezieht sich lediglich auf die Notwendigkeit, daß die Möglichkeit vorgesehen werden muß, beide Arten von Werten abspeichern zu können. Für ein allgemeines Transformationssystem wird man also beide Kellerarten bereitstellen müssen.

Der systematischste Ansatz für die Auflösung von Kaskaden und komplizierten Mischformen besteht darin, zunächst den Rekursionsablauf zu 'entflechten' (mit Hilfe eines Parameterkellers) und anschließend diese entflochtene Form (abstrakte Schachtelung) in eine repetitive zu transformieren, wobei die wesentlichen Informationen über den Ablauf der Rekursion (Analyse der Prozedur) in einem Protokollkeller abgespeichert werden.

Der Nachteil dieses Ansatzes besteht darin, daß nur in sehr einfachen Fällen ein oder beide Keller als überflüssig erkannt werden, wenn algorithmisch äquivalente Formen existieren, die ohne Einführung von stacks (leicht erkennbar) aufgelöst werden können. In vielen Fällen, in denen die Einführung von stacks theoretisch nicht notwendig ist, wird man auf individuelle Überlegungen angewiesen sein, um eine einfache Umformung darzustellen.

Es gibt jedoch einige Ansätze, die sich mit der Systematisierung derartiger Umformungen befassen. Hier ist vor allem die bei Bauer und Wössner als 'allgemeiner Ansatz' bezeichnete Methode zu nennen, die zuerst von Darlington und Burstall in umfassenderer Bedeutung unter dem Namen unfold/fold-Methode behandelt wurde. Anwendungen und Weiterentwicklungen dieses Ansatzes finden sich unter anderem bei Feather [Feat 77] [Feat 79] , Clark/Darlington [ClDa 77] und Schmitz [Schm 78] (siehe Kapitel 6).

Als weitere Ansätze sind zu nennen die 'Arithmetisierung des Ablaufs' und die 'Technik der Wertverlaufstabellierung' ([BaWö 81], Kapitel 4.3.2 und 4.3.4). Diese beiden Ansätze scheinen jedoch nicht mächtig genug, um eine abstraktere Methode (oder Theorie) zu begründen.

Auf Grund der weit über kaskadenartige Rekursionen hinausragenden Anwendungsmöglichkeiten der unfold/fold-Methode wird diese gesondert behandelt (siehe Kapitel 6).

Auch die Entflechtung ist eine umfassendere Methode, die in engem Zusammenhang mit der geschachtelten Rekursion steht. Zum einen ist das Ergebnis einer vollständigen Entflechtung stets eine geschachtelte Form, zum anderen können auch komplizierte, ohne stack nicht auflösbare Schachtelungen in dieser Weise behandelt werden.
Für derartige, komplizierte kaskadenartige und geschachtelte Funktionen sowie Mischformen findet die Transformation in zwei Schritten statt. Im ersten Schritt wird eine 'entflochtene Form' erzeugt (möglicherweise unter Verwendung von Parameterkellern), und erst im zweiten Schritt wird diese entflochtene Form in eine repetitive überführt.

Der erste Schritt dieser Transformation soll zunächst für eine einfache Kaskade gezeigt werden, dann für die allgemeinste Kaskadenform, und schließlich soll ein abstraktes Beispiel die Entflechtung einer kompliziert geschachtelten Funktion illustrieren. Konkrete Beispiele findet man in Kapitel 4.4.

4.3.2 Entflechtung von Kaskaden

Die Überführung einer entflochtenen Form in eine repetitive (Schritt 2) soll im nächsten Abschnitt gesondert behandelt werden, da sowohl entflochtene Kaskaden als auch entflochtene Schachtelungen und Mischformen betroffen sind.

Das Muster für eine einfache Kaskade sieht wie folgt aus:

funct C ≡ (λ x)p:
    if B(x) then phi(C(K_1(x)), C(K_2(x)), E(x)) else H(x) fi

Anmerkung:
Bei parallelen Aufrufen rekursiver Funktionen sollte schon beim Matchen geprüft werden, ob die Argumente (hier $K_1(x)$ und $K_2(x)$) identisch sind. Wenn dies der Fall ist, kann eventuell in einen einfacheren Rekursionstyp (hier z.B. in einen linearen, falls $K_1(x) \equiv K_2(x)$) umgeformt werden.

Die Einführung von Hilfsvariablen in der oben gegebenen kaskadenartigen Funktion liefert die folgende "detaillierte" Form:

```
funct C ≡ (λ x)ρ:
      if B(x)
      then λ x1 ≡ K1(x);   ρ z1 ≡ C(x1);
           λ x2 ≡ K2(x);   ρ z2 ≡ C(x2);
           phi(z1, z2, E(x))
      else H(x)
      fi
```

Um diese Form weiter untersuchen zu können, muß zuerst der Begriff der 'Entflochtenheit' geklärt werden.

Definition (nach [BaWö 81]): Die detaillierte Form (s.o.) einer rekursiven Rechenvorschrift heißt entflochten, wenn keiner der Parameter und keine der zur Detaillierung eingeführten Hilfsvariablen sowohl vor als auch nach ein und demselben rekursiven Aufruf verwendet wird.

Sieht man sich diese Definition genau an, so ist schon intuitiv klar, daß eine entflochtene Form in eine repetitive umgewandelt werden kann, wobei kein Parameterkeller mehr nötig ist, sondern lediglich ein Parameterregister. Allerdings müssen die "Rücksprungadressen" der rekursiven Funktionsaufrufe in einem Protokollkeller gestapelt werden. Dazu jedoch später.

Die oben gegebene detaillierte Form einer kaskadenartigen Funktion ist also nach unserer Definition nicht entflochten, da der Parameter x und die Hilfsvariable z_1 die entsprechende Bedingung verletzen.

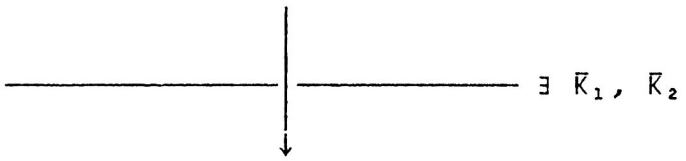
Eine sogenannte 'Einbettung' bereitet die Entflechtung von x vor. Der entscheidende Schritt dieser Einbettung besteht darin, zusätzlich noch den Parameterwert, mit dem die Rechenvorschrift aufgerufen wurde, als Resultat abzuliefern.

Daraus ergibt sich die folgende Form:

```
funct C ≡ (λ x)ρ:
      b
      where (λ a, ρ b) ≡ C*(x)
      funct C* ≡ (λ x)(λ, ρ):
            if B(x)
            then λ x1 ≡ K1(x); (λ y1, ρ z1) ≡ C*(x1);
                 λ x2 ≡ K2(x); (λ y2, ρ z2) ≡ C*(x2);
                 (x, phi(z1, z2, E(x)))
            else (x, H(x))
            fi
```

Hierbei gilt offensichtlich: $y_1 = K_1(x)$ und $y_2 = K_2(x)$.

D.h., falls die Umkehrfunktionen $\overline{K}_1$ und $\overline{K}_2$ von K_1 und K_2 existieren, läßt sich x aus y_1 und y_2 jeweils wieder zurückgewinnen und die Funktion hinsichtlich des Parameters x im Ablauf entflechten:




```
funct C ≡ (λ x)ρ:
      b
      where (λ a, ρ b) ≡ C*(x)
      funct C* ≡ (λ x)(λ, ρ):
            if B(x)
            then λ x1 = K1(x); (λ y1, ρ z1) ≡ C*(x1);
                 λ x2 = K2(x); (λ y2, ρ z2) ≡ C*(x2);
                 (K̄2(y2), phi(z1, z2, E(K̄2(y2))))
            else (x, H(x))
            fi
```

Für z_1 muß ein stack eingeführt werden, da keine Möglichkeit besteht, mit einer Umkehrfunktion zu arbeiten.

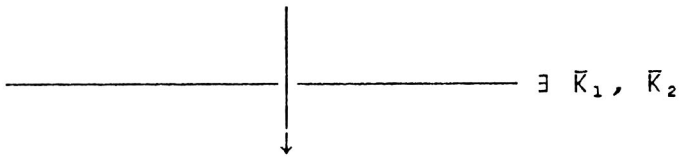
Daraus ergibt sich die Form:

```
funct C ≡ (λ x)ρ:
      b
      where (λ a, stack ρ sb, ρ b) ≡ C*(x, empty)
      funct C* ≡ (λ stack ρ sz) (λ, stack ρ, ρ):
         if B(x)
         then (λ x1, stack ρ sr1) ≡ (K1(x), sz);
              (λ y1, stack ρ sz1, ρ z1) ≡ C*(x1, sr1);
              (λ x2, stack ρ sr2) ≡ (K2(x), sz & z1);
              (λ y2, stack ρ sz2, ρ z2) ≡ C*(x2, sr2);
              (x, sz, phi(z1, z2, E(x)))
         else (x, sz, H(x))
         fi
```

wobei folgende Äquivalenzen gelten:

$$sz_1 = sr_1 = sz$$
$$sz_2 = sr_2 = sz \;\&\; z_1$$
$$sz = \text{rest } sz_2$$
$$z_1 = \text{top } sz_2$$

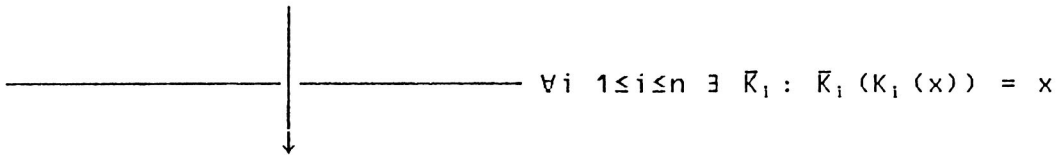
Die Funktion läßt sich nun unter Ausnutzung dieser Äquivalenzen in eine vollständig entflochtene Form überführen:




```
funct C ≡ (λ x)ρ:
      b
      where (λ a, stack ρ sb, ρ b) ≡ C*(x, empty)
      funct C* ≡ (λ x, stack ρ sz) (λ, stack ρ, ρ):          M9
         if B(x)
         then (λ x1, stack ρ sr1) ≡ (K1(x), sz);
              (λ y1, stack ρ sz1, ρ z1) ≡ C*(x1, sr1);
              (λ x2, stack ρ sr2) ≡ (K2(K̄1(y1)), sz1 & z1);
              (λ y2, stack ρ sz2, ρ z2) ≡ C*(x2, sr2);
              (K̄2(y2), rest sz2, phi(top sz2, z2, E(K̄2(y2))))
         else (x, sz, H(x))
         fi
```

Gleich anschließend soll versucht werden die allgemeinste Form einer kaskadenartigen Funktion in eine entflochtene Form zu überführen. Wieder sei darauf hingewiesen, daß schon beim Matchen alle Argumente K_i paarweise auf Gleichheit untersucht werden sollten, um die Funktion eventuell schon vor dem Entflechtungsversuch zu vereinfachen.

```
funct C ≡ (λ x)ρ:
      if B(x)
      then phi(C(K1(x)),..., C(Kn(x)), E(x))
      else H(x)
      fi
```




```
funct C ≡ (λ x)ρ:
      b
      where (λ a, stack ρ sb, ρ b) ≡ C*(x, empty)
      funct C* ≡ (λ x, stack ρ sz) (λ, stack ρ, ρ):
         if B(x)
         then (λ x1, stack ρ sr1) ≡ (K1(x), sz);
              (λ y1, stack ρ sz1, ρ z1) ≡ C*(x1,sr1);
              (λ x2, stack ρ sr2) ≡ (K2(K̄1(y1)), sz1 & z1);
              (λ y2, stack ρ sz2, ρ z2) ≡ C*(x2,sr2);
              (λ x3, stack ρ sr3) ≡ (K3(K̄2(y2)), sz2 & z2);
              ..........
              (λ xn, stack ρ srn) ≡
                                   (Kn(K̄n-1(yn-1)), szn-1 & zn-1);
              (λ yn, stack ρ szn, ρ zn) ≡ C*(xn,srn);
              (K̄n(yn), rest^(n-1) szn, phi(top rest^(n-2) szn,
                                  ....., top rest° szn, zn, E(K̄n(yn)))
        else (x, sz, H(x))
        fi
```

Anmerkungen:

a) Ob die Umkehrfunktionen existieren, und falls ja, welches diese Umkehrfunktionen sind, muß als Zusatzinformation - wie bei der Transformation der linearen Rekursion - durch Anfragen an den Benutzer in Erfahrung gebracht werden.

b) Falls nicht alle Umkehrfunktionen existieren, muß für jede nicht existierende Umkehrfunktion ein stack vorgesehen werden, bzw. der oben verwendete Parameterkeller entsprechend erweitert werden.

Regeln für die Entflechtung kaskadenartiger Rekursionen:

Regel27 kaskadenartig
$\bar{K}_i$ für einige $1 \leq i \leq n$ CUOTS

==> M9
{∀ K_i ¬∃ $\bar{K}_i$: kellere den Parameter x_i
und ersetze $\bar{K}_i(y_i)$ durch x_i}
Schema entflochen
bzw. geschachtelt

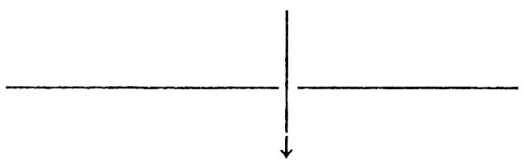
Die entflochtene Form kann dann mit Hilfe der entsprechenden Regeln für die geschachtelt rekursiven Funktionen in eine repetitive Form transformiert werden.

4.3.3 Beispiel für die Entflechtung einer kompliziert geschachtelten rekursiven Funktion

Entflochten werden soll die folgende linear verschränkt geschachtelte Funktion:

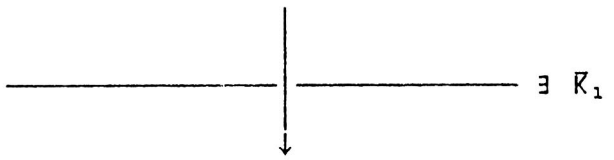
```
funct F ≡ (λ x)p:
      if B(x)
      then phi(F(psi(F(K1(x)), K2(x))), E(x))
      else H(x)
      fi
```

Zu diesem Zweck muß sie zunächst in eine detaillierte Form überführt werden:



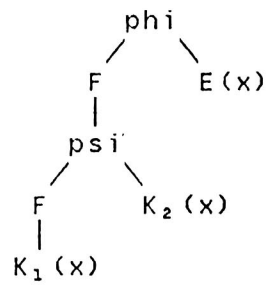
```
funct F ≡ (λ x)p:
      if B(x)
      then x1 ≡ K1(x);    z1 ≡ F(x1);
           x2 ≡ K2(x);    z2 ≡ psi(z1, x2);
           x  ≡ phi(z2, E(x))
      else H(x)
      fi
```

Da der Parameter x der Entflochtenheitsbedingung nicht genügt, muß für x ein stack eingeführt werden.




```
funct F ≡ (λ x)p:
      b
      where (λ a, stack λ sa, p b) ≡ F*(x, empty)
      funct F* ≡ (λ x, stack λ sx) (λ, stack λ, p):
            if B(x)
            then (λ x1, stack λ sx1) ≡ (K1(x), sx);
                 (λ y1, stack λ sy1, p z1) ≡ F*(x1, sx1);
                 (λ x2, stack λ sx2) ≡
                              (psi(z1, k2(K̄1(y1))), sy1 & K̄1(y1));
                 (λ y2, stack λ sy2, p z2) ≡ F*(x2, sx2);
                 (top sy2, rest sy2, phi(z2, E(top sy2)))
            else (x, sx, H(x))
            fi
```

An diesem Beispiel wird deutlich, daß nur eine geschachtelte Form F, die mehrstellige Funktionen enthält, in denen F als Argument vorkommt, nicht entflochten sein kann.
Noch deutlicher wird diese Abhängigkeit, wenn man sich die entsprechenden Baumdiagramme ansieht. Treten nur Funktionen mit einem Argument auf, so entsteht ein degenerierter Baum, der stets entflochten ist.



Das Baumdiagramm zeigt deutlich, daß der Parameter x entflochten werden muß.

Anmerkungen:
a) an einem solchen Baumdiagramm kann die Detaillierung einer kompliziert geschachtelten Funktion unmittelbar abgelesen werden.
b) eine tiefere Analyse des Beispiels liefert durch Arithmetisierung des Ablaufs eine repetitive Form ohne stacks, falls Parameter als Laufvariablen zu erkennen sind, die während des Ablaufs einer systematischen Veränderung unterliegen. Es ist jedoch nicht zu sehen, wie eine derartige Analyse von einer Maschine systematisch geleistet werden könnte.

4.3.4 Transformation einer entflochtenen Form in eine repetitive

Die allgemeine entflochtene Form sieht wie folgt aus:

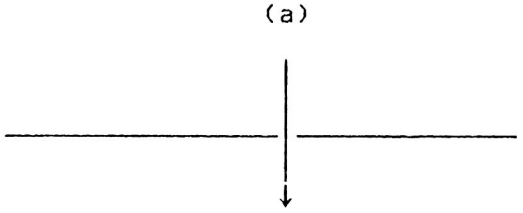
```
funct F ≡ (λ x)λ:
      if B(x)
      then F(S_n(F(S_n-1(.....(F(S_1(x>                    (a)
      else H(x)
      fi
```

bzw.:

```
funct F ≡ (λ x)λ:
      if B(x)
      then S_n(F(S_n-1(.....(F(S_1(x>                      (b)
      else H(x)
      fi
```

wobei die S_i einfache Funktionen sind.

Zunächst soll versucht werden die unter (a) aufgeführte Form zu transformieren.

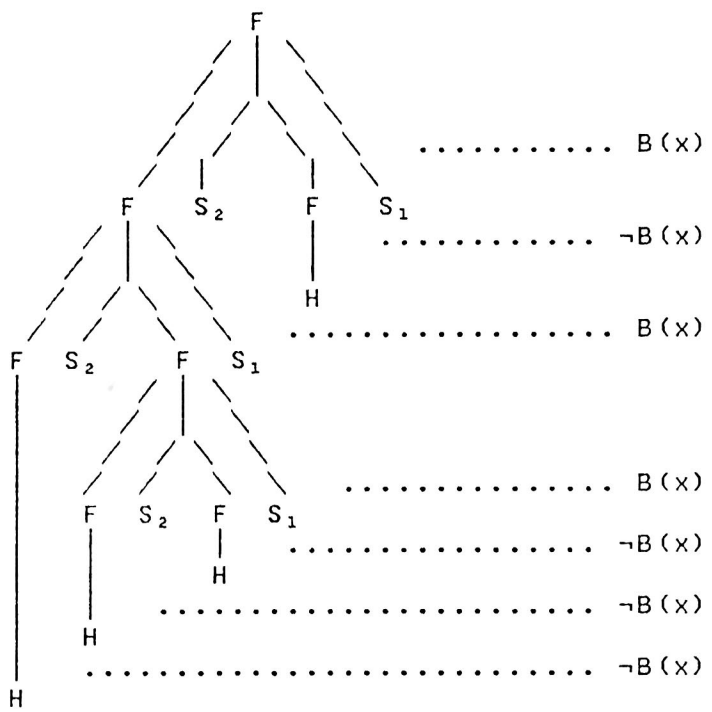



```
funct F ≡ (λ x)λ:
      F* ≡ (x, empty)
      where
      funct F* ≡ (λ x, stack {1,...,n} sn)λ:
            if B*(x, sn)
            then F*(S*(x, sn))
            else H*(x, sn)
            fi                                              M10
      where
      H*(λ x, stack {1,...,n} sn) ≡ H(x)
      B*(λ x, stack {1,...,n} sn) bool ≡ B(x) and sn≠empty
      S*(λ x, stack {1,...,n} sn) (λ, stack {1,...,n}) ≡
            if ¬B(x) and sn≠empty
            then (S_k(H*(x, sn)),
                  where Index k = top sn
                  if top sn ≠ n
                  then (rest sn) & ((top sn) + 1)
                  else (rest sn)
                  fi )
            else (S_1(x), sn & 2)
            fi
```

Damit ist die in Kapitel 4.3 mit 'verschränkt geschachtelt' bezeichnete Form in eine repetitive transformiert worden.
Bei dieser Transformation wurde jedoch nicht von der Struktur ausgegangen, die die Zeichenkette, die die noch nicht berechneten Teile der rekursiven Aufrufe repräsentiert, unter allen möglichen Bedingungsfolgen annehmen kann, sondern die Transformation ist am Ablauf orientiert. Der stack enthält somit nicht mehr die aufeinanderfolgenden S_i's, zwischen denen jeweils ein F auftritt, das im nächsten Schritt entweder expandiert oder reduziert werden muß, sondern die "Rücksprungadressen" S_i, d.h. die Stellen, an denen weitergearbeitet werden muß, wenn die dahinterliegende Zeichenkette abgearbeitet ist.

Betrachten wir als Beispiel den Ablauf für die verschränkt geschachtelte (entflochtene) Form 'F(S₂(F(S₁(x))))'
mit der Bedingungsfolge:
B(x), ¬B(x), B(x), B(x), ¬B(x), ¬B(x), ¬B(x).

a) Baumdarstellung



b) Ablauftabelle

Bedingung	Funktion (abgearbeitete Teile unterstrichen)	Keller
	F	( )
B(x)	F S_2 F $\underline{S_1\ x}$	(2)
¬B(x)	F $\underline{S_2\ H\ S_1\ x}$	( )
B(x)	F S_2 F $\underline{S_1\ S_2\ H\ S_1\ x}$	(2)
B(x)	F S_2 F S_2 F $\underline{S_1\ S_1\ S_2\ H\ S_1\ x}$	(2,2)
¬B(x)	F S_2 F $\underline{S_2\ H\ S_1\ S_1\ S_2\ H\ S_1\ x}$	(2)
¬B(x)	F $\underline{S_2\ H\ S_2\ H\ S_1\ S_1\ S_2\ H\ S_1\ x}$	( )
¬B(x)	$\underline{H\ S_2\ H\ S_2\ H\ S_1\ S_1\ S_2\ H\ S_1\ x}$	( )

Anschließend soll der zweite etwas schwierigere Fall (b) der in Kapitel 4.2 mit 'linear verschränkt geschachtelt' bezeichneten entflochtenen Form untersucht werden.

Bei der strukturorientierten Methode wurde ein reduzierter, zweistelliger stack benutzt, wobei die erste Stelle jedes stack-Elements die einfachen S_i (genauer gesagt nur den Index i) enthielt, und die zweite Stelle die Anzahl der daran in ununterbrochener Folge anschließenden S_1's. Bei der ablauforientierten Methode steht man vor ähnlichen Problemen. Auch hier muß eine Anhäufung von S_1's erkannt werden, um diese Folge ohne Unterbrechung abzuarbeiten. Verwendet man einen stack wie in Fall (a), dann ist zwar durch die Betrachtung mehrerer (der obersten) stack-Elemente zu erkennen, wo eine Anhäufung auftritt, hat man jedoch den stack entsprechend abgearbeitet, so ist nicht mehr erkennbar, ob eine ununterbrochene Folge da war, bzw. wo diese Folge beendet ist. Daher muß auch bei dieser Methode ein modifizierter stack benutzt werden, wobei ein zusätzliches Symbol '$' eingeführt wird, das das Ende einer solchen Folge signalisiert.

Schon der einfache Fall mit der Schachtelungstiefe 2 (siehe [BaWÖ 82], Kapitel 6.1.3) zeigt, daß grundsätzlich zwischen zwei verschiedenen Rekursionsschritten (Prozeduren) unterschieden werden muß.

1.) Rekursion A, die nur in Abhängigkeit von der Bedingung B(x) expandiert, und
2.) Rekursion B, die in Abhängigkeit vom Inhalt des stacks reduziert und anschließend entweder sich selbst oder Rekursion A aufruft.

Für die Auflösung der linear verschränkt geschachtelten Form ist daher die in Kapitel 4.2 erwähnte strukturorientierte Lösung mit vollständigem stack eventuell vorteilhafter.

Strukturorientierte Lösung mit vollständigem stack
(Muster M11):

```
funct F ≡ (λ x)λ:
      b
      where (λ b, stack ρ sn) ≡ F*(x, $)
      funct F*(λ x, stack ρ sn)(λ, stack ρ):
            if sn ≠ empty
            then if top sn ≠ $
                 then F*(S_k(x), (rest sn))
                      where Index k = top sn
                 else if B(x)                                    M11
                      then F*(S_1(x),
                                 ((rest sn & (n, $, n-1,..., 2, $))
                      else if top(rest sn) ≠ empty
                           then F*(S_r(H(x)), (rest(rest sn)))
                                where Index r = top(rest sn)
                           else (H(x), (rest sn))
                           fi
                       fi
                 fi
            else (x, sn)
            fi
```

Es werden daher folgende Regeln für die Transformation der entflochtenen verschränkt geschachtelten und linear verschränkt geschachtelten Funktionen in repetitive Form gebraucht:

Regel28 entflochten
geschachtelt (verschränkt)

==> M10

Regel29 entflochten
geschachtelt (linear verschränkt)

==> M11

4.4 Beispiele für die Transformation der verschiedenen Rekursionsarten

An dieser Stelle soll gezeigt werden, wie die transformierten Funktionen für die in Kapitel 3.2 bereits erwähnten Beispiele für lineare, geschachtelte und kaskadenartige Rekursion aussehen.

4.4.1 Transformation der Fakultätsfunktion

```
funct FAK ≡ (nat x) nat:
      if x ≠ 0
      then x * FAK(x-1)
      else 1
      fi
```

4.4.1.1 Auflösung durch Um-Klammerung

Da E(x) ≠ { }, d.h. phi nicht einstellig ist, wird der Versuch der Auflösung durch Um-Klammerung nicht übersprungen (Regel1).

Abgeprüft wird nun, ob es eine Funktion psi gibt, sodaß phi(phi(r, s), t) = (phi(r psi(s, t)) gilt. phi ist in diesem Fall die Multiplikation. Wird psi ebenfalls als Multiplikationsfunktion gewählt, so ist diese Bedingung erfüllt.

Die durch die Ausführung von Regel2 transformierte Fakultätsfunktion sieht dann wie folgt aus:

```
funct FAK ≡ (nat x) nat:
      if x ≠ 0 then G(x-1, x) else 1 fi
      where
      funct G ≡ (nat x, nat z) nat:
            if x ≠ 0 then G((x-1), (x*z)) else (1*z) fi
```

Auch alle Bedingungen von Regel3 sind erfüllt, da wir bereits im vorhergehenden Schritt psi = phi gewählt haben und für die Multiplikation das neutrale Element '1' existiert.

Nach Anwendung dieser Regel hat die Fakultätsfunktion die folgende endgültige Form:

```
funct FAK ≡ (nat x) nat:
      G(x,1)
      where
      funct G ≡ (nat x, nat z) nat:
            if x ≠ 0 then G(x-1, x*z) else (1*z) fi
```

Ein angeschlossener Simplifier würde natürlich noch (1*z) durch z ersetzen (siehe Kapitel 7).

4.4.1.2 Auflösung durch Operandenvertauschung

Werden die Regeln in der geordneten Reihenfolge ihrer Nummern auf Anwendbarkeit überprüft, so würde die Fakultätsfunktion auf jeden Fall durch Um-Klammerung aufgelöst werden. Wir überspringen jedoch die entsprechenden Regeln und zeigen, daß die gegebene Definition auch mit Hilfe der Operandenvertauschung transformiert werden kann.

Beginnen wir mit Regel5. Das Schema sei noch nicht aufgelöst und phi (die Multiplikation) ist nicht einstellig. Überprüft wird die Funktion jetzt auf erweiterte Rechtskommutativität im definierten Sinn, d.h. es soll eine Funktion psi gesucht werden, sodaß gilt: phi(psi(r, s), t) = psi(phi(r, t), s). Diese Bedingung ist erfüllt, da die Multiplikation rechtskommutativ ist. Nach Regel6 kann also die Fakultätsfunktion entsprechend Muster M2 transformiert werden:

```
funct FAK ≡ (nat x) nat:
      G(x,F)
      where *1
      funct G ≡ (nat x, nat z) nat:
            if x≠0
            then G((x-1), (z*x))
            else z
            fi

            wobei ein Wert für F noch gefunden werden muß.
```

Ferner ist bei der Fakultätsfunktion der else - Teil, H(x), konstant. Daher wird nach Regel8 für F dieser konstante Wert eingesetzt und *1 gelöscht. Die Fakultätsfunktion hat schließlich die folgende repetitive Form:

```
funct FAK ≡ (nat x) nat:
      G(x,1)
      where
      funct G ≡ (nat x, nat z) nat:
            if x≠0
            then G((x-1), (z*x))
            else z
            fi
```

4.4.1.3 Auflösung durch Funktionsumkehr

Wir überspringen jetzt sowohl die Regeln für die Um-Klammerung als auch die für die Operandenvertauschung. Begonnen wird daher mit Regel10.
Alle Voraussetzungen sind erfüllt, da die Umkehrfunktion für die Subtraktion die Addition ist. Es kann also wie folgt transformiert werden (Muster M3):

```
funct FAK ≡ (nat x) nat:
      R(x0, H(x0))
      where *1
      funct R ≡ (nat y, nat z) nat:
            if (y≠x)
            then R((y+1), ((y+1)*z))
            else z
            fi
```

x_0 kann aus der Terminierungsbedingung bestimmt werden, und $H(x_0)$ ist die Konstante 1. Daher gilt nach Regel11:

```
funct FAK ≡ (nat x) nat:
      R(0,1)
      where
      funct R ≡ (nat y, nat z) nat:
            if (y≠x)
            then R((y+1), ((y+1)*z))
            else z
            fi
```

4.4.1.4 Auflösung durch Funktionsumkehr unter Verwendung eines stacks

Die erste für diese Auflösung zuständige Regel ist Regel13.
Da jede linear rekursive Funktion unter Verwendung eines stacks in eine repetitive Form überführt werden kann, wird ohne einschränkende Bedingungen in eine Form entsprechend Muster M5 transformiert:

```
funct FAK ≡ (nat x) nat:
      R(P(x,empty))
      where
      funct P ≡ (nat x, stack nat sn)(stack nat, nat):
            if x≠0
            then P((x-1), sn ∝ x)
            else (sn, 1)
            fi
      funct R ≡ (stack nat sy, nat z) nat:
            if sy ≠ empty
            then R(rest sy, ((top sy)*z,)
            else z
            fi
```

4.4.2 Transformation der '91-Funktion' von Manna

```
funct F91 ≡ (nat x) nat:
      if x ≤ 100
      then F91(F91(x+11))
      else (x-10)
      fi
```

Zunächst wird in Regel14 abgeprüft, ob die Eigenschaft 0, d.h. $\forall x \in \lambda$: B(g(x)) => B(x), gilt.

λ	= nat	B(x)	= x≤100
g(x)	= x-10	B(g(x))	= (x-10)≤100

Wenn (x-10)≤100 gilt, dann soll daraus folgen, daß auch x<100 gilt. Diese Bedingung ist offensichtlich nicht erfüllt. Daher bleibt Wert(0) unverändert gleich FALSE.

Für die Anwendung von Regel15 ist die Eigenschaft 1 Voraussetzung, d.h. $\forall x \in \lambda$: g(f(x)) = f(g(x)).

f(x)	= x+11	f(g(x))	= (x-10)+11 = x+1
g(x)	= x-10	g(f(x))	= (x+11)-10 = x+1

Die Äquivalenz ist offensichtlich, damit wird Regel15 ausgeführt, und Wert(1) wird auf TRUE gesetzt. Regel16 kommt nicht zur Anwendung, da bereits die Bedingung Wert(0) = TRUE nicht erfüllt ist. Regel17 fragt Wert(1) = TRUE ab. Diese Bedingung ist erfüllt. Nun kommt noch die Eigenschaft 2 hinzu, d.h. $\forall x \in \lambda$: B(g(f(x))) => B(x).

Wenn (x+1)≤100 gilt, dann soll daraus folgen, daß auch x≤100 gilt. Diese Implikation ist offensichtlich immer wahr. Damit gilt auch Eigenschaft 2, und Regel17 wird ausgeführt, d.h. Wert(2) wird auf TRUE gesetzt.

Regel18 setzt voraus, daß Wert(1) und Wert(2) auf TRUE gesetzt wurden (das trifft in diesem Beispiel zu), und prüft zusätzlich noch die Eigenschaft 3 ab, d.h. $\forall x \in \lambda$ gilt: B(x) => B(g(x)).

Wenn x≤100 gilt, dann soll daraus folgen, daß auch (x-10)≤100 gilt. Auch diese Bedingung ist für alle x in unserem Beispiel erfüllt, womit Regel18 ausgeführt werden kann. Wert(3) wird auf TRUE gesetzt, und die Funktion F91 kann in ihre endgültige Fassung transformiert werden.

```
funct F91 ≡ (nat x) nat:
      if x≤100
      then F91((x+11)-10)
      else (x-10)
      fi
```

Auch hier würde ein angeschlossener Simplifier ((x+11)-10) in (x+1) verkürzen (siehe Kapitel 7).

4.4.3 Transformation der 'Fibonaccifunktion'

```
funct FIBO ≡ (pnat x) pnat:
      if x>2
      then (FIBO(x-2) + FIBO(x-1))
      else 1
      fi
```

Zunächst fragt Regel27 ab, ob für alle K_i Umkehrfunktionen existieren.

$$K_1(x) = x-2 \quad \text{und} \quad K_2(x) = x-1$$

Offensichtlich existieren beide Umkehrfunktionen, und der Benutzer des Systems kann sie sofort angeben:

$$\bar{K}_1(x) = x+2 \quad \text{und} \quad \bar{K}_2(x) = x+1.$$

Dementsprechend wird transformiert und kein Parameter muß gekellert werden.

```
funct FIBO ≡ (pnat x) pnat:
      if x>2
      then (pnat x1) ≡ (x-2);   pnat z1 ≡ FIBO(x-2);
           (pnat x2) ≡ (x-1);   pnat z2 ≡ FIBO(x-1);
           (z1 + z2)
      else 1
      fi
```

Diese Form der Fibonaccifunktion ist nicht entflochten, da weder der Parameter x noch die Hilfsvariable z_1 der Entflochtenheitsbedingung (siehe Kapitel 4.3.2) genügen.
Es wird daher in eine Form transformiert, die den Parameterwert, mit dem die Funktion aufgerufen wird, als Resultat mit abliefert (Einbettung).

```
funct FIBO ≡ (pnat x) pnat:
      b
      where FIBO*(x) ≡ (pnat a, pnat b)
      funct FIBO* ≡ (pnat x) (pnat, pnat):
            if x>2
            then  pnat (x1)          ≡ x-2
                 (pnat y1, pnat z1) ≡ FIBO*(x1);
                  pnat (x2)          ≡ x-1;
                 (pnat y2, pnat z2) ≡ FIBO*(x2);
                  (x, z1 + z2)
            else (x, 1)
            fi
```

Hier gilt offensichtlich $x-2 = y_1$ und $x-1 = y_2$.
Das oben angegebene Muster kann daher unter Verwendung der bereits bekannten Umkehrfunktionen in eine bezüglich des Parameters x entflochtene Form transformiert werden.

```
funct FIB0 ≡ (pnat x) pnat:
      b
      where FIB0*(x) ≡ (pnat a, pnat b)
      funct FIB0* ≡ (pnat x) (pnat, pnat):
            if x>2
            then  pnat(x1)            ≡ x-2;
                  (pnat y1, pnat z1)  ≡ FIB0*(x1);
                   pnat(x2)           ≡ x-1;
                  (pnat y2, pnat z2)  ≡ FIB0*(x2);
                  (K̄2(y2), (z1 + z2))
            else (x, 1)
            fi
```

Für z_1 muß ein stack eingeführt werden, da keine Möglichkeit besteht mit einer Umkehrfunktion zu arbeiten.

```
funct FIB0 ≡ (pnat x) pnat:
   b
   where (pnat a, stack pnat sb, pnat b) ≡ FIB0*(x, empty)
   funct FIB0* ≡
         (pnat x, stack pnat sz) (pnat, stack pnat, pnat):
      if x>2
      then (pnat x1, stack pnat sr1) ≡ (x-2, sz);
           (pnat y1, stack pnat sz1, pnat z1) ≡ FIB0*(x1, sr1);
           (pnat x2, stack pnat sr2) ≡ (x-1, sz & z1);
           (pnat y2, stack pnat sz2, pnat z2) ≡ FIB0*(x2, sr2);
           (x, sz, (z1 + z2))
      else (x, sz, 1)
      fi
```

Da nun $\bar{K}_1(x)=x+2$ und $\bar{K}_2(x)=x+1$ existieren , kann diese Funktion in eine vollständig entflochtene Form überführt werden:

```
funct FIB0 ≡ (pnat x) pnat:
   b
   where (pnat a, stack pnat sb, pnat b) ≡ FIB0*(x, empty)
   funct FIB0* ≡
         (pnat x, stack pnat sz) (pnat, stack pnat, pnat):
      if x>2
      then (pnat x1, stack pnat sr1) ≡ (x-2, sz);
           (pnat y1, stack pnat sz1, pnat z1) ≡ FIB0*(x1, sr1);
           (pnat x2, stack pnat sr2) ≡ (K2(K̄1(y1)), sz1 & z1);
           (pnat y2, stack pnat sz2, pnat z2) ≡ FIB0*(x2, sr2);
           (K̄2(y2), rest sz2, (top sz2 + z2))
      else (x, sz, 1)
      fi
```

Damit haben wir eine linear verschränkt geschachtelte Funktion erhalten, die nach den entsprechenden Regeln aufgelöst werden kann.

4.5 Schlußbemerkung zu den Transformationen

In diesem Kapitel wurden Transformationsschemata für lineare, geschachtelte und kaskadenartige Funktionen angegeben und deren Anwendung an einigen Beispielen verdeutlicht. Damit sei der Abschnitt über Transformationen und Schemata abgeschlossen. Der Vollständigkeit halber soll jedoch noch auf einige Artikel hingewiesen werden, die sich mit diesem Thema beschäftigen.

Unter dem Titel 'The equivalence of certain computations' veröffentlichte D.C. Cooper 1966 [Coop 66] einen Artikel, in dem rekursive Definitionen unter Verwendung bedingter Ausdrücke mit iterativen verglichen werden und ein allgemeines Äquivalenztheorem aufgestellt und bewiesen wird. In diesem Artikel finden sich bereits die Grundideen zur Um-Klammerung und zur Operandenvertauschung.

R.S. Bird verfaßte 1977 zwei Papiere mit den Titeln 'Notes on recursion elimination' [Bird 77A] und 'Improving programs by the introduction of recursion' [Bird 77B]. Der letztere Artikel beschreibt die Programmtransformationstechnik 'recursion introduction', die auf zwei Algorithmen, die Pattern-Match-Probleme lösen, angewandt wird. Dabei werden Algorithmen, die stacks benutzen, zunächst in rekursive Rechenvorschriften transformiert, und anschließend werden diese Rekursionen eliminiert (tabulation), um effizientere Abläufe zu erhalten. Der erste Artikel von Bird befaßt sich mit der Transformation kaskadenartiger Rekursionen in iterative Formen unter Verwendung eines oder mehrerer stacks. Entsprechende Beispiele sind das Traversieren von Bäumen und Sortieralgorithmen. Ein ähnlicher Ansatz findet sich bereits 1973 bei Haskell in dem Artikel 'Efficient implementation of a class of recursively defined functions' [Hask 73], wobei jedoch hauptsächlich die Möglichkeit der Einschränkung der verwendeten stacks auf eine feste Größe untersucht wird.

Ein weiteres Papier von Bird mit dem Titel 'Recursion elimination with variable parameters' [Bird 78] beschäftigt sich mit dem Problem der Parameterübergabe bei der Auflösung von Rekursionen. Als Beispiele werden iterative Versionen des Aufbauens eines balancierten Baumes in den Programmiersprachen ALGOL 68 und PASCAL gegeben.

Tuero Hikita setzte sich 1979 in dem Artikel 'On a class of recursive procedures and equivalent iterative ones' [Hiki 79] mit wechselseitig rekursiven Prozeduren und der Transformation der rekursiven Lösung des 'Türme-von-Hanoi'-Problems auseinander. Dieses Problem wird von Hikita in Anlehnung an die bereits 1976 verfaßte Veröffentlichung von H. Partsch und P. Pepper 'A family of rules for recursion removal' [PaPe 76] behandelt.

Aus dem Jahre 1977 stammt ein Artikel von J.S. Rohl 'Converting a class of recursive procedures into non-recursive ones' [Rohl 77]. Weiter entwickelt hat Rohl diesen Ansatz 1981 in seinem Papier 'Eliminating recursion from combinatoric procedures' [Rohl 81]. Eine dritte Veröffentlichung von diesem Autor beinhaltet die Transformation linear rekursiver Definitionen in iterative: 'The elimination of linear recursion: a tutorial' [Rohl 80].

Dieses ist nur eine kleine Auswahl aus der unüberschaubaren Menge von Artikeln über Rekursionen, deren Elimination und Optimierung. Auch die in diesem Kapitel gegebenen Regeln enthalten nicht alle Ergebnisse, da das Ziel zunächst war, ein System zu entwerfen, das die Muster gängiger rekursiv definierter Funktionen in Klassen einteilt, und für diese Klassen die Möglichkeiten der Transformation in repetitive Formen darzustellen. Zukünftigen Untersuchungen bleibt es überlassen, dieses Regelsystem zu erweitern und zu verfeinern unter Berücksichtigung entsprechender Effizienzvergleiche.

5. Beweise für die Korrektheit der verwendeten Transformationsschemata

Für die Beweise werden zwei Definitionen gebraucht:

Definition 1

$N: \lambda \rightarrow N_0 \cup \{\Leftrightarrow\}$
$N(x) := \min\{n \mid B(K^n(x)) = \text{false}\}$
$N(x) = \{\Leftrightarrow\}$, falls kein solches n existiert.
N(x) bezeichnet also die Rekursionstiefe, in der die Bedingung zum ersten Mal nicht mehr erfüllt ist.

Definition 2

Zwei Funktionen L und L' erfüllen die Eigenschaft Eig(n), wenn gilt:

$\forall x \in \lambda: \quad N(x) = n \Rightarrow L(x) = L'(x)$

5.1 Beweise für das linear rekursive Muster

5.1.1 Um-Klammerung

Zu zeigen ist: $\forall n \in N_0$ gilt Eig(n)

Induktionsanfang: n = 0 trivial
n = 1 trivial

Induktionsvoraussetzung: Eig(n-1) gilt für ein $n \geq 1$

Induktionsschluß: Sei $x \in \lambda$ mit $N(x) = n$.

Nach dem linear rekursiven Grundmuster gilt (wenn B(x) = true):

$L(x) = phi(L(K(x)), E(x))$
$= phi(G(K^2(x), E(K(x)), E(x))$ * Induktionsvoraussetzung

Nach dem transformierten Muster gilt:

$L(x) = G(K(x), E(x))$
$= G(K^2(x), psi(E(K(x)), E(x)))$ ** Definition von G

Die Gleichheit von * und ** ergibt sich aus dem folgenden Lemma.

Lemma 1:

$\forall r,s,t \in \lambda \quad \forall n \in N_0$ gilt:
$N(r) = n \Rightarrow G(r,psi(s,t)) = phi(G(r,s),t)$
und beide Berechnungen terminieren in n Schritten

Beweis für Lemma 1:

Induktionsanfang: $\forall x \in \lambda \quad N(x) = 0 \Rightarrow$

G(r, psi(s,t))	= phi(H(r), psi(s,t))	nach Definition von G
	= phi(phi(H(r),s),t)	Seitenbedingung des Transformationsschemas
	= phi(G(r,s),t)	nach Definition von G

N(x) = 1 analog

Induktionsvoraussetzung:
Lemma 1 gilt für $\forall$ r,s,t $\in \lambda$ mit N(r) = n-1 (n≥2)

Induktionsschluß:

G(r, psi(s,t))	= G(K(r), psi(E(r), psi(s,t)))	nach Definition von G
	= phi(G(K(r), E(r)), psi(s,t))	Induktionsvoraussetzung da N(K(r)) = n-1
	= phi(phi(G(K(r), E(r)), s), t)	Seitenbedingung des Transformationsschemas
	= phi(G(K(r), psi(E(r), s), t)	Induktionsvoraussetzung, da N(K(r)) = n-1
	= phi(G(r,s), t)	nach Definition von G

Bemerkung:
Die Behauptung der simultanen Termination wird durch die Fälle N(x) = 0 und N(x) = 1 bewiesen.

5.1.2 Operandenvertauschung

Induktionsanfang: Für $x \in \lambda$ und N(x) = 0 ist die Äquivalenz offensichtlich.

Induktionsvoraussetzung: Gelte Eig(0),...,Eig(n-1) für ein n≥1

Induktionsschluß:

Aus dem linearen Schema ergibt sich

L(x)	= phi(L(K(x), E(x))	
	= phi(G(K(x), co), E(x)) *	nach Induktionsvoraussetzung, da N(K(x))=n-1

Aus dem transformierten Schema ergibt sich

L(x)	= G(x, co)	
	= G(K(x), psi(co, E(x))) **	

Es bleibt die Äquivalenz von * und ** zu zeigen.
Dies gelingt mit den folgenden Lemmata.

Lemma 2:
(Hilfssatz für den Beweis von Lemma 3)

$\forall\, n \in N$ $psi(phi^{n-1}(co, b_{n-1},\ldots,b_1,a) = phi^n(co,a,b_{n-1},\ldots,b_1)$
wobei psi wie in Kapitel 4.1.2 definiert ist, und
$phi^1(x_1, x_2) := phi(x_1, x_2)$
$phi^{n+1}(x_1,\ldots, x_{n+2}) := phi(phi^n(x_1,\ldots, x_{n+1})\ x_{n+2})$.

Beweis:

Induktionsanfang:
n=1 ist der zweite Teil der Seitenbed. der Transformation.
n=2 $psi(phi(co,b,a)) = psi(phi(co,a,b)$ nach Seitenbed. 1
$= phi^2(co,a,b)$ nach Seitenbed. 2

Induktionsvoraussetzung: Gelte die Behauptung für ein $n \geq 1$

Induktionsschluß:

$phi^n(co,a,b_{n-1},\ldots,b_1)$
$= phi(phi^{n-1}(co,a,b_{n-1},\ldots,b_2,a),b_1)$
$= phi(psi(phi^{n-2}(co,b_{n-1},\ldots,b_2,a),b_1))$ nach Ind.vorauss.
$= phi(psi(phi^{n-2}(co,b_{n-1},\ldots,b_2),a),b_1))$
$= psi(phi(phi^{n-2}(co,b_{n-1},\ldots,b_2),b_1),a))$ nach Seitenbed. 1
$= psi(phi^{n-1}(co,b_{n-1},\ldots,b_2),b_1),a))$

Lemma 3:

a) $\forall\, n \geq 1$
$\forall\, x \in \lambda$ mit $N(x) = n$
$\forall\, b_i \in \lambda$ und $\forall\, r = 1,\ldots,n$
gilt:
$phi^r(G(K^r(x),co),b_r,\ldots,b_1) = G(K^r(x),phi^r(co,b_r,\ldots,b_1))$

b) $\forall\, x \in \lambda$ gilt:
$(N(x) = \infty \Rightarrow$ beide Berechnungen terminieren nicht)

Induktionsanfang:
Sei $x \in \lambda$ mit $N(x) = n = 1$
$phi(G(K(x), co), b) = phi(co,b)$ da $N(K(x)) = 0$
$= psi(co,b)$ nach Seitenbed. 2
$= G(K(x), psi(co,b))$ da $K(x) = 0$
$= G(K(x), phi(co,b))$

Induktionsvoraussetzung: Gelte Aussage (a) für ein $n \geq 2$

Induktionsschluß:

Der Beweis für die Gültigkeit von a) für alle $x \in \lambda$ mit $N(x) = n$ erfolgt durch "Rückwärtsinduktion" über r.

r=n folgt sofort aus $N(K^n(x)) = 0$

Gelte die Behauptung für n, n-1,...., r für ein r≥2

$phi^{r-1}(G(K^{r-1}(x), co), b_{r-1},...,b_1)$

$= phi^{r-1}(G(K^r(m), psi(co, E(K^{r-1}(x))), b_{r-1},...,b_r)$
G expandiert (man beachte: $N(K^{r-1}(x))\geq 0$)

$= phi^r(G(K^r(x), co), E(K^{r-1}(x)), b_{r-1},...,b_1)$
nach Voraussetzung mit $K^{r-1}(x)$ für x und 1 für r
(man beachte: $N(K^{r-1}(x))\leq n-1$ und Seitenbed. 2)

$= G(K^r(x), phi^r(co, E(K^{r-1}(x)), b_{r-1},...,b_1)$
nach Induktionsvoraussetzung der Rückwärtsinduktion

$= G(K(K^{r-1}(x)), psi(phi^{r-1}(co,b_{r-1},...,b_1, E(K^{r-1}(x))))$
nach Lemma 2

$= G(K^{r-1}(x), phi^{r-1}(co, b_{r-1},...,b_1))$
nach Definition von G wegen $N(K^{r-1}(x))>0$

b) ist offensichtlich

Bemerkung: Mit r=1 ergibt sich aus Lemma 3 die Äquivalenz der Gleichungen * und **.

Bewiesen wurde hier nur der Fall H = const. (Regel7).
Die beiden anderen Auflösungsmöglichkeiten können analog bewiesen werden. Zu zeigen wäre nur, daß ein korrekter Endwert vorberechnet wurde, für Fall (b) - Regel8 nach der Bedingung $B(x) \equiv (x \neq x_1)$ und für Fall (c) - Regel9 durch die Vorberechnung mit Hilfe der Funktion F. Dieser Nachweis ist jedoch offensichtlich und braucht daher an dieser Stelle nicht ausführlich behandelt zu werden.

5.1.3 Funktionsumkehr

Für diesen Beweis wird das folgende Lemma gebraucht:

Lemma 4 :

$$\forall\, x \in \lambda \quad N(x) \Leftrightarrow \quad \Rightarrow \quad \forall\, i\ (0\leq i\leq N(x)) \quad R(K^i(x), L(K^i(x)), x) = L(x)$$

Beweis für Lemma 4:
(begrenzte Induktion über i=0,...,N(x))

Induktionsanfang:
i=0: $K^0(x) = x \Rightarrow R(K^0(x), L(K^0(x)), x) = L(x)$
nach Def. von R

Induktionsvoraussetzung:
$R(K^i(x), L(K^i(x)), x) = L(x)$ für ein i mit $0 \le i < N(x)$

Induktionsschluß:

$R(\bar{K}(K^i(x)), L(\bar{K}(K^i(x))), x)$
$\Rightarrow$ if $\bar{K}(K^i(x)) \neq x$
then $R(K^i(x), phi(L(\bar{K}(K^i(x)), E(K^i(x)), x)))$
else $L(\bar{K}(K^i(x)))$ nach Def. von R
fi
$=$ if $\bar{K}(K^i(x)) \neq x$
then $R(K^i(x), L(K^i(x)), x)$
else $L(\bar{K}(K^i(x)))$ $i<N(x)$ und nach Def. von L
fi
$=$ if $\bar{K}(K^i(x)) \neq x$
then $L(x)$
else $L(\bar{K}(K^i(x)))$ Induktionsvoraussetzung
fi
$= L(x)$ da $\bar{K}(K^i(x)) \neq x \equiv$ true,
denn $K^{i+1}(x) = x \Rightarrow N(x) = \infty$

Beweis der Korrektheit der Transformation

Behauptung: $R(x_0, H(x_0), x) = L(x)$

$m_0 = K^{N(m)}(m)$ $H(m_0) = L(m_0)$

$\Rightarrow * = R(m_0, H(m_0), m) = R(K^{N(m)}(m), L(K^{N(m)}(m), m)$
$= L(x)$

offensichtlich gilt:
L terminiert $\Leftrightarrow$ R terminiert
d.h. falls L nicht terminiert gilt:

a) $\forall x \in N_0$ $K^n(x) \neq x_1$
$\Rightarrow \forall n \in N_0\ \bar{K}^n(x_1) \neq x$
$\Rightarrow$ R terminiert nicht

b) R terminiert nicht und damit das gesamte zweite Muster nicht.

5.1.4 Funktionsumkehr unter Verwendung eines stacks

Die Funktionsumkehr unter Einführung eines Stapels ist in [BaWö 81] auf den Seiten 294/295 dargestellt. Die Herleitung der Transformationsregeln erfolgt dort in drei Schritten.

Schritt 1: Einbettung in gleichwertige Vorschrift mit redundantem Parameter des Typs 'stack λ'.

```
funct L ≡ (λ x)p:
      L*(x, empty)
      where
      funct L* ≡ (λ x, stack λ sx)p:
            if B(x) then phi(L*(K(x), sx & x), E(x))
                    else H(x) fi
```

Dies liefert mit $K^*(x, sx) = (K(x), sx \& x)$ die formale Voraussetzung dafür, die Funktionsumkehr (ohne stack) anzuwenden, denn es gilt:

$\overline{K}^*(K^*(x, sx)) = (x, sx)$ wobei
$\overline{K}^*(x, sx) = (top\ sx, rest\ sx)$

Schritt 2: Anwendung der Funktionsumkehr ohne stack auf L*

Schritt 3: Vereinfachung des Ergebnisses

5.2 Beweise für die geschachtelt rekursiven Muster

5.2.1 einfach geschachtelt, Schachtelungstiefe n=2

5.2.1.1 Grundauflösung, ohne Nebenbedingung

Der Beweis für die Äquivalenz der beiden Muster bei der Transformation der einfach geschachtelten Rekursion mit der Schachtelungstiefe n=2 in eine repetitive Form (Grundauflösung) ist als Grenzfall enthalten in dem Beweis für die Äquivalenz der beiden Muster bei der Transformation der einfach geschachtelten Rekursion mit der Schachtelungstiefe n≥2 in eine repetitive Form (siehe Beweis 5.2.2).

5.2.1.2 Auflösung unter Eigenschaft 0, Überführung in eine lineare Rekursion

Diese Transformation kann unter der folgenden Bedingung durchgeführt werden:

$$\forall\, x \in \lambda:\ B(g(x)) \Rightarrow B(x)$$

$$D := \{x \in \lambda \mid N(x) < \infty\}$$

Es ist offensichtlich, daß aufgrund der oben angegebenen Bedingung $\forall\, x \in D:\ F(x) = g^{n+1}(f^n(x))$ sowohl für das geschachtelt rekursive als auch für das linear rekursive Muster des Transformationsschemas gilt.

Für alle x, die nicht in D liegen, terminiert keine der beiden Formen.

5.2.1.3 Auflösung unter Eigenschaft 0 und 1

Der Rekursionsverlauf habe die folgende Situation ergeben:

$$F(x) = F^n f^{n-1}(x) \text{ für ein } n \geq 1$$

und es gelte: $B(f^{n-1}(x)) = \text{false}$, wobei $f^{n-1}(x) = y$

Angenommen, $B(g^k(y)) = \text{true}$ für ein $k \geq 1$

$\Rightarrow B(y) = \text{true}$ (durch mehrfache Anwendung von Eigenschaft 0)
$\Rightarrow$ Widerspruch

Folglich gilt $\forall\, k \geq 1\ B(g^k(y)) = \text{false}$:

$$\begin{aligned} F(x) &= F^n f^{n-1}(x) \\ &= F^n(y) \\ &= F^{n-1}(g(y)) && \text{da } B(y) = \text{false} \\ &\ldots\ldots\ldots\ldots\ldots && \ldots\ldots\ldots\ldots\ldots\ldots \\ &= g^n(y) && \text{da } B(g^{n-1}(y)) = \text{false} \\ &= g^n f^{n-1}(y) \end{aligned}$$

Die Termination von F(x) ist also äquivalent zu:

$$\exists\, k \geq 0 \quad B(f^k(x)) = \text{false}$$

Berücksichtigt man noch die Eigenschaft 1, dann ist wegen der trivialen Struktur von K offensichtlich $K(x,x) = F(x)$ erfüllt.

5.2.1.4 Auflösung unter Eigenschaft 1,2 und 3

1.) Da im Rekursionsverlauf von F für jedes neu erzeugte 'F' später 'g' eingesetzt werden muß, und stets die Anzahl der F's um 1 größer ist als die Anzahl der f's, gilt offensichtlich wegen Eigenschaft 1:

$F(x) = g^{m+1}(f^m(x))$ für ein $m \geq 0$ oder F(x) terminiert nicht.

Und unmittelbar einzusehen ist:

$G(x) = g^{N+1}(f^N(x))$ für ein $N \geq 0$ oder G(x) terminiert nicht.

Jedes Zwischenresultat der Rekursion von F(x) hat die Gestalt:

(*) $F(x) = F^{n-i+1}(g^i(f^n(x)))$ für $n \in N_0$, $i \in \{0,\ldots,n+1\}$

deshalb gilt: F terminiert <=> $\exists\, j \in N_0$, sodaß für alle Zwischenresultate der Art (*) gilt: $n \leq j$.

2.) G(x) terminiert nicht, dann gilt:

$\forall\, r \in N_0$ $B((gf)^r(x)) = true$

Angenommen, F(x) terminiert, dann hat das letzte Zwischenresultat der Rekursion die Form $F(x) = F(g^m(f^m(x)))$, bevor mit $B((gf)^m(x)) = false$ die Rekursion abgeschlossen wird, und das ist ein Widerspruch.

Also gilt: G(x) terminiert nicht ==> G(x) = F(x)

3.) G(x) terminiere, dann $\exists\, N \in N_0$: $N := \min\{n \mid B((gf)^n(x)) = false\}$

$B((gf)^N(x)) = false$ ==> $B(g^{N+1}(f^{N+1}(x))) = false$
wegen Eigenschaft 1 und Eigenschaft 2

==> $B(g^r(f^{N+1}(x))) = false$
für $r=0,\ldots,N+i$ wegen Eigenschaft 3

==> In jedem Zwischenresultat (*) der Rekursion von F gilt: $n \leq N$

==> F terminiert und $m \leq N$

Nun bleibt nur noch zu zeigen: $m \geq N$

F terminiert, dann hat das letzte Zwischenresultat der Rekursion die Form $F(x) = F(g^m(f^m(x)))$, bevor mit $B(g^m(f^m(x))) = false$ die Rekursion abgeschlossen wird.

$B(g^m(f^m(x))) = false \Longrightarrow B((gf)^m(x)) = false$ nach Eigenschaft 1

$\Longrightarrow m \geq N$

5.2.2 einfach geschachtelt, Schachtelungstiefe n≥2

Gezeigt werden soll, daß die folgende Gleichung gilt:

$$Q(x,i) = F^i(x)$$

a) Für $i \geq 1$ ergibt die Anwendung des Operators F^{i-1} auf die geschachtelte Form:

$$F^i(x) = \underline{\text{if}}\ B(x)\ \underline{\text{then}}\ F^{k+i-1}(f(x))\ \underline{\text{else}}\ F^{i-1}(g(x))\ \underline{\text{fi}}$$

$$F^0(x) = x$$

wobei k eine beliebige Schachtelungstiefe ist.

b) Insgesamt gilt dann für $i \in nat$:

```
F^i(x) = if i≠0 then if B(x)
                     then F^(k+i-1)(f(x))
                     else F^(i-1)(g(x))
                     fi
                else x
                fi
```

c) Setze $R(x,i) = F^i(x)$

Dann ergibt sich für R aus (b) :

```
R(x,i) = if i≠0 then if B(x)
                     then R(f(x), k+i-1))
                     else R(g(x), i-1)
                     fi
                else x
                fi
```

$\Longrightarrow R \equiv Q$

$\Longrightarrow Q(x,1) = F^1(x) = F(x)$

5.2.3 verschränkt geschachtelt, Schachtelungstiefe n=2

Gezeigt werden soll, daß die folgende Gleichung gilt:

$$Q(x,i) = (Fh)^{i-1}\ (F(x))$$

a) Für $i \geq 1$ ergibt die Anwendung des Operators $(Fh)^{i-1}$ auf die verschränkt geschachtelte Form:

```
(Fh)^1(x) = if B(x)
            then (Fh)^1(F(f(x)))
            else (Fh)^(i-1)(g(x))
            fi

(Fh)^0(x) = x
```

b) Insgesamt gilt dann für $i \in$ nat:

```
(Fh)^(i-1)(F(x)) = if i≠0
                   then if B(x)
                        then (Fh)^(i+1)(f(x))
                        else (Fh)^(i-1)(g(x))
                        fi
                   else x
                   fi
```

c) Sei $R(x,i) = (Fh)^{i-1}(F(x))$

Dann ergibt sich für R aus (b):

```
R(x,i) = if i≠0
         then if B(x)
              then R(f(x), i+1)
              else if i=1
                   then g(x)
                   else R(h(g(x)), i-1)
                   fi
              fi
         else x
         fi
```

d) Durch einfache Umformung ergibt sich:

```
R(x,i) = if i≠0
         then if ¬B(x) and i=1
              then g(x)
              else R( if B(x)
                      then (f(x), i+1)
                      else (h(g(x)), i-1)
                      fi
              fi
         else x
         fi
```

$\Rightarrow R \equiv Q$

$\Rightarrow Q(x,1) = (Fh)^{0}F(x) = F(x)$

5.2.4 verschränkt geschachtelt, Schachtelungstiefe n>2

$F^* : \lambda \times \text{stack}\ \{h_1,\ldots,h_n\} \longrightarrow \lambda$

$F^*(x,(h_{i_1},\ldots,h_{i_r})) := F(h_{i_1}(\ldots F(h_{i_r}(F(x>$

wobei $h_{i_1},\ldots,h_{i_r}$ der stack-Inhalt ist, mit h_{i_r} als top-Element

$F^*(x, \text{empty}) := F(x)$

Der Rekursionsverlauf von F habe folgende Situation ergeben:

$F(x) = F(h_{i_1}(\ldots(F(h_{i_r}(F(y>$

$sr := (h_{i_1},\ldots,h_{i_r})$

Dann gilt: $F^*(y, sr) = F(x)$

a) Sei nun B(y) wahr, dann gilt:

$F(x) = F(h_{i_1}\ldots\ldots(F(h_{i_r}(F(h_1\ldots(h_n(F(f(y>$
$= F^*(f(y), s_r \,\&\, h_1 \,\&\, \ldots \&\, h_n)$

b) Sei nun ¬B(y) wahr und k≥1, dann gilt:

$F(x) = F(h_{i_1}(F\ldots(F(h_{i_r-1}(F(h_{i_r}(g(y>$
$= F^*(\text{top}\ s_r(g(y)), \text{rest}\ s_r)$

c) Sei nun ¬B(y) wahr und F(x) = F(y), d.h. k=0, dann gilt:

$F^*(y, \text{empty}) = F(y) = g(y)$

Folglich gilt für Q definiert durch:

```
funct Q ≡ (λ x, stack {h1,...,hn} sx) λ:
      if B(x)
      then Q(f(x), sx & h1 &...& hn)
      else if sx ≠ empty
           then Q(top sx(g(x)), rest sx)
           else g(x)
           fi
        fi

   Q(x,empty) = F(x)
```

5.2.5 linear verschränkt geschachtelt, Schachtelungstiefe n≥2

Der Rekursionsverlauf habe eine der folgenden Situationen ergeben:

Fall 1 $F(x) = h_{i_1}(h_1^{r_1}(F\ldots(h_{i_n}(h_1^{r_n}(F(y>$ für ein n≥1

Fall 2 $F(x) = F(y)$ mit $y=x$

Fall 3 $F(x) = y$

i=0 für Fall 2
i=1 für Fall 1 und Fall 3

$s = (h_{i_1 r_1}) \,\&\, \ldots\ldots \,\&\, (h_{i_n r_n})$ für Fall 1 (d.h. falls n≥0)
s = empty für Fall 2 und Fall 3

Dann gilt allgemein: $F(x) = F^*(y,s,i)$

Gelte nun für $F^*(y,s,i)$

a) i=0, dann liegt der Fall 2 vor (x=y)
und es gelte zusätzlich

a_1) s = empty

a_{11}) B(y) = true
$F^*(y,s,i) = F(x)$
$= h_1(f(h_2.....h_n(F(f(x>$
$= F^*(f(y), s \,\&\, (h_1,0) \,\&\, \ldots \,\&\, (h_n,0)), 1)$

a_{12}) B(y) = false
$F^*(y,s,i) = F(y) = g(y)$

a_2) s ≠ empty tritt nicht auf

b) i=1, dann liegt der Fall 1 oder Fall 3 vor

b_1) s = empty, d.h. es liegt Fall 3 vor

b_{11}) B(y) = true
F*(y,empty,1) = y = F(x)

b_{12}) B(y) = false
wie b_{11}, d.h. unter b_1 hat der Wert B(x) keine Bedeutung

b_2) s ≠ empty, d.h. es liegt der allgemeine Fall 1 vor:

b_{21}) B(y) = true
$F^*(y,s,1) = h_{i_1}(h_1^{r_1}(F....h_{i_n}(h_1^{r_n}(F(y>$
$= h_{i_1}(h_1^{r_1}(F....F(h_{i_n}(h_1^{r_n}(h_1(F(h_2.....h_n(F(f(y>$
$= F^*(f(y), rest\ s\ \&\ (top_1\ s, top_2\ s + 1)$
$\&\ (h_2,0)\ \&\ ...\ \&\ (h_n,0),\ 1)$

b_{22}) B(y) = false
$F^*(y,s,1) = h_{i_1}(h_1^{r_1}(F...h_{i_k}(h_1^{r_k}(F(y>$
$= h_{i_1}(h_1^{r_1}(F...h_{i_{k-1}}(h_1^{r_{k-1}}(F(h_{i_k}(h_1^{r_k}(g(y>$
$= F^*(h_{i_k}(h_1^{r_k}(g(y))),\ rest\ s,\ 1)$

mit der Hilfsfunktion H aus dem Rekursionsmuster für Q gilt dann:

$h_{i_k}(h_1^{r_k}(g(y))) = top_1\ s(H(top_2\ s,\ g(y))$

Die '1' als drittes Argument ist für k≥2 klar, für k=1 gilt:

$F^*(y,s,1) = h_{i_1}(h_1^{r_1}(F(y)))$
$= h_{i_1}(h_1^{r_1}(g(y)))$ nach Voraussetzung
$= F^*(h_{i_1}(h_1^{r_1}(g(y))),\ empty,\ 1)$ nach Definition von F*

Damit ist gezeigt, daß F* exakt dem Rekursionsmuster von Q genügt.

Darüberhinaus gilt:

F*(x,empty,0) = F(x), womit die Identität Q ≡ F gezeigt ist.

5.3 Beweise für die kaskadenartigen Muster

5.3.1 Entflechtung, Faktor n=2

Der Beweis für die Entflechtung einer kaskadenartigen Rekursion mit dem Faktor n=2 ist in dem Beweis 5.3.2 für die Entflechtung einer kaskadenartigen Rekursion mit dem Faktor n≥2 enthalten.

5.3.2 Entflechtung, Faktor n≥2

Es seien folgende Definitionen gegeben:

$I_n := \{1,.....,n\}$
$M_0 := \{x \in \lambda \mid B(x) = false\}$
$M_1 := \{x \in \lambda \mid B(x) = true \Longrightarrow \forall\, i \in I_n \quad B(K^i(x)) = false\}$
$M_r := \{x \in \lambda \mid \forall\, i: I_{r-1} \to I_n \quad \forall\, m \in I_n \quad B(K^{i_1}...K^{i_{r-1}}(x)) = true \Longrightarrow B(K^m K^{i_1}...K^{i_{r-1}}(x)) = false\}$

n=0 ==> $F^*(x, sz) = (x, sz, H(x))$
$F(x) = H(x) = PI_3\,(F^*(x, sz))$

wobei PI_3 die 3.Projektion sei.

n-1 => n $\quad \forall\, x \in M_n$

falls B(x) = false wie n=0

falls B(x) = true:
folgende Zusammenhänge im Rekursionsmuster von F* sind offensichtlich:

$sr_1 = sz$
$sz_i = sr_i \qquad i=1,...,n$
$sr_{i+1} = sz_i \;\&\; z_i \qquad i=1,...,n-1$
$y_i = K_i(x)$, d.h. $\bar{K}_i(y_i) = x$
$x_i = K_i(x)$

Daraus ergibt sich, daß nach Durchlauf des then-Teils gilt:

$sz_n = sz \;\&\; z_1 \;\&\; ... \;\&\; z_{n-1}$

und die relevante dritte Komponente der then-Teil-Berechnung lautet deshalb:

$phi(z_1,...,z_{n-1}, z_n, E(x))$.

$x_i = K^i(x) \in M_{n-1}$ und deshalb gilt:

$z_i = PI_3\, F^*(x_i, sr_i) = F(x_i) = F(K^i(x))$ nach Induktionsvoraussetzung

Folglich ergibt die Berechnung des then-Teils:

$phi(F(K^1(x)),.....,F(K^n(x)), E(x))) = F(x)$

Ist x nicht Element aus der Vereinigung aller M_i mit $0 \leq i \leq \infty$, dann haben die Rekursionsablaufbäume von F und F* offensichtlich unendliche Tiefe, d.h. F und F* terminieren nicht bei Anwendung auf x bzw. (x,empty).

---

6. Die unfold/fold-Methode

6.1 Allgemeine Beschreibung

Die unfold/fold-Methode wurde von Darlington und Burstall [DaBu 77] entwickelt und dient dazu rekursive Programme, die in Form von Rekursionsgleichungen 1.Ordnung geschrieben sind, in effizientere Formen zu transformieren. Darlington und Burstall gingen davon aus, daß Programmierer zunächst verständliche Programme schreiben, die dann durch Umformung in effizientere transformiert werden, jedoch auf Kosten der Transparenz. Das Problem dabei ist, diesen Prozeß systematisch vornehmen zu können, um jedes rekursive Programm auf die Möglichkeit der Effizienzsteigerung durch Transformationsmethoden untersuchen zu können.

Ausgehend von einem Programmsystem, das versucht jegliche Rekursion durch eine Iteration zu ersetzen und u.a. redundante Berechnungen zu eliminieren, versuchten Darlington und Burstall weitere Transformationsschemata zu finden, die eine Effizienzsteigerung erwarten lassen. Diese Überlegungen beruhen hauptsächlich auf semantischen Eigenschaften der zu transformierenden Programme. Diese Eigenschaften lassen sich nicht grundsätzlich automatisch feststellen, sodaß das implementierte System von Darlington und Burstall halbautomatisch funktioniert, wobei gewisse Anfragen an den Benutzer gemacht werden. Die vom Benutzer gegebenen Informationen führen dann zu weiteren Transformationen, und einige Beispiele zeigen, daß diese Methode zumindest ein guter Ansatz dafür ist, allgemeine Methoden der Effizienzsteigerung rekursiver Programme zu entwickeln.

Das unfold/fold-Modell beinhaltet verschiedene Transformationsregeln. Zunächst - wie der Name schon sagt - das 'unfolding' und das 'folding'. 'Unfolding' bedeutet im übertragenen Sinne das 'Expandieren' einer Seite einer Rekursionsgleichung, 'folding' entsprechend das 'Reduzieren'. Diese Art der Reduktion wurde auch schon von Manna und Waldinger in ihrer Arbeit über Programmsynthese [MaWa 75] unabhängig von Darlington und Burstall entwickelt. Auch in späteren Anwendungen der unfold/fold Methode, z.B. bei L. Schmitz [Schm 78], wird dieser Vorgang unter dem Begriff Programmsynthese weiterentwickelt. Einige dieser Anwendungsbeispiele sollen jedoch später noch detaillierter beschrieben werden.

Um die von Darlington und Burstall entwickelten Transformationsalgorithmen verständlich machen zu können, brauchen wir zunächst die folgenden Begriffsdefinitionen, die durch ein anschließendes Beispiel verdeutlicht werden sollen:

Instanziierung. Dabei wird eine Substitution einer bestehenden Gleichung erzeugt, indem Werte eingesetzt (Variablen instanziiert) werden.

Expansion (Unfolding). Wenn E<=E' und F<=F' Gleichungen sind, und eine Instanz von E in F' auftritt , dann ersetzt man dieses Auftreten der Instanz von E durch die entsprechende Instanz von E', woraus sich F'' ergibt, und fügt die Gleichung F<=F'' den bisher gegebenen Gleichungen hinzu.

Reduktion (Folding). Wenn E<=E' und F<=F' Gleichungen sind, und eine Instanz von E' in F' auftritt , dann ersetzt man dieses Auftreten einer Instanz von E' durch die entsprechende Instanz von E, woraus sich F'' ergibt, und fügt die Gleichung F<=F'' den bisher gegebenen Gleichungen hinzu.

Definitionen. Hierbei wird eine neue Rekursionsgleichung eingeführt, deren linke Seite keiner Instanz einer linken Seite irgendeiner vorhergehenden Gleichung ist.

Abstraktion. Hierbei wird eine where-Clause eingeführt durch Ableitung einer neuen Gleichung aus einer vorhergehenden.
Aus E<=E' erhält man:

$$E <= E'(u_1/F_1,\ldots,u_n/F_n) \text{ where } \langle u_1,\ldots,u_n\rangle = \langle F_1,\ldots,F_n\rangle$$

Gesetze. Gleichungen können transformiert werden, indem spezielle mathematische Gesetze wie Assoziativität, Kommutativität usw. angewendet werden.

Die ersten drei und die letzten beiden Transformationen sind einfach zu mechanisieren. Lediglich die vierte Möglichkeit birgt einige Hindernisse. Hierin liegt der eigentlich heuristische Teil der unfold/fold-Methode, der vom Benutzer unterstützt werden muß, obgleich auch die Reihenfolge der Anwendung aller (anderen) Transformationsschritte heuristisch gesteuert wird. Die Erzeugung einer solchen Definition kann auch nicht gleichgesetzt werden mit einer Transformation im üblichen Sinne, sondern wird mehr im Sinne einer Hilfsgleichung benutzt.

Ein Beispiel, in dem alle diese Schritte (bis auf die Anwendung algebraischer Gesetze) vorkommen, ist die Transformation eines einfachen Rekursionsschemas für die Berechnung der Fibonaccizahlen.

```
1. f(0)    <= 1
2. f(1)    <= 1
3. f(x+2)  <= f(x+1) + f(x)                          gegeben

4. g(x)    <= <f(x+1), f(x)>                  Definition (eureka)
5. g(0)    <= <f(1), f(0)>                    Instanziierung von 4
           <= <1,1>                           Expansion mit 1 und 2
6. g(x+1)  <= <f(x+2), f(x+1)>                Instanziierung von 4
           <= <f(x+1) + f(x), f(x+1)>         Expansion mit 3
           <= <u+v,u> where
                      <u,v>=<f(x+1), f(x)>    Abstraktion
           <= <u+v,u> where <u,v>=g(x)        Reduktion mit 4
7. f(x+2)  <= u+v where
                 <u,v>=<f(x+1), f(x)>         Abstraktion von 3
           <= u+v where <u,v>=g(x)            Reduktion mit 4
```

Die neue Definition der Berechnung der Fibonaccizahlen sieht nun wie folgt aus:

```
f(0)   <= 1
f(1)   <= 1
f(x+2) <= u+v where <u,v>=g(x)
g(0)   <= <1,1>
g(x+1) <= <u+v,u> where <u,v>=g(x)
```

Nach dieser Definition lassen sich die Fibonaccizahlen in einer Zeit berechnen, die linear mit x wächst und nicht mehr exponentiell.

Für die Anwendungsstrategie der oben genannten Regeln (Transformationsschritte) geben Darlington und Burstall (DaBu 77) zunächst einen Grundalgorithmus an, der dann unter Verwendung der Ergebnisse einiger Testläufe weiterentwickelt wird.

Algorithmus 1 (Grundalgorithmus)

1. Stelle alle notwendigen Definitionen auf.
2. Bilde Instanzen.
3. Expandiere jede Instanz wiederholt.
4. Nach jeder Expansion versuche algebraische Gesetze (wie Assoziativität, Kommutativität usw.) anzuwenden und where - Abstraktionen durchzuführen.
5. Reduziere wiederholt.

Wie an diesem Algorithmus deutlich wird, sind das Expandieren und das Reduzieren zunächst die einzigen vollständig mechanisierbaren Schritte. Die Schritte 1 und 2 setzt man im allgemeinen voraus. Es bleibt also Schritt 4 auf Mechanisierbarkeit zu untersuchen. Die unklaren (nicht-mechanisierbaren) Stellen werden von Darlington und Burstall mit 'eureka' gekennzeichnet, was soviel heißt wie 'ich hab's gefunden'. Damit wird angedeutet, welche Hilfestellung der Benutzer dem System einstweilen geben muß.

Beim Experimentieren mit Anwendungsstrategien für die Transformationsregeln machten Darlington und Burstall einige hilfreiche Beobachtungen, die sie für eine effektivere Fassung ihres Grundalgorithmus nutzten:

a) Fast alle optimierenden Transformationen bestehen aus einer Reihe von Expansionen, Umformungen und anschließenden Reduktionen,

b) Die Anwendung algebraischer Gesetze und der where - Abstraktionen kann normalerweise zurückgehalten werden bis direkt vor einen Reduktionsschritt.

Kombiniert man die Anwendung allgemeiner Gesetze und die where - Abstraktionen mit dem Reduktionsprozeß, d.h. wendet man sie nur an, wenn damit sicher die Voraussetzung für eine Reduktion geschaffen wird (forced folding), so spart man sich einige unnötige Versuche. Daraus ergibt sich der zweite Algorithmus:

Algorithmus 2 (1. erweiterte Fassung des Grundalgorithmus)

1. Stelle alle notwendigen Definitionen auf.
2. Bilde Instanzen.
3. Führe eine Expansion oder eine Umformung durch.
   Wiederhole diesen Schritt beliebig oft.
4. Führe eine beliebige erzwungene Reduktion durch.
   Wiederhole Schritt 4 solange bis keine Reduktion mehr möglich ist.

Eine weitere Beobachtung führt zum dritten Algorithmus:

c) falls sicher ist, daß das gegebene Gleichungssystem nicht uneingeschränkt expandiert werden kann, kann das Reduzieren zurückgestellt werden, bis alle möglichen Expansionen gemacht worden sind.

Algorithmus 3 (2. erweiterte Fassung des Grundalgorithmus')

1. Stelle alle notwendigen Definitionen auf.
2. Bilde Instanzen.
3. Expandiere jede Gleichung bis keine weitere Expansion mehr möglich ist.
4. Für jede Instanz einer Gleichung, die verbessert werden soll
   a) expandiere solange bis keine Expansion mehr möglich ist,
   b) forme um (allgemeine Gesetze , where - Abstraktionen) und gehe zu Schritt c oder Schritt 3,
   c) führe irgendeine beliebige erzwungene Reduktion durch, wiederhole Schritt c solange bis keine Reduktion mehr möglich ist.

Diese beiden zuletzt genannten Algorithmen sind von Darlington und Burstall in einem System implementiert worden [DaBu 76]. Auf diesem Stand der Entwicklung wurden dem Benutzer noch folgende Angaben abverlangt:

a) Liste der Gleichungen mit allen notwendigen Definitionen (eurekas),
b) Relevante Umformungsgesetze,
c) Liste aller geeigneten Instanzen linker Seiten von Gleichungen, auf denen das System arbeiten soll.

Für die zukünftige Entwicklung des Systems sahen Darlington und Burstall als wichtigsten Punkt die Automatisierung der Erzeugung neuer Definitionen. Eine Idee zur Lösung dieses Problems ist die Generierung des Berechnungsraumes durch alle Gleichungen bis auf eine bestimmte Stufe und ein anschließender Vergleich von Knoten dieses Baumes. Werden vergleichbare Knoten dieses Baumes gefunden, so wird nach einer Substitution gesucht, die angewandt auf den im Baum weiter unten stehenden Knoten den der Wurzel näheren ergibt (zum Vergleich siehe das System ZAP [Feat 79]).

Bisher enthält das System lediglich eine Matchroutine, die überprüft, ob eine Reduktion durchgeführt werden kann. Diese vergleicht zwei Ausdrücke und versucht eine Substitution zu finden, die den einen Ausdruck in den anderen überführt. Dabei ist unter anderem eingeschlossen das Erkennen des Schlusses von n auf n+1, das Erkennen von Formen, die durch die Anwendung des Assoziativ- bzw. Kommutativgesetzes ineinander überführbar sind, sowie das Matchen von where - Abstraktionen.

Ferner sollten sich zukünftige Untersuchungen des Systems darauf richten, den Matcher so zu erweitern, daß er automatisch Hilfsfunktionen erzeugt.

Eine der interessantesten Fähigkeiten des Systems ist das Konvertieren einer Rekursion in eine iterative Form (siehe Beispiel - Transformation der Berechnung der Fibonaccizahlen).

Eine Menge von Funktionsdefinitionen $f_1,...,f_r$ heißt iterativ, wenn für jede Gleichung $f_i(x_1,...,x_n)<=E$ entweder E kein f_i enthält, oder sie die Form $f_k(E_1,...,E_n)$ hat und $E_1,..,E_n$ kein f_i enthält, oder ein bedingter Ausdruck vorliegt, dessen Alternativen einer dieser Formen entsprechen. Jede iterative Definition läßt sich trivialerweise als Schleife schreiben.

Das System von Darlington und Burstall kann die Transformation einer rekursiven in eine iterative Form nicht automatisch durchführen, da bei jedem Schritt eine neue Definition eingeführt werden muß. Alle Definitionen haben aber ein ähnliches Muster und sind Verallgemeinerungen der ursprünglichen Funktionsgleichungen.

Als Beispiel betrachte man die Verallgemeinerung der Fakultätsfunktion.
Gegeben sei die folgende Definition:

```
1. fact(0)   <= 1
2. fact(n+1) <= (n+1) * fact(n)
```

Eingeführt werden soll durch Verallgemeinerung von n+1 auf n eine neue Funktion f:

```
3. f(n,u)    <= u * fact(n)              Definition (eureka)
4. f(0,u)    <= u                        instanziiere, expandiere
5. f(n+1,u)  <= u * ((n+1) * fact(n))    instanziiere, expandiere
             <= f(n,u * (n+1))           Assoziativität von *
                                         und Reduktion mit 3
6. fact(n+1) <= f(n,n+1 )                Reduktion von 2 unter
                                         Verwendung von 3
```

Die Definition, die sich aus 1,6,4 und 5 zusammensetzt, ist iterativ.

Um eine noch kürzere Definition zu erhalten, müßte man Gleichung 1 und 6 durch fact(n) <= f(n,1) ersetzen. Dazu braucht man eine zusätzliche Regel, die von den Autoren als 'Redefinitionsregel' bezeichnet wird.
Diese Redefinitionsregel erlaubt im wesentlichen die Umkehrung unserer bisherigen Transformationen.

Beispiel: Haben wir die folgende Definition:

```
f(0)   <= 0
f(n+1) <= f(n),
```

dann wäre eine einfachere Form:

```
f(n)   <= 0.
```

Diese ist jedoch nicht mit den bisher möglichen Transformationsregeln zu erreichen, sondern erfordert eine Umkehrung dessen, eine Redefinition. Implementiert wurde diese Regel von Darlington und Burstall nicht.

Anmerkung.
Bezüglich der Nutzung der Verallgemeinerung mit entgegengesetzter Intention, d.h. zur Transformation iterativer in rekursive Formen, sei auf Boyer/Moore [BoMo 75], Moore [Moor 75] und Aubin [Aubi 75] verwiesen.

Als noch offene Probleme, die zu weiteren Untersuchungen Anlaß geben, formulierten Darlington und Burstall 1976 im Anschluß an ihre Arbeit die folgenden Fragen:

1. Gibt es irgendwelche formalen Charakterisierungen für die Klasse von Programmverbesserungen (Effizienzsteigerungen), die unsere Transformationen umschließen?

2. Welches sind notwendige und welches hinreichende Bedingungen dafür, daß unsere Transformationen eine Effizienzsteigerung erzeugen?
   Gibt es einen allgemeinen Beweis?

Ihre weiteren Betrachtungen wollten Darlington und Burstall konzentrieren auf die Mechanisierung der Verallgemeinerung alter Definitionen zu neuen und auf das Problem, das aus der Strukturierung der Optimierung großer Programme folgt.

Die meisten Anwendungsbeispiele der unfold/fold-Methode sind in Zusammenhang mit dem Begriff Programmsynthese zu finden.

Unter Programmsynthese können wir die Übertragung einer gegebenen abstrakten Spezifikation in eine konkrete Berechnungsstruktur verstehen.

Darlington benutzte diese Technik z.B., um aus einer abstrakten Definition für das Sortieren mehrere verschiedenen konkrete Sortieralgorithmen abzuleiten [Darl 76], Schmitz [Schm 78] dagegen entwickelte auf ähnliche Weise mehrere verschiedene konkrete Algorithmen zur Berechnung der transitiven Hülle einer Relation.

Zur Ergänzung werden diese und einige weitere Beispiele für die Anwendungsmöglichkeiten der unfold/fold-Methode in den folgenden Abschnitten betrachtet, und die allgemeinen Beschreibungen sollen an dieser Stelle zunächst abgeschlossen sein.

6.2 Betrachtungen und Beispiele zur unfold/fold-Methode nach Martin S. Feather

Martin S. Feather betrachtete die unfold/fold-Methode 1979 auf einer breiten Ebene und entwickelte das ZAP-Transformationssystem [Feat 79]. Er bearbeitete damit mehrere nicht-triviale Programme, um die interessantesten und wichtigsten Gesichtspunkte eines allgemeinen Transformationssystems darzustellen.
In seiner Arbeit 'A System for Program Transformation' stellt er gut lesbare Programme mit einer modularen Struktur effizienten, aber unübersichtlichen Programmen gegenüber. Die Phasentrennung bei der Programmentwicklung wird in Abschnitt 6.2.1 anhand des Telegrammproblems aufgezeigt. In der ersten Phase wird ein lesbares jedoch uneffizientes Programm geschrieben, das sogenannte Protoprogramm, in der zweiten wird dieses in ein effizienteres transformiert, wobei die Korrektheit erhalten bleibt (bezüglich der Korrektheit von Transformationen unter Anwendung der unfold/fold-Methode siehe [Kott 82]).

Die Implementierung von Protoprogrammen ist im allgemeinen nicht eindeutig in dem Sinne, daß mehrere verschiedene effizientere Programme daraus abgeleitet werden können (siehe Programmsynthese von Sortierprogrammen bei Darlington [Darl 76]). Ein weiterer Punkt, der bei der Entwicklung eines derartigen Transformationssystems berücksichtigt werden sollte, ist, daß für die Transformation nicht mehr Arbeit aufgewendet werden sollte als für das sofortige Schreiben eines effizienten Programms nötig wäre.

Alle Programme werden von Feather in der von Darlington und Burstall entwickelten Sprache NPL geschrieben [Burst 77]. Diese Sprache ist rein funktional, d.h. sie zeichnet sich dadurch aus, daß alle ihre Objekte einen Wert liefern, sie keine Zuweisungen und Seiteneffekte enthält und der Grundprogrammiermodus rekursiv ist, was zu einer klaren und korrekten Programmierung beitragen soll.
Die geforderte Modularität der Protoprogramme dient der einfachen Aufspaltung des Gesamtproblems in Teilprobleme (siehe Telegrammproblem, Kap. 6.2.1), die simultan bearbeitet werden können.

Das ZAP-Programmtransformationssystem von Feather ist eine Erweiterung des Systems von Darlington und Burstall. Es entstand aus der Motivation, die transformatorische Methodologie zu testen und Probleme aufzudecken, die sich aus der zunehmenden Größe von Programmen ergeben. Abgesehen vom Telegrammproblem wurden ein kleiner Compiler und ein Textformatierer sowie viele kleinere Beispielprogramme transformiert.

Untersucht und beschrieben wurden von Feather unter anderem der Grad der Automation, d.h. der Umfang, in dem das System versucht, die Transformationen zu automatisieren, die zugrundeliegenden Transformations- bzw. Manipulationsmethoden, die Strukturierung der Transformationsphase, die Kommandosprache und eingebaute Standardwert-Generatoren (Default-Generatoren, siehe Kap. 6.2.4)

Ein Vergleich des Automationsgrades zeigt, daß das von Manna und Waldinger 1977 implementierte DEDALUS-System [MaWa 77] vollkommen automatisch läuft, dabei jedoch sehr restriktiv bezüglich der zulässigen Spezifikationen ist. Das System von Darlington und Burstall [DaBu 76] arbeitet mit geringer Benutzerunterstützung, wird jedoch unhandlich bei größeren Programmen, da der Benutzer überhäuft wird mit einer Vielzahl von Entscheidungen, die eine gute Kenntnis des Transformationsprozesses voraussetzen. Das von Bauer et al. 1977 ausgearbeitete CIP-System [BPPW 77] wird vollständig vom Benutzer gesteuert, somit wird auch hier der Benutzer zumindest bei größeren Programmen leicht überlastet. Und das ZAP-System ist zwischen diesen Extremen einzuordnen, da der Benutzer zwar an einigen Stellen eingreifen muß, die Anzahl dieser Hilfestellungen aber so gering wie möglich gehalten wird.
Dies wird einerseits erreicht, indem auf der Benutzerebene nur größere Transformationsschritte deutlich werden, die im Grunde aus langen Folgen einzelner Manipulationen bestehen, und somit der Benutzer nicht unnötig mit detaillierten Spezifikationen belastet wird. Andererseits stellt das System sogenannte Default-Generatoren zur Verfügung, die der Benutzer aufrufen kann, wenn er der Ansicht ist, daß die nächsten Transformationsschritte einfach genug sind, um automatisch in Angriff genommen werden zu können.

6.2.1 Das Telegrammproblem

Martin S. Feather griff 1977 die unfold/fold-Methode auf, um sie auf das Telegrammproblem anzuwenden.
Die Sprache, in der er das Telegrammproblem formulierte, ist, wie bereits erwähnt, NPL. Sie benutzt Rekursionsgleichungen, um Ausdrücke zu berechnen. Dabei dienen die linken Seiten als Muster, um Argumente zu vergleichen und Variablen zu binden und, falls dies erfolgreich war, nach der rechten Seite auszuwerten. Feather verwendet das von Darlington und Burstall 1976 entwickelte und implementierte Transformationssystem, das die drei Hauptschritte Expandieren (unfolding), Reduzieren (folding) und Anwendung relevanter algebraischer Gesetze enthält.

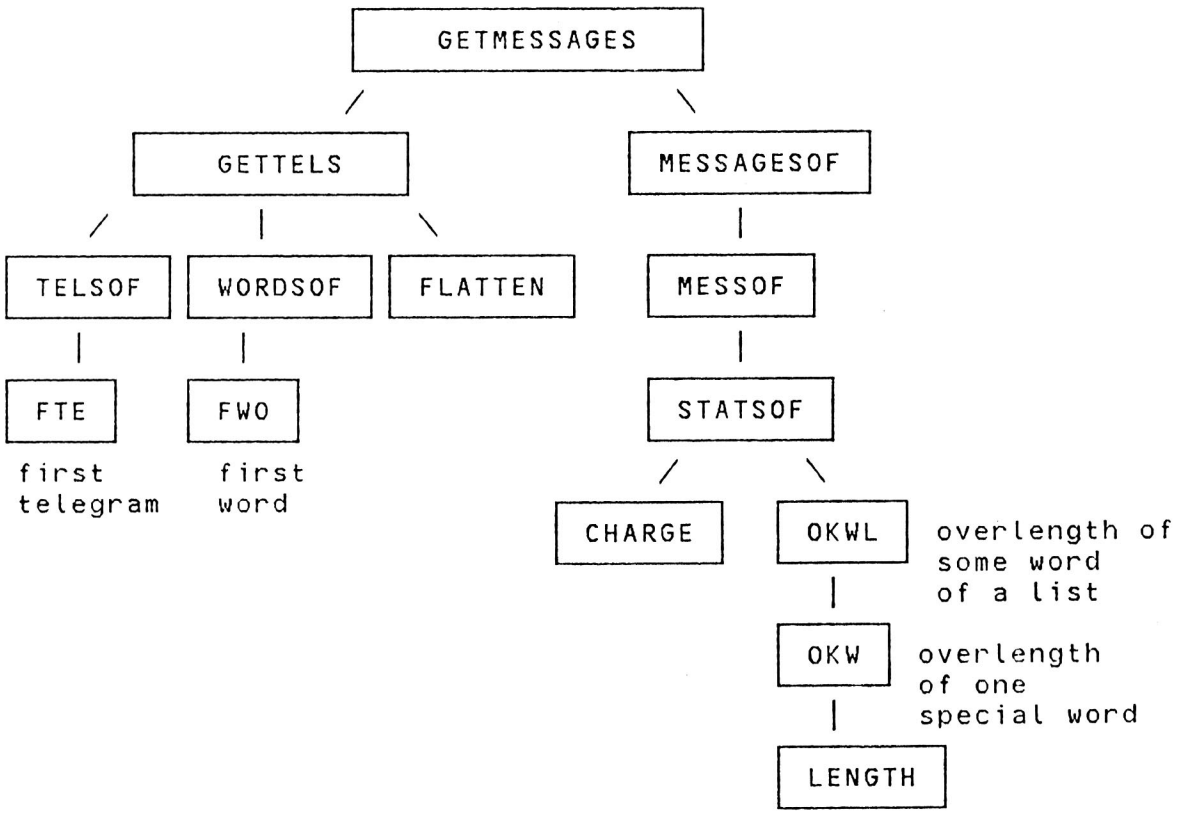
Das Telegrammproblem, das ursprünglich von Henderson und Snowdon [HeSn 72] als Programmierübung gedacht war, wird wie folgt spezifiziert:

> Es soll eine Anzahl von Telegrammen verarbeitet werden. Diese Telegramme sind als Eingabe in Form eines Musters aus Buchstaben, Ziffern und Leerzeichen verfügbar. Die einzelnen Wörter der Telegramme werden von einer Reihe von Leerzeichen getrennt, und jedes Telegramm schließt mit dem Wort "ZZZZ". Die Folge der zu bearbeitenden Telegramme wird beendet mit dem leeren Telegramm, d.h. einem Telegramm ohne Wörter.
> Bestimmt werden soll für jedes Telegramm die Anzahl der anzurechnenden Wörter, wobei das abschließende Wort "ZZZZ" nicht gebührenpflichtig ist und Wörter mit mehr als 12 Buchstaben

als überlang betrachtet werden.
Das Ergebnis der gesamten Verarbeitung soll eine ordentlich aufbereitete Liste der Telegramme sein, jedes versehen mit seiner Wortanzahl und einer Mitteilung darüber, ob überlange Wörter auftreten, und falls ja, wieviele.

Um den Umfang der Bearbeitung nicht unnötig zu vergrößern, verzichtet Feather auf die ordentliche Aufbereitung und fordert als Ergebnis lediglich eine Liste der Telegramme mit der entsprechenden Statistik.

Bei der Spezifikation werden zunächst alle Datentypdefinitionen, dann alle Funktionsdefinitionen und schließlich alle erforderlichen Rekursionsgleichungen angegeben.
Der Gesamtprozeß beginnt mit dem Aufruf der Funktion GETMESSAGES und kann durch folgendes Strukturdiagramm dargestellt werden:



Die Spezifikation von Feather entspricht einer lesbaren, aber sehr ineffizienten Lösung des Problems. Um einen effizienteren Ablauf zu erreichen, gibt es zunächst zwei Möglichkeiten.
Die erste betrifft die Einbettung zweier oder mehrerer Funktionen in eine umfassendere. Dabei wird versucht diese Funktionen zu kombinieren, d.h. eine neue Funktion zu finden, für die eine rekursive Definition angezeigt ist. Diesen Transformationsprozeß nennt Feather COMBINING.

Die zweite Möglichkeit betrifft verschiedene Funktionen, die jedoch die gleichen Argumente benutzen. Dieser Transformationsprozeß wird TUPLING genannt und verbindet zwei Funktionen so, daß sie gleichzeitig berechnet werden können.

Um nun diese beiden Möglichkeiten der Transformation geeignet anwenden zu können, ist eine Strategie zur Strukturierung sinnvoll. Zu diesem Zweck wird das Gesamtproblem in Teilprobleme zerlegt, die individuell in Angriff genommen werden können und später wieder zu einer Gesamtlösung zusammengesetzt werden. Damit soll die Steuerung für den Benutzer erleichtert werden, d.h. der Benutzer gibt die Definitionen (in der Terminologie von Darlington und Burstall 'eurekas' genannt), nach denen das System fragt, für jedes Teilproblem an und behält damit eine bessere Übersicht.

Die von Feather vorgeschlagene Strategie sieht vor, daß 'von innen nach außen' gearbeitet wird, d.h. enthält die Definition einer Funktion G eine Funktion F, dann verbessere zunächst F bevor G verbessert wird, bzw. wenn kombiniert werden soll, dann paarweise ebenfalls 'von innen nach außen'.

Diese Strategie ergibt für das Telegrammproblem die folgenden Schritte:

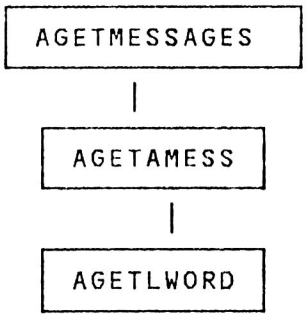
1. verbessere WORDSOF
2. verbessere TELSOF
3. verbessere GETMESSAGES
   a) kombiniere WORDSOF mit FLATTEN => GETWORDS
   b) kombiniere TELSOF mit GETWORDS => GETTELS
   c) kombiniere MESSAGESOF mit GETTELS => GETMESSAGES
4. konvertiere in eine iterative Form, die geeignet ist zur Übersetzung in eine imperative Sprache.

wobei das 'Kombinieren' zweier Funktionen dem oben beschriebenen COMBINING-Prozeß entspricht.

Die Steuerung der Transformation der Teilprobleme läuft über die Betrachtung der Fälle, die durch die Werte, die die Argumente annehmen können, gegeben sind. Einige Fälle können einfach berechnet werden, für andere muß nach einer rekursiven Definition gesucht werden. Um diese Rekursionen zu finden, werden Muster verwendet, die automatisch generiert oder vom Benutzer vorgeschlagen werden können. Gibt es keine passenden Muster, so besteht für den Benutzer die Möglichkeit Hilfsfunktionen einzuführen, für die das System jeweils einen neuen Funktionsnamen generiert.

Mit Hilfe der oben beschriebenen Transformationen kann nun die ursprüngliche, ineffiziente Fassung des Telegrammproblems in eine effizientere übertragen werden.

Die Aufrufstruktur der einzelnen Funktionen der transformierten Lösung des Telegrammproblems sieht dann wie folgt aus:



Anmerkung:
Es muß an dieser Stelle jedoch darauf hingewiesen werden, daß derartige Strukturdiagramme die Kompliziertheit der einzelnen Funktionen verdecken (die Rekursionsgleichungen der transformierten Lösung sind erheblich länger und unübersichtlicher), dennoch spiegeln sie auf eine eindrucksvolle Weise die Strukturveränderungen wider, die durch Transformation erreicht werden können.

6.2.2 Das ZAP-System

Die von Darlington und Burstall entwickelte und von Feather erweiterte Transformationsmethode besteht insgesamt gesehen darin, Folgen kleiner Manipulationen auf Rekursionsgleichungen anzuwenden, wobei die sechs bereits beschriebenen Manipulationstypen benutzt werden:
Definitionen, Bildung von Instanzen, Expandieren, Reduzieren, Anwendung allgemeiner mathematischer Gesetze und Abstraktion.

In einer abschließenden Phase des gesamten Transformationsprozesses muß dieses Teilsystem, das ausschließlich auf Rekursionsgleichungen aufbaut, verlassen werden, damit das verbesserte Rekursionsgleichungsprogramm in eine konventionelle imperative Programmiersprache konvertiert werden kann.

Der wichtigste Punkt des ZAP-Systems ist jedoch nach Feather eine geeignete Steuerung der einzelnen Manipulationsschritte. Er schlägt an dieser Stelle zwei Möglichkeiten vor: die zielgerichtete Transformation und die mustergesteuerte Transformation.
Bei der zielgerichteten ist das Protoprogramm gegeben und der Benutzer muß für die Teilprobleme gewünschte Ergebnisse (Endausdrücke) festsetzen, wonach es dem System überlassen wird, die Manipulationen zu finden, die den mittleren Teil ausfüllen. Zu diesem Zweck werden Anfangs- und Endausdruck vollständig durch unfolding und die Anwendung allgemeiner algebraischer Gesetze expandiert und dann verglichen (gematcht).
Bei der mustergesteuerten Transformation arbeitet das System lediglich als Verifizierer für die spezifizierten Transformationsschritte. Hierbei wird die Schwierigkeit umgangen, ei-

nen bestimmten Endausdruck festzusetzen, indem nur ein Muster dieses Endausdrucks gegeben wird, und dieses Muster funktionale Variablen enthalten darf, die Teilausdrücke ersetzen. Anfangsausdruck und Endmuster werden wie bei der zielgerichteten Transformation behandelt. Ist der Matchvorgang erfolgreich, werden die dabei entstehenden Bindungen der funktionalen Variablen dazu benutzt den Endausdruck durch Instanziierung des Endmusters zu erzeugen.

Funktionale Variablen werden innerhalb von Mustern auf zwei unterschiedliche Arten verwendet. Sie können einerseits mit Konstruktoren verglichen werden und andererseits mit Funktionen, die entweder während der Expansion nicht betrachtet werden oder ausdrücklich als "verwendbar" deklariert wurden. Verwendbar heißt in diesem Fall, die Funktion darf im Endausdruck wieder vorkommen.

Bei der Definition neuer Funktionen werden die Phasen
a) spezifizieren der neuen Funktion und
b) Transformation des ursprünglichen Ausdrucks
zusammengefaßt, wobei eine Variable an die Definition der neuen Funktion gebunden wird, und jedes Auftreten dieser Variablen innerhalb eines Musters durch einen Aufruf der neuen Funktion ersetzt wird.

6.2.3 Die ZAP-Sprache

Gesteuert wird das gesamte System mit Kommandos der ZAP-Sprache. Kommandofolgen werden Metaprogramme genannt. Ihr Protokoll dokumentiert die ausgeführten Transformationsschritte und kann gespeichert und später wiederverwendet werden, falls z.B. Modifikationen des Transformationsprozesses angezeigt sind. Dazu ist zu bemerken, daß ein unpassendes Metaprogramm nicht dazu führen kann, daß falsche Transformationen durchgeführt werden, sondern lediglich dazu, daß der Transformationsprozeß stecken bleibt oder ein Programm entsteht, das immer noch äquivalent ist zum ursprünglichen, aber vielleicht nicht so effizient wie es nach Abarbeitung eines geeigneten Metaprogramms sein könnte. Diese Tatsache spricht für die Zuverlässigkeit des Transformationsprozesses.

Bleibt der Transformationsprozeß hängen, so gibt es zwei Möglichkeiten für einen weiteren Versuch.
Zum einen kann die Spezifikation dieselbe bleiben, und das Metaprogramm wird modifiziert.
Zum zweiten kann das Protoprogramm verändert und die neue Fassung mit dem alten oder geringfügig geänderten Metaprogramm getestet werden. Zudem ist es möglich, in Abhängigkeit vom Grad der Veränderungen, die am Protoprogramm vorgenommen werden, nur Teile des gesamten Transformationsprozesses erneut laufen zu lassen. Zumindest brauchen in Metaprogrammen nur die Teile eventuell verändert zu werden, die unmittelbar die veränderten Teile des Protoprogramms betreffen.

Feather ist der Ansicht, daß bei der Transformation nichttrivialer Programme eine bestimmte Transformationsphase dazu geeignet ist, einige lokale Überprüfungen des gesamten Programms zu machen. Ein Metaprogramm besteht demnach hauptsächlich aus Folgen von Kontextblöcken, in denen Kommandos die relevanten Details für diesen Kontext festhalten und Transformationen stattfinden. Innerhalb von Kontextblöcken werden zu diesem Zweck folgende Kommandos benutzt:

UNFOLD gefolgt von Funktionsnamen deutet an, welche Funktionsgleichungen benutzt werden sollen, um den Ausdruck und das Muster bei der Transformation zu expandieren. Um nicht nur Gleichungen einer Funktion, sondern alle Gleichungen, die eine bestimmte Funktion enthalten, zu verwenden, gibt es das Kommando UNFOLDALL.

USING gefolgt von Funktionsnamen gibt an, welche Funktionen als "verwendbar" (s.o.) gelten, wobei zwischen uneingeschränkt (default) und eingeschränkt verwendbaren Funktionen unterschieden wird.

LEMMATA gefolgt von Eigenschaften und Reduktionsregeln geben Informationen, die beim Expandieren benutzt werden können (Assoziativität, Kommutativität, Identität von Funktionen).

Hierin liegt die einzige Möglichkeit unkorrekte Transformationen durchzuführen, wenn z.B. der Benutzer behauptet, eine Funktion sei assoziativ, obgleich sie es nicht ist. Um dies auszuschließen, könnte z.B. jedes verwendete Lemma über einen angeschlossenen theorem-prover verifiziert werden, was jedoch 1979 noch nicht vorgesehen war.

6.2.4 Default-Generatoren des ZAP-Systems

Zum gleichen Zeitpunkt waren dem System die folgenden drei Default-Generatoren zugänglich:

Typ-Information (stellt einfache Informationen über Datentypen bereit),
Hauptargumente (unterbricht die Transformation einer Funktion zum Zwecke der Betrachtung ihrer Einzelargumente),
Muster (liefert einfache Muster für Transformationen).

Der Typ-Information-Default-Generator ist dazu da, für jeden verwendeten Datentyp einfache Fallunterscheidungen zu machen, um festzustellen, für welche Werte der Argumente die Funktion direkt berechnet werden kann, und für welche Werte versucht werden muß, eine Rekursion zu finden. Ist dieser Default-Generator nicht dazu in der Lage, kompliziertere Unterscheidungen vorzuschlagen, so muß der Benutzer wieder eingreifen, indem er entweder die einzelnen Fälle und Muster explizit beim Transformieren oder vorher spezielle Typ-Information angibt.

Der Hauptargument-Default-Generator bezieht sich auf die Spezifikation von Zielen innerhalb von Transformationsblöcken, wobei die Typ-Information, gleichgültig ob default oder benutzergegeben, verwendet wird. Die Initiierung dieses Prozesses ist durch ein spezielles Schlüsselwort vor den entsprechenden Argumenten möglich.

Der Muster-Default-Generator erzeugt Pattern, die aus der funktionalen Variablen bestehen mit allen dazugehörigen freien Variablen der linken Seite des Ziels, die als entsprechende Argumente gelten, und rekursiven Aufrufen der Funktion, die transformiert werden soll. Die rekursiven Aufrufe werden gebildet, indem die Argumente auf der linken Seite durch den Fall, der von der Typ-Information vorgeschlagen wurde, ersetzt werden. Ein spezielles Schlüsselwort muß auch hier vor die entsprechenden Argumente gesetzt werden.

Versuche haben gezeigt, daß viele einfache Transformationen durch Kombination dieser Defaults ohne Benutzerunterstützung ausgeführt werden können.

6.2.5 Zwei weitere Beispiele nach Martin S. Feather

Nachdem die Transformation des Telegrammproblems schon in ausreichendem Maße beschrieben worden ist, sollen zwei weitere Beispiele kurz angedeutet werden, die von Feather mit Hilfe des ZAP-Systems transformiert wurden.

Ein einfacher Compiler, der eine abstrakte Syntax einer blockstrukturierten Sprache in Maschinencode überträgt. Da die Sprache Blöcke, while-Schleifen, bedingte Ausdrücke und Zuweisungen enthält, müssen Codeerzeugung zur Berechnung von Ausdrücken, Sprünge und Tests für Schleifen und bedingte Ausdrücke, sowie Identifikation korrekter Inkarnationen von Variablen innerhalb von Blöcken einbezogen und die damit zusammenhängenden Probleme untersucht werden.

Die ursprüngliche Spezifikation von Feather zerlegt das Protoprogramm in mehrere verschiedene einfache Phasen, die wie zu erwarten einen sehr ineffizienten Compiler beschreiben. Durch Transformation unter Benutzung des ZAP-Systems entsteht ein konventioneller Zwei-Stufen-Compiler. Das Protoprogramm enthält 60 Rekursionsgleichungen und ist damit umfangreicher als das Protoprogramm des Telegrammproblems, das aus 34 Rekursionsgleichungen besteht.

Ein Textformatierer, der Texte bearbeitet, die mit Kommandos durchsetzt sind, und der einen sauber formatierten Output erzeugt. Die zugrundeliegende Spezifikation ist dem Buch 'Software-Tools' von Kerrighan und Plaugher [KePl 76] entnommen, und man kann sagen, daß der Formatierer alle üblichen Kommandos und Funktionen enthält. Das von Feather geschriebene Protoprogramm unterscheidet sich jedoch von der vorgegebenen Lösung, da die Aufgabe des Textformatierens in ihre elementaren Bestandteile zerlegt wurde, was zu einem vollkommen unakzeptablen Programm geführt hat, wenn Effizienzmaßstäbe angelegt werden.

Das Protoprogramm enthält über 200 Rekursionsgleichungen. Übertragen wird es jedoch vom ZAP-System in eine Version, die vergleichbar ist mit der von Kerrighan und Plaugher.

Die Aufrufstruktur des Endprogramms ist auch für dieses Programm wesentlich kompakter als die des Protoprogramms.

6.3 Eine Synthese verschiedener Sortieralgorithmen nach John Darlington (1976)

Wie schon an anderer Stelle erwähnt macht Darlington mit der Synthese verschiedener Sortieralgorithmen einen ähnlichen Versuch wie später Lothar Schmitz [Schm 78] bezüglich verschiedener Algorithmen zur Berechnung der transitiven Hülle einer Relation.

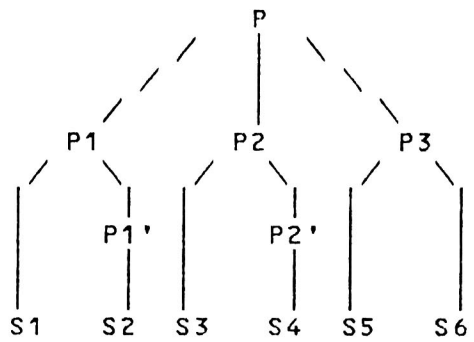
Ausgehend von einer einfachen, uneffizienten mathematischen Definition des Sortiervorgangs werden allgemein bekannte effiziente Sortieralgorithmen abgeleitet, die sich durch grundlegend unterschiedliche Strukturen auszeichnen.
In diesem Abschnitt sollen nicht die einzelnen Manipulationsschritte bzw. Manipulationsfolgen dieser Ableitung dargestellt werden , da die Transformationsmethodik des von Darlington und Burstall entwickelten unfold/fold-Ansatzes bereits eingehend beschrieben und an anderen Beispielen erläutert wurde, sondern es soll lediglich darauf hingewiesen werden, welche unterschiedlichen Strukturen aus einer einfachen Definition mit Hilfe der unfold/fold-Methode erzeugt werden können.

Der Sortiervorgang wird von Darlington als Selektion der geordneten Permutation einer Menge definiert. Die zu sortierenden Objekte sind Funktionen, die eine Folge ganzer Zahlen in eine Menge von Atomen mit einer totalen Ordnung < abbilden. Zunächst wird die Menge Perm aller Permutationen der gegebenen Menge erzeugt, dann wird durch Herausfiltern der ungeordneten Funktionen die geordnete Permutation selektiert.

Die Definitionsmenge sieht nach Clark/Darlington [ClDa 77] wie folgt aus:

```
sort(X)      <= y sodaß perm(x,y) & ord(y)
perm (x,y)   <= ∀ u (u∈ <—> u∈y)
ord(y)       <= ∀ (u,v) (u vor v in y —> u≤v)
u ∈ nil      <= false
u ∈ [x]      <= u = x
u ∈ x< >y    <= u ∈ x oder u ∈ y wobei < > das Anein-
                                anderhängen zweier
                                Listen bedeutet
u vor v in nil   <= false
u vor v in [x]   <= false
u vor v in x< >y <= u vor v in x
                    oder u vor v in y
                    oder (u∈x & v∈y)
```

a) Bei Darlington [Darl 76] werden aus dieser Definition drei Zwischenalgorithmen abgeleitet (P1, P2 und P3).
P1 wird für Selection Sort und P2 für Insertion Sort modifiziert.
Aus P1 wird dann Quicksort (S1), aus P1' Selection Sort (S2), aus P2 Merge Sort (S3), aus P2' Insertion Sort (S4) und aus P3 werden Exchange Sort (S5) und Bubble Sort (S6) abgeleitet.
Somit ergibt sich für die Synthese die folgende Struktur:



Die Ableitung der ersten vier Algorithmen ist relativ symmetrisch, während sich die letzten beiden Algorithmen in ihrer Art unterscheiden. Der Unterschied der Synthese von P1 und P2 ist, daß in P1 die Rekursion über eine 'Dekomposition' (Dekomposition einer Menge S in zwei disjunkte Mengen S1 und S2, sodaß gilt S1 ∪ S2 = S) gemacht wird, während sie in P2 über eine 'Dekonstruktion' (S = s + S1) läuft. Derselbe Unterschied liegt zwischen Quick Sort und Selection Sort und trennt auch Merge Sort von Insertion Sort.
Exchange Sort und Bubble Sort unterscheiden sich in der Art wie sie den Raum aller Permutationen durchlaufen bis sie die geordnete Permutation erzeugt haben.

Bemerkung: an dieser Stelle sei noch darauf hingewiesen, daß Darlington diese Synthese per Hand durchgeführt hat, da zu diesem Zeitpunkt noch kein automatisches Transformationssystem für derartige Programmsynthesen zur Verfügung stand.

Als Zukunftsperspektive wird von Darlington auf mögliche weitere Untersuchungen hingewiesen, wobei z. B. zu hoffen ist, daß bei Weiterverfolgung jedes möglichen Weges hinter Verzweigungspunkten (wie P1, P2 und P3) automatisch neue, effiziente Algorithmen entwickelt werden können, die mit keinem bisher bekannten übereinstimmen.

b) Bei Clark/Darlington [ClDa 77] werden die Algorithmen darin unterschieden, ob sie auf der input- oder auf der output-Variablen arbeiten. Merge Sort z.B. arbeitet auf der output-, Quick Sort dagegen auf der input-Variablen. Hier werden analoge Entscheidungen beschrieben zu Darlington [Darl 76], jedoch in einer abstrakteren Terminologie. Insertion Sort wird daher ebenfalls mit Merge Sort in Verbindung gebracht und Selection Sort wird von Quicksort abgeleitet. In diesem Artikel bleiben Bubble Sort und Exchange Sort unberücksichtigt.

6.4 Algorithmen zur Berechnung der transitiven Hülle einer Relation nach Lothar Schmitz (1978)

Mit seinem Versuch aus einer einfachen Spezifikation zur Berechnung der transitiven Hülle einer Relation mehrere verschiedene effiziente Algorithmen abzuleiten, macht Schmitz [Schm 78] einen ähnlichen Ansatz der Programmsynthese wie Darlington 1976 bezüglich der Ableitung effizienter Sortieralgorithmen aus einer recht abstrakten Spezifikation des Sortiervorgangs. Für diese Synthese benutzt Schmitz ebenfalls die bereits beschriebenen Manipulationen der unfold/fold-Technik.

Zunächst wird die allgemeine Definition der transitiven Hülle einer Relation gegeben.

Sei V eine endliche Menge und $R \subseteq V^2$ eine binäre Relation über V, wobei die Elemente von V Knoten genannt werden und die Elemente von R Kanten. Dann ist die transitive Hülle R^+ von R die Menge aller Paare $\langle x,y \rangle \in V^2$, die durch einen Weg in R verbunden sind.

Für die Programmsynthese definiert Schmitz noch einige Sätze, die die Äquivalenz von Wegen, den Ausschluß innerer Zyklen, den kürzesten Weg und die Dekomposition von Wegen betreffen.
Das Ziel der Synthese ist, aus der gegebenen Spezifikation geeignete Rekursionsgleichungen abzuleiten, wobei die Terminierung der zu entwickelnden Programme sichergestellt werden muß.
Dabei verwendet Schmitz eine Strategie, die - wie bei Darlington und Burstall bereits erwähnt - forced-folding (erzwungene Reduktion) genannt wird.
Unter Anwendung der für die unfold/fold-Methode charakteristi-

schen Manipulationen entwickelt Schmitz drei verschiedene Syntheseversuche.
Aus diesen Ansätzen entstehen schließlich vier verschiedene, effiziente Rekursionsgleichungsalgorithmen, die sich alle durch eine unterschiedliche Struktur auszeichnen (siehe Dalington - Sortieralgorithmen).

In dem von Schmitz zu dieser Programmsynthese vorliegenden Papier findet sich jedoch keine ausgiebige Betrachtung der Manipulationsschritte und Möglichkeiten der unfold/fold-Methode - abgesehen von der besonderen Erwähnung und Anwendung des forced-folding.

6.5 Abschließende Bemerkungen zur unfold/fold-Methode

Hiermit sind meiner Meinung nach die wichtigsten Artikel, die zur Zeit bezüglich der unfold/fold-Methode und ihrer Anwendungsmöglichkeiten zur Verfügung stehen, kurz beschrieben und zusammengefaßt. Die von Darlington und Burstall entwickelten Algorithmen scheinen für eine Implementierung auszureichen. Verbesserungsmöglichkeiten liegen in der Ansammlung weiterer Heuristiken und dem Anschluß eines Verifizierers für die Überprüfung der vom Benutzer eingegebenen Lemmata. Weitere Ergebnisse können nach meinem Ermessen nur experimentell erarbeitet werden.

7. Zusammenfassung und zukünftige Entwicklungen

In dieser Arbeit wurde gezeigt, unter welchen Bedingungen mit Hilfe von Programmtransformationen rekursiv definierte Funktionen in repetitive überführt werden können. Dabei wurde deutlich, daß sich die Untersuchungen auf zwei grundlegende Probleme konzentrieren:

1. die vollständige Elimination von Rekursionen unter Verwendung syntaktischer und semantischer Informationen über die konkrete, zu verarbeitende Funktion, wobei kein stack eingeführt wird, und
2. die Optimierung der Verwendung von stacks für rekursive Funktionen, bei deren Auflösung in eine repetitive Form nicht auf das Kellern von Parameterwerten und/oder Protokollvariablen verzichtet werden kann.

Zukünftige Untersuchungen sollten sich darauf richten, nach weiteren Bedingungen zu forschen, unter denen rekursiv definierte Funktionen ohne Verwendung von stacks in repetitive überführt werden können, da die Problemverlagerung von der Programm- auf die Datenstruktur nur eine Pseudolösung für das eigentliche Problem sein kann.

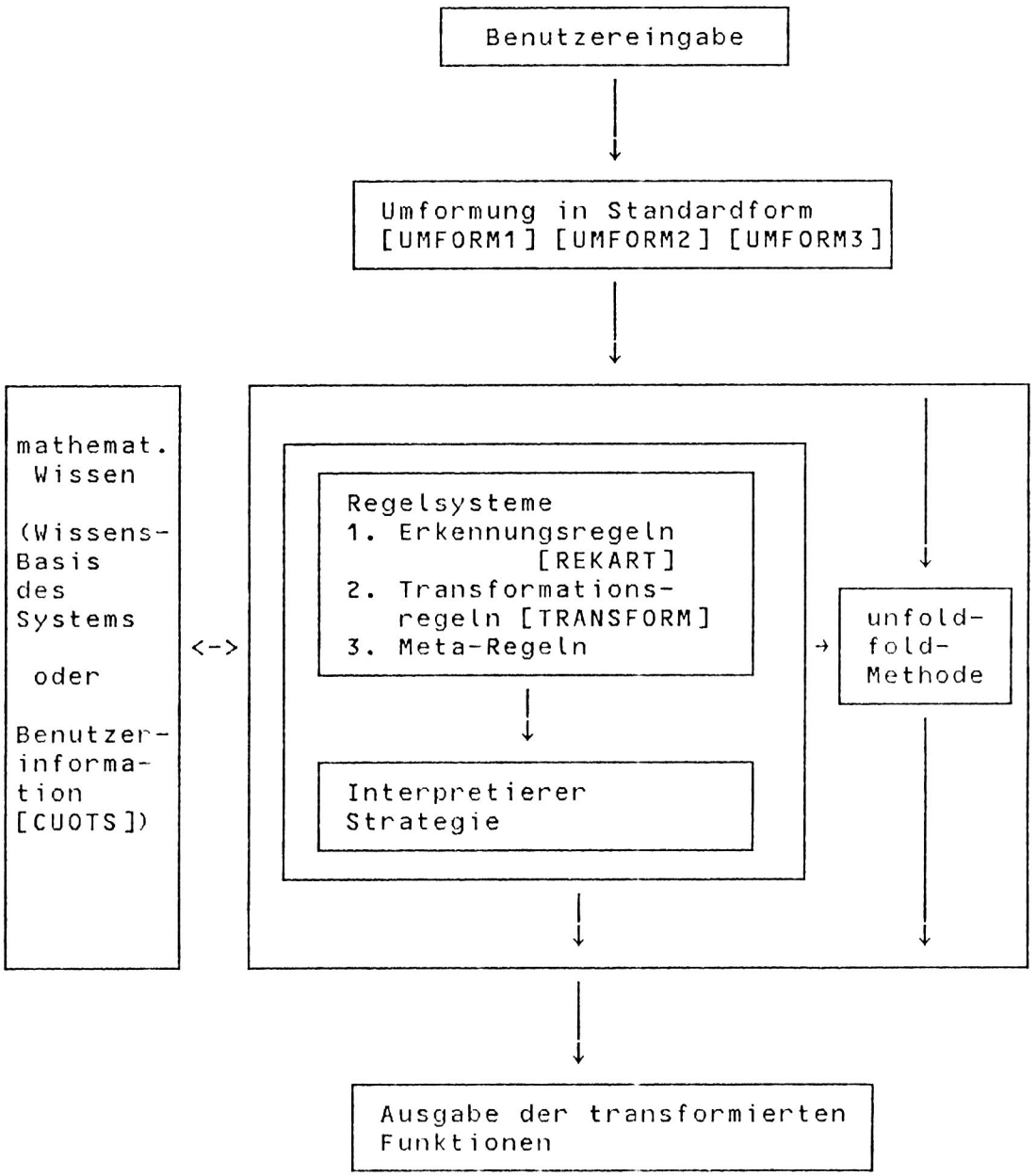
In Kapitel 3 wurde ein Regelsystem erstellt, das die Art einer rekursiven Funktion bestimmt, und in Kapitel 4 ein weiteres, das in Abhängigkeit von dieser Art die Rekursion eliminiert. Korrektheitsbeweise für die hierbei verwendeten Transformationsschemata vervollständigen diesen Teil der Arbeit.

Weiterhin ist eine Methode zur Effizienzsteigerung rekursiver Rechenvorschriften vorgestellt worden , die auf der Verwendung von Heuristiken beruht und in einem Gesamtsystem auf das zuvor beschriebene Transformationsregelsystem aufgesetzt werden kann.

Um den Transformationsprozeß selbst zu optimieren, kann es sinnvoll sein, das Programm um ein Metaregelsystem zu erweitern. Ein solches Metaregelsystem steuert die Anwendung der Regeln, wobei die sequentielle Reihenfolge der Prüfung auf Anwendbarkeit der Regeln nicht mehr notwendige Voraussetzung ist.

Ferner sei darauf hingewiesen, daß die Überlegungen zur Optimierung der verwendeten stacks nicht den Anspruch auf Vollständigkeit erheben. Es sind lediglich Vorschläge, wie ein stack organisiert werden kann, um Speicherplatz und Rechenzeit einzusparen. Andere Organisationsformen sind durchaus denkbar.

Ein Gesamtsystem, das als Eingabe Spezifikationen der Sprache ASPIK [BeVo 83] erhält und transformierte repetitive Funktionen in derselben Spezifikationssprache ausgibt, könnte folgenden Aufbau haben:



Dabei könnten die beiden beschriebenen Methoden zur Elimination von Rekursionen alternativ angewendet werden oder sequentiell. Sequentiell bedeutet in diesem Fall, daß zunächst versucht wird, eine Funktion mit dem in Kapitel 4 beschriebenen Transformationsregelsystem zu transformieren, und erst, wenn dieses

System versagt, die unfold/fold-Methode anzuwenden.

Denkbar ist auch, eine gegebene rekursive Funktion in mehrere verschiedene repetitive Formen zu transformieren (siehe Kapitel 4.4.1 - Fakultätsfunktion). In diesem Fall müßte eine Agenda eingeführt werden, die Zwischenzustände des Transformationsprozesses an Entscheidungspunkten beinhaltet.

Schließlich soll ein kleines Beispiel eine Grenze des vorgeschlagenen Systems aufzeigen, falls es als stand-alone Programm benutzt wird.

Angenommen, wir haben die folgende linear rekursive Funktionsdefinition:

```
funct F ≡ (nat x) nat:
      if x>0
      then x + F(x-1)
      else 0
      fi
```

Diese Funktion kann von unserem Regelsystem durch Um-Klammerung in die folgende Form transformiert werden (da '+' assoziativ, psi = phi = Addition gewählt werden kann, und das neutrale Element der Addition '0' ist):

```
funct F ≡ (nat x) nat:
      G(x,0)
      where
      funct G ≡ (nat x, nat z) nat:
            if x>0
            then G(x-1, z+x)
            else z+0
            fi
```

Regel2 und Regel3 führen zu diesem Ergebnis.

Sieht man sich die erste Form jedoch genauer an, und berücksichtigt die Semantik der gegebenen Funktion, so erkennt man, daß es sich um die Summenbildung aller natürlichen Zahlen von 1 bis n handelt.
Ein Simplifier, der den Schluß von x auf x-1 beinhaltet, würde diese Form erkennen und die einfache, nicht-rekursive Summenberechnung nach Gauss F(x) ≡ (nat x) nat: x/2*(x+1) einsetzen.

Zu bemerken ist noch, daß von Informatikern immer wieder die Forderung nach rekursiven Programmiersprachen erhoben wird, um dem Programmierer das Formulieren von Programmen zu erleichtern und Programme übersichtlicher gestalten zu können. Gleichzeitig erhebt sich natürlich die Frage, wie diese 'einfachen' Rechenvorschriften auch eine einfache Ausführbarkeit gewährleisten könnten. Compiler werden daher mit Implementierungen versehen, die bezüglich rekursiver Funktionen fast ausschließlich auf der Verwendung von stacks beruhen. Aber bereits 1968 wurde z.B. von D.W. Barron in seinem Buch 'Recursive Techniques in Programming' [Barr 68] die Notwendigkeit erkannt, andere Lösungen zu finden:

> "...It is not clear whether work along these lines will lead to practically useful results, in the sense of being able to automatically transform recursive procedure into equivalent nonrecursive procedure. It is more likely that changes in hardware will remove the inefficiency that presently attaches to recursive procedures. One thing, however, is certain. If programs are written to perform these transformations, those programs will themselves be recursive."
>
> D.W. Barron
> (aus: Recursive Techniques in Programming)

Anhang1

Regelbasis für das Erkennen der Rekursionsart

M 1.1 der Funktionsname tritt in der Rekursion genau einmal auf
==> einfache Rekursion

M 1.2 der Funktionsname tritt in der Rekursion mehrfach auf
==> mehrfache Rekursion

M 2.1 einfach rekursiv
das erste Element des rekursiven Operationsteils ist der Funktionsname der zu untersuchenden Funktion (rekursiver Aufruf)
==> repetitiv
Art bestimmt

M 2.2 einfach rekursiv
¬(Art bestimmt)
das erste Element des rekursiven Operationsteils ist der Name einer Funktion, die den rekursiven Funktionsaufruf als ein Argument enthält
==> linear
Art bestimmt

M 3.1 mehrfach rekursiv
der rekursive Funktionsname tritt in der Definition genau zweimal auf
der zweite rekursive Funktionsaufruf tritt innerhalb des Argumentbereichs des ersten rekursiven Funktionsaufrufs auf
==> geschachtelt
Schachtelungstiefe ST = 2
Art bestimmt

M 3.2 mehrfach rekursiv
der rekursive Funktionsname tritt n-mal in der Definition auf (n>2)
der nächste rekursive Funktionsaufruf tritt jeweils im Argumentbereich des vorhergehenden auf
==> geschachtelt
Schachtelungstiefe ST = n
Art bestimmt

M 3.3 mehrfach rekursiv
der rekursive Funktionsname tritt in der Definition genau zweimal auf
der zweite rekursive Funktionsaufruf tritt unabhängig vom (parallel zum) ersten rekursiven Funktionsaufruf, d.h. außerhalb des Argumentbereichs des ersten, auf
==> kaskadenartig
Faktor F = 2
Art bestimmt

M 3.4 mehrfach rekursiv
der rekursive Funktionsname tritt n-mal in der Definition auf (n>2)
der nächste rekursive Funktionsaufruf tritt jeweils außerhalb des Argumentbereichs des vorhergehenden auf
==> kaskadenartig
Faktor F = n
Art bestimmt

M 3.5 ¬(Art bestimmt)
==> Muster nicht erkannt (Fehlermeldung)

M 4.1 geschachtelt
∃ einfache Funktion h, sodaß mindestens ein rekursiver Aufruf im Argumentbereich von h liegt
==> verschränkt

M 4.2 geschachtelt
∃ einfache Funktion phi, sodaß der erste rekursive Aufruf im Argumentbereich von phi liegt
==> linear

Anhang 2

Regelbasis für die Transformationen

```
Regel1      E(x) = { }
       ==> phi einstellig
           (phi rechtskommutativ)

Regel2      ¬(phi einstellig)
            ∃ psi: ν × ν → ν                                    CUOTS
            phi(phi(r, s), t) = phi(r, psi(s, t))
       ==> M1
            Schema aufgelöst
            Um-Klammerung

Regel3      ¬(phi einstellig)
            Um-Klammerung
            ρ = ν                                               CUOTS
            phi(phi(r, s), t) = phi(r, phi(s, t))
            ∃ e ∈ ρ: phi(r, e) = r
       ==> ersetze in M1 die erste if-then-else - Clause
            durch G(x,e)

Regel4      ¬(Schema aufgelöst)
            phi einstellig
       ==> psi := phi
            rechtskommutativ
            psi generiert

Regel5      ¬(Schema aufgelöst)
            ¬(phi einstellig)
            ∃ psi: ρ × ν → ρ                                    CUOTS
            phi(psi(r, s), t) = psi( phi(r, t), s)
       ==> setze psi gleich user-input
            rechtskommutativ
            psi generiert

Regel6      ¬(Schema aufgelöst)
            rechtskommutativ
            psi generiert
       ==> M2

Regel7      ¬(Schema aufgelöst)
            rechtskommutativ
            psi generiert
            H ≡ const.
       ==> ersetze F durch const.
            lösche *1
            Schema aufgelöst
```

Regel8 ¬(Schema aufgelöst)
rechtskommutativ
psi generiert
(B(x) = (x ≠ x_1)) CUOTS
==> ersetze F durch H(x_1)
lösche *1
Schema aufgelöst

Regel9 ¬(Schema aufgelöst)
rechtskommutativ
psi generiert
==> ersetze F durch F(x)
ersetze * durch
funct F ≡ (λ x)p:
if B(x) then F(K(x)) else H(x) fi
Schema aufgelöst

Regel10 ¬(Schema aufgelöst)
¬(phi einstellig)
∃ $\bar{K}$: ∀ x ∈ {K^i(x) | i∈N_0, x∈λ}: $\bar{K}$(K(x))=x CUOTS
==> M3
Umkehrfunktion

Regel11 Umkehrfunktion
B(x) = (x ≠ x_1) CUOTS
==> ersetze *1 durch λ x_0 ≡ x_1
Schema aufgelöst

Regel12 Umkehrfunktion
¬(Schema aufgelöst)
==> ersetze *1 durch
λ x_0 ≡ P(x)
where
funct P ≡ (λ x)λ:
if B(n) then P(K(n)) else n fi

Regel13 ¬(Schema aufgelöst)
¬(phi einstellig)
==> M4
Schema aufgelöst

Regel14 geschachtelt (einfach)
Schachtelungstiefe 2 CUOTS
es gilt die Eigenschaft 0
==> Wert(0):=TRUE

Regel15 geschachtelt (einfach)
Schachtelungstiefe 2 CUOTS
es gilt die Eigenschaft 1
==> Wert(1):=TRUE

```
Regel16     geschachtelt (einfach)
            Schachtelungstiefe 2
            Wert(0) = TRUE
            Wert(1) = FALSE
        ==> M7
            phi einstellig
            ersetze 'geschachtelt (einfach)
              durch 'linear'

Regel17     geschachtelt (einfach)
            Schachtelungstiefe 2
            Wert(1) = TRUE
            es gilt die Eigenschaft 2                    CUOTS
        ==> Wert(2):=TRUE

Regel18     geschachtelt (einfach)
            Schachtelungstiefe 2
            Wert(1) = TRUE
            Wert(2) = TRUE
            es gilt die Eigenschaft 3                    CUOTS
        ==> Wert(3):=TRUE
            M5
            ersetze 'geschachtelt (einfach)'
              durch 'Schema aufgelöst'

Regel19     ¬(Schema aufgelöst)
            geschachtelt (einfach)
            Schachtelungstiefe 2
            Wert(0) = TRUE
            Wert(1) = TRUE
        ==> M6
            ersetze 'geschachtelt (einfach)'
              durch 'Schema aufgelöst'

Regel20     geschachtelt
            ¬(Schema aufgelöst)
        ==> M8

Regel21     geschachtelt (einfach)
            Schachtelungstiefe y>2
        ==> fülle die Lücken in M8:
            z := 1
            z-Sorte := nat i
            BED := i≠0
            *1 := { }
            *3 := (f(x), i+Schachtelungstiefe-1)
            *4 := (g(x), i-1))
            *5 := { }
            *6 := x
            ersetze 'geschachtelt (einfach)'
              durch 'Schema aufgelöst'
```

```
Regel22     geschachtelt (verschränkt)
            Schachtelungstiefe y=2
        ==> fülle die Lücken in M8:
            z := 1
            z-Sorte := nat i
            BED := i≠0
            *1 := if ¬B(x) and i=1 then g(x) else
            *3 := (f(x), i+1)
            *4 := (h(g(x)), i-1))
            *5 := { }
            *6 := x
            ersetze 'geschachtelt (verschränkt)'
              durch 'Schema aufgelöst'

Regel23     geschachtelt (verschränkt)
            Schachtelungstiefe y>2
        ==> fülle die Lücken in M8:
            z := empty
            z-Sorte := stack *2 s
            BED := ¬(¬B(x) and s=empty)
            *1 := {}
            *6 := g(x)

Regel24     geschachtelt (verschränkt)
            Schachtelungstiefe y=3
        ==> fülle die Lücken in M8:
            *2 := nat
            *3 := if top s = 0
                  then (f(x), (((rest s) & 1) & 0))
                  else (f(x), (((rest s) & ((top s) + 1) & 0))
                  fi
            *4 := if top s = 0
                  then ((h(g(x))), (rest s))
                  else (L((top s), g(x)), (rest s))
                  fi
            *5 := where
                  funct L ≡ (nat n, λ y)λ:
                        if n>0
                        then L(n-1, l(y))
                        else y
                        fi
            ersetze 'geschachtelt (verschränkt)'
              durch 'Schema aufgelöst'

Regel25     geschachtelt (verschränkt)
            Schachtelungstiefe y>3
        ==> fülle die Lücken im (erweiterten) Muster M8:
            *2 := {h1,...,hn}
            *3 := (f(x), s & h1 &.....& hn)
            *4 := ((top s(g(x))), (rest s))
            *5 := { }
            ersetze 'geschachtelt (verschränkt)'
              durch 'Schema aufgelöst'
```

```
Regel26    geschachtelt (linear verschränkt)
      ==>  fülle die Lücken in M8:
           z := empty,0
           z-Sorte := stack ({h1,..,hn} × nat s), {0,1} v
           BED := ¬((¬B(x) or v=1) and s=empty)
           *1 := {}
           *3 := if s ≠ empty
                 then (f(x), (rest s) & ((top1 s), (top2 s))
                                      & (h2,0) &...& (hn,0), 1)
                 else (f(x), s & (h1,0) &...& (hn,0), 1)
                 fi
           *4 := ((top1 s)(H((top2 s), g(x)), (rest s), 1)
           *5 := where
                 funct H ≡ (nat n, λ y)λ:
                        if n>0
                        then H(n-1, h1(y))
                        else y
                        fi
           *6 := if v=1 then x else g(x) fi
           ersetze 'geschachtelt (linear verschränkt)'
             durch 'Schema aufgelöst'
```

```
Regel27    kaskadenartig
           Umkehrfunktionen                                  CUOTS
      ==>  M9
           {∀ Ki | ¬∃ K̄i : kellere xi und K̄i (yi ):=xi }
           Schema entflochen
             bzw. geschachtelt (verschränkt)
             bzw. geschachtelt (linear verschränkt)

Regel28    entflochten
           geschachtelt (verschränkt)
      ==>  M10

Regel29    entflochten
           geschachtelt (linear verschränkt)
      ==>  M11
```

Anhang 3

Endgültige Muster für die Transformationen

wobei hinter dem Namen des Musters die ursprüngliche Form und die Bedingungen, unter denen in das angegebene Muster transformiert werden kann, aufgeführt sind:
Muster 'Name' (ursprüngliche Form; Bedingungen).

Muster M1 (linear; Bedingungen für die Um-Klammerung)

```
funct L ≡ (λ x)ρ:
      if B(x) then G(K(x),E(x)) else H(x) fi
      where
      funct G ≡ (λ x, ν z)ρ:
            if B(x) then G(K(x), psi(E(x),z))
                    else phi(H(x), z) fi
```

Muster M2 (linear; Bedingungen für die Operandenvertauschung)

```
funct L ≡ (λ x)ρ:
      G(x,F)
      where *1
      funct G ≡ (λ x,ρ z)ρ:
            if B(x) then G(K(x), psi(z, E(x)))
                    else z fi
```

Muster M3 (linear; Bedingungen für die Funktionsumkehr)

```
funct L ≡ (λ x)ρ:
      R(x0, H(x0), x)
      where *1
      funct R ≡ (λ y, ρ z, λ x)ρ:
            if (y ≠ x)
            then R(K̄(y), phi(z, E(K̄(y))), x)
            else z
            fi
```

Muster M4 (linear; Bedingungen für die Funktionsumkehr mit stack)

```
funct L ≡ (λ x)ρ:
      R(P(x, empty))
      where
      funct P ≡ (λ x, stack λ sx) (stack λ, ρ):
            if B(x)
            then P(K(x), sx & x)
            else (sx, H(x))
            fi.
      funct R ≡ (stack λ sy, ρ z)ρ:
            if sy ≠ empty
            then R(rest sy, phi(z, E(top sy)))
            else z
            fi
```

Muster M5 (einfach geschachtelt;)

```
funct F ≡ (λ x)λ:
      if B(x) then F(g(f(x))) else g(x) fi
```

Muster M6 (einfach geschachtelt; Eigenschaft 0 und 1)

```
funct F ≡ (λ x)λ:
      K(x,x)
      where
      funct K ≡ (λ x, λ z)λ:
            if B(x)
            then K(f(x), g(f(z)))
            else g(z)
            fi
```

Muster M7 (einfach geschachtelt; Eigenschaft 0)

```
funct F ≡ (λ x)λ:
      if B(x) then g(F(f(x))) else g(x) fi
```

Muster M8 (geschachtelt;)

```
funct F ≡ (λ x)λ:
      Q(x,z)
      where
      funct Q ≡ (λ x, z-Sorte)λ:
            if BED
            then *1
            Q( if B(x)
               then *3
               else *4
               fi )
            else *6
            fi
            *5
```

Muster M9 (Kaskade, Faktor n=2;)

```
funct C ≡ (λ x)ρ:
      b
      where (λ a, stack ρ sb, ρ b) ≡ C*(x, empty)
      funct C* ≡ (λ x, stack ρ sz) (λ, stack ρ, ρ):
         if B(x)
         then (λ x1, stack ρ sr1) ≡ (K1(x), sz);
              (λ y1, stack ρ sz1, ρ z1) ≡ C*(x1, sr1);
              (λ x2, stack ρ sr2) ≡ (K2(K̄1(y1)), sz1 & z1);
              (λ y2, stack ρ sz2, ρ z2) ≡ C*(x2, sr2);
              (K̄2(y2), rest sz2, phi(top sz2, z2, E(K̄2(y2))))
         else (x, sz, H(x))
         fi
```

Muster M10 (entflochten, verschränkt geschachtelt;)

```
funct F ≡ (λ x)λ:
      F* ≡ (x, empty)
      where
      funct F* ≡ (λ x, stack {1,...,n} sn)λ:
            if B*(x, sn)
            then F*(S*(x, sn))
            else H*(x, sn)
            fi
      where
      H*(λ x, stack {1,...,n} sn) ≡ H(x)
      B*(λ x, stack {1,...,n} sn) bool ≡ B(x) and sn≠empty
      S*(λ x, stack {1,...,n} sn) (λ, stack {1,...,n}) ≡
            if ¬B(x) and sn≠empty
            then (S_k(H*(x, sn)),
                  where Index k = top sn
                  if top sn ≠ n
                  then (rest sn) & ((top sn) + 1)
                  else (rest sn)
                  fi )
            else (S_1(x), sn & 2)
            fi
```

Muster M11 (entflochten, linear verschränkt geschachtelt;)

```
funct F ≡ (λ x)λ:
      b
      where (λ b, stack ρ sn) ≡ F*(x, $)
      funct F*(λ x, stack ρ sn)(λ, stack ρ):
            if sn ≠ empty
            then if top sn ≠ $
                 then F*(S_k(x), (rest sn))
                      where Index k = top sn
                 else if B(x)
                      then F*(S_1(x),
                                ((rest sn & (n, $, n-1,..., 2, $))
                      else if top(rest sn) ≠ empty
                           then F*(S_r(H(x)), (rest(rest sn)))
                                where Index r = top(rest sn)
                           else (H(x), (rest sn))
                           fi
                       fi
                  fi
            else (x, sn)
            fi
```

LITERATURLISTE

[Aubi 75] Aubin, R.
Some generalisation heuristics in proofs by induction
Proc. IRIA Symp. Proving and improving programs, pp. 197-208
Arc-et-Senan, Frankreich, 1975

[BPPW 77] Bauer, F.L.; Partsch, H.; Pepper, P.; Wössner, H.
Notes on the project CIP: outline of a transformation system
TUM-INFO-7729, Institut für Informatik
Technische Universität München
München, 1977

[BaWö 81] Bauer, F.L.; Wössner, H.
Algorithmische Sprache und Programmentwicklung
Springer, Berlin 1981

[Barr 68] Barron, D.W.
Recursive techniques in programming
Computer Monographs
McDonald, London, 1968

[Barr 71] Barron, D.W.
deutsche Ausgabe von [Barr 68]
Rekursive Techniken in der Programmierung
Computer Monographien
Hanser, München, 1971

[BeVo 83] Beierle, Ch.; Voss, A.
Parametrization-by-use for hierarchicaly structured objects
SEKI-Projekt
Fachbereich Informatik
Universität Kaiserslautern, Mai 1983

[Bird 77A] Bird, R.S.
Notes on recursion elimination
CACM, Vol. 20, No. 6, pp. 434-439
New York, 1977

[Bird 77B] Bird, R.S.
Improving programs by the introduction of recursion
CACM, Vol. 20, No. 11, pp. 856-863
New York, 1977

[Bird 78] Bird, R.S.
Recursion elimination with variable parameters
The Computer Journal, Vol. 22, No. 2, pp. 151-154
New York, 1978

[BoMo 75] Boyer, R.S.; Moore, J.S.
Proving theorems about LISP functions
JACM Vol. 22, No. 1, pp. 129-144
New York, 1975

[Burs 77] Burstall, R.M.
Design considerations for a functional programming language
Proc. of Infotech state of the art conference, pp. 45-57
Kopenhagen, 1977

[ClDa 77] Clark, Keith; Darlington, John
Algorithm analysis through synthesis
Draft
Edinburgh, 1977

[Coop 66] Cooper, D.C.
The equivalence of certain computations
The Computer Journal, Vol. 9, pp. 45-52
New York, 1966

[Darl 76] Darlington, John
A synthesis of several sorting algorithms
Acta Informatica, No. 11, pp. 1-30
Springer, London 1978
und
D.A.I. Research Report No. 23
Edinburgh, 1976

[DaBu 76] Darlington, John; Burstall, R.M.
A system which automatically improves programs
Proceedings: Third Int. Joint Conference on Artifical Intelligence, pp. 479-485
Californien, 1973

[DaBu 77] Darlington, John; Burstall, R.M.
A transformation system for developing recursive programs
JACM Vol. 24, No. 1, pp. 44-67
New York, 1977

[Feat 77] Feather, Martin S.
Program transformation applied to the telegram problem
D.A.I. Research Report No. 47
Edinburgh, 1977

[Feat 79] Feather, Martin S.
A system for program transformation
D.A.I. Research Paper No. 124
Edinburgh, 1979

[Gerl 83] Gerlach, M.
A second-order matching procedure for the practical use in a program transformation system
SEKI-Projekt
Fachbereich Informatik
Universität Kaiserslautern, Juni 1983

[Hask 73] Haskell, R.
Efficient implementation of a class of recursively defined functions
The computer Journal, Vol. 18, No. 1, pp. 23-29
New York, 1973

[HeSn 72] Henderson, P.; Snowdon, R.
An experiment in structured programming
BIT 12, pp. 38-53, 1972

[Hiki 79] Hikita, Teruo
On a class of recursive procedures and equivalent iterative ones
Acta Informatica, No. 12, pp. 305-320
Springer, London, 1979

[HuLa 78] Huet, G.; Lang B.
Proving and applying program transformations expressed with 2nd order patterns
Acta-Informatica, No.11, pp. 31-55
Springer, London, 1978

[KePl 76] Kernighan, B.W.; Plaugher, P.J.
Software Tools
Addison-Wesley, Reading (MA), 1976

[Kilg 81] Kilgour, A.C.
Generalized non-recursive traversal of binary trees
Software - practice and experience, Vol. 11, pp. 1299-1306
Wiley & Sons, New York, 1981

[Kott 82] Kott, Laurent
Unfold/fold program transformations
Rapport de Recherche
(publication interne no. 173)
Laboratoire d'Informatique
Rennes, 1982

[MaNV 72] Manna, Zohar; Ness, Stephen; Vuillemin, Jean
Inductive methods for proving properties of programs
ACM, SIGPLAN notices, Vol. 7, No. 1, pp. 27-50
New York, 1972

[MaWa 75] Manna, Zohar; Waldinger, R.J.
Knowledge and reasoning in program synthesis
Artificial Intelligence Journal Vol. 6, No. 2, pp. 175-208
North Holland, Amsterdam, 1975

[MaWa 77] Manna, Zohar; Waldinger, R.J.
The automatic synthesis of recursive programs
Proceedings of ACM SIGART-SIGPLAN Symposium on
Artificial Intelligence and Programming Languages,
pp. 29-36
New York, 1977

[Moor 75] Moore, J.S.
Introducing iteration into the pure LISP theorem
prover
CLS-74-J Xerox Palo Alto Research Center (PARC)
Palo Alto, Califormia, 1975

[PaPe 76] Partsch, Helmut; Pepper, Peter
A family of rules for recursion removal
Information processing letters, Vol. 5, No. 6,
pp. 174-177
North Holland, Amsterdam, 1976

[PaHe 70] Paterson, M.S.; Hewitt, C.E.
Comparative Schematology
Record of Project MAC
Conference on concurrent systems and parallel
computation, pp. 119-128
ACM, New York, 1970

[Rohl 77] Rohl, J.S.
Converting a class of recursive procedures into
non-recursive ones
Software - practice and experience, Vol. 7,
pp. 231-238
Wiley & Sons, New York, 1977

[Rohl 80] Rohl, J.S.
The elimination of linear recursion: a tutorial
Proceedings of the 3rd Australian Computer Conference
1980

[Rohl 81] Rohl, J.S.
Eliminating recursion from combinatoric procedures
Software - practice and experience, Vol. 11,
pp. 803-817
Wiley & Sons, New York, 1981

[Schm 78] Schmitz, Lothar
An exercise in program synthesis:
Algorithms for computing the transitive closure
of a relation
Bericht Nr. 7801
Hochschule der Bundeswehr
München, 1978

[Stro 70] Strong, H.R.
Translating recursion equations into flow-charts
Journal of Computer and System Sciences 5,
pp. 254-285
New York, 1970

In der Literaturliste wurden folgende Abkürzungen benutzt:

ACM	Association for Computing Machinery
JACM	Journal of the ACM
CACM	Communications of the ACM
SIGART	Special Interest Group on ARTifical Inteligence
SIGPLAN	Special Interest Group on Programming LANguages
TUM-INFO	Informationen der Technische Universität München
D.A.I	Department of Artifical Intelligence University of Edinburgh
IRIA	Institut de Recherche d'Informatique et d'automatique
BIT	Nordisk tidskrift for informations behandling
Proc.	Proceedings
Symp.	Symposium
Int.	International
Vol.	Volume (Band)
No. Nr.	Number (Nummer)
pp.	Pages (Seiten)

Stichwortverzeichnis

ablauforientierte Methode 58 f.
Abstraktion 85
Agenda 105
allgemeiner Ansatz 50
Anfangswert 30
Anwendungsstrategie 86
Arithmetisierung des Ablaufs 50, 56
ASPIK-Sprache 9
Assoziativität 27
Aufrufstruktur 95

Baumdiagramme 56
Bedingungsfolgen 44
Beweise 69 ff.
Bubble Sort 100

case-Konstrukte 8
CIP-System 92
COMBINING 93
Compiler 98
CUOTS (call-user-or-tablesearch) 27

DEDALUS-System 92
Default-Generatoren 97
Definitionen 85
Dekomposition 100
Dekonstruktion 100
Detaillierung 51, 56

Einbettung 51, 65, 93
einfache Funktionen 8
Elimination von Rekursionen 6
Entflechtung 50 f., 66, 82
eureka 87
Exchange Sort 100
Expansion 42, 84 f.

FAKTOR (Anzahl paralleler rekursiver Aufrufe) 18
Fakultätsfunktion 61, 89
Fibonaccifunktion 65, 86
Flußdiagramme 6
Funktionsumkehr ohne stack 30, 62, 72
Funktionsumkehr mit stack 32, 63, 73

Gesetze 85

hängende Operationen 28

if-then-else-Konstrukte 8
Insertion Sort 100
Instanziierung 85
iterative Funktionsdefinition 89

knowledge-based-system 6
Kontextblöcke 97

LEMMATA 97

Merge Sort 100
Metaprogramme 96
Modularität 91

neutrales Element 27
NPL 91

Operandenvertauschung 28, 62, 70

Parameterkeller 49
Parameterwerte 30
Permutation 99
Programmsynthese 84, 90, 101
Programmtransformation 6
Protokollkeller 49
Protoprogramm 91

Quicksort 100

Rechtskommutativität 28
recursion introduction 67
Redefinitionsregel 89
redundante Berechnungen 84
Reduktion 42, 84 f.
Regelaufbau
    Aktionsteil 16
    Bedingungsteil 16
Regelbasis 8
Regelsystem 7
Rekursion
    explizite, direkte 8
    implizite, indirekte 8
    wechselseitige 10
    einfache 17
    mehrfache 17
    lineare 14
    einfach geschachtelte 15, 34, 74 ff.
    verschränkt geschachtelte 15, 38, 78
    linear verschränkt geschachtelte 15, 44, 80
    kaskadenartige 16, 49, 82
Rekursionsgleichungen 1.Ordnung 84
repetitive Funktionsdefinition 14
Rückwärtsinduktion 71

Seiteneffekte 11
Selection Sort 100
Selektion 99
Sortieralgorithmen 99
stack 32
Standardform 11
Stelligkeit 21, 24
STI (Schachtelungstiefe) 18
Strukturdiagramme 93
strukturorientierte Methode 58,59

tabulation 67
Telegrammproblem 92
Textformatierer 99
Transformationen 24
  mustergesteuerte 95
  zielgerichtete 95
Transformationsblöcke 98
Transformationsschema 24
transitive Hülle 101
Transparenz 84
TUPLING 94

Umkehrfunktionen 30
Um-Klammerung (Klammernverschiebung) 26, 61, 69
unfold/fold-Methode 84 ff.
  unfolding 85
  folding 85
  forced-folding 85, 101
UNFOLD 97
UNFOLDALL 97
USING 97

Verwendbarkeit 96

Wertverlaufstabellierung 50

Zählervariablen 41
ZAP-Sprache 96
ZAP-System 95

91-Funktion (von Manna) 64