Kapitel 5

Automatisierte Softwareentwicklung

In den vorherigen Kapiteln haben wir die Grundlagen für die Implementierung eines universellen Basisinferenzsystems geschaffen, mit dem Schlußfolgerungen über alle Bereiche der Mathematik und Programmierung formal gezogen und zum Großteil automatisiert werden können. Alle allgemeinen Techniken sind von ihrer Konzeption her vorgestellt und im Hinblick auf ihre prinzipiellen Möglichkeiten und Grenzen ausführlich diskutiert worden.

Im Anfangskapitel hatten wir als Motivation für die Entwicklung von formalen Kalkülen, Inferenzsystemen und Beweisstrategien besonders die rechnergestützte Softwareentwicklung hervorgehoben. Wir wollen nun auf dieses Ziel zurückkommen und uns mit der praktischen Anwendbarkeit von Inferenzsystemen als Unterstützung bei der Entwicklung garantiert korrekter Software befassen. Wir wollen vor allem Techniken vorstellen, mit denen das allgemeine Beweisentwicklungssystem zu einem leistungsfähigen semi-automatischen System für die Synthese von Programmen aus formalen Spezifikationen ausgebaut werden kann.

Viele dieser Techniken sind in den letzten Jahrzehnten zunächst unabhängig von formalen logischen Kalkülen entstanden[1] und ausgetestet worden. Wir werden sie daher zunächst losgelöst von einer typentheoretischen Formalisierung – wohl aber angepaßt an die verwendete Notation – besprechen und erst im Anschluß daran diskutieren, auf welche Art die bekannten Synthesestrategien in den allgemeinen logischen Formalismus integriert[2] werden können, der die Korrektheit einer Synthese sicherstellen kann.

Programmsynthese geht davon aus, daß die Beschreibung des zu lösenden Problems bereits als eine Spezifikation vorliegt, die in einer präzisen formalen Sprache formuliert ist. Bevor man sich also mit den einzelnen Techniken und Verfahren der Programmsynthese beschäftigen kann, muß man wissen, auf welche Arten sich eine Formalisierung informal gegebener Begriffe und Zusammenhänge realisieren läßt. Die Formalisierung mathematischer Theorien, die wir in Abschnitt 5.1 exemplarisch besprechen werden, bildet die Grundlage einer präzisen Beschreibung von Programmierproblemen und ebenso aller Verfahren, die aus dieser Beschreibung einen korrektes Programm generieren sollen. Während sich Abschnitt 5.1 auf die Formalisierung verschiedener Anwendungsbereiche konzentriert, werden wir in Abschnitt 5.2 die formalen Grundbegriffe vorstellen, die für eine einheitliche Betrachtung verschiedenster Programmsyntheseverfahren erforderlich sind.

Die folgenden drei Abschnitte sind dann den den verschiedenen Methoden zur Synthese von Programmen aus formalen Spezifikationen gewidmet. Abschnitt 5.3 betrachtet die Verfahren, die auf dem bereits

[1]Die Literatur zur Programmsynthese ist extrem umfangreich und es ist nahezu unmöglich, einen Überblick über alle Verfahren zu behalten. Die wichtigsten dieser Verfahren sind in [Green, 1969, Burstall & Darlington, 1977, Manna & Waldinger, 1979, Manna & Waldinger, 1980, Bibel, 1980, Gries, 1981, Hogger, 1981, Bibel & Hörnig, 1984, Dershowitz, 1985, Bates & Constable, 1985, Franova, 1985, Smith, 1985b, Heisel, 1989, Neugebauer *et.al.*, 1989, Smith & Lowry, 1990, Galmiche, 1990, Fribourg, 1990, Bibel, 1991, Franova & Kodratoff, 1991, Lowry, 1991, Smith, 1991a] beschrieben worden. Die Unterschiede zwischen diesen Verfahren liegen vor allem in den konkreten Schwerpunkten, der Effizienz ihrer Implementierung und der Art der notwendigen Benutzerinteraktion begründet.

[2]Die Integration von Synthesestrategien in einen allgemeinen logischen Formalismus ist derzeit ein aktuelles Forschungsgebiet. Die hier präsentierten Wege beschreiben die derzeit in Untersuchung befindlichen Möglichkeiten und sind keineswegs vollends ausgereift. Für eine Vertiefung im Rahmen von Studien- und Diplomarbeiten ist dieses Thema daher besonders geeignet.

Notation	Meaning
$\mathbb{B}$, true, false	Data type of boolean expressions, explicit truth values
$\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$	Boolean connectives
$\forall$x$\in$S.p, $\exists$x$\in$S.p	Limited boolean quantifiers (on finite sets and sequences)
if p then a else b	Conditional
Seq(α)	Data type of finite sequences over members of α
null?, $\in$, $\sqsubseteq$	Decision procedures: emptiness, membership, prefix
[], [a], [i..j], [a_1...a_n]	Empty/ singleton sequence, subrange, literal sequence former
a.L, L·a	prepend a, append a to L
[f(x)|x$\in$L$\wedge$p(x)], |L|, L[i]	General sequence former, length of L, i-th element,
domain(L), range(L)	The sets {1..|L|} and {L[i]|i$\in$domain(L)}
nodups(L)	Decision procedure: all the L[i] are distinct (no duplicates)
Set(α)	Data type of *finite* sets over members of α
empty?, $\in$, $\subseteq$	Decision procedures: emptiness, membership, subset
$\emptyset$, {a}, {i..j}, {a_1...a_n}	Empty set, singleton set, integer subset, literal set former
S+a, S-a	element addition, element deletion
{f(x)|x$\in$S$\wedge$p(x)}, |S|	General set former, cardinality
S$\cup$T, S$\cap$T, S\T	Union, intersection, set difference
$\bigcup$FAMILY, $\bigcap$FAMILY	Union, intersection of a family of sets

Abbildung 5.1: Erweitertes Vokabular

bekannten Prinzip "Beweise als Programme" basieren. Abschnitt 5.4 bespricht dann Syntheseverfahren auf der Basis äquivalenzerhaltender Formeltransformationen, wobei wir besonders auch auf das LOPS-Verfahren [Bibel, 1980] eingehen werden. Abschnitt 5.5 schließlich behandelt Verfahren, die auf einer Instantiierung schematischer Algorithmen beruhen, und illustriert diese am Beispiel dreier Algorithmenentwurfsstrategien, die in das KIDS System des Kestrel Instituts in Palo Alto [Smith, 1990, Smith, 1991a] – aus praktischer Hinsicht derzeit das erfolgreichste aller Systeme – integriert sind. Die Möglichkeiten einer nachträglichen Optimierung synthetisch erzeugter Algorithmen besprechen wir dann in Abschnitt 5.6.

Die hier vorgestellten Techniken sind bisher nur wenig systematisiert und vereinheitlicht worden, da sie größtenteils unabhängig voneinander entstanden sind. Der interessierte Leser ist daher meist auf ein intensives Studium einer Vielfalt von Originalarbeiten angewiesen. Mit diesem Kapitel soll versucht werden, die Thematik in einer möglichst einheitlichen Weise darzustellen. Wegen der großen Bandbreite von Denkweisen, die hinter den einzelnen Verfahren stehen, können wir jedoch nicht allen Ansätzen gleichermaßen gerecht werden und sind sogar gezwungen, manche von ihnen gänzlich zu ignorieren.

ACHTUNG: *Dieser Teil des Skriptes ist bis auf weiteres noch in unvollständigem Zustand. Die Abschnitte 5.1, 5.2, und 5.3 beschreiben nur das Notwendigste, was für ein Verständnis der Hauptteile von Bedeutung ist. Eine endgültige Version wird etwas ausführlicher auf die in der Vorlesung angesprochenen Details eingehen.*

5.1 Verifizierte Implementierung mathematischer Theorien

Dieser Abschnitt befaßt sich vor allem mit den konservativen Erweiterungen der Typentheorie, die für eine Formalisierung von Grundbegriffen der wichtigsten Anwendungsbereiche erforderlich sind, welche in der Programmierung immer wieder vorkommen. Neben der Einführung von Vokabular (siehe Abbildung 5.1) geht es hierbei vor allem darum, die grundlegensten Gesetze dieser Begriffe als verifizierte Lemmata der NuPRL-Bibliothek zu formalisieren.

Notwendigerweise muß dabei über eine Systematik beim Aufbau mathematischer Theorien gesprochen werden (vor allem auch darüber, welche Lemmata fundamental sind und welche nicht) und über die technischen Hilfsmittel, wie diese Systematik in einem formalen System umgesetzt werden kann. Hauptsächlich geht es dabei um eine Unterstrukturierung des Wissens in Theorien *und um die Anordnung von Lemmata in der Wissensbank – zum Beispiel durch eine Systematische Vergabe von Lemma-Namen entsprechend der Theorie, dem Hauptoperator und dem Nebenoperator, die in dem Lemma vorkommen. Diese Systematik, die im Detail derzeit noch erforscht wird, liegt der Anordnung der in Anhang B zusammengestellten Definitionen und Lemmata zugrunde.*

5.2 Grundkonzepte der Programmsynthese

In diesem Abschnitt werden vor allem die Grundbegriffe vorgestellt, die für eine einheitliche Betrachtung verschiedenster Programmsyntheseverfahren erforderlich sind. Hierzu werden zunächst die einzelnen Phasen einer systematischen Programmentwicklung angesprochen, um zu verdeutlichen, welche Begriffe unbedingt erforderlich sind.

Eine Präzisierung der Begriffe "formale Spezifikation, Programm, Korrektheit und Erfüllbarkeit von Spezifikationen" als Grundlage jeglicher weiteren Verarbeitung im Syntheseprozeß schließt sich an und wird an einem Beispiel illustriert. Anschließend wird ein kurzer Überblick über Synthesekonzepte und historisch relevante Verfahren gegeben, der es erleichtert, die späteren Kapitel einzuordnen.

Definition 5.2.1 (Spezifikationen, Programme und Korrektheit)

1. *Eine (formale) Spezifikation spec ist ein Tupel (D,R,I,O), wobei*
   - *D ein Datentyp ist (der Eingabebereich oder Domain),*
   - *R ein Datentyp (der Ausgabebereich oder Range),*
   - *I ein Prädikat über D (die Input-Bedingung an zulässige Eingaben) und*
   - *O ein Prädikat über D×R die Output-Bedingung an erlaubte Ausgaben)*
2. *Ein (formales) Programm p besteht aus einer Spezifikation spec=(D,R,I,O) und einer berechenbaren Funktion body: $D \not\to R$ (dem Programmkörper).*
3. *Ein Programm p=((D,R,I,O), body) heißt korrekt, falls für alle zulässige Eingaben $x \in D$ mit I(x) die Bedingung O(x,body(x)) erfüllt ist.*

Da es sich in beiden Fällen um formale Konzepte handelt, müssen formale Sprachen zur Beschreibung der Komponenten von Spezifikationen und Programmen benutzt werden. Man unterscheidet im allgemeinen die *Spezifikationssprache*, mit der Ein- und Ausgabebereich, sowie Ein- und Ausgabebedingung formuliert werden von der *Programmiersprache*, in welcher der Programmkörper beschrieben wird. Bei der Erstellung automatisierter Programmsynthesesysteme hat es sich jedoch als sinnvoll herausgestellt, in beiden Fällen dieselbe Sprache zu verwenden oder, genauer gesagt, die Programmiersprache als (berechenbaren) Teil der Spezifikationssprache[3] anzusehen.

Da die Tupelschreibweise *((D,R,I,O), body)* im allgemeinen nur sehr schwer zu lesen ist, verwenden wir in konkreten Fällen meist eine Notation, die zusätzlich einige Schlüsselworter zur syntaktischen Auftrennung zwischen den Komponenten und die Abstraktionsvariablen enthält. Da mengenwertige Funktionen bei der Synthese eine große Rolle spielen werden, geben wir auch hierfür eine besondere Notation an.

Definition 5.2.2 (Syntaktisch aufbereitete formale Notationen)

1. *Eine formale Spezifikation spec=$(D, R, \lambda x.I[x], \lambda x,y.O[x,y])$[4] eines Programms f wird auch durch die Notation*

   FUNCTION $f(x{:}D){:}R$ WHERE $I[x]$ RETURNS y SUCH THAT $O[x,y]$

   beschrieben. Wenn $I[x]$=`true` ist, darf der `WHERE`*-Teil auch entfallen.*

2. *Die Notation*

   FUNCTION $f(x{:}D)$:Set(R) WHERE $I[x]$ RETURNS $\{ y \mid O[x,y] \}$

   steht als Abkürzung für die Spezifikation

   FUNCTION $f(x{:}D)$:Set(R) WHERE $I[x]$ RETURNS S SUCH THAT S = $\{ y{:}R \mid O[x,y] \}$

[3]Bei einer Integration in die Typentheorie ist dies automatisch gewährleistet. Die Spezifikationssprache geht nur in dem Sinne über die Programmiersprache hinaus, daß hier auch unentscheidbare Prädikate (also Funktionen mit Bildbereich $\mathbb{P}_i$) zugelassen sind, wo die Programmiersprache boole'sche Funktionen erwartet. Dieser Unterschied wird allerdings nur selten ausgenutzt.

[4]Es sei daran erinnert, daß $O[x,y]$ Platzhalter für einen beliebigen Ausdruck ist, in dem x und y frei vorkommen dürfen.

3. *Ein Programm* p=((D, R, ,λx.$I[x]$, $\lambda x,y$.$O[x,y]$), `letrec` f(x)=$body[f,x]$) *wird beschrieben durch die Notation*

   FUNCTION f(x:D):R WHERE $I[x]$ RETURNS y SUCH THAT $O[x,y]$ = $body[f,x]$

Damit ist das Ziel der Programmsynthese klar. Es geht darum zu einer gegebenen formalen Spezifikation *spec*=(`D.R,I,O`) eine berechenbare Funktion `body` zu bestimmmen, so daß insgesamt das Programm

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

korrekt ist. Da wir später zeigen wollen, wie man Programmsyntheseverfahren in den formalen Rahmen der Typentheorie integrieren können, müssen wir dieses Ziel als Ziel eines konstruktiven Beweises ausdrücken. Dies ist nicht schwer, da "es ist `body` zu bestimmmen" mit einem Nachweis der Existenz der Funktion `body` gleichgesetzt werden kann. Wir sagen in diesem Fall, daß die Sezifikation *erfüllbar* (oder *synthetisierbar*) ist.

Definition 5.2.3 (Synthetisierbarkeit von Spezifikationen)

Eine formale Spezifikation spec=(`D.R,I,O`) *heißt erfüllbar (synthetisierbar), wenn es eine Funktion* `body:D`$\not\to$`R` *gibt, die das Programm p*=(*spec*,`body`) *korrekt werden läßt.*

Wir wollen das Vorgehen bei der formalen Spezifikation eines Problems an einem Beispiel illustrieren.

Beispiel 5.2.4 (Costas-Arrays: Problemstellung)

In [Costas, 1984] wurde erstmals eine Klasse von Permutationen beschrieben, die für die Erzeugung leicht wiederzuerkennender Radar- und Sonarsignale besonders gut geeignet sind. Seitdem sind eine Reihe der kombinatorischen Eigenschaften dieser sogenannten *Costas-Arrays* untersucht worden, aber eine allgemeine Konstruktionsmethode konnte noch nicht angegeben werden. Das Problem der Aufzählung von Costas-Arrays muß daher durch Suchalgorithmen gelöst werden. Ein effizientes (d.h. nicht-exponentielles) Verfahren konnte erstmals 4 Jahre nach der Beschreibung der Costas-Arrays in [Silverman *et.al.*, 1988] gegeben werden. Das Problem ist mithin schwierig genug, um sich als Testbeispiel für Syntheseverfahren zu eignen. Andererseits erlaubt die mathematische Beschreibung eine relativ einfache Formalisierung des Problems.

Ein *Costas Array der Ordnung n* ist eine Permutation `p` von $\{1..n\}$ in deren Differenzentafel keine Zeile doppelt vorkommende Elemente besitzt. Dabei ergibt sich die erste Zeile der Differenzentafel aus den Differenzen benachbarter Elemente, die zweite aus den Differenzen der Elemente mit Abstand 2 usw. Ein Costas Array der Ordnung 6 und seine Differenzentafel ist unten als Beispiel gegeben.

2	4	1	6	5	3	p
-2	3	-5	1	2		Zeile 1
1	-2	-4	3			Zeile 2
-4	-1	-2				Zeile 3
-3	1					Zeile 4
-1						Zeile 5
						Zeile 6

Das *Costas Arrays Problem* ist die Frage nach einem effizienten Algorithmus, der bei Eingabe einer natürlichen Zahl n *alle* Costas Arrays der Ordnung n bestimmt.

Um dieses Ziel formal zu spezifizieren, müssen wir zunächst alle Begiffe formalisieren, die in der Problemstellung genannt sind, aber noch nicht zum erweiterten Standardvokabular. Anschließend sollten die wichtigsten Gesetze dieser Konzepte als Lemmata formuliert werden, um für die Verwendung während einer Synthese bereitzustehen (sie müssten ansonsten *während* der Synthese hergeleitet werden). Parallel dazu kann die Problemstellung durch eine formale Spezifikation beschrieben werden.

– Die Konzepte *Permutation* und *Differenzentafel* gehören bisher nicht zum formalisierten Vokabular. Eine Liste L ist Permutation einer Menge S, wenn sie alle Elemente von S enthält, aber keine doppelten Vorkommen. Die Differenzentafel einer Liste L schreiben wir zeilenweise auf und verwenden hierzu die Bezeichnung `dtrow(`L`,`j`)` (difference table of L, row j). Eine solche Zeile besteht aus der Differenz der Elemente von L mit Abstand j.

```
perm(L,S)     ≡   nodups(L) ∧ range(L)=S
dtrow(L,j)    ≡   [L[i]-L[i+j] | i∈[1..|L|-j]]
```

– Die wichtigsten mathematischen Gesetze der Permutationen ergeben sich nach Auffalten der Definition aus den Gesetzen von `nodups` und `range` und brauchen nicht mehr aufgeführt zu werden.

Für `dtrow` stellen wir – wie immer – vor allem distributive Eigenschaften bezüglich Kombination mit anderen Standardoperationen auf endlichen Folgen auf und erhalten die folgenden Lemmata.

```
∀L,L':Seq(ℤ).∀i:ℤ.∀j:ℕ.
1.            dtrow([],j)   = []
2. j≤|L|  ⇒  dtrow(i.L,j) = (i-L[j]).dtrow(L,j)
3. j≠0    ⇒  dtrow([i],j) = []
4. L⊑L'   ⇒  dtrow(L,j)   ⊑ dtrow(L',j)
5. j≥|L|  ⇒  dtrow(L,j)   = []
6. j≤|L|  ⇒  dtrow(L·i,j) = dtrow(L,j)·(L[|L|+1-j]-i)
```

– Damit sind alle Begriffe geklärt und wir können die formale Spezifikation aufstellen: das Programm soll bei Eingabe von $n \in \mathbb{Z}$ alle Permutationen `p`∈`Seq(ℤ)` von $\{1..n\}$ berechnen, deren Differenzentafeln in keiner Zeile j doppelte Elemente haben. Dies macht nur dann Sinn, wenn $n \geq 1$ ist.

Da Zeilen in der Differenzentafel von `p` per Definition endliche Folgen sind, können wir die Funktion `nodups` verwenden. Zudem reicht es, Zeilenindizes zwischen 1 und n bzw. aus `domain(p)` zu betrachten, da bei größeren Indizes `dtrow(p,`j`) = []` ist. Das erlaubt die Verwendung des beschränkten All-Quantors und wir erhalten als Ausgabebedingung

```
perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))
```

Damit haben wir alle Komponenten der Problemspezifikation bestimmt, setzen diese in das Beschreibungsschema für mengenwertige Programmspezifikationen aus Definition 5.2.2 ein und erhalten die folgende formale Spezifikation des Costas Arrays Problems.

```
FUNCTION Costas (n:ℤ):Seq(ℤ)  WHERE n≥1
  RETURNS { p: | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j)) }
```

5.3 Programmentwicklung durch Beweisführung

Die grundsätzliche Denkweise des Prinzips "Beweise als Programme" ist eines der fundamentalen Konzepte der intuitionistischen Typentheorie. Wir hatten in Abschnitt 3.4 bereits ausführlich darüber gesprochen.

In diesem Abschnitt soll der Zusammenhang zu den eben eingeführten Konzepten der formalen Spezifikation gezogen werden, um eine Vergleichbarkeit zu anderen Syntheseparadigmen zu erreichen. Konkrete Synthesestrategien auf der Basis des Prinzips "Beweise als Programme" haben üblicherweise einen Induktionsbeweiser als Kernbestandteil. Wir werden dies nur kurz am Beispiel des Oyster/Clam *Systems illustrieren und dann im wesentlichen auf Literatur verweisen, da eine ausführliche Behandlung von Induktionsbeweisern ein Vorlesungsthema für sich ist.*

Zum Schluß sprechen wir über einige Probleme einer "reinen" Synthese von Programmen durch Beweisführung, wobei vor allem das niedrige Niveau der Inferenzschritte und die Schwierigkeiten bei der Konstruktion allgemeinrekursiver Programme zur Sprache kommen.

5.4 Programmentwicklung durch Transformationen

Während das Paradigma "Beweise als Programme" sich im wesentlichen an der Frage nach einer verifizierten Korrektheit von Programmen orientierte, ist eine Steigerung der Effizienz von Programmen der ursprüngliche Hintergrund von transformationsbasierten Verfahren. Dies liegt im wesentlichen daran, daß es für viele Problemstellungen durchaus sehr leicht ist, eine korrekte prototypische Lösung zu erstellen, die sich aufgrund ihrer *abstrakten Beschreibungsform* auch sehr leicht verifizieren und modifizieren läßt. Diese Eleganz erkauft man sich jedoch oft mit einer relativ großen Ineffizienz.

Aus diesem Grunde hat man sich schon sehr früh mit der Frage befaßt, wie man Programme systematisch in effizientere Programme mit gleichem extensionalen Verhalten *transformieren*[5] kann [Darlington, 1975, Burstall & Darlington, 1977, Clark & Sickel, 1977]. Die Fortsetzung dieser Idee auf die Synthese von Programmen aus formalen Spezifikationen [Manna & Waldinger, 1975, Darlington, 1975, Manna & Waldinger, 1979] basiert vor allem auf dem Gedanken, daß eine Spezifikation eigentlich auch als ein sehr ineffizientes – genauer gesagt, nicht ausführbares – Programm betrachtet werden kann, das nun dahingehend verbessert werden muß, daß alle nichtausführbaren Bestandteile durch effizientere ersetzt werden müssen, also durch Ausdrücke, die von einem Computer berechnet werden konnten.

Seit dem Aufkommen des Programmierens in der Sprache der Prädikatenlogik – also der Denkweise der Programmiersprache Prolog – lassen sich Programmtransformationen mehr oder weniger mit Transformationen logischer Formeln identifizieren[6] und sind somit für logische Inferenzsysteme, *Rewrite-Techniken* und Theorembeweiser-basierte Verfahren handhabbar geworden. Viele Verfahren haben aus diesem Grunde auch eine logische Programmierprache als Zielsprache und habe sich zur Aufgabe gesetzt, die Problemspezifikation in eine den Hornklauseln verwandte Formel umzuwandeln, die dann direkt in ein Prolog-Programm übertragen werden kann. Im Gegensatz zur Konstruktion von Programmen durch Beweise läßt sich das Ziel einer Synthese jedoch nicht so deutlich fixieren. Es wird vielmehr durch die internen Vorgaben der Strategien definiert, welche die Anwendung von Transformationen als *Vorwärts*inferenzen steuern.

Aufgrund der Dominanz logik-basierter Verfahren bei der Synthese durch Transformationen und zugunsten einer besseren Vergleichbarkeit mit den anderen Syntheseparadigmen wollen im folgenden eine logische Beschreibungsform für transformationsbasierte Verfahren verwenden. Alle andersartigen Ansätze lassen sich ohne Probleme in diese Form übersetzen.

5.4.1 Synthese durch Transformation logischer Formeln

Aus logischer Sicht besteht eine Synthese durch Transformationen im wesentlichen daraus, daß die Ausgabebedingung einer Spezifikation als ein *neues Prädikat* aufgefaßt wird, dessen Eigenschaften nun mit Hilfe von (bedingten) Äquivalenztransformationen der Ausgabebedingung hergeleitet werden sollen, bis diese Eigenschaften konkret genug sind, um eine rekursive Beschreibung des neuen Prädikates anzugeben, in der nur noch "auswertbare" Teilformeln auftreten. Interpretiert man das neue Prädikat nun als den Kopf eines Logikprogramms so kann die rekursive Beschreibung als der entsprechende Programmkörper angesehen werden, der nun im Sinne der Logik-Programmierung auswertbar ist.

---

[5] Programmtransformationssysteme mit dem Ziel der Optimierung von Programmen [Burstall & Darlington, 1977, Broy & Pepper, 1981, Partsch & R., 1983, Krieg-Brückner *et.al.*, 1986, Bauer & others, 1988, Krieg-Brückner, 1989] sind mittlerweile ein eigenes Forschungsgebiet geworden, auf dem sehr viel wertvolle Arbeit geleistet wurde. Manche von ihnen verstehen sich durchaus auch als Programmentwicklungssysteme, die den Weg von ineffizienten zu effizienten Spezifikationen von Algorithmen unterstützen. Auch wenn es aus praktischer Sicht durchaus legitim ist, direkt mit einer prototypische Lösung anstelle einer formalen deskriptiven Spezifikation zu beginnen, findet eine Synthese im eigentlichen Sinne hierbei jedoch nicht statt.

[6] Nichtsdestotrotz variieren die individuellen Formalismen sehr stark, da viele Ansätze auch auf der Basis einer funktionalen Sichtweise entwickelt wurden. Die wichtigsten Verfahren sind beschrieben in [Manna & Waldinger, 1975, Hogger, 1978, Manna & Waldinger, 1979, Barstow, 1979, Bibel, 1980, Hogger, 1981, Kant & Barstow, 1981, Dershowitz, 1985, Balzer, 1985, Sato & Tamaki, 1989, Lau & Prestwich, 1990].

Die Vorgehensweise bei der Synthese eines durch D, R, I und O spezifierten Algorithmus ist also die folgende.

1. Zunächst wird ein neues Prädikat F über $D{\times}R$ definiert durch die Formel

   $\forall \mathtt{x}{:}D.\forall \mathtt{y}{:}R.\ I[x] \quad\Rightarrow\quad \mathtt{F(x,y)} \Leftrightarrow O[x,y]$[7]

   Diese Formel hat die äüßere Gestalt einer bedingten Äquivalenz und wird für spätere Inferenzen der Wissensbank des Systems (temporär) hinzugefügt.

2. Unter Verwendung aller aus der Wissensbank bekannten Äquivalenzen – einschließlich der soeben definierten – wird die obige Formel nun so lange transformiert, bis eine Formel der Gestalt

   $\forall \mathtt{x}{:}D.\forall \mathtt{y}{:}R.\ I[x] \quad\Rightarrow\quad \mathtt{F(x,y)} \Leftrightarrow O^*[F,x,y]$

   erreicht wurde, wobei O^* nur aus erfüllbaren Prädikaten und rekursiven Vorkommen von F bestehen darf. Zugunsten einer besseren Übersichtlichkeit darf O^* hierbei auch in mehrere Teilprobleme zerlegt worden sein, die durch separate, neu eingeführte Prädikate beschrieben werden. Am Ende des Transformationsprozesses müssen für deren Definitionsformeln allerdings die gleichen Bedingungen gelten, so daß diese Zerlegung wirklich rein "kosmetischer" Natur ist.

   Welche Regeln angewandt werden müssen, um dieses Ziel zu erreichen, ist zunächst nicht weiter festgelegt.[8] Dies ist eher eine Frage der konkreten Synthesestrategie, welche die anzuwendenden Regeln nach einer internen Heuristik auswählt.

3. Zur Erzeugung eines Programms aus der erreichten Zielformel gibt es nun zwei Möglichkeiten. Man kann die Formel mehr oder weniger direkt *als Logik-Programm interpretieren* und muß in diesem Falle nur die Syntax entsprechend anpassen. Es besteht aber auch die Möglichkeit, durch Anwendung von *Programmformationsregeln*, die logische Konnektive in Programmstrukturen übersetzen, imperative oder funktionale Programme zu erzeugen.

Bei dieser Art der Synthese spielen sogenannte *äquivalenzerhaltende Transformationen* eine zentrale Rolle. Diese basieren im wesentlichen auf bedingten Äquivalenzen und Gleichheiten der Gestalt

$$\forall \mathtt{z}{:}\mathtt{T}.\ P[z] \;\Rightarrow\; Q[z] \Leftrightarrow Q'[z] \qquad \text{bzw.} \qquad \forall \mathtt{z}{:}\mathtt{T}.\ P[z] \;\Rightarrow\; t[z] = t'[z],$$

die von nahezu allen Lemmata des Anhang B eingehalten wird. Dabei werden diese Äquivalenzen und Gleichheiten im allgemeinen als gerichtete *Rewrite-Regeln* behandelt, die meist von links nach rechts gelesen werden. Sie ersetzen in einer Formel jedes Vorkommen einer Teilformel $Q[z]$ durch $Q'[z]$ (bzw. eines Terms $t[z]$ durch $t'[z]$, sofern im Kontext dieser Teilformel die Bedingung $P[z]$ nachgewiesen[9] werden kann. Dabei bestehen die Äquivalenzen und Gleichheiten, auf die das Inferenzsystem zugreifen kann, aus den *vorgefertigten Lemmata* der Wissensbank, den *Definitionen neu erzeugter Prädikate*, elementaren *logischen Tautologien* und Abstraktionen, sowie aus dynamisch erzeugten Kombinationen dieser Äquivalenzen.

Da die Leistungsfähigkeit von Syntheseverfahren mittels Äquivalenztransformationen wesentlich von den eingesetzten Strategien abhängt, richtet sich die derzeitige Forschung vor allem auf effiziente Rewrite-Techniken und Heuristiken, sowie auf den Entwurf leistungsfähiger Transformationsregeln. Die Algorithmenschemata, die wir in Abschnitt 5.5 ausführlich behandeln werden, können als eine sehr ausgereifte Form einer Transformationsregel auf hohem Abstraktionsniveau angesehen werden.

---

[7]Die Verwendung der Notation $O[x,y]$ soll verdeutlichen, daß es sich bei O um eine komplexere Formel mit freien Variablen x und y handelt, während F tatsächlich für einen Prädikatsnamen steht.

[8]Auf den ersten Blick erscheint dies sehr ziellos. Bei der Suche nach Beweisen einer Aussage ist man jedoch im Prinzip in derselben Lage, denn es steht auch nicht fest, welche Regeln zum Beweis nötig sind. Ausschließlich das Ziel – es muß ein Beweis für die gegebene Aussage sein – steht fest. Ein wichtiger Unterschied ist aber, daß die Anzahl der anwendbaren Regeln bei Beweisverfahren erheblich kleiner und ein zielorientiertes Top-Down Suchverfahren möglich ist – solange man nicht in größerem Maße auf Lemmata angewiesen ist. In diesem Fall steht man vor derselben Problematik wie bei der Synthese durch Transformationen.

[9]Auf diese bedingten Äquivalenzen und Gleichheiten werden wir im Rahmen kontextabhängiger Simplifikationen in Abschnitt 5.6.1 noch einmal zurückkommen. Im Prinzip würde es reichen, anstelle von Äquivalenzen auch umgekehrte Implikationen zu verwenden, wodurch sich dann eine *Verfeinerung* des Programms ergibt. Hierbei verliert man jedoch zwangsläufig mögliche Lösungen, was nicht immer gewünscht sein mag.

Wir wollen an einem Beispiel illustrieren, wie man mit Hilfe von äquivalenerhaltenden Transformationen Programme aus formalen Spezifikationen erzeugen kann. Auf die Strategie, mit deren Hilfe wir die anwendbaren Äquivalenzen finden, wollen wir dabei vorerst nicht eingehen, sondern nur demonstrieren, daß das Paradigma “Synthese durch Transformationen” zumindest aus theoretischer Sicht vollständig ist.

Beispiel 5.4.1 (Maximale Segmentsumme)

In Beispiel 3.4.6 auf Seite 147 hatten wir das Problem der Berechnung der maximalen Summe $m = \Sigma_{i=p}^{q} L[i]$ von Teilsegmenten einer nichtleeren Folge L ganzer Zahlen formalisiert und mit Hilfe des Prinzips “Beweise als Programme” gelöst. Dabei hatten wir uns auf eine ausführliche Analyse des Problems gestützt, die wir in Beispiel 1.1.1 auf Seite 2 aufgestellt hatten. Wir wollen nun zeigen, daß sich dieses Problem durch Verwendung von Äquivalenztransformationen auf ähnliche Weise lösen läßt.

Wir beginnen mit der Definition eines neuen Prädikats maxseg, das wir über eine Äquivalenz einführen.

$$\forall L{:}Seq(\mathbb{Z}).\forall m{:}\mathbb{Z}.\ L\neq[]\ \Rightarrow\ maxseg(L,m)\ \Leftrightarrow\ m = MAX(\bigcup\{\{\Sigma_{i=p}^{q}L[i]\ |p\in\{1..q\}\}\ |\ q\in\{1..|L|\}\})$$

Gemäß unserer Analyse in Beispiel 1.1.1 beginnen wir damit, daß wir das Problem generalisieren und simultan zur Berechnung der maximalen Segmentsumme immer auch die maximale Summe a von Anfangssegmenten berechnen. Wir führen hierzu ein Hilfsprädikat max_aux mit drei Variablen L, m und a ein, und stützen maxseg hierauf ab: m ist die maximale Segmentsumme von L, wenn man ein a angeben kann, so daß max_aux(L,m,a) gilt.

$$\forall L{:}Seq(\mathbb{Z}).\forall m{:}\mathbb{Z}.\ \ L\neq[]\ \Rightarrow\ maxseg(L,m)\ \Leftrightarrow\ \exists a{:}\mathbb{Z}.\ max_aux(L,m,a)$$

$$\forall L{:}Seq(\mathbb{Z}).\forall m,a{:}\mathbb{Z}.\ L\neq[]\ \Rightarrow\ max_aux(L,m,a)\ \Leftrightarrow$$
$$m = MAX(\bigcup\{\{\Sigma_{i=p}^{q}L[i]\ |p\in\{1..q\}\}\ |\ q\in\{1..|L|\}\})\ \ \wedge\ a = MAX(\{\Sigma_{i=1}^{q}L[i]\ |q\in\{1..|L|\}\})$$

Unter der Voraussetzung, daß max_aux für jede Wahl von L erfüllbar ist, ist dies tatsächlich eine Äquivalenzumformung. In den folgenden Schritten betrachten wir nur noch das Prädikat max_aux. Wir beginnen mit einer Fallanalyse über die Gestalt von L und verwenden hierzu das Lemma

$$\forall L{:}Seq(\mathbb{Z}).\ L\neq[]\ \Rightarrow\ L=[first(L)]\ \vee\ rest(L)\neq[]$$

Da die Vorbedingung dieses Lemmas erfüllt ist, können wir auf der rechten Seite der Äquivalenz die Disjunktion L=[first(L)] ∨ rest(L)≠[] ergänzen und die disjunktive Normalform bilden.

$$\forall L{:}Seq(\mathbb{Z}).\forall m,a{:}\mathbb{Z}.\ L\neq[]\ \Rightarrow\ max_aux(L,m,a)\ \Leftrightarrow$$
$$L=[first(L)]\ \wedge\ m = MAX(\bigcup\{\{\Sigma_{i=p}^{q}L[i]\ |p\in\{1..q\}\}\ |\ q\in\{1..|L|\}\})$$
$$\wedge\ a = MAX(\{\Sigma_{i=1}^{q}L[i]\ |q\in\{1..|L|\}\})$$
$$\vee\ rest(L)\neq[]\ \ \wedge\ m = MAX(\bigcup\{\{\Sigma_{i=p}^{q}L[i]\ |p\in\{1..q\}\}\ |\ q\in\{1..|L|\}\})$$
$$\wedge\ a = MAX(\{\Sigma_{i=1}^{q}L[i]\ |q\in\{1..|L|\}\})$$

Nun verwenden wir (aufbereitete) Lemmata über die Kombination von maximaler Segment- bzw. Anfangssumme und einelementigen Listen und Listen mit nichtleerem Rest.

– Die maximale Segmentsumme einer Liste [i] ist i.

$$\forall L{:}Seq(\mathbb{Z}).\forall i,m{:}\mathbb{Z}.\ L=[i]\ \Rightarrow\ m = MAX(\bigcup\{\{\Sigma_{i=p}^{q}L[i]\ |p\in\{1..q\}\}\ |\ q\in\{1..|L|\}\})\ \ \Leftrightarrow\ m=i$$

– Die maximale Anfangssumme einer Liste [i] ist i.

$$\forall L{:}Seq(\mathbb{Z}).\forall i,a{:}\mathbb{Z}.\ L=[i]\ \Rightarrow\ a = MAX(\{\Sigma_{i=1}^{q}L[i]\ |q\in\{1..|L|\}\})\Leftrightarrow\ a=i$$

– Die maximale Segmentsumme einer Liste L=i.L', wobei L' nichtleer ist, ist das Maximum der maximalen Segmentsumme von L' und der maximalen Anfangssumme von L.

$$\forall L{:}Seq(\mathbb{Z}).\forall i,a{:}\mathbb{Z}.\ rest(L)\neq[]\ \ \Rightarrow\ \ m = MAX(\bigcup\{\{\Sigma_{i=p}^{q}L[i]\ |p\in\{1..q\}\}\ |\ q\in\{1..|L|\}\})$$
$$\Leftrightarrow\ \ \exists a{:}\mathbb{Z}.\ a = MAX(\{\Sigma_{i=1}^{q}L[i]\ |q\in\{1..|L|\}\})$$
$$\wedge\ \exists m'{:}\mathbb{Z}.\ m' = MAX(\bigcup\{\{\Sigma_{i=p}^{q}rest(L)[i]\ |p\in\{1..q\}\}\ |\ q\in\{1..|rest(L)|\}\})$$
$$\wedge\ m=max(m',a)$$

– Die maximale Anfangssumme einer Liste L=i.L', wobei L' nichtleer ist, ist das Maximum von i und der Summe von i und der maximalen Anfangssumme von L'

$$\forall L{:}Seq(\mathbb{Z}).\forall i,a{:}\mathbb{Z}.\ rest(L)\neq[]\ \ \Rightarrow\ \ a = MAX(\{\Sigma_{i=1}^{q}L[i]\ |q\in\{1..|L|\}\})$$
$$\Leftrightarrow\ \exists a'{:}\mathbb{Z}.\ a' = MAX(\{\Sigma_{i=1}^{q}rest(L)[i]\ |q\in\{1..|rest(L)|\}\})\ \ \wedge\ \ a=max(first(L),\ a'+first(L))$$

Wenden wir diese Lemmata nacheinander als Transformationen auf die obige Formel an (wobei wir im dritten Schritt kein neues a mehr einführen müssen), so ergibt sich

$$\forall\texttt{L:Seq}(\mathbb{Z}).\forall\texttt{m,a:}\mathbb{Z}.\ \texttt{L}\neq\texttt{[]}\ \Rightarrow\ \texttt{max_aux(L,m,a)}\ \Leftrightarrow$$
$$\texttt{L=[first(L)]}\ \wedge\ \texttt{m=first(L)}\ \wedge\ \texttt{a=first(L)}$$
$$\vee\ \texttt{rest(L)}\neq\texttt{[]}\ \wedge\ \exists\texttt{m':}\mathbb{Z}.\ \texttt{m'=MAX(}\bigcup\{\{\Sigma_{i=p}^{q}\texttt{rest(L)[i]}\,|\,\texttt{p}\in\{\texttt{1..q}\}\}\,|\,\texttt{q}\in\{\texttt{1..|rest(L)|}\}\})$$
$$\wedge\ \exists\texttt{a':}\mathbb{Z}.\ \texttt{a'=MAX(}\{\Sigma_{i=1}^{q}\texttt{rest(L)[i]}\,|\,\texttt{q}\in\{\texttt{1..|rest(L)|}\}\})$$
$$\wedge\ \texttt{a=max(first(L), a'+first(L))}$$
$$\wedge\ \texttt{m=max(m',a)}$$

Den ersten Fall können wir aufgrund der Gleichung `m = first(L)` zu `L=[m]` $\wedge$ `m = first(L)` $\wedge$ `a = m` vereinfachen. Aufgrund des Lemmas `L=[m]` $\Rightarrow$ `m = first(L)` kann das mittlere Konjunkt dann entfallen. Im zweiten Fall erlaubt uns der Kontext `rest(L)`$\neq$`[]`, die Definition von `max_aux` für `rest(L)` wieder zurückzufalten. Wir erhalten insgesamt

$$\forall\texttt{L:Seq}(\mathbb{Z}).\forall\texttt{m:}\mathbb{Z}.\ \ \texttt{L}\neq\texttt{[]}\ \Rightarrow\ \texttt{maxseg(L,m)}\ \Leftrightarrow\ \exists\texttt{a:}\mathbb{Z}.\ \texttt{max_aux(L,m,a)}$$

$$\forall\texttt{L:Seq}(\mathbb{Z}).\forall\texttt{m,a:}\mathbb{Z}.\ \texttt{L}\neq\texttt{[]}\ \Rightarrow\ \texttt{max_aux(L,m,a)}\ \Leftrightarrow$$
$$\texttt{L=[m]}\ \wedge\ \texttt{a=m}$$
$$\vee\ \texttt{rest(L)}\neq\texttt{[]}\ \wedge\ \exists\texttt{m',a':}\mathbb{Z}.\ \texttt{max_aux(rest(L),m',a')}$$
$$\wedge\ \texttt{a=max(first(L), a'+first(L))}\ \wedge\ \texttt{m=max(m',a)}$$

Dieses Formelpaar kann nun ohne großen Aufwand schrittweise in ein logisches Programm überführt werden. Hierzu lassen wir alle Quantoren entfallen, da in logischen Programmen alle Variablen der linken Seite all-quantifiziert und alle zusätzlichen Variablen der rechten Seite existentiell quantifiziert sind. Die Vorbedingungen können im Programmcode ebenfalls entfallen. Dies führt zu der vereinfachten Form

```
maxseg(L,m) ⇔ max_aux(L,m,a)

max_aux(L,m,a) ⇔
     L=[m] ∧ a=m
   ∨ rest(L)≠[]  ∧ max_aux(rest(L),m',a') ∧ a=max(first(L), a'+first(L)) ∧  m=max(m',a)
```

Nun zerlegen wir die Disjunktion in zwei separate Bedingungsklauseln, was wiederum der Semantik von logischen Programmen entspricht. In der letzten Klausel kann dann die Bedingung `rest(L)`$\neq$`[]` entfallen, da sie der Nichtanwendbarkeit der zweiten Klauselfolgt. Da Prolog-Programme – ähnlich zum ML-Abstraktionsmechanismus– Listen automatisch strukturell zerlegen können schreiben wir `i.L'` anstelle von `L`, um uns die Notationen `first` und `rest` zu ersparen. Dies liefert

```
maxseg(L,m) ⇔ max_aux(L,m,a)

max_aux(L,m,a)     ⇐  L=[m] ∧ a=m
max_aux(i.L',m,a)  ⇐  max_aux(L',m',a') ∧ a=max(i, a'+i) ∧  m=max(m',a)
```

Die Semantik logischer Programme erlaubt nun, in der zweiten Klausel die Gleichungen in den Programmkopf zu verlagern. Da Funktionen in logischen Programmen nicht erlaubt sind, ersetzen wir die Funktion `max` durch das entsprechende prädikative Programm `MAX`, dessen dritte Komponente die Ausgabe der Berechnung des Maximums zweier Zahlen ist. Nun ersetzen wir noch $\wedge$ durch ein Komma und $\Leftrightarrow$ und $\Leftarrow$ durch `:-` und wir haben ein Prolog-Programm zur Berechnung der maximalen Segmentsumme einer nichtleeren Folge ganzer Zahlen erzeugt.

```
maxseg(L,m)      :- max_aux(L,l,m).
max_aux([m],m,m).
max_aux(a.L',l,m):- max_aux(L,m',l'),max(a,l'+a,l),max(l,m',m).
```

In diesem Beispiel ist noch sehr vieles von Hand geschehen und auch die Auswahl der Lemmata war sehr speziell. Auch wäre ohne eine vorherige Analyse das Problem nicht zu lösen gewesen, da man sonst sehr wahllos nach anwendbaren Lemmata gesucht hätte. Diese Möglichkeit steht einer maschinell gesteuerten Synthese, in der alle Schritte durch eine Strategie bestimmt werden müssen, ebensowenig zur Verfügung wie die auf das Problem zugeschnittenen Lemmata. Eine Strategie muß daher nach wesentlich allgemeineren Kriterien vorgehen und eine Methodik verfolgen, die in manchen Fällen zwar etwas umständlich sein mag, aber (fast) immer zum Erfolg führt.

Mit einer konkreten Synthesestrategie auf der Basis äquivalenzerhaltender Transformationen werden wir uns im nächsten Abschnitt befassen. Zuvor wollen wir im Rest dieses Abschnitts jedoch noch den abschließenden Schritt einer Synthese, also der Erzeugung konkreter Programme diskutieren. In Beispiel 5.4.1 hatten wir zur Illustration ein logisches Programm generiert. Wir wollen die hierbei eingesetzten *Programmformationsregeln* kurz zusammenstellen.

Strategie 5.4.2 (Formation logischer Programme)
Eine Menge bedingter Äquivalenzen der Gestalt

$$\forall \mathtt{x}{:}D.\forall \mathtt{y}{:}R.\quad I[x] \;\Rightarrow\; \mathtt{F(x,y)} \;\Leftrightarrow\; O^*[\mathtt{F},\mathtt{F}_i,x,y]$$
$$\forall \mathtt{x}_1{:}D_1.\forall \mathtt{y}_1{:}R_1.\quad I_1[x_1] \;\Rightarrow\; \mathtt{F}_1(\mathtt{x}_1,\mathtt{y}_1) \;\Leftrightarrow\; O_1^*[\mathtt{F},\mathtt{F}_i,x_1,y_1]$$
$$\vdots$$
$$\forall \mathtt{x}_n{:}D_n.\forall \mathtt{y}_n{:}R_n.\quad I_n[x_n] \;\Rightarrow\; \mathtt{F}_n(\mathtt{x}_n,\mathtt{y}_n) \;\Leftrightarrow\; \mathtt{O}_n^*[\mathtt{F},\mathtt{F}_i,x_n,y_n]$$

wobei auf der linken Seite nur atomare Formeln stehen, wird durch sukzessive Anwendung der folgenden Regeln in ein logisches Programm umgewandelt.

- *Die äußeren Allquantoren entfallen.*
  (Alle Variablen im Programmkopf sind implizit allquantifiziert.)

- *Existenzquantoren auf der rechten Seite der Äquivalenz entfallen.*
  (Alle neuen Variablen im Programmkörper sind implizit existenzquantifiziert.)

- *Alle Vorbedingungen entfallen*
  (Vorbedingungen werden vom Programm nicht überprüft)

- *Äquivalenzen mit einer unerfüllbaren rechten Seite entfallen*
  (Der Programmaufruf würde keine Lösung finden können)

- *Disjunktionen auf der rechten Seite werden zerlegt in zwei sequentielle Äquivalenzen mit der gleichen*[10] *linken Seite. Hierzu wird die rechte Seite zunächst auf disjunktive Normalform gebracht.*
  (Zwei Klauseln mit demselben Programmkopf gelten als Alternativen zur Lösung des Problems und werden der Reihe nach abgearbeitet.)

- *Funktionesaufrufe der Gestalt* `y=g(x)` *werden durch die entsprechenden (bereits bekannten) prädikativen Programmaufrufe der Gestalt* `G(x,y)` *ersetzt.*
  (Logische Programme kennen nur prädikative Programme)

- *Zerlegungen von Standarddatentypen wie* `first(L)/rest(L)` *werden durch Strukturmuster wie* `x.L'` *anstelle von* `L` *im Programmkopf ersetzt.*
  (Eine strukturelle Zerlegung durch Pattern-Matching geschieht automatisch)

- *Gleichheiten zwischen Variablen der rechten Seite können durch entsprechende Substitutionen der Variablen auf der linken Seite der Äquivalenz ersetzt werden.*
  (Durch Pattern-Matching wird der Ausgabevariablen der Wert der entsprechenden Eingabe zugewiesen.)

Für logische Programme sind die Formationsregeln verhältnismäßig einfach, da im wesentlichen nur noch syntaktische Anpassungen an die Konventionen logischer Programmiersprachen vorgenommen werden müssen. Für die Erzeugung andersartiger Programme sind die Programmformationsregeln ein wenig komplizierter, da nun logische Konnektive in *Programmstrukturen* übersetzt werden können. Zugunsten einer Vergleichbarkeit der Paradigmen geben wir diese Regeln nun auch für die funktionale Programmierung an.

[10] Der Sonderfall, daß die rechte Seite nur aus Disjunktionen einiger F_i besteht, kann optimiert werden. Da nun mehrere Klauseln nur aus dem Aufruf eines F_i bestehen, können all diese F_i umbenannt werden in dasselbe Prädikat. Die dadurch entstehenden redundanten Klauseln können entfernt werden.

Dabei gehen wir wie bei der Formation logischer Programme davon aus, daß alle vorkommenden atomaren Formeln bereits als erfüllbar bekannt oder rekursive Varianten einer der linken Seiten sind. Die Formationsregeln müssen also nur angeben, wie man aus bekannten Teillösungen eine Gesamtlösung zusammensetzt, welche die zusammengesetzte Spezifikation erfüllt. Hierzu geben wir für jedes logische Konnektiv eine eigene Regel an, wobei – ebenfalls wie bei logischen Programmen – Implikationen und Negationen nicht individuell verarbeitet werden können. Wir erwarten also am Ende der Transformationen eine gewisse Normalform.

Die einfachste Formationsregel gibt es für die *Konjunktion*. Wenn O einzeln erfüllbar ist, und die Ausgabebedingung $O[x,y] \wedge P(x,y)$ erfüllt werden soll, so kann man P als eine Art Filter für die Lösungen von O betrachten, die man zuvor berechnet hat. Dies bedeutet, daß für alle Eingaben, deren Lösung auch die Bedingung P erfüllt, einfach diese Lösung übernommen wird. Eine Konjunktion in der Ausgabebedingung entspricht also einer *Restriktion* der Menge der legalen Eingaben.

Lemma 5.4.3 (Formationsregel für die Konjunktion)

Es sei spec$=(D,R,I,O)$ *eine beliebige Spezifikation,* $P\colon D\times R\to\mathbb{B}$ *ein Prädikat. Die Funktion* $body\colon D\not\to R$ *erfülle die Spezifikation* FUNCTION F(x:D):R WHERE $I[x]$ RETURNS y SUCH THAT $O[x,y]$. *Dann wird die Spezifikation*

FUNCTION F(x:D):R WHERE $I[x]$ RETURNS y SUCH THAT $O[x,y] \wedge P(x,y)$

von der Funktion body *erfüllt, falls für alle* $x\in D$ *mit* $I[x]$ *die Eigenschaft* $P(x,body(x))$ *gilt.*

Diese Regel hat natürlich nur einen sehr engen Anwendungsbereich, da sie nicht darauf eingeht, ob für die kombinierte Ausgabebedingung eine andere Lösungsfunktion besser geeignet wäre. Eine angemessenere Regel kann man eigentlich nur für mengenwertige Programme angeben, da nur bei diesen Programmen ausgedrückt werden kann, daß die *Menge* der Lösungen eingeschränkt wird zu $\{\, y \mid y\in body(x) \wedge P(x,y)\}$.

Eine *Disjunktion* hatte bei logischen Programmen zu zwei alternativen Programmdefinitionsteilen geführt, die gemäß der Sematik logischer Programme der Reihe nach verarbeitet werden: die zweite Alternative wird nur dann aufgerufen, wenn die erste nicht zum Erfolg führt. Dies bedeutet, daß die zweite Alternatie genau dann aufgerufen wird, wenn die (nicht explizit vorhandene) Eingabebedingung der ersten nicht mehr erfüllt ist. In einer funktionalen Denkweise entspricht dies einer *Fallunterscheidung*: die Lösungsfunktion des ersten Disjunkts wird aufgerufen, wenn ihre Vorbedingung erfüllt ist und ansonsten die Lösungsfunktion der zweiten.

Lemma 5.4.4 (Formationsregel für die Disjunktion)

Es seien spec$=(D,R,I,O)$ *und* spec'$=(D,R,I',O')$ *beliebige Spezifikationen, die von den Funktionen* $body\colon D\not\to R$ *bzw.* $body'\colon D\not\to R$ *erfüllt werden. Dann wird die Spezifikation*

FUNCTION F(x:D):R WHERE $I[x] \vee I'[x]$ RETURNS y SUCH THAT $O[x,y] \vee O'[x,y]$

erfüllt von der Funktion λx. if I[x] then $body$(x) else $body'$(x).

Der *Existenzquantor* führte bei logischen Programmen dazu, daß für die neue Variable ein Wert bestimmt werden mußte, der dann zur Bestimmung der Lösung weiter verarbeitet wurde. Hierbei gibt es zwei Lesarten.

- Im ersten Fall läßt sich die Ausgabebedingung zerlegen in ein e Teilbedingung O, bei der die neue Variable eine Ausgabe beschreibt, und eine Teilbedingung O', bei der sie als Eingabe verwendet wird. Der Existenzquantor dient dann der Beschreibung einer *kaskadischen Berechnung*.

- Im anderen Fall ist die neue Variable nur eine zusätzliche Ausgabevariable eines anderen Prädikates O, das nicht weiter untergliedert werden kann. Bei diesem Prädikat handelt es sich dann um eine G*eneralisierung* des Problems und der Existenzquantor deutet an, daß nur eine der Ausgabevariablen von Interesse ist.

Lemma 5.4.5 (Formationsregeln für den Existenzquantor)

1. *Es seien* spec=(D,R,I,O) *und* spec'=$(D{\times}R,R',I',O')$ *beliebige Spezifikationen, die von den Funktionen* $body{:}D\not\to R$ *bzw.* $body'{:}D{\times}R\not\to R'$ *erfüllt werden. Dann wird die Spezifikation*

   FUNCTION F(x:D):R' WHERE $I[x]$ RETURNS z SUCH THAT $\exists y{:}R.\ O[x,y] \wedge O'(x,y,z)$

   erfüllt λx. $body'$(x,$body$(x)), *falls für alle* $x\in D$ *mit* $I[x]$ *die Eigenschaft* $I'(x,body(x))$ *gilt.*

2. *Es sei* spec=$(D,R'{\times}R,I,O)$ *eine beliebige Spezifikationen, die von der Funktionen* $body{:}D\not\to R'{\times}R$ *erfüllt werde. Dann wird die Spezifikation*

   FUNCTION F(x:D):R WHERE $I[x]$ RETURNS z SUCH THAT $\exists y{:}R'.\ O[x,y,z]$

   von der Funktion λx. $body$(x).1 *erfüllt*

Eine weitere Formationsregel ergibt sich aus der Tatsache, daß Transformationen meist auf eine *rekursive Beschreibung* des Programms abzielen. Während diese bei Logikprogrammen ohne weitere Änderung übernommen werden kann (das Prädikat 'ist' das Programm), muß in einer funktionalen Denkweise ein rekursives Programm erzeugt werden. Hierbei gilt die rekursive Beschreibung der Ausgabebedingung als Voraussetzung für die Formationsregel, taucht aber in der Spezifikation nicht so direkt auf wie in den bisherigen Regeln. Für die totale Korrektheit des generierten Programms muß zudem sichergestellt sein, daß die Rekursion auch *terminiert*, daß also die Eingabe nicht beliebig reduziert werden kann. Die folgende Formationsregel beschreibt die einfachste Version der Erzeugung rekursiver Programme: die Ausgabebedingung läßt sich zerlegen in eine rekursive Variante und eine Reihe von Bedingungen, in denen die rekursiv berechnete Teillösung zu einer Gesamtlösung zusammengesetzt wird.

Satz 5.4.6 (Formationsregel für rekursiv zerlegbare Prädikate)

Es sei spec=$(D,R'{\times}R,I,O)$ *eine beliebige Spezifikationen,* f_d:D$\not\to$D *wohlfundierte 'Reduktionsfunktion',*[11] O_C: $D{\times}D{\times}R{\times}R\to\mathbb{B}$ *und* $body{:}D{\times}D{\times}R\not\to R$. *Gilt für alle* $x,x_r\in D$, $y,y_r\in R$ *mit* $I[x]$

1. $I[x] \Rightarrow I[f_d(x)/x]$
2. $I[x] \Rightarrow O[x,y] \Leftrightarrow \exists y_r{:}R.\ O[f_d(x),y_r] \wedge O_C(x,f_d(x),y_r,y)$
3. $I[x] \Rightarrow O_C(x,x_r,y_r,\ body(x,x_r,y_r))$

so wird die Spezifikation

FUNCTION F(x:D):R WHERE $I[x]$ RETURNS z SUCH THAT $O[x,y]$

erfüllt von der Funktion letrec f(x) = $body$(x, f_d(x), f(f_d(x)))

Diese Regel entspricht der Formationsregel für Divide & Conquer Algorithmen, die wir in Abschnitt 5.5.4 ausführlicher besprechen werden. Natürlich ist sie eine drastische Vereinfachung, da im Allgemeinfall das Ergebnis der Transformationen auch mehrere rekursive Varianten der Ausgabebedingung auf der rechten Seite enthalten kann. Die entsprechende allgemeine Regel ist nur wenig anders, bedarf aber einiger zusätzlicher[12] Formalia. Anhand einer Analyse der inneren Struktur rekursiver Funktionen macht man sich relativ schnell klar, daß diese Regel vollständig ist, daß sich also jede berechenbare Funktion schreiben läßt als

letrec f(x) = let $x_1,..,x_n$ = f_d(x) in $body$(x, $x_1,..,x_n$, f(x_1), ..., f(x_n))

und somit einer rekursiven Zerlegung der Ausgabebedingung in der obengenannten Art entspricht.

Theorem 5.4.6 beschreibt die wesentliche häufigere $\wedge$-*Reduktion* eines Problems, bei der die Spezifikation in eine Konjunktion von Bedingungen zerlegt wird und zur Berechnung einer einzelnen Lösung *alle* Teillösungen nötig sind. Möglich ist aber auch eine sogenannte $\vee$-*Reduktion*, bei der die verschiedenen Lösungen der Teilprobleme jeweils direkt zu einer Lösung des Gesamtproblems beitragen. In einer rein funktionalen Denkweise

[11] Eine Reduktionsfunktion f_d:D$\not\to$D heißt wohlfundiert, wenn es keine unendlichen absteigenden Ketten der Form x, f_d(x), f_d(f_d(x)), f_d(f_d(f_d(x))), ... gibt.

[12] Die Reduktionsfunktion f_d müsste so beschrieben werden, daß sie die Eingabe x in eine Menge reduzierter Eingaben abbildet. Anstelle von O müsste auf der rechten Seite eine Menge von Bedingungen stehen, deren Ein- und Ausgaben entsprechend von O_C weiterverwendet werden. Eine entsprechende Notation zur Kennzeichnung dieser Mengen müßte erst eingeführt werden.

kann dies nur durch die Verwendung mengenwertiger Funktionen (im Sinne von Definition 5.2.2.2) exakt wiedergegeben werden, da andernfalls die entstehende Fallunterscheidung wieder einer $\wedge$-Reduktion entspricht. Überhaupt muß festgestellt werden, daß Synthese durch Transformation mehr auf die Beschreibung *aller Lösungen* einer Ausgabebedingung abzielt und somit logische Programme oder mengenwertige Funktionen als Zielsprache adäquater erscheinen als Funktionen, die nur eine Lösung berechnen. Der hierzu notwendige Formalismus ist bisher allerdings noch nicht aufgestellt worden und bleibt ein Ziel für zukünftige Forschungen.

Wir wollen die allgemeine Diskussion von Synthese durch Transformationen mit dem Beispiel einer Programmformation unter Verwendung der obigen Regeln abschließen.

Beispiel 5.4.7 (Maximale Segmentsumme: Programmformation)

In Beispiel 5.4.1 hatten wir das Problem der maximalen Segmentsumme durch Transformationen in folgende rekursive Beschreibung umgewandelt.

```
∀L:Seq(ℤ).∀m:ℤ.   L≠[] ⇒  maxseg(L,m) ⇔ ∃a:ℤ. max_aux(L,m,a)

∀L:Seq(ℤ).∀m,a:ℤ. L≠[] ⇒  max_aux(L,m,a) ⇔
     L=[first(L)]  ∧ m=first(L) ∧ a=first(L)
   ∨ rest(L)≠[]    ∧ ∃m',a':ℤ. max_aux(rest(L),m',a')
                   ∧ a=max(first(L), a'+first(L))  ∧  m=max(m',a)
```

`max_aux` ist eine Generalisierung von `maxseg` im Sinne von Lemma 5.4.5.2 und besitzt selbst eine rekursive Beschreibung. Um Theorem 5.4.6 anzuwenden, müssen wir nur noch die dort vorkommenden Parameter mit den aktuellen Komponenten in Beziehung setzen.

`x` entspricht `L`, `y` entspricht dem Paar `m,a`, $\texttt{f}_d$`(x)` entspricht $\texttt{L}_r$`=rest(L)`, $\texttt{y}_r$ entspricht `m',a'` und $\texttt{O}_C$`(L,`$\texttt{L}_r$`,m',a',m,a')` ist die Bedingung

```
   L=[first(L)] ∧ m=first(L) ∧  a=first(L)
 ∨ Lr≠[] ∧ a=max(first(L),a'+first(L)) ∧ m=max(m',a)
```

$\texttt{f}_d$ ist offensichtlich wohlfundiert und eine Lösung für $\texttt{O}_C$ ist die Funktion

```
λL,Lr,m',a'. if Lr=[] then (first(L),first(L))
                      else let a=max(first(L), a'+first(L)) in  (max(m',a),a)
```

Insgesamt ergibt sich somit folgende funktionale Lösung des Problems der maximalen Segmentsumme

```
FUNCTION Maxseg(L:Seq(ℤ)):ℤ  WHERE L≠[] RETURNS m
  SUCH THAT m=MAX(⋃{ {Σ_{i=p}^q L[i] |p∈{1..q}} | q∈{1..|L|} })
=  snd(Max_aux(L))

FUNCTION Max_aux(L:Seq(ℤ)):ℤ×ℤ  WHERE L≠[]  RETURNS m,l
  SUCH THAT m=MAX(⋃{ {Σ_{i=p}^q L[i] |p∈{1..q}} | q∈{1..|L|} })  ∧  a=MAX({Σ_{i=1}^q L[i] |q∈{1..|L|} })
= if rest(L)=[] then (first(L),first(L))
                else let (m',l') = Max_aux(rest(L)) in
                       let a=max(first(L), a'+first(L)) in
                         (max(m',a),a)
```

5.4.2 Die LOPS Strategie

LOPS – eine Abkürzung für logische Programmsynthese – ist ein transformationsbasiertes Syntheseverfahren, welches auf einer Reihe von Strategien basiert, die erstmalig in [Bibel, 1978, Bibel *et.al.*, 1978, Bibel, 1980, Bibel & Hörnig, 1984] beschrieben wurden. Den Mittelpunkt dieses Verfahrens bilden zwei Strategien namens GUESS-DOMAIN und GET-REC. GUESS-DOMAIN versucht einen Teil der Spezifikation zu bestimmen, mit dessen Hilfe man das Problem so in kleinere Teilprobleme zerlegen kann, daß sich die Gesamtlösung aus Lösungen der Teilprobleme ergibt. Auf der Basis dieser Zerlegung versucht GET-REC dann eine rekursive Lösung des Problems zu erzeugen. Diese beiden Strategien werden unterstützt durch eine Reihe weiterer Strategien zur Vor- und Nachverarbeitung sowie zur Vereinfachung von Teilproblemen. Aus dem Resultat der Transformationen wird in einem abschließenden Schritt ein Programm erzeugt, wobei man im wesentlichen der im vorigen Abschnitt beschriebenen Methode folgt.

Im Gegensatz zu den meisten Ansätzen ist die LOPS Strategie ein typischer Vertreter eines Verfahrens aus der *künstlichen Intelligenz*. Es basiert im wesentlichen auf *syntaktischen Suchverfahren*, die – zugunsten einer Begrenzung des Suchraumes – durch semantische Informationen und Heuristiken gesteuert werden. Dies wird vor allem durch die Vorstellung motiviert, daß ein Mensch bei der Programmierung ähnliche Fähigkeiten einsetzt wie bei der Lösung anderer Probleme und daß somit *allgemeine* Techniken, die auf Raten von Lösungen und Reduktionen auf geklöste Probleme basieren, Anwendung finden. Wir wollen im folgenden die LOPS-Strategie anhand eines einfachen Leitbeispiels entwickeln und zum Schluß in geschlossener Weise zusammenstellen.

Beispiel 5.4.8 (Maximum-Problem)

Das Maximum-Problem besteht darin, einen Algorithmus zu generieren, der aus einer gegebenen nichtleeren Menge S das maximale Element m bestimmt. Wie immer beginnt eine Synthese mit der Definition eines neuen Prädikats max, das wir über eine Äquivalenz einführen.

$$\forall S{:}Set(\mathbb{Z}).\forall m{:}\mathbb{Z}.\ S\neq\emptyset \ \Rightarrow\ max(S,m) \ \Leftrightarrow\ (m\in S \ \wedge\ \forall x\in S.\ x\leq m)$$

S wird als Input-Variable und m als Output-Variable gekennzeichnet. Die Ausgabebedingung ist nicht auswertbar und bedarf daher einer Transformation.

5.4.2.1 GUESS-DOMAIN

Als ein typischer KI-Ansatz beginnt eine LOPS-Synthese mit dem Versuch, Informationen über die Ausgabewerte zu *raten*. Ein solches Raten kann erfolgreich sein, was bedeutet, daß man entweder die gesamte Lösung oder zumindest einen Teil davon geraten hat, oder fehlschlagen, was uns erlaubt, die geratenen Werte aus der Menge der zu betrachtenden Möglichkeiten zu streichen. In jedem Falle bedeutet das Raten aber einen Informationsgewinn für die weiteren Syntheseschritte.

Beispiel 5.4.9

Für die Lösung des Maximum-Problems können wir versuchen, die gesamte Lösung zu raten. Wir führen hierzu eine neue "Guess"-Variable g ein und drücken die Tatsache, daß wir entweder erfolgreich geraten haben oder nicht, durch die Tautologie $g{=}m \vee g\neq m$ aus.

Natürlich müssen die Möglichkeiten für die Auswahl von g auf irgendeine Weise limitiert werden, die mit der ursprünglichen Problemstellung zu tun hat, da ansonsten aus algorithmischer Sicht nichts gewonnen wird. Wir begrenzen daher das Raten auf Werte $g\in S$ und nehmen diese "Domain"-Bedingung für g zusätzlich in die Formel mit auf. Insgesamt wird das Maximum-Problem durch den Rateschritt in die folgende Formel transformiert.

$$\forall S{:}Set(\mathbb{Z}).\forall m{:}\mathbb{Z}.\ S\neq\emptyset \ \Rightarrow\ max(S,m) \ \Leftrightarrow\ \exists g{:}\mathbb{Z}.\ g\in S \ \wedge\ m\in S \wedge \forall x\in S.x\leq m \ \wedge\ (g{=}m \vee g\neq m)$$

Nun trennen wir die beiden Fälle auf, indem wir die Disjunktion $g{=}m \vee g\neq m$ über die ursprüngliche Ausgabebedingung distributieren.

$$\forall S{:}Set(\mathbb{Z}).\forall m{:}\mathbb{Z}.\ S\neq\emptyset \ \Rightarrow$$
$$max(S,m) \ \Leftrightarrow\ \exists g{:}\mathbb{Z}.\ g\in S \wedge\ (m\in S \wedge \forall x\in S.x\leq m \wedge g{=}m) \ \vee\ (m\in S \wedge \forall x\in S.x\leq m \wedge g\neq m)$$

Damit ist klar, wie das Problem gelöst werden muß: wir müssen zunächst einen Wert $g\in S$ bestimmen und dann die beiden Fälle einzeln lösen, wobei wir g nun als Eingabewert betrachten dürfen. Um dies deutlicher zu machen, führen wir zwei neue Prädikate max1 und max2 ein, welche den Erfolgs- bzw. Mißerfolgsfall kennzeichnen und zerlegen wir das Problem in drei Teilprobleme.

$$\forall S{:}Set(\mathbb{Z}).\forall m{:}\mathbb{Z}.\ S\neq\emptyset \ \Rightarrow\ max(S,m) \ \Leftrightarrow\ \exists g{:}\mathbb{Z}.\ g\in S \ \wedge\ (max1(S,g,m) \vee max2(S,g,m))$$
$$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.\ S\neq\emptyset \ \wedge\ g\in S \ \Rightarrow\ max1(S,g,m) \ \Leftrightarrow\ m\in S \ \wedge\ \forall x\in S.\ x\leq m \ \wedge\ g{=}m$$
$$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.\ S\neq\emptyset \ \wedge\ g\in S \ \Rightarrow\ max2(S,g,m) \ \Leftrightarrow\ m\in S \ \wedge\ \forall x\in S.\ x\leq m \ \wedge\ g\neq m$$

Die Lösung des ersten Problems steht fest, sobald die beiden anderen gelöst sind, da es leicht ist, einen Wert $g\in S$ zu berechnen. Das zweite Problem ist trivial zu lösen, da hier die Ausgabe m identisch mit der Eingabe g sein soll. Uns bleibt also noch die Lösung des Mißerfolgsfalls, wobei uns nun erheblich mehr Informationen zur Verfügung stehen als zu Beginn der Synthese.

Aus syntaktischer Sicht ist Raten also dasselbe wie die Einführung einer neuen Guess-Variablen g. Dieses Raten muß einerseits mit dem ursprünglichen Problem in Beziehung gesetzt werden, damit man die geratenen Werte auch verwenden kann, und andererseits auf irgendeine Weise limitiert werden, damit es algorithmisch einigermaßen effizient durchgeführt werden kann.

Für das erste muß man eine mögliche Relation[13] t zwischen der Guess-Variablen g und der Ausgabevariablen y fixieren, die nach dem Raten entweder erfüllt ist oder nicht. Im einfachsten Fall ist dies, wie in Beispiel 5.4.9 illustriert, die Gleichheit. Wenn die Ausgaben jedoch strukturierte Objekte wie Listen, Mengen, Bäume etc. sind, kann man kaum davon ausgehen, die gesamte Ausgabe in einem Schritt zu raten. Daher muß es auch möglich sein, nur einen Teil der Ausgabe zu raten, den man z.B. in einem Schritt auf dem Bildschirm darstellen könnte, und die Relation t entsprechend als $g \in S$, g=first(S), etc. zu wählen. Da hierfür nur wenige sinnvolle Möglichkeiten bestehen, die im wesentlichen nur von der Struktur der Ausgaben abhängt, empfiehlt es sich, die Informationen über zulässige *Guess-Schemata* in einer Wissensbank abzulegen.

Definition 5.4.10 (Guess-Schema)

Ein Guess-Schema ist ein 4-Tupel $(R, G, t, \mathit{displayable})$, *wobei* R *und* G *Datentypen sind,* $t\colon R{\times}G \to \mathbb{B}$, $\mathit{displayable}\colon R \to \mathbb{B}$ *und die Eigenschaft* $\forall y{:}R.\ \mathit{displayable}(y) \Leftrightarrow \exists g{:}G.t(g,y)$ *erfüllt ist.*

Ein Guess-Schema enthält also alle wichtigen Standardinformationen über mögliche Arten, Informationen über einen Ausgabewert zu beschreiben. Während einer Synthese wird ein solches Schema passend zu einem vorgegebenen Ausgabetyp R ausgewählt und dann instantiiert. Die Komponente *displayable* ist eine feste Zusatzinformation, die benötigt wird um festzustellen, ob GUESS überhaupt anwendbar ist oder ob zunächst ein Vorverarbeitungsschritt (siehe Abschnitt 5.4.2.3) durchgeführt werden muß, weil manche Ausgabewerte keine darstellbaren Anteile besitzen, die man raten könnte. Sie könnte im Prinzip aus t hergeleitet werden. Es ist aber effizienter, sie im Schema mit abzuspeichern.

Während die Beziehung zwischen Guess-Variable g und Ausgabevariable y also aus der Wissensbank genommen wird, muß die Begrenzung des Ratens durch eine *Domain-Condition* an g heuristisch bestimmt werden. Eine Heuristik *DOMAIN* muß hierzu eine Bedingung DC bestimmen, die es erlaubt, die geratenen Werte effizient zu bestimmen und die Möglichkeit falscher Rateschritte klein zu halten, ohne dabei aber darstellbare Anteile möglicher Lösungen aus der Betrachtung zu eliminieren. Präzisiert man diese Bedingungen, so ergeben sich folgende Anforderungen an die Heuristik DOMAIN.

1. Raten muß berechenbar sein, d.h. es muß gelten

   FUNCTION $F_D C(x{:}D){:}G$ WHERE $I[x]$ RETURNS y SUCH THAT $DC(x,y)$ ist erfüllbar

   Dies beschränkt die heuristische Suche auf die Auswahl von Prädikaten, die *als erfüllbar bekannt* sind.

2. Alle darstellbaren Anteile möglicher Lösungen müssen durch Raten erreichbar sein, d.h. es muß gelten

   $\forall x{:}D.\forall g{:}G.\ I[x] \Rightarrow (\exists y{:}R.\ O[x,y] \wedge t(g,y)) \Rightarrow DC(x,g)$[14]

   Dies bedeutet, daß sich $DC(x,g)$ normalerweise als eine Vereinfachung der Formel $O[x,y] \wedge t(g,y)$ ergibt, wozu normalerweise (limitierte) Vorwärtsinferenzen eingesetzt werden: man versucht aus der Formel $O[x,y] \wedge t(g,y)$ alle Eigenschaften von x und g zu bestimmen, die sich durch Anwendung weniger Lemmata beweisen lassen. Im einfachen Fall ($t(g,y)$ ist g=y) wählt man üblicherweise eine *echte* Teilmenge der Konjunkte aus $O(x,g)$.

3. Die Menge der zu ratenden Werte $\{g{:}G \mid DC(x,g)\}$ sollte klein sein.

   Hierdurch wird insbesondere auch die Menge der Mißerfolge beim Raten, also $\{g{:}G \mid DC(x,g) \wedge \neg t(g,y)\}$ für jede Lösung y klein gehalten.

4. Raten sollte effizient sein, d.h. es sollte eine schnelle Lösung existieren für das Problem

   FUNCTION $F_D C(x{:}D){:}G$ WHERE $I[x]$ RETURNS y SUCH THAT $DC(x,y)$

[13] Der Name t deutet an, daß diese Relation in tautologischer Weise zur Formel hinzugefügt wird.

[14] In diesem Fall gilt $\{g{:}G \mid \exists y{:}R.\ O[x,y] \wedge t(g,y)\} \subseteq \{g{:}G \mid DC(x,g)\}$

Die letzten beiden Kriterien dienen vor allem der Entscheidungshilfe, wenn mehrere Möglichkeiten für die Auswahl der Domain-Bedingung DC existieren. Sie geben Kriterien für einen Vergleich der offenstehenden Möglichkeiten an und verlangen Abschätzungen, die sich auf semantische Information stützen. Da derartige Informationen derzeit nur sehr unvollständig von einem automatischen System bestimmt werden können, ist in dieser Phase von DOMAIN ein Benutzereingriff sinnvoll.

Für die erfolgreiche Durchführung eines Rateschrittes sind insgesamt also die Wahl eines Guess-Schemas der Wissensbank und die heuristische, zum Teil benutzerunterstützte Auswahl der Domain-Bedingung erforderlich. Die eigentliche Guess-Transformation auf der Basis dieser Informationen verläuft dann ganz schematisch ab: es wird die Domain-Bedingung und die Tautologie zu der Ausgabebedingung ergänzt, die Disjunktion über die Ausgabebedingung distributiert und schließlich das ganze in drei Teilprobleme zerlegt. All dies läßt sich wie folgt in einem Schritt zusammenfassen.

Definition 5.4.11 (GUESS-Transformation)

Es sei spec=(D,R,I,O) *eine beliebige Spezifikation,* $GS = (R, G, t, displayable)$ *ein Guess-Schema und* $DC: D \times G \to \mathbb{B}$*. Dann ist die durch* GS *und* DC *bestimmte* *GUESS-Transformation* *definiert als Transformation der Spezifikationsformel*

$$\forall x{:}D.\forall y{:}R.\ I[x] \ \Rightarrow\ F(x,y) \Leftrightarrow O[x,y]$$

in die Formelmenge

$$\forall x{:}D.\forall y{:}R.\ I[x] \Rightarrow F(x,y) \Leftrightarrow \exists g{:}G.\ DC(x,g) \wedge (F_1(x,g,y) \vee F_2(x,g,y))$$
$$\forall x{:}D.\forall g{:}G.\forall y{:}R.\ I[x] \wedge DC(x,g) \Rightarrow F_1(x,g,y) \Leftrightarrow O[x,y] \wedge t(g,y)$$
$$\forall x{:}D.\forall g{:}G.\forall y{:}R.\ I[x] \wedge DC(x,g) \Rightarrow F_2(x,g,y) \Leftrightarrow O[x,y] \wedge \neg t(g,y)$$

Es läßt sich verhältnismäßig einfach nachweisen, daß diese Transformation tatsächlich äquivalenzerhaltend ist.

5.4.2.2 GET-REC

Nach der Anwendung von GUESS-DOMAIN sind nur noch die beiden neu definierten Teilprobleme zu untersuchen. Von diesen ist normalerweise eines entweder trivial erfüllbar wie imm Falle des Maximum-Problems oder relativ schnell als unlösbar nachzuweisen. Eine Überprüfung dieser beiden Möglichkeiten wird daher unmittelbar im Anschluß an GUESS-DOMAIN durchgeführt. Für die verbleibenden ungelösten Teilprobleme wird nun versucht, Rekursion einzuführen.

Beispiel 5.4.12

Nach der GUESS-Phase ist nur noch das zweite Teilproblem `max2` des Maximum-Problems ungelöst.

$$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.\ S\neq\emptyset \wedge g\in S \Rightarrow max2(S,g,m) \Leftrightarrow m\in S \wedge \forall x\in S.\ x\leq m \wedge g\neq m$$

Wir versuchen nun, die rechte Seite der Äquivalenz in eine rekursive Variante der Ausgangsformel $m\in S \wedge \forall x\in S. x\leq m$ umzuschreiben. Dazu benötigen wir natürlich Informationen darüber, welche Arten von Rekursion auf einer endlichen Menge `S` im Zusammenhang mit einem Element $g\in S$ überhaupt möglich sind, um eine terminierende rekursive Beschreibung des Problems zu erlauben.

Ein einfache sallgemeines Schema zur Reduktion endlicher Mengen mit Hilfe eines Elementes besteht darin, dieses Element aus der Menge hinauszunehmen. Ein solcher Schritt kann nur endlich oft wiederholt werden, bis die Menge leer ist, was bedeutet, daß eine Rekursion wohlfundiert ist.

Wir versuchen daher, Lemmata zu finden, mit denen wir die Formel $m\in S \wedge \forall x\in S.\ x\leq m \wedge g\neq m$ im Kontext aller Vorbedingungen in eine Formel der Gestalt $m\in S\text{-}g \wedge \forall x\in S\text{-}g.\ x\leq m \wedge \ldots$ transformieren können. Zuvor müssen wir jedoch auch die Eingabebedingung $S\neq\emptyset$ mit Hilfe von Lemmata zerlegen in die Form $S\text{-}g\neq\emptyset \vee \ldots$. Damit können wir die Eingabewerte bestimmen, die für eine rekursive Verarbeitung zulässig sind, während wir die anderen direkt behandeln müssen.

Wir suchen also in der Wissensbank gezielt nach Lemmata über Mengendifferenz und leere Menge (B.1.10.7, negiert), Elementrelation (B.1.9.5) und beschränkten Allquantor (B.1.11.6):

B.1.10.7	$S\text{-}g{=}\emptyset \Leftrightarrow S{=}\{g\} \vee S{=}\emptyset$
negiert	$S\text{-}g{\neq}\emptyset \Leftrightarrow S{\neq}\{g\} \wedge S{\neq}\emptyset$
Nach Ergänzung der Disjunktion $S{=}\{g\}$:	$S{=}\{g\} \vee S\text{-}g{\neq}\emptyset \Leftrightarrow S{\neq}\emptyset$
B.1.9.5	$m{\in}S\text{-}g \Leftrightarrow g{\neq}m \wedge m{\in}S$
B.1.11.6	$\forall x{\in}S\text{-}g.\ x{\leq}m \wedge g{\leq}m \Leftrightarrow \forall x{\in}S.\ x{\leq}m$

Nach Anwendung dieser Lemmata und Zerlegung der Disjunktion in der Eingabebedingung erhalten wir

$$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.\ S{=}\{g\} \wedge g{\in}S \Rightarrow max2(S,g,m) \Leftrightarrow m{\in}S \wedge \forall x{\in}S.\ x{\leq}m \wedge g{\neq}m$$
$$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.\ S\text{-}g{\neq}\emptyset \wedge g{\in}S \Rightarrow max2(S,g,m) \Leftrightarrow m{\in}S\text{-}g \wedge \forall x{\in}S\text{-}g.\ x{\leq}m \wedge g{\neq}m \wedge g{\leq}m$$

In der ersten Teilformel stellt sich sofort heraus, daß die Bedingungen $m{\in}S \wedge g{\neq}m$ unter der Voraussetzung $S{=}\{g\}$ widersprüchlich sind. In der zweiten Teilformel können wir nun die Definition des Prädikates `max` mit `S-g` und `m` zurückfalten, da Ein- und Ausgabebedingung in rekursiver Form vorliegen. Zusätzlich fassen wir $g{\neq}m \wedge g{\leq}m$ zusammen. Damit ergeben sich insgesamt die folgenden vier Teilformeln

$$\forall S{:}Set(\mathbb{Z}).\forall m{:}\mathbb{Z}.\ S{\neq}\emptyset \Rightarrow max(S,m) \Leftrightarrow \exists g{:}\mathbb{Z}.\ g{\in}S \wedge (max1(S,g,m) \vee max2(S,g,m))$$
$$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.\ S{\neq}\emptyset \wedge g{\in}S \Rightarrow max1(S,g,m) \Leftrightarrow \forall x{\in}S.\ x{\leq}g \wedge g{=}m$$
$$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.\ S{=}\{g\} \wedge g{\in}S \Rightarrow max2(S,g,m) \Leftrightarrow false$$
$$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.\ S\text{-}g{\neq}\emptyset \wedge g{\in}S \Rightarrow max2(S,g,m) \Leftrightarrow max(S\text{-}g,m) \wedge g{<}m$$

wobei wir die Ausgabebedingung von `max1` ebenfalls durch Substitutionen und Entfernung von Redundanz vereinfacht haben. Nun erweisen sich alle Bestandteile als auswertbar und wir können mit der Strategie 5.4.2 das folgende Logikprogramm zur Lösung des Maximumproblems erzeugen.

```
max(S,M)        :- member(G,S), max-aux(S,G,M).
max-aux(S,G,M) :- setminus(S,G,SminusG), max(SminusG,M), less(G,M),!.
max-aux(S,M,M) :- setless(S,M).
```

Im Gegensatz zu GUESS-DOMAIN besteht die Strategie GET-REC zur Einführung von Rekursion nicht aus einer festgelegten Transformation sondern ist durch ihr Ziel – die Einführung von Rekursion[15] – definiert. Dieses Ziel ist, genau besehen, eine *disjunktive Zerlegung der Eingabebedingung* in eine rekursive Variante der ursprünglichen Eingabebedingung und einer Bedingung an Eingaben, für die Rekursion nicht in Frage kommt, sowie eine *konjunktive Zerlegung der Ausgabebedingung* in eine rekursive Variante der ursprünglichen Ausgabebedingung und zusätzlichen Anforderungen. Zusätzlich soll erreicht werden, daß außer den Ausgangsbedingungen nur noch erfüllbare Prädikate vorkommen.

Die Transformation mit dem Ziel der Rekursion wird im wesentlichen durch eine Suche nach geeigneten Lemmata erreicht, in denen die Beziehung zwischen einer reduzierten und der nichtreduzierten Form der vorkommenden Prädikate behandelt wird. Dies verlangt vor allem nach einer guten Strukturierung der Wissensbank und einer entsprechenden Suchmethode, mit der anwendbare Lemmata schnell aufgespürt[16]

Wesentlich ist aber natürlich auch die Frage, welche Arten von Rekursion überhaupt möglich sind, um einen sinnvollenden terminierenden Algorithmus zu generieren. Da Rekursionseinführung nur dann Sinn macht, wenn auch die geratene Information zur Problemreduktion verwendet wird, wird die Art der Rekursion im allgemeinen von der Guessvariablen und der Eingabevariablen abhängen, also aus einer Reduktionsfunktion bestehen, die zu $x{\in}D$ und $g{\in}G$ einen Wert $x{\searrow}g$ bestimmt, der kleiner ist als `x`. Natürlich muß diese Funktion*wohlfundiert* sein, da sonst keine Terminierungsgarantie gegeben werden könnte. Da letzteres zur Laufzeit einer Synthese eine sehr hohe Belastung des Inferenzsystems bedeuten würde und Rekursionen normalerweise ohnehin nur von den Typen `D` der Eingabewerte und `G` der Guessvariablen abhängen, empfiehlt es sich, auch diese Information als Rekursionsschema in einer Wissensbank abzulegen.

Definition 5.4.13 (Rekursionsschema)

Ein Rekursionsschema ist ein Tripel $(D, G, \searrow)$*, wobei* D *und* G *Datentypen sind und* $\searrow: D{\times}G \to D$ *eine wohlfundierte Reduktionsfunktion.*

[15] Der Name *GET-REC* Goal-oriented Equivalence Transformation) deutet an, daß die Transformation zielorientiert ist.

[16] Eine gute Methode ist die Verwendung von Hashtabellen, die nach dem Kriterium äuserer/innerer Operator erstellt werden. Die Lemmata des Anhang B sind nach einem solchen Kriterium sortiert.

Diese Definition behandelt nur die einfache Form einer Rekursion. Im Allgemeinfall kann eine rekursive Zerlegung einer Eingabe auch mehrere "kleinere" Eingabewerte erzeugen (vergleiche die Diskussion auf Seite 236). Die allgemeinste Form eines Rekursionsschemas besteht daher aus einer wohlfundierten Reduktionsfunktion $\searrow: D \times G \rightarrow \texttt{Set}(D)$, was formal allerdings etwas aufwendiger zu handhaben ist. Unter Verwendung all dieser Informationen geht die Strategie GET-REC insgesamt wir folgt vor.

Strategie 5.4.14 (GET-REC)
Es sei $\forall \texttt{x}{:}D.\forall \texttt{y}{:}R.\ I[\texttt{x}] \Rightarrow \texttt{F(x,y)} \Leftrightarrow O[\texttt{x,y}]$ *die Ausgangsformel der Synthese und* G *der Typ der in der GUESS-Phase bestimmten zusätzlichen Variablen. Die Strategie GET-REC transformiert eine gegebene Spezifikationsformel der Gestalt* $\forall \texttt{x}{:}D.\forall \texttt{g}{:}G.\forall \texttt{y}{:}R.\ \texttt{I'(x,g)} \Rightarrow \texttt{F'(x,g,y)} \Leftrightarrow O'[\texttt{x,g,y}]$ *mit den folgenden Schritten in eine rekursive Form*

1. *Wähle ein Rekursionsschema* $(D,G,\searrow)$ *aus der Wissensbank.*

2. *Durch gezielte Suche nach anwendbaren Lemmata über die Beziehung zwischen einer reduzierten und der nichtreduzierten Form der vorkommenden Prädikate schreibe die Eingabebedingung um in eine Formel der Gestalt* $\texttt{I}_b\texttt{(x,g)} \vee I[\texttt{x}\searrow\texttt{g} / \texttt{x}]$ *und die Ausgabebedingung um in eine Formel der Gestalt* $\exists \texttt{y}_r.\ O[\texttt{x}\searrow\texttt{g,y}_r / \texttt{x,y}]) \wedge \texttt{O}_r\texttt{(x,g,y}_r\texttt{,y)}$

3. *Spalte die nichtrekursiven Lösungen ab, ersetze die Konjunktionsglieder aus* O *durch* F *und erzeuge insgesamt die Äquivalenzformeln*
   $\forall \texttt{x}{:}D.\forall \texttt{g}{:}G.\forall \texttt{y}{:}R.\ \texttt{I}_b\texttt{(x,g)} \Rightarrow \texttt{F'(x,g,y)} \Leftrightarrow O'[\texttt{x,g,y}]$
   $\forall \texttt{x}{:}D.\forall \texttt{g}{:}G.\forall \texttt{y}{:}R.\ \texttt{I(x}\searrow\texttt{g)} \Rightarrow \texttt{F'(x,g,y)} \Leftrightarrow \exists \texttt{y}_r.\ \texttt{F(x}\searrow\texttt{g,y}_r\texttt{)} \wedge \texttt{O}_r\texttt{(x,g,y}_r\texttt{,y)}$

4. *Vereinfache die entstandenen Formeln soweit wie möglich und überprüfe die Auswertbarkeit.*

5.4.2.3 Unterstützende Strategien

Die Strategien GUESS und GET-REC bilden den Kernbestandteil jeder mit LOPS durchgeführten Synthese. In einfachen Fällen reichen sie in der bisher beschriebenen Form aus, um das vorgegebene Problem zu lösen. Normalerweise sind jedoch einige unterstützende Techniken notwendig, um eine LOPS-Synthese erfolgreich zum Ziel zu führen. Neben einigen allgemeinen Techniken wie *Normalisierung*, *Simplifikation* durch gerichtete Lemmata (vergleiche Abschnitt 5.6.1), *Unterproblem-Erzeugung* für komplexere Ausdrücke und Ersetzung nichtauswertbarer Prädikate durch äquivalente auswertbare Prädikate (GET-EP) geht es hierbei vor allem um Strategien zur Unterstützung von GUESS-DOMAIN im zusammenhang mit strukturierten Ausgabewerten.

So kann es zum Beispiel bei induktiven Datenstrukturen Ausgabewerte ohne darstellbare Anteile geben, die durch einen Vorverarbeitungsschritt (*Preprocessing*) abgespalten und separat gelöst werden müssen. Bei mehreren Ausgabewerten (Produkten) ist es sinnvoller, das Raten auf einen Ausgabewert zu fokussieren. Hierzu muß man entweder die Anzahl der Ausgabevariablen durch Bestimmung funktionaler Abhängigkeiten *reduzieren* (GET-RNV), die Ausgabebedingungen separieren, um die Ausgabevariablen separat behandeln zu können (GET-SOC), oder eine hierarchische Lösung versuchen, indem zunächst die bestgeeigneteste Ausgabevariable für das Raten ausgewählt wird (CHVAR) und dann in einer inneren Schleife die anderen Ausgabevariabln behandelt werden, wobei die Abhängigkeiten zwischen den Ausgabevariablen ebenfalls berücksichtigt werden (DEPEND). Wir wollen diese Unterstützungsstrategien im folgenden kurz besprechen.

Preprocessing für GUESS

Bei der bisherigen Beschreibung der Strategie GUESS sind wir davon ausgegangen, daß das Raten einer Lösung bzw. eines Teils davon, im Prinzip immer möglich ist. Für induktive Ausgabedatentypen ist dies jedoch nicht immer der Fall, da zu diesen meist auch ein "leeres" Basisobjekt gehört, welches keinerlei darstellbare Anteile besitzt. Derartige Sonderfälle müssen daher durch einen Vorverarbeitungsschritt abgespalten werden, bevor das eigentliche Raten beginnen kann.

Technisch bedeutet dies, daß man zunächst diejenigen Eingabewerte bestimmen muß, die zu einem Ausgabewert ohne darstellbare Anteile gehören. Während für alle anderen Eingaben die übliche GUESS-DOMAIN Strategie anwendbar ist, muß für diese "primitiven" Werte eine direkte eine Lösung bestimmt werden. Im Allgemeinfall ist dies jedoch nicht sehr schwer, da die Menge $\{\mathtt{y}{:}R \mid \neg\mathtt{displayable(y)}\}$ der Ausgabewerte ohne darstellbare Anteile meist einelementig ist.

Die zentrale Aufgabe des Vorverarbeitungsschrittes ist somit, die primitiven Eingaben zu identifizieren, also ein Prädikat `prim` zu bestimmen mit den Eigenschaften

$$\forall \mathtt{x}{:}D.\ (\exists \mathtt{y}{:}R.\ I[\mathtt{x}] \wedge O[\mathtt{x,y}] \wedge \neg\mathtt{displayable(y)}) \Rightarrow \mathtt{prim(x)}$$
$$\forall \mathtt{x}{:}D.\ I[\mathtt{x}] \wedge \mathtt{prim(x)} \Rightarrow \exists \mathtt{y}{:}R.\ O[\mathtt{x,y}]$$

Im allgemeinen bestimmt man `prim` durch Vereinfachung der Formel $I[\mathtt{x}] \wedge O[\mathtt{x,y}] \wedge \neg\mathtt{displayable(y)}$, wozu normalerweise (limitierte) Vorwärtsinferenzen[17] eingesetzt werden. Mit diesem Prädikat transformiert man nun die Spezifikationsformel

$$\forall \mathtt{x}{:}D. \forall \mathtt{y}{:}R.\ I[\mathtt{x}] \Rightarrow \mathtt{F(x,y)} \Leftrightarrow O[\mathtt{x,y}]$$

in die Formelmenge

$$\forall \mathtt{x}{:}D. \forall \mathtt{y}{:}R.\ I[\mathtt{x}] \wedge \mathtt{prim(x)} \Rightarrow \mathtt{F(x,y)} \Leftrightarrow O[\mathtt{x,y}]$$
$$\forall \mathtt{x}{:}D. \forall \mathtt{y}{:}R.\ I[\mathtt{x}] \wedge \neg\mathtt{prim(x)} \Rightarrow \mathtt{F(x,y)} \Leftrightarrow O[\mathtt{x,y}]$$

Das erste dieser beiden Teilprobleme besitzt üblicherweise eine sehr einfache Lösung – nämlich genau das eine `y`, für das ¬displayable(y) gilt. Das zweite Teilproblem kann nun in der bisher bekannten Weise weiterverarbeitet werden. Wir wollen diese Technik an dem Vorverarbeitungsschritt bei der Synthese von Sortieralgorithmen illustrieren.

Beispiel 5.4.15 (Synthese von Sortieralgorithmen – Vorverarbeitung)

Das Problem der Sortierung von Listen ganzer Zahlen besteht darin, zu einer gegebenen Liste eine Umordnung der Elemente zu finden, die geordnet ist. Dabei ist eine Liste L *geordnet*, wenn ihre Elemente in aufsteigender Reihenfolge angeordnet sind, was sich wie folgt formal definieren läßt

$$\mathtt{ordered}(L) \equiv \forall \mathtt{i} \in \{\mathtt{1..}|L|\mathtt{-1}\}.\ L\mathtt{[i]} \leq L\mathtt{[i+1]}$$

Da die Umordnung von Listen durch das in Anhang B.2 definierte boolesche Prädikat `rearranges` repräsentiert wird, können wir das Sortierproblem nun durch die folgende Äquivalenzformel pezifizieren.

$$\forall \mathtt{L,S{:}Seq}(\mathbb{Z}).\ \mathtt{true} \Rightarrow \underline{\mathtt{SORT(L,S)}} \Leftrightarrow \mathtt{rearranges(L,S)} \wedge \mathtt{ordered(S)}$$

Jedes GUESS-Schema der Wissensbank muß die Ausgabewerte mit darstellbaren Anteilen charakterisieren. Unabhängig von der konkreten Tautologierelation t sind dies genau die nichtleeren Listen ganzer Zahlen, d.h. es gilt $\neg\mathit{displayable}(\mathtt{S}) \equiv \mathtt{S=[]}$. Bevor wir also die Strategie GUESS-DOMAIN auf das Sortierproblem anwenden können, müssen wir den Bereich der "primitiven" Eingaben bestimmen, also ein Prädikat $\mathtt{prim{:}Seq}(\mathbb{Z})\to\mathbb{B}$ herleiten mit der Eigenschaft

$$\exists \mathtt{S{:}Seq}(\mathbb{Z}).\ \mathtt{rearranges(L,S)} \wedge \mathtt{ordered(S)} \wedge \mathtt{S=[]} \Rightarrow \mathtt{prim(L)}$$

Die Substitution von `S` durch `[]` und die Anwendung des Lemmas über die Umordnung leerer Listen (Lemma B.2.26.1) liefert $\mathtt{prim(L)} \equiv \underline{\mathtt{L=[]}}$. Die zweite Bedingung

$$\mathtt{L=[]} \Rightarrow \exists \mathtt{S{:}Seq}(\mathbb{Z}).\ \mathtt{rearranges(L,S)} \wedge \mathtt{ordered(S)}$$

hat trivialerweise die Lösung $\mathtt{S} \equiv \mathtt{[]}$, da es nur eine Art von Listen ohne darstellbare Anteile gibt. Nach der Vorverarbeitung muß nun das folgende Teilproblem des Sortierproblems mit der üblichen LOPS-Strategie gelöst werden.

$$\forall \mathtt{L,S{:}Seq}(\mathbb{Z}).\ \mathtt{L} \neq \mathtt{[]} \Rightarrow \underline{\mathtt{SORT(L,S)}} \Leftrightarrow \mathtt{rearranges(L,S)} \wedge \mathtt{ordered(S)}$$

[17] Diese Strategie könnte zum Beispiel mit Hilfe einer allgemeinen Technik zur Bestimmung der *Vorbedingungen* für die Gültigkeit einer logischen Formel, die in [Smith, 1985b, Section 3.2] beschrieben ist, automatisiert werden.

GET-RNV – Reduce Number of Output Variables

Bei Syntheseproblemen mit mehreren Ausgabewerten würde das raten verhältnismäßig kompliziert, wenn gleichzeitig alle Ausgabewerte geraten werden müssten. Die Strategie GUESS ist daher darauf ausgelegt, jeweils nur einen einzelnen Ausgabewert zu verarbeiten. Um dies zu unterstützen, versucht die Strategie GET-RNV, die Anzahl der Ausgabevariablen des Problems dadurch zu verringern, daß *unmittelbar erkennbare Abhängigkeiten* zwischen den Variablen aufgespürt werden und eine *Ausgabevariable als Funktion der anderen Variablen* beschrieben wird. Dies bedeutet, daß eine Spezifikationsformel der Form

$$\forall \texttt{x}{:}D.\forall \texttt{(y,y')}{:}R{\times}R'.\ I[x] \ \Rightarrow\ \texttt{F(x,y,y')} \Leftrightarrow O[\texttt{x,y,y'}]$$

transformiert wird in eine Formelmenge der Art

$$\forall \texttt{x}{:}D.\forall \texttt{(y,y')}{:}R{\times}R'.\ I[x] \ \Rightarrow\ \texttt{F(x,y,y')} \Leftrightarrow \texttt{F'(x,y)} \wedge \texttt{y'=g}[\texttt{x,y}]$$
$$\forall \texttt{x}{:}D.\forall \texttt{y}{:}R.\ I[x] \ \Rightarrow\ \texttt{F'(x,y)} \Leftrightarrow O[\texttt{x,y,g[x,y]} / \texttt{x,y,y'}]$$

Die Bestimmung der funktionalen Abhängigkeit `y'=g[x,y]` beschränkt sich hierbei auf Zusammenhänge, die unmittelbar mit Hilfe weniger Lemmata nachzuweisen sind. Ansonsten ist diese Strategie nicht anwendbar.

Beispiel 5.4.16 (Reduktion des FIND-Problems)

Das Problem, zu einer gegebenen Menge `S` von ganzen Zahlen das `i`-t-größte Element `a` zu bestimmen, ist nicht ganz leicht zu formalisieren, wenn man nur die elementaren Mengenoperationen (ohne den allgemeinen Set-Former) zur Verfügung stehen hat. In diesem Falle lautet die naheliegenste Formalisierung, daß `S` in zwei Mengen `S1` und `S2` sowie das Element `a` zerlegbar sein muß, wobei `S1` nur aus Elementen besteht, die kleiner sind als `a`, und `S2` nur aus größeren Elementen, wobei `S2` genau `i-1` Elemente haben muß. Diese Charakterisierung führt zu folgender Spezifikation des FIND-Problems

$$\forall \texttt{S,i}.\forall \texttt{S1,a,S2}.\ \texttt{i}\in\{\texttt{1..|S|}\} \Rightarrow \texttt{FIND((S,i),(S1,a,S2))}$$
$$\Leftrightarrow\ \texttt{S=S1}\cup\{\texttt{a}\}\cup\texttt{S2} \ \wedge\ \forall \texttt{x}\in\texttt{S1.x<a} \ \wedge\ \forall \texttt{x}\in\texttt{S2.a<x} \ \wedge\ \texttt{|S2|=i-1}$$

Dieses Problem enthält drei Ausgabevariablen, von denen zumindest eine redundant ist, da sie vollständig durch `S, a` und die andere Ausgabemenge beschrieben werden kann. Genau dies festzustellen ist die Aufgabe von GET-RNV: da `S1, S2` und $\{\texttt{a}\}$ disjunkt sind, läßt sich `S1` sich darstellen als `S1 = S\(S2+a)`, was zu folgender reduzierten Form des Problems führt.

$$\forall \texttt{S,i}.\forall \texttt{S1,a,S2}.\ \texttt{i}\in\{\texttt{1..|S|}\} \Rightarrow \texttt{FIND((S,i),(S1,a,S2))}$$
$$\Leftrightarrow\ \texttt{FIND'((S,i),(S1,a))} \ \wedge\ \texttt{S1=}\boxed{\texttt{S}\backslash\texttt{(S2+a)}}$$
$$\forall \texttt{S,i}.\forall \texttt{S2,a}.\ \texttt{i}\in\{\texttt{1..|S|}\} \Rightarrow \texttt{FIND'((S,i),(S2,a))}$$
$$\Leftrightarrow\ \texttt{a}\in\texttt{S} \ \wedge\ \texttt{S2}\subseteq\texttt{S-a} \ \wedge\ \forall \texttt{x}\in\boxed{\texttt{S}\backslash\texttt{(S2+a)}}\texttt{.a<x} \ \wedge\ \forall \texttt{x}\in\texttt{S2.x>a} \ \wedge\ \texttt{|S2|=i-1}$$

GET-SOC – Separate Output Conditions

Da die Strategie GET-RNV schnell an die Grenzen ihrer Anwendungsmöglichkeiten stößt, versucht die Strategie GET-SOC als nächstes, die Ausgabebedingung $O[\texttt{x,y,y'}]$ in unabhängige zu zerlegen, in denen jeweils nur eine der beiden Ausgabevariablen vorkommt. Dies bedeutet, daß eine Spezifikationsformel der Form

$$\forall \texttt{x}{:}D.\forall \texttt{(y,y')}{:}R{\times}R'.\ I[x] \ \Rightarrow\ \texttt{F(x,y,y')} \Leftrightarrow O[\texttt{x,y,y'}]$$

transformiert wird in eine Formelmenge der Art

$$\forall \texttt{x}{:}D.\forall \texttt{(y,y')}{:}R{\times}R'.\ I[x] \ \Rightarrow\ \texttt{F(x,y,y')} \Leftrightarrow \texttt{F1(x,y)} \wedge \texttt{F2(x,y')}$$
$$\forall \texttt{x}{:}D.\forall \texttt{y}{:}R.\ I[x] \ \Rightarrow\ \texttt{F1(x,y)} \Leftrightarrow O_1[\texttt{x,y}]$$
$$\forall \texttt{x}{:}D.\forall \texttt{y'}{:}R'.\ I[x] \ \Rightarrow\ \texttt{F2(x,y')} \Leftrightarrow O_2[\texttt{x,y'}]$$

Diese Zerlegung von $O[\texttt{x,y,y'}]$ in $O_1[\texttt{x,y}]$ und $O_2[\texttt{x,y'}]$ muß auf rein syntaktischer Basis durchführbar sein. Die einfachste Methode hierfür ist, jeweils diejenigen Konjunktionsglieder von $O[\texttt{x,y,y'}]$ aufzusammeln, in denen ausschließlich `y` bzw. `y'` als Ausgabevariablen vorkommen. Wenn diese Technik versagt, ist die Strategie GET-SOC nicht anwendbar.

CHVAR – Choose Output-Variable

Bei Problemspezifikationen mit mehreren Ausgabewerten, bei denen weder GET-RNV noch GET-SOC zum Ziel führen, muß versucht werden, eine hierarchische Lösung des Problems zu generieren. Dies bedeutet, daß die Berechnung des zweiten Ausgabewertes ein Teilproblem während der Berechnung des ersten werden wird, wobei dieses Teilproblem erst nach der Anwendung von GUESS-DOMAIN auf die "äußere" Variable angegangen wird, also nachdem für diese Variable ein geratener Wert als zusätzliche Eingabe vorliegt. Die Hauptfrage, die hierfür zu klären ist, welche der Ausgabevariablen als erstes zu behandeln ist.

Das Verfahren CHVAR, mit dem diese Variable bestimmt wird, ist verhältnismäßig aufwendig und benötigt eine Reihe semantischer Zusatzinformationen, die eigentlich nur von einem Benutzer des Synthesesystems gegeben werden können. Bei einer Spezifikationsformel der Form

$$\forall \mathtt{x}{:}D.\forall (\mathtt{y},\mathtt{y'}){:}R\times R'.\ I[x] \ \Rightarrow\ \mathtt{F(x,y,y')} \ \Leftrightarrow\ O[\mathtt{x,y,y'}]$$

Wählt man für jede der beiden Ausgabevariablen `y` und `y'` ein Guess-Schema aus der Wissensbank und führt die Heuristik DOMAIN aus. Anschließend vergleicht man die Anzahl der Werte der Ausgabebereiche R und R', die durch Raten erreicht werden können, also

$$|\,\{\mathtt{y}{:}R \mid \exists \mathtt{g}{:}G.\ DC(\mathtt{x,g}) \wedge t(\mathtt{g,y})\}\,| \quad \text{und} \quad |\,\{\mathtt{y'}{:}R' \mid \exists \mathtt{g'}{:}G'.\ DC'(\mathtt{x,g'}) \wedge t'(\mathtt{g',y'})\}\,|$$

und wählt diejenige Ausgabevariable, bei der diese Anzahl geringer ist. Dies hat zur Folge, daß bei der Bestimmung aller Lösungen des Problems die Anzahl der äußeren Rekursionen des generierten Algorithmus kleiner ist, was – hoffentlich – zu einem effizienteren Algorithmus führt.

DEPEND – Bestimme Abhängigkeiten der Ausgaben

Nachdem CHVAR die äußere Variable für GUESS-DOMAIN festgelegt hat und der äußere Rateschritt ausgeführt worden ist, versucht die Strategie DEPEND die Abhängigkeiten zwischen der äußeren und der inneren Ausgabevariablen zu bestimmen. Dieses Ziel ähnelt dem der Strategie GET-RNV, verlangt aber ein aufwendigeres Verfahren.

Es sei o.B.d.A. `y` die von CHVAR ausgewählte Ausgabevariable für den äußeren Rateschritt. Dann ist nach der Anwendung von GUESS-DOMAIN die folgende Formelmenge generiert worden.

$$\forall \mathtt{x}{:}D.\forall (\mathtt{y},\mathtt{y'}){:}R\times R'.\ I[x] \ \Rightarrow\ \mathtt{F(x,y,y')} \ \Leftrightarrow\ \exists \mathtt{g}{:}G.\ DC(\mathtt{x,g}) \wedge (\mathtt{F_1(x,g,y,y')} \vee \mathtt{F_2(x,g,y,y')})$$
$$\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall (\mathtt{y},\mathtt{y'}){:}R\times R'.\ I[\mathtt{x}] \wedge DC(\mathtt{x,g}) \ \Rightarrow\ \mathtt{F_1(x,g,y,y')} \ \Leftrightarrow\ O[\mathtt{x,y,y'}] \wedge t(\mathtt{g,y})$$
$$\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall (\mathtt{y},\mathtt{y'}){:}R\times R'.\ I[\mathtt{x}] \wedge DC(\mathtt{x,g}) \ \Rightarrow\ \mathtt{F_2(x,g,y,y')} \ \Leftrightarrow\ O[\mathtt{x,y,y'}] \wedge \neg\mathtt{t(g,y)}$$

Die Strategie DEPEND versucht nun die maximale Teilbedingung von O[x,y,y'] zu bestimmen, die *beide* Teilprobleme lösbar macht, wenn `x,g` und `y` als Eingabevariablen vorausgesetzt werden.

Die Heuristik hierfür geht relativ grob vor: es wird die größte Menge O'[x,y,y'] von Konjunktionsgliedern aus O[x,y,y'] gewählt, in denen `y'` vorkommt. Anschließend wird versucht, die beiden reduzierten Probleme

$$\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall \mathtt{y}{:}R.\ \forall \mathtt{y'}{:}R'.\ I[\mathtt{x}] \wedge DC(\mathtt{x,g}) \ \Rightarrow\ \mathtt{F'_1(x,g,y,y')} \ \Leftrightarrow\ O'[\mathtt{x,y,y'}] \wedge t(\mathtt{g,y})$$
$$\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall \mathtt{y}{:}R.\ \forall \mathtt{y'}{:}R'.\ I[\mathtt{x}] \wedge DC(\mathtt{x,g}) \ \Rightarrow\ \mathtt{F'_2(x,g,y,y')} \ \Leftrightarrow\ O'[\mathtt{x,y,y'}] \wedge \neg\mathtt{t(g,y)}$$

mit Eingabevariablen `x,g` und `y` zu synthetisieren, wobei man sich wider auf "leicht zu findende" Lösungen konzentriert. Sind $\mathtt{f_1}$ und $\mathtt{f_2}$ die zugehörigen generierten Lösungsfunktionen, so wird – wie bei GET-RNV – im ursprünglichen Problem `y'` durch $\mathtt{f_i(x,g,y)}$ ersetzt, was zu folgender Formelmenge führt.

$$\forall \mathtt{x}{:}D.\forall (\mathtt{y},\mathtt{y'}){:}R\times R'.\ I[x] \ \Rightarrow\ \mathtt{F(x,y,y')} \ \Leftrightarrow\ \exists \mathtt{g}{:}G.\ DC(\mathtt{x,g}) \wedge (\mathtt{F_1(x,g,y,y')} \vee \mathtt{F_2(x,g,y,y')})$$
$$\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall (\mathtt{y},\mathtt{y'}){:}R\times R'.\ I[\mathtt{x}] \wedge DC(\mathtt{x,g}) \ \Rightarrow\ \mathtt{F_1(x,g,y,y')} \ \Leftrightarrow\ \mathtt{F^*_1(x,g,y)} \ \wedge\ \mathtt{y'=f_1(x,g,y)}$$
$$\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall \mathtt{y}{:}R.\ I[\mathtt{x}] \wedge DC(\mathtt{x,g}) \ \Rightarrow\ \mathtt{F^*_1(x,g,y)} \ \Leftrightarrow\ O[\mathtt{x,y,f_1(x,g,y)}\,/\,\mathtt{x,y,y'}] \wedge t(\mathtt{g,y})$$
$$\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall (\mathtt{y},\mathtt{y'}){:}R\times R'.\ I[\mathtt{x}] \wedge DC(\mathtt{x,g}) \ \Rightarrow\ \mathtt{F_2(x,g,y,y')} \ \Leftrightarrow\ \mathtt{F^*_2(x,g,y)} \ \wedge\ \mathtt{y'=f_2(x,g,y)}$$
$$\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall \mathtt{y}{:}R.\ I[\mathtt{x}] \wedge DC(\mathtt{x,g}) \ \Rightarrow\ \mathtt{F^*_2(x,g,y)} \ \Leftrightarrow\ O[\mathtt{x,y,f_2(x,g,y)}\,/\,\mathtt{x,y,y'}] \wedge \neg t(\mathtt{g,y})$$

5.4.2.4 Das Verfahren als Gesamtstrategie

Mit den soeben vorgestellten Zusatzstrategien lassen sich die Hauptstrategien GUESS-DOMAIN und GET-REC auf viele verschiedene Problemstellungen anpassen, wodurch das LOPS-Verfahren – zumindest aus theoretischer Sicht – zu einem recht mächtigen Syntheseverfahren wird. Wir fassen die bisher einzeln vorgestellten Strategien nun zu einer Gesamtstrategie zusammen.

Strategie 5.4.17 (LOPS–Verfahren)
Gegeben sei eine Problemspezifikation spec=(D, R, I, O).

1. *Definiere das Prädikat* F *durch* $\forall$x:D.$\forall$y:R. $I[x] \Rightarrow$ F(x,y) $\Leftrightarrow O[x,y]$ *und markiere* x *als Eingabe.*
2. *GUESS-Phase:*
   - *Besteht* y *aus mehreren Ausgabevariablen, so versuche eine Problemreduktion mit GET-RNV, GET-SOC oder CHVAR/DEPEND*

   (a) *Wähle ein GUESS-Schema* $(R, G, t,$ *displayable*$)$ *mit Ausgabetyp* R *aus der Wissensbank*

   (b) *Enthält* R *nichtdarstellbare Ausgaben, so spalte diese durch Bestimmung eines Prädikats* prim *ab.*

   (c) *Bestimme die Domain-Condition* DC *mit der Heuristik DOMAIN*

   (d) *Transformiere die Ausgangsformel durch Hinzufügen der geratenen Information und spalte das Problem in Teilprobleme gemäß Definition 5.4.11.*

   Markiere g *als Eingabevariable der neuen Hilfsprädikate und behandle alle Teilprobleme einzeln.*
3. *Teste die Auswertbarkeit und Widersprüchlichkeit einzelner Teilprobleme heuristisch. Der Nachweis muß in wenigen Schritten durchführbar sein.*

   Dabei führt einfaches Guessing immer zu trivial lösbaren Teilproblemen im Erfolgsfall und induktive Datenstrukturen oft zu widersprüchlichen Teilprobleme im Mißerfolgsfall.
4. *GET-REC-Phase (siehe Strategie 5.4.14 auf Seite 242)*

   (a) *Wähle ein Rekursionsschema* $(D, G, \searrow)$ *aus der Wissensbank.*

   (b) *Durch gezielte Suche nach Lemmata über die Beziehung zwischen einer reduzierten und der nichtreduzierten Form der vorkommenden Prädikate schreibe die Formel um in eine rekursive Variante der Ausgangsformel. Dabei kann eine Abspaltung nichtrekursiver Lösungen erforderlich sein.*
5. *Vereinfache die entstandenen Formeln soweit wie möglich und überprüfe die Auswertbarkeit. Starte LOPS erneut, wenn eine Formel nicht auswertbar ist.*
6. *Konstruiere aus der rekursiven Formel ein Programm*

Aus Sicht der Künstlichen Intelligenz ist das LOPS-Verfahren ein sehr vielseitiges und mächtiges Konzept, da sehr verschiedenartige Problemstellungen mit nur wenigen Grundstrategien auf recht elegante Weise gelöst werden können. Diese Eleganz gilt bisher jedoch nur auf dem Papier, da es in der Tat sehr viel Intelligenz benötigt, um eine LOPS-Synthese erfolgreich zum Ziel zu führen, und noch nicht geklärt ist, wie diese Schritte schematisiert und in eine automatisch auszuführende Methode umgewandelt werden können. Auf einen ungeübten Anwender wirkt das LOPS-Verfahren daher oft recht ziellos, da ihm keine festen Regeln zur Verfügung stehen, nach denen er die nötigen Umformungen auswählen kann. Die Angabe solcher zuverlässiger Regeln und ihre Einbettung in ein formales Konzept ist daher noch ein offenes Forschungsproblem.

5.4.3 Integration in das allgemeine Fundament

Wir wollen das nur andeuten

Satz 5.4.18 (Integration von LOPS – simple Variante)

Es sei ...

Ist (D,R,↘) *ein wohlfundiertes Rekursionsschema und gilt*

$\forall$x:D.$\forall$g:R. I(x) $\Leftrightarrow$ I_{base}(x,g) $\vee$ I(x↘g)
$\forall$x:D.$\forall$g,y,y_r:R. DC(x,g) $\wedge\neg$O(x,g) $\wedge$ g$\neq$y $\Rightarrow$
O(x,y) $\Leftrightarrow$ O(x↘g,y_r) $\wedge$ O_{rec}(x,g,y_r,y)

und sind die folgenden Programme korrekt

```
FUNCTION f_DC(x:D):R  WHERE I(x) RETURNS g  SUCH THAT DC(x,g)              = f_dc(x)
FUNCTION f_BASE(x,g:D×R):R  WHERE I_base(x) ∧DC(x,g) ∧¬O(x,g)
  RETURNS y  SUCH THAT O(x,y)  ∧  g≠y                                      = f_base(x,g)
FUNCTION f_REC(x,g,y_r:D×R×R):R  WHERE I(x↘g) ∧DC(x,g) ∧¬O(x,g) ∧O(x↘g,y_r)
  RETURNS y  SUCH THAT g≠y  ∧  O_rec(x,g,y_r,y)                            = f_rec(x,g,y_r)
```

dann ist das folgende rekursive Programm korrekt

```
FUNCTION f(x:D):R  WHERE I(x) RETURNS y  SUCH THAT O(x,y)
=  let g=f_dc(x) in  if O(x,g) then g
             else if I_base(x,g) then f_base(x,g)
                  else f_rec(x,g,f(x↘g))
```

Erweiterung: allgemeines GUESS, komplexere Rekursion, mengenwertige Funktionen
↦ Einbettung von LOPS durch Theoreme
↦ Strategien = Bestimmung der notwendigen Parameter

Versuch der Umsetzung in ein formales System

1. vie EQ trafo des Theorems – vorstellen

Ebenenwechsel und algo-theorie – am simplen LOPS vorführen (hierzu mehr in 5.5)

bergang: das bestreben um die Verbesserung von LOPS führt in die Gleiche Richtung wie Algo-Theorien. Nur so terminiert es.

5.4.4 Synthese durch Tranformationen im Vergleich

Auch wenn die Verwendung von Transformationsregeln und Rewrite-Techniken anstelle von starren Kalkül-Regeln vom Prinzip her wesentlich flexiblere und effizientere Systeme verspricht, wurde in praktischer Hinsicht mit Verfahren zur Synthese durch Transformationen bisher nicht mehr erreicht als mit dem Prinzip "Beweise als Programme". Dies liegt im wesentlichen daran, daß sich viele Transformationen sich auch als Beweistaktiken (sogenannte *Konversionen*, siehe [Jackson, 1993c, Kapitel 8.8]) in beweisbasierte Systeme integrieren lassen. Zum anderen

5.5 Programmentwicklung durch Algorithmenschemata

Praktische Erfahrungen haben gezeigt, daß der Umgang mit Synthesesystemen, die auf der Basis des Prinzips "Beweise als Programme" operieren oder Programme durch Anwendung von äquivalenzerhaltenden Transformationen erzeugen, sehr mühsam ist und daß man über die Synthese einfachster Beispiele[18] nicht hinauskommt, ohne von einem Benutzer massive Eingriffe in den Syntheseprozeß zu verlangen. Der Grund hierfür liegt im wesentlichen darin, daß in beiden Ansätzen *allgemeine Verfahren* die Anwendung elementarer logischer Inferenzen steuern und zur Unterstützung bestenfalls Lemmata über verschiedene Anwendungsbereiche heranziehen. Eine Verarbeitung von wirklichem Programmierwissen, also von Erkenntnissen über Programmstrukturen und ihre Eigenschaften, findet jedoch nicht statt. Aus diesem Grunde ist eine Steuerung derartiger Systeme durch einen "normalen" Benutzer" kaum möglich, da dies tiefe Einsichten in die Strategie und Vertrautheit mit der Denkweise der zugrundeliegenden Kalküle verlangt. Da Programmierer im allgemeinen aber keine Logiker sind, kann von einer echten Unterstützung bei der Entwicklung korrekter Programme noch nicht gesprochen werden.

Ein Konzept, das sich deutlich stärker nach den Bedürfnissen der Praxis richtet, ist der Versuch, Programme durch Verwendung von *Algorithmenschemata* zu synthetisieren. Es basiert auf der Erkenntnis, daß in Lehrbüchern über die Methodik des Programmierens [Knuth, 1968, Knuth, 1972, Knuth, 1975, Gries, 1981] sehr stark auf die Struktur von Algorithmen eingegangen wird, die für eine Lösung bestimmter Probleme besonders geeignet ist, und ebenso auf Methoden, derartige Algorithmen zu entwickeln. Programmiertechniken wie *Generate-and-Test, Divide-and-Conquer, Problemreduktion, Binärsuche, Backtracking, Dynamische Programmierung, Siebe, etc.* sind den meisten Programmierern wohl vertraut und es gibt nur sehr wenige algorithmische Probleme, die nicht mit einer dieser Techniken elegant gelöst werden können. Syntheseverfahren, die für gewöhnliche Programmierer verständlich und kontrollierbar sein sollen, sollten sich daher an diesen Erkenntnissen der Programmiermethodik orientieren und Wissen über die Struktur von Algorithmen verarbeiten. Ein Benutzer sollte die Algorithmenstruktur seiner Wahl festlegen können und das Synthesesystem sollte die Strategien bereitstellen, die Komponenten eines derartigen Algorithmus zu bestimmen und die Bedingungen für seine Korrektheit zu überpüfen. Durch die engeren Vorgaben ist dieses Vorgehen erheblich zielgerichteter als Verfahren, die auf allgemeinen Techniken (wie Induktionsbeweiser oder die LOPS Strategie) aufgebaut sind, und führt daher auch schneller zum Ziel.

Obwohl dieser Gedanke an sich schon relativ alt ist, hat es lange gedauert, ihn in einem erfolgreichen und leistungsfähigen System umzusetzen. Das Hauptproblem hierbei war vor allem, ein tragfähiges theoretisches Fundament zu schaffen, welches die Korrektheit schematisch erzeugter Algorithmen sicherstellt und es ermöglichte, einen Großteil der Beweislast ein für alle Mal im Voraus abzuhandeln und somit aus dem eigentlichen Syntheseprozeß herauszunehmen. Zudem war es nötig, die Struktur verschiedener Klassen von Algorithmen zu analysieren, die Komponenten dieser Algorithmen innerhalb einer einheitliche Beschreibung zu identifizieren, Axiome für die Korrektheit der Algorithmen aufzustellen und die Korrektheit des schematischen Algorithmus nachzuweisen. Schließlich mußten Strategien zur systematischen Erzeugung der fehlenden Komponenten entwickelt werden, die sich – der praktischen Durchführbarkeit wegen – meistens auf vorgefertigte Teilinformationen stützen, ansonsten aber in einem formalen logischen Kalkül eingebettet sind und somit als Beweis- oder Transformationsregel eines sehr hohen Abstraktionsniveaus angesehen werden können.

In den letzten Jahren sind die wichtigsten algorithmischen Strukturen wie *Divide-and-Conquer* [Smith, 1983a, Smith, 1985a, Smith, 1987a], *Lokalsuche* [Lowry, 1988, Lowry, 1987, Lowry, 1989, Lowry, 1991], *Globalsuchalgorithmen* [Smith, 1987b, Smith, 1991a], *Dynamische Programmierung* [Smith, 1991b] und *Problemreduktionsge-*

[18]Nicht selten hört man das Argument, daß Synthesesysteme aus diesem Grunde keinerlei praktische Relevanz besitzen. Man könne ebensogut die wenigen Beispiele direkt in einer Wissensbak abspeichern und dann nach bedarf herausholen.

Im Prinzip ist die Programmentwicklung durch Algorithmenschemata eine konsequente Fortsetzung dieses Gedankens: man speichert tatsächlich vorgefertigte Lösungen in einer Wissensbank ab. Allerdings läßt die Vielzahl der möglichen Algorithmen es nicht zu, diese komplett abzuspeichern. Statt dessen muß man von den Besonderheiten konkreter Algorithmen abstrahieren und Schemata abspeichern, bei denen einige Parameter noch einzusetzen sind. Die Strategie läuft dann darauf hinaus, diese Parameter im konkreten Fall zu bestimmen.

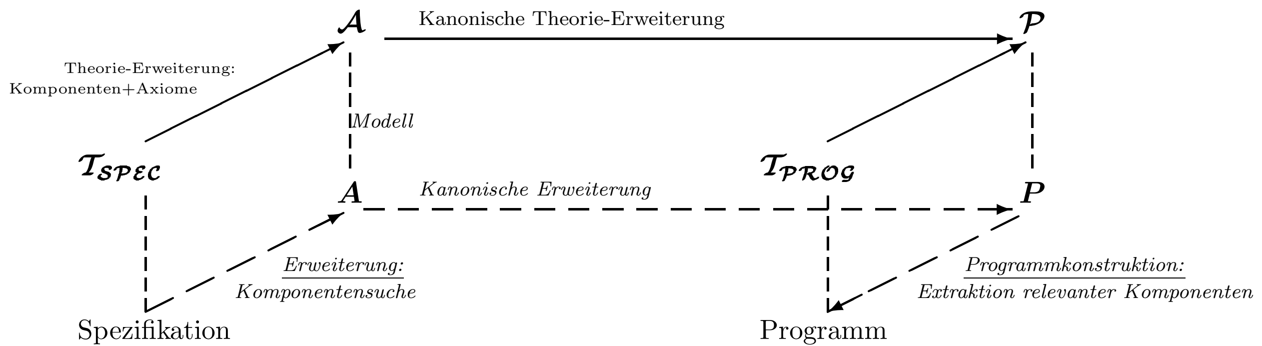


Abbildung 5.2: Synthese durch Erweiterung algebraischer Theorien

neratoren [Smith, 1991c] unabhängig voneinander untersucht worden. Sie haben zu einer Reihe erfolgreicher Algorithmenentwurfsstrategien geführt, die alle in das KIDS System [Smith, 1990, Smith, 1991a] integriert wurden. Die praktischen Erfahrungen mit diesem System zeigen, daß der Einsatz von Algorithmenschemata keineswegs eine Einengung der Möglichkeiten eines Synthesesystems bedeutet sondern vielmehr zu flexiblen und praktisch erfolgreichen Verfahren führen kann.

Wir wollen im folgenden zunächst die theoretischen Grundlagen erklären, die eine formal korrekte Synthese mit Algorithmenschemata ermöglichen. Anschließend werden wir dieses Paradigma am Beispiel von drei Synthesestrategien erläutern und dabei zeigen, wie sich diese Vorgehensweise in ein allgemeines Inferenzsystem wie NuPRL integrieren läßt.

5.5.1 Algorithmenschemata als Algebraische Theorien

Eine formal präzise Grundlage für die Synthese von Algorithmen mit Hilfe von Algorithmenschemata läßt sich am leichtesten im Kontext algebraischer Theorien beschreiben, die für die Beschreibung formaler Spezifikationen [Guttag & Horning, 1978, Sannella & Tarlecki, 1987, Sannella & Tarlecki, 1988] ein gebräuchliches Hilfsmittel sind. Die prinzipielle Vorgehensweise ist schematisch in Abbildung 5.2 dargestellt.

- In der algebraischen Sichtweise werden Spezifikationen als Modelle einer allgemeinen *Problemtheorie* $\mathcal{T}_{\mathcal{SPEC}}$ betrachtet und das Ziel einer Synthese ist, dieses Objekt durch Hinzunahme weiterer Datentypen und Operationen zu einem Modell A einer abstrakten Algorithmentheorie $\mathcal{A}$ zu erweitern.

- Die abstrakte Algorithmentheorie $\mathcal{A}$ wird unabhängig von dem eigentlichen Syntheseprozeß entworfen. Sie beschreibt Namen von Datentypen und Operationen sowie Axiome, die es ermöglichen, innerhalb dieser Theorie Theoreme über die Korrektheit abstrakter Algorithmen zu beweisen, die sich vollständig mit den Bestandteilen der Theorie $\mathcal{A}$ beschreiben lassen.

  Formal bedeutet dies, daß sich die Theorie $\mathcal{A}$ auf kanonische Weise zu einer *Programmtheorie* $\mathcal{P}$ fortsetzen läßt, die neben den Komponenten der Problemtheorie $\mathcal{T}_{\mathcal{SPEC}}$ auch noch einen Programmkörper, seine Axiome und eine Beschreibung durch Komponenten der Theorie $\mathcal{A}$ beinhaltet und eine Erweiterung der Standardprogrammtheorie $\mathcal{T}_{\mathcal{PROG}}$ (mit den Mindestanforderungen an Programme) ist.

- Die Erzeugung eines konkreten Algorithmus benötigt somit nur noch die Instantiierung der abstrakten Namen der Theorie $\mathcal{A}$ durch die konkreten Objekte des Modells A, was gleichbedeutend ist mit der kanonischen Erzeugung eines Modells P von $\mathcal{P}$, aus dem dann die relevante Komponente – nämlich das Programm – extrahiert wird.

Beispiel 5.5.1 (Divide-and-Conquer Algorithmen)

Divide-and-Conquer Algorithmen, die wir in Abschnitt 5.5.4 ausführlicher betrachten werden, lösen ein Problem durch Reduktionen mit Hilfe von Dekompositionsoperationen. Formal lassen sie sich als Programme mit der folgenden Grundstruktur beschreiben:

```
FUNCTION F(x:D):R   WHERE I(x) RETURNS y  SUCH THAT O(x,y)
= if primitive(x)  then Directly-solve(x)  else (Compose ∘ G×F ∘ Decompose)(x)
```

hierbei sind `Decompose, G, Compose, Directly-solve` und `primitive` Parameter, die im Verlaufe einer Synthese noch zu bestimmen sind.

Die Theorie $\mathcal{A}_{D\&C}$ der Divide-and-Conquer Algorithmen enthält neben den Namen `D, R` für Ein- und Ausgabebereich und den Namen `I, O` für Ein- und Ausgabebedingungen also noch die Namen `Decompose, G, Compose, Directly-solve` und `primitive` als zusätzliche Operationen. Hinzu kommen zwei zusätzliche Namen für Datentypen, die für eine korrekte Typisierung erforderlich sind und natürlich die Axiome, welche die Korrektheit des obigen Algorithmus garantieren.

Synthese bedeutet somit, aufgrund der konkreten Werte für `D, R, I` und `O` konkrete Operationen zu finden, welche anstelle von `Decompose, G, Compose, Directly-solve` und `primitive` eingesetzt werden können und dann das obige Schema entsprechend zu belegen.

Das allgemeine Konzept operiert also auf zwei Ebenen. Auf der Ebene der *algorithmischen Theorien* werden die grundsätzlichen Zusammenhänge formal beschrieben[19] und überprüft. Auf der Ebene der konkreten Probleme wird dann die eigentliche Synthese ausgeführt, wobei der schwierigste Schritt – die Erzeugung des Algorithmus – nun ganz schematisch und *ohne weitere Inferenzen* geschehen kann. Spezialisierte Synthesestrategien können sich daher auf die Erzeugung von Modellen der entsprechenden algorithmischen Theorie konzentrieren. Um dies zu präzisieren, wollen wir einige Definitionen aus der Theorie der algebraischen Spezifikationen aufgreifen.

Definition 5.5.2 (Formale Theorien und Modelle)

1. *Eine Signatur Σ ist ein Paar (S,Ω), wobei S eine Familie von Namen für Sorten (Datentypen) und Ω eine Familie von typisierten Operationsnamen ist.*

2. *Eine formale Theorie $\mathcal{T}$ besteht aus einer Signatur Σ, welche das Vokabular der Theorie beschreibt und einer Menge Ax von Axiomen für die genannten Sorten und Operationen.*

3. *Eine Theorie $\mathcal{T}_1$ erweitert eine Theorie $\mathcal{T}_2$, wenn alle Sortennamen, Operationsnamen und Axiome von $\mathcal{T}_2$ – bis auf konsistente Umbenennungen – auch in $\mathcal{T}_1$ existieren.*

4. *Eine Struktur T für $\mathcal{T}$ besteht aus einer Familie nichtleerer Datentypen und einer Familie von Operationen, welche die in $\mathcal{T}$ genannten Typisierungen respektieren.*

5. *Eine Struktur T_1 für $\mathcal{T}_1$ erweitert eine Struktur T_2 für $\mathcal{T}_1$, wenn alle Datentypen und Operationen von T_2 auch in T_1 existieren und dort die gleichen (in $\mathcal{T}_2$ genannten) Typbedingungen erfüllen.*

6. *Ein Modell T $\mathcal{T}$ ist eine Struktur für $\mathcal{T}$, deren Datentypen und Operationen alle Axiome aus $\mathcal{T}$ erfüllen.*

Formale (algebraische) Theorien sind somit nichts anderes als formale Theorien über Objekte, von denen wir nicht mehr kennen als ihren Namen, ihre Typisierung und ihre Axiome. Strukturen sind Belegungen dieser Namen durch konkrete typkorrekte Objekte und Modelle sind Belegungen der Namen durch konkrete

[19]Im Prinzip ist die abstrakte algorithmischen Theorie nichts anderes als eine geringfügig formalere Beschreibung der zentralen Komponenten eines Algorithmus einer bestimmten Klasse. Wir haben immer dann, wenn wir allgemein über Spezifikationen, Programme und Strategien gesprochen haben, abstrakte Namen `D, R` für Ein- und Ausgabebereich und abstrakte Namen `I, O` für Ein- und Ausgabebedingungen verwendet und implizit vorausgesetzt, daß es sich hierbei um *Platzhalter* für die tatsächlichen Bereiche und Bedingungen handelt. Agorithmischen Theorien sind somit nichts anderes als formale Theorien über derartige Platzhalter, von denen wir nicht mehr kennen als ihren Namen und ihre Axiome.

Objekte, die neben der Typkorrektheit auch noch die Axiome erfüllen. Wir wollen diese Begriffe anhand zweier formaler Theorien illustrieren, welche das Konzept der Spezifikationen und das der Programme im Kontext algebraischer Theorien repräsentieren.

Definition 5.5.3 (Problem- und Programmtheorien)

1. $\underline{\mathcal{T}_{\mathcal{SPEC}}}$ = ({D,R}, {I:D→$\mathbb{B}$, O:D×R→$\mathbb{B}$}, ∅) *ist die algebraische Theorie der Spezifikationen.*
2. $\underline{\mathcal{T}_{\mathcal{PROG}}}$ = ({D,R}, {I: D→$\mathbb{B}$, O: D×R→$\mathbb{B}$, body: D$\not\to$R}, {∀x:D. I(x) ⇒ O(x,body(x)) }) *ist die algebraische Theorie der Programme*
3. *Eine Theorie $\mathcal{P}$ heißt Problemtheorie, wenn sie $\mathcal{T}_{\mathcal{SPEC}}$ erweitert.*
4. *Eine Theorie $\mathcal{P}$ heißt Programmtheorie, wenn sie $\mathcal{T}_{\mathcal{PROG}}$ erweitert.*

Eine Problemtheorie[20] ist also eine beliebige algebraische Theorie, deren Modelle unter anderem eine Programmspezifikation enthalten, und eine Programmtheorie ist eine algebraische Theorie, deren Modelle unter anderem ein korrektes Programm enthalten. Ein Modell einer Programmtheorie, welches ein Modell P einer Problemtheorie erweitert, enthält also eine Lösung der in P genannten Spezifikation.

Beispiel 5.5.4

Strukturen und Modelle für Problem- und Programmtheorien sind nichts anderes als eine etwas umsortierte Schreibweise für Spezifikationen und Programme im Sinne von Definition 5.2.1. Die Repräsentation einer Spezifikation des Sortierproblems

```
FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) WHERE true
  RETURNS S  SUCH THAT rearranges(L,S) ∧ ordered(S)
```

wird somit dargestellt[21] als folgendes Modell von $\mathcal{T}_{\mathcal{SPEC}}$

{Seq(ℤ), Seq(ℤ)}, {λL. true, λL,S. rearranges(L,S) ∧ ordered(S)}

Die Repräsentation eines Selectionsort-Programms stellt eine Erweiterung dieses Modells durch Hinzunahme einer weiteren Komponente dar:

```
{Seq(ℤ), Seq(ℤ)}, {λL. true, λL,S. rearranges(L,S) ∧ ordered(S),
                    letrec sort(L) = if L=[] then [] else let g=min(L) in g.sort(L-g)}
```

Eine Erweiterung wäre auch

{Seq(ℤ), Seq(ℤ)}, {λL. true, λL,S. rearranges(L,S) ∧ ordered(S), letrec sort(L) = []}

Diese Erweiterung wäre zwar immer noch eine Struktur für $\mathcal{T}_{\mathcal{PROG}}$, da Typkorrektheit nach wie vor erfüllt ist. Ein Modell aber ist sie nicht, da das Axiom von $\mathcal{T}_{\mathcal{PROG}}$ verletzt ist.

Die in Beispiel 5.5.4 charakterisierte Umwandlung von Spezifikationen und Programmen in Modelle algebraischer Theorien wird durch das folgende Lemma gerechtfertigt.

Lemma 5.5.5

Es sei spec=(D,R,I,O) *eine beliebige Spezifikation und* $body{:}D\not\to R$. *Dann gilt*

1. *Das Programm* FUNCTION F(x:D):R WHERE $I(x)$ RETURNS y SUCH THAT $O(x,y)$ = $body(x)$ *ist genau dann korrekt, wenn* $\underline{T_{spec,\ body}} = (\{D, R\}, \{I, O, body\})$ *ein Modell von $\mathcal{T}_{PROG}$ ist.*
2. *Die Spezifikation* FUNCTION F(x:D):R WHERE $I(x)$ RETURNS y SUCH THAT $O(x,y)$ *ist genau dann erfüllbar, wenn es ein Modell P von $\mathcal{T}_{PROG}$ gibt, welches* $\underline{T_{spec}} = (\{D, R\}, I, O\})$ *erweitert.*

[20]Man beachte, daß die Formulierung der Theorie $\mathcal{T}_{\mathcal{SPEC}}$ eine große Verwandtschaft zu der Formalisierung des Konzeptes "Spezifikation" (siehe Übung 7) durch Konstrukte der Typentheorie besitzt. Diese Verwandtschaft zwischen algebraischen Theorien und den noch formaleren typentheoretischen Ausdrücken (die keine informalen Beschreibungen zulassen) weist einen Weg für eine Integration algorithmischer Theorien und der zugehörigen Synthesestrategien in einen allgemeines formales Inferenzsystem.

[21]Wir verwenden die Mengenschreibweise, um anzudeuten, daß es auf die Reihenfolge der Objekte einer Familie eigentlich nicht ankommt. Allerdings muß der Bezug zwischen einer Komponente und seinem Namen in der abstrakten Theorie erkennbar bleiben, so daß man bei einer Implementierung besser auf Listen zurückgreift.

Aufgrund von Lemma 5.5.5 können wir Programmsynthese tatsächlich als Erweiterung algebraischer Strukturen beschreiben. Das Konzept der Algorithmentheorien als Hilfsmittel für eine systematische Erweiterung von Spezifikationen läßt sich mit den soeben eingeführten Begriffen nun relativ leicht beschreiben: eine Algorithmentheorie ist eine Problemtheorie, die eine kanonische Erweiterung zu einer Programmtheorie besitzt.

Definition 5.5.6 (Algorithmentheorien)

Eine Problemtheorie $\mathcal{A}$ heißt Algorithmentheorie, wenn es ein Programmschema `BODY` *gibt mit der Eigenschaft*

- `BODY` *weist jeder Struktur A für $\mathcal{A}$ eine Funktion aus* $D_A \not\to R_A$ *zu,*
- *Für jedes Modell A von $\mathcal{A}$ ist das folgende Programm korrekt.*[22]
  FUNCTION F(x:D_A):R_A WHERE I_A(x) RETURNS y SUCH THAT O_A(x,y) = BODY(A)(x)

  Dabei sei $\underline{D_A}$ *(bzw.* $\underline{R_A}$, $\underline{I_A}$, $\underline{O_A}$*) diejenige Komponente der Struktur A, welche in der Theorie $\mathcal{A}$ dem Namen* D *(bzw.* R, I, O*) entspricht.*

Algorithmenschemata sind also nichts anderes als Funktionen, die in einem schematischen Algorithmus jeden Namen durch ein konkretes Objekt eines Modells A ersetzen und so eine Standardlösung der Spezifikation $spec_A$ = (D_A, R_A, I_A, O_A) generieren. Eine Algorithmentheorie ist nichts anderes als eine Beschreibung von Komponenten und Axiomen, welche die Existenz eines solchen Algorithmenschemas garantieren.

Beispiel 5.5.7 (Formale Theorie der Divide-and-Conquer Algorithmen)

Die formale Theorie der Divide-and-Conquer Algorithmen (vergleiche Beispiel 5.5.1) enthält neben den Bestandteilen des eigentlichen Algorithmus noch ene Reihe zusätzlicher Komponenten, die für eine Spezifikation der Bestandteile des Algorithmus und für einen Nachweis der Terminierung benötigt werden. Alle Komponenten und Axiome ergeben zusammengefaßt die formale Algorithmentheorie $\mathcal{T}_{D\&C} = (S_{D\&C}, \Omega_{D\&C}, Ax_{D\&C})$ mit den folgenden Bestandteilen:

$S_{D\&C}$ ≡ {D, R, D', R'}

$\Omega_{D\&C}$ ≡ {I: D→𝔹, O: D×R → 𝔹, O_D: D×D'×D → 𝔹, I': D'→𝔹, O': D'×R' → 𝔹, O_C: R'×R×R → 𝔹,
≻: D×D → 𝔹, Decompose: D → D'×D, G: D'→R', Compose: R'×R → R,
Directly-solve:D→R, primitive:D→𝔹
}

$Ax_{D\&C}$ ≡ {FUNCTION F_p(x:D):R WHERE I(x) ∧ primitive(x) RETURNS y SUCH THAT O(x,y)
= Directly-solve(x) korrekt,
O_D(x,y',y) ∧ O(y,z) ∧ O'(y',z') ∧ O_C(z,z',t) ⇒ O(x,t),
FUNCTION F_d(x:D):D'×D WHERE I(x) ∧ ¬primitive(x)
RETURNS y',y SUCH THAT I'(y') ∧ I(y) ∧ x≻y ∧ O_D(x,y',y) = Decompose(x) korrekt,
FUNCTION F_c(z,z':R×R'):R RETURNS t SUCH THAT O_C(z,z',t) = Compose(z,z') korrekt,
FUNCTION F_G(y':D'):R' WHERE I'(y') RETURNS z' SUCH THAT O'(y',z') = G(y') korrekt,
≻ ist wohlfundierte Ordnung auf D
}

Das abstrakte Programmschema für diese Algorithmentheorie ist eine Abbildung von einer Struktur A = ({D, R, D', R'}, {I, O, O_D, I', O', O_C, ≻, Decompose, G, Compose, Directly-solve, primitive}) in den Algorithmus

```
FUNCTION F(x:D):R   WHERE I(x) RETURNS y  SUCH THAT O(x,y)
= if primitive(x)  then Directly-solve(x)  else (Compose ∘ G×F ∘ Decompose)(x)
```

BODY(A) ist korrekt, wenn A alle obengenannten Axiome erfüllt, also ein Modell von $\mathcal{T}_{D\&C}$ ist.

[22] Gemäß Lemma 5.5.5 könnten wir als algebraisch formulierte Bedingung auch schreiben: "({D_A, R_A}, {I_A, O_A, BODY(A)}) ist ein Modell für $\mathcal{T}_{PROG}$". Eine schematische Beschreibung der Funktion BODY durch Komponenten von $\mathcal{A}$ muß sich daher ausschließlich mit den Axiomen aus $\mathcal{A}$ als korrekt im Sinne des Axiomes für $\mathcal{T}_{PROG}$ nachweisen lassen: die Korrektheit ist also ein Theorem der Theorie $\mathcal{A}$.

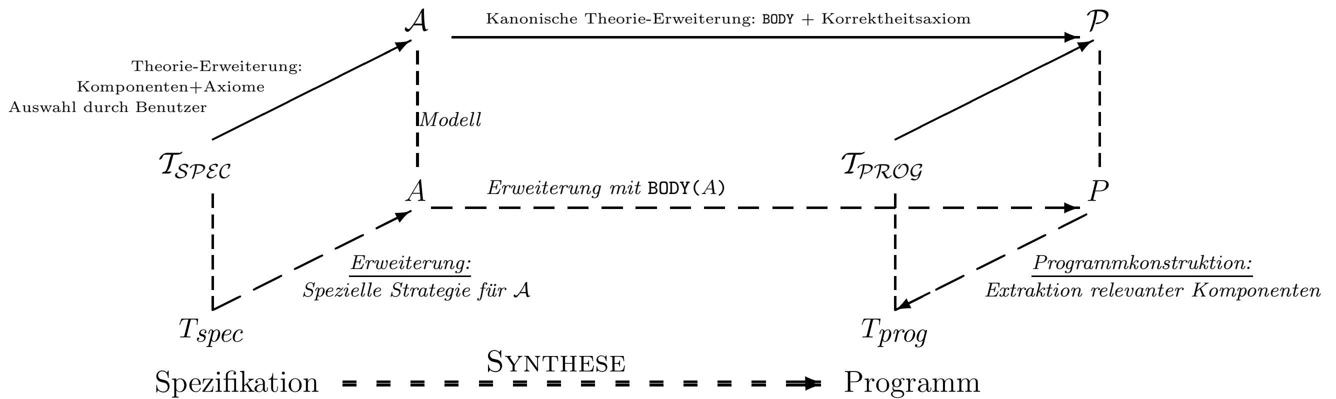


Abbildung 5.3: Synthese mit Algorithmenschemata: Generelle Methode

Mit den bisher gegebenen Definitionen ist es nicht sehr schwierig eine allgemeine Strategie zur Synthese von Programmen mit Hilfe von Algorithmenschemata zu berschreiben und formal zu rechtfertigen. Ihre Grundlage ist der folgende Satz.

Satz 5.5.8

Eine Spezifikation spec=(D,R,I,O*) ist genau dann erfüllbar, wenn es eine Algorithmentheorie* $\mathcal{A}$ *und ein Modell* A *für* $\mathcal{A}$ *gibt, welches die Struktur* T_{spec} *erweitert.*

Beweis: Es sei $\mathcal{A}$ eine Algorithmentheorie und A ein Modell für $\mathcal{A}$, welches die Struktur $T_{spec} = (\{D, R\}, \{I, O\})$ erweitert. Gemäß Definition 5.5.2 ist dann *spec*=$\texttt{spec}_A$, also $\texttt{D}_A=D$, $\texttt{R}_A=R$, $\texttt{I}_A=I$ und $\texttt{O}_A=O$ und für das zu $\mathcal{A}$ gehörige abstrakte Programmschema BODY ist dann das Programm

```
FUNCTION F(x:D):R  WHERE I(x) RETURNS y  SUCH THAT  O(x,y) = BODY(A)(x)
```

korrekt und somit ist die Spezifikation *spec* erfüllbar. □

Satz 5.5.8 rechtfertigt die folgende generelle Methode zur Synthese von Programmen, die in Abbildung 5.3 (eine Präzisierung von Abbildung 5.2) schematisch dargestellt ist.

Strategie 5.5.9 (Synthese mit Algorithmenschemata: Generelle Methode)

Gegeben sei eine Problemspezifikation FUNCTION F(x:D):R WHERE I(x) RETURNS y SUCH THAT O(x,y)

1. *Wähle eine Algorithmentheorie* $\mathcal{A}$ *aus der Wissensbank*
2. *Erweitere* $T_{spec} = (\{D, R\}, \{I, O\})$ *durch Hinzufügen von Komponenten und Überprüfung von Axiomen zu einem Modell* A *von* $\mathcal{A}$
3. *Bestimme* BODY(A) *und liefere als Antwort das Programm*

   ```
   FUNCTION F(x:D):R  WHERE I(x) RETURNS y  SUCH THAT  O(x,y) = BODY(A)(x)
   ```

   Dieses Programm ist garantiert korrekt.

Der erste Schritt dieser Strategie ist im wesentlichen eine Entwurfsentscheidung, die einem Benutzer des Systems überlassen werden sollte. Der zweite Schritt dagegen sollte dagegen durch eine Strategie weitestgehend automatisch durchgeführt werden.

Natürlich ist die soeben beschriebene Methode noch sehr allgemein, da sie ja noch nicht sagt, wie denn nun die zusätzlichen Komponenten des Modells A von $\mathcal{A}$ bestimmt werden sollen. Grundsätzlich sollten die entsprechenden Teilstrategien auch sehr stark auf die gewählten Algorithmentheorien zugeschnitten werden, um effizient genug zu sein. Dennoch hat es sich als sinnvoll herausgestellt, nicht nur bei der Auswahl der Algorithmentheorien als solche auf vorgefertigtes Wissen zurückzugreifen sondern dies auch bei der Bestimmung eines konkreten Modells zu tun, also zu jeder Algorithmentheorie eine Reihe von möglichst allgemeinen Modellen in einer Wissensbank abzulegen und diese dann auf ein gegebenes Problem zu *spezialisieren*. Auf

diese Art kann man sich eventuell die Überprüfung komplizierter Axiome – wie etwa der Wohlfundiertheit von $\succ$ bei Divide-and-Conquer Algorithmen – ersparen.

Es gibt eine Reihe von allgemeinen Techniken, bereits bekannte Modelle auf ein neues Problem anzuwenden, durch die Strategie 5.5.9 ein wenig verfeinert werden kann.

- Ein allgemeines Modell kann durch Verwendung von *Typvariablen* in ein generisches Modell für eine Klasse von Algorithmen verwandelt werden.
- Durch *Reduktion* kann ein Problem auf ein spezielleres abgebildet werden, für das bereits eine Lösung vorliegt ("Operator Match").
- Bei mengenwertigen Problemstellungen (Suche nach *allen* Lösungen einer Ausgabebedingung) kann ein allgemeines Modell auf eine neue Problemstellung *spezialisiert* werden.

Da die Vorgehensweise im ersten Fall naheliegend ist, beschränken wir uns im folgenden auf Reduktion und Spezialisierung.

Definition 5.5.10 (Reduktion und Spezialisierung)

Es seien spec=(D,R,I,O) und spec'=(D',R',I',O') zwei beliebige Spezifikationen

1. *spec ist reduzierbar auf spec' (im Zeichen spec⊴spec'), wenn gilt*

   $\forall$x:D. I(x) $\Rightarrow$ $\exists$x':D'. I'(x') $\wedge$ $\forall$y':R'. O'(x',y') $\Rightarrow$ $\exists$y:R.O(x,y)

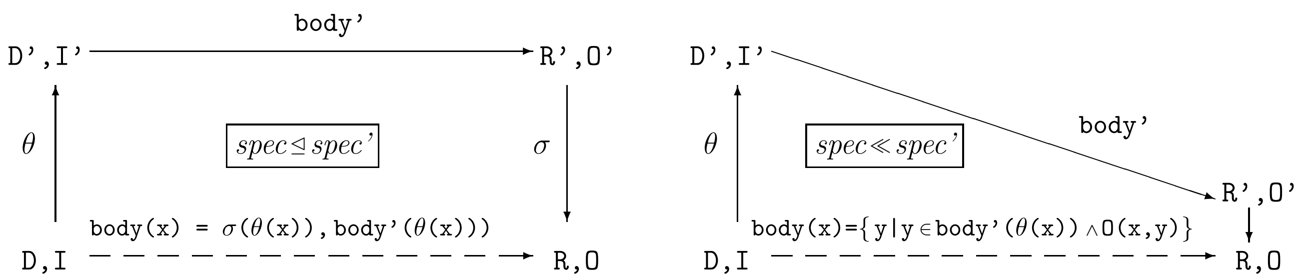
2. *spec spezialisiert spec' (im Zeichen spec≪spec'), wenn R' eine Obermenge von R ist und gilt*

   $\forall$x:D. I(x) $\Rightarrow$ $\exists$x':D'. I'(x') $\wedge$ $\forall$y:R. O(x,y) $\Rightarrow$ O'(x',y)

   Man sagt auch, daß spec' die Spezifikation spec generalisiert

Es ist also *spec* reduzierbar auf *spec'*, wenn die Eingabebedingung von *spec* (nach Transformation) stärker ist als die von *spec'* und die Ausgabebedingung schwächer ist. *spec* spezialisiert *spec'*, wenn sowohl die Ein- als auch die Ausgabebedingung stärker ist, wobei keine Transformation der Ausgaben stattfindet.

Beide Definitionen enthalten eine typische Anwendung des Prinzip "Beweise als Programme" in seiner elementarsten Form. Ein formaler Nachweis der Reduzierbarkeit induziert zwei Transformationsfunktionen θ:D→D' und σ:D'×R'→R, die zur Transformation eines bekannten Algorithmus gemäß dem untenstehen Diagramm verwendet werden können. Ein Nachweis der Spezialisierung induziert eine Transformation θ:D→D', die zusammen mit der Ausgabebedingung O einen bekannten mengenwertigen Algorithmus in einen Lösungsalgorithmus für die Ausgangsspezifikation verwandeln kann.



Die Kombination dieser Erkenntnisse mit dem in Satz 5.5.8 allgemeinen Verfahren führt zu zwei Verfeinerungen des allgemeinen Syntheseverfahrens, deren Basis die folgenden zwei Sätze sind.

Satz 5.5.11 (Operator Match)

Eine Spezifikation spec=(D,R,I,O) ist genau dann erfüllbar, wenn es eine Algorithmentheorie $\mathcal{A}$ und ein Modell A für $\mathcal{A}$ gibt, mit der Eigenschaft spec⊴$\texttt{spec}_A$.

Satz 5.5.11 erweitert Satz 5.5.8 insofern, daß die Bedingung $spec$=$\texttt{spec}_A$ zu $spec \trianglelefteq \texttt{spec}_A$ abgeschwächt wurde, daß also anstelle von Gleichheit zwischen Spezifikation und dem Spezifikationsanteil des Modells nur noch Reduzierbarkeit gefordert wird. Entsprechend kann auch die Synthesestrategie verfeinert werden.

Strategie 5.5.12 (Synthese mit Algorithmenschemata und Reduktion)

Gegeben sei eine Problemspezifikation FUNCTION F(x:D):R WHERE I(x) RETURNS y SUCH THAT O(x,y)

1. *Wähle eine Algorithmentheorie $\mathcal{A}$ aus der Wissensbank*
2. *Wähle ein allgemeines Modell A von $\mathcal{A}$ aus der Wissensbank*
3. *Beweise* (D,R,I,O) $\trianglelefteq \texttt{spec}_A$ *und extrahiere aus dem Beweis zwei Transformationsfunktionen σ und θ*
4. *Spezialisiere* BODY(A) *und liefere als Antwort das Programm*
   FUNCTION F(x:D):R WHERE I(x) RETURNS y SUCH THAT O(x,y) = σ(θ(x),BODY(A)(θ(x)))

   Dieses Programm ist garantiert korrekt.

Dieses Verfahren läßt sich immer dann anwenden, wenn man eine Vorstellung davon hat, auf welche Art man das zu lösende Problem verallgemeinern möchte. Für die Auswahl von A ist daher eine gewisse Benutzersteuerung notwendig. So könnte man zum Beispiel Berechnungen über natürlichen Zahlen auf Berechnungen über Listen reduzieren, da die Zahl n isomorph zur Liste [1..n] ist, und auf diese Art einige Reduktionsstrategien für Listen auf Zahlen übertragen. Weitere Strategien zur Auswahl von A hängen im allgemeinen jedoch sehr von der konkreten Algorithmentheorie ab.

Für mengenwertige Problemstellungen läßt sich die Technik der Generalisierung einsetzen. Der folgende Satz sagt im wesentlichen, wie dies geschehen kann.

Satz 5.5.13 (Synthese durch Generalisierung)

> *Eine Spezifikation* FUNCTION F(x:D):Set(R) WHERE I(x) RETURNS { z | O(x,z)} *ist genau dann erfüllbar, wenn es eine Algorithmentheorie $\mathcal{A}$ und ein Modell A für $\mathcal{A}$ gibt, das (D,R,I,O) generalisiert.*

Man beachte, daß die Generalisierung sich auf die Ausgabebedingung O bezieht und nicht etwa auf { z | O(x,z)}. Da bei der Generalisierung die der Ausgabebereich des zu wählenden Modells bis auf Obermengenbildung bereits festliegt, läßt sich die allgemeine Strategie 5.5.9 wie folgt verfeinern.

Strategie 5.5.14 (Synthese mit Algorithmenschemata und Spezialisierung)

Gegeben sei eine Problemspezifikation FUNCTION F(x:D):R WHERE I(x) RETURNS { z | O(x,z)}

1. *Wähle eine Algorithmentheorie $\mathcal{A}$ aus der Wissensbank.*
2. *Wähle ein allgemeines Modell A von $\mathcal{A}$ aus der Wissensbank mit der Eigenschaft $R \subseteq R_A$.*
3. *Beweise* (D,R,I,O) $\trianglelefteq \texttt{spec}_A$ *und extrahiere aus dem Beweis eine Transformationsfunktion θ*
4. *Spezialisiere* BODY(A) *und liefere als Antwort das Programm*
   FUNCTION F(x:D):R WHERE I(x) RETURNS { z | O(x,z)} = { y | y ∈ BODY(A)(θ(x)) ∧ O(x,y) }

   Dieses Programm ist garantiert korrekt.

Der Anwendungsbereich dieser Grundstrategie, die sich je nach Algorithmentheorie noch ein wenig verfeinern läßt, ist erstaunlich groß. Sie hat besonders im Zusammenhang mit Globalsuch-Algorithmen (siehe Abschnitt 5.5.2) zu einer sehr effizienten Synthesestrategie geführt, in der außer dem Nachweis der Generalisierung und einer ähnlich leichten Überprüfung von sogenannten Filtern praktisch keine logischen Schlußfolgerungen – insbesondere also keine Induktionsbeweise – anfallen.

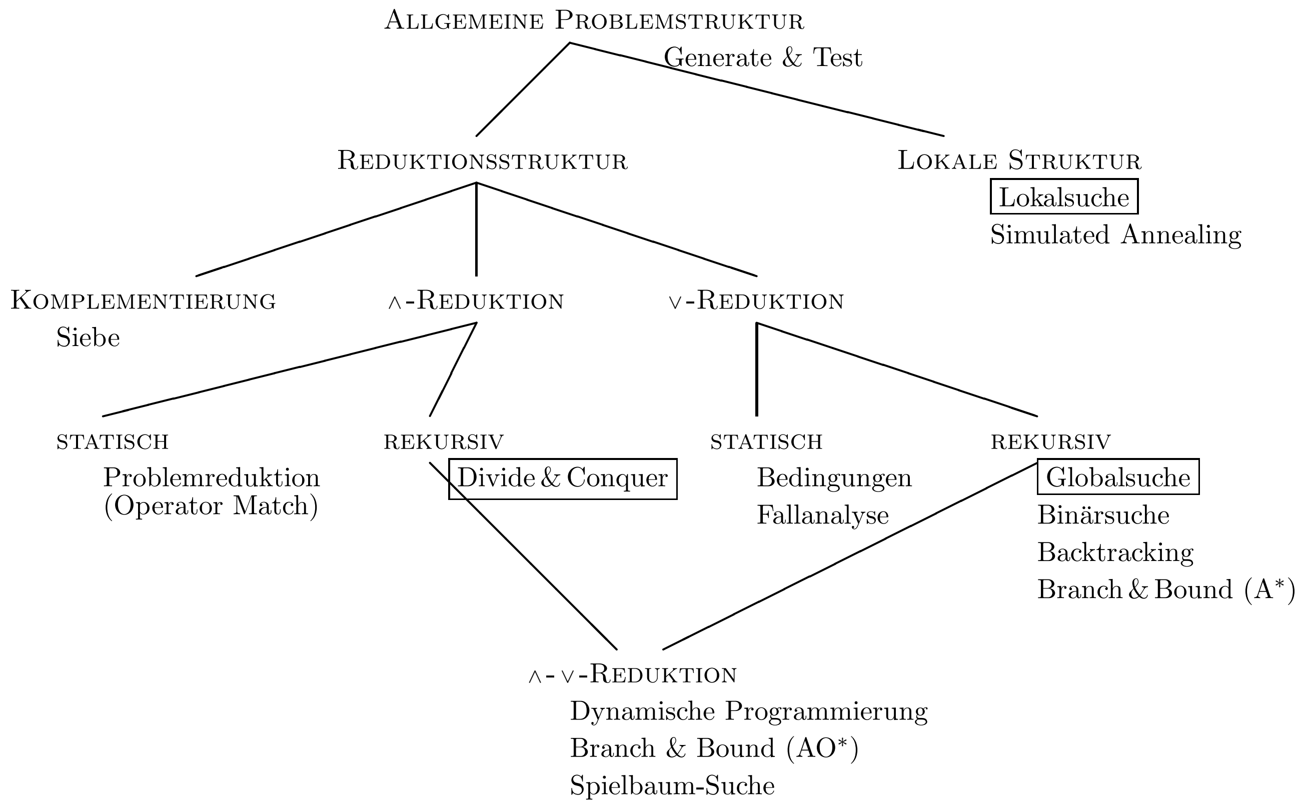


Abbildung 5.4: Hierarchie algorithmischer Theorien

Im Laufe der letzten Jahre ist eine Reihe algorithmischer Strukturen ausführlich untersucht und in der Form von Algorithmentheorien formalisiert worden. Dabei wurden auch Zusammenhänge zwischen den einzelnen Formalisierungen entdeckt, die sich mehr oder weniger direkt aus der hierachischen Anordnung der einzelnen Algorithmenstrukturen (siehe Abbildung 5.4) ergeben. Es erscheint daher sinnvoll, nicht nur einzelne Algorithmentheorien zu betrachten, sondern auch Mechanismen zur Verfeinerung von Theorien und zur Generierung von Entwurfsstrategien zu entwickeln. Derartige Mechanismen werden zur Zeit erforscht [Smith, 1992, Smith, 1993] und sollen in der Zukunft zu einer erheblichen Flexibilitätssteigerung von Synthesesystemen führen.

Wir wollen im folgenden nun einige konkrete Algorithmentheorien – Globalsuchalgorithmen, Divide-and-Conquer Algorithmen und Lokalsuchalgorithemn – im Detail betrachten und hieran die Leistungsfähigkeit dieses Paradigmas demonstrieren.

5.5.2 Globalsuch-Algorithmen

Problemlösung durch Aufzählen einer Menge von Lösungskandidaten ist eine weitverbreitete Programmiertechnik. *Globalsuche* ist eine Aufzählungsmethode, die als Verallgemeinerung von Binärsuche, Backtracking, Branch-and-Bound, Constraint Satisfaction, heuristischer Suche und ähnlichen Suchverfahren angesehen werden kann, und – zumindest aus theoretischer Sicht – zur Lösung jeder beliebigen Problemstellung eingesetzt werden kann.

Abstrahiert man von den Besonderheiten individueller Präsentationen so stellt sich heraus, daß die Grundidee, die all diese Algorithmen gemeinsam haben, eine Manipulation gewisser Mengen von Lösungskandidaten ist: man beginnt mit einer *Initialmenge*, die alle möglichen Lösungen enthält und *zerteilt* diese in kleinere Kandidatenmengen, aus denen man schließlich konkrete Lösungen *extrahieren* und überprüfen kann. Überflüssige Kandidatenmengen können durch *Filter* eliminiert werden. Der gesamte Prozeß wird wiederholt, bis keine weitere Aufspaltung der Kandidatenmengen mehr möglich ist.

Nur in den seltensten Fällen werden die Kandidatenmengen explizit dargestellt, da zu Beginn üblicherweise unendliche Mengen von Lösungskandidaten betrachtet werden müssen. Aus diesem Grunde werden Kandidatenmengen in Globalsuchalgorithmen im allgemeinen durch sogenannte *Deskriptoren* beschrieben – also durch endliche Objekte, die ein charakterisierendes Merkmal dieser Mengen repräsentieren – und die Zugehörigkeitsrelation zwischen Ausgabewerten und Kandidatenmengen durch ein allgemeineres *Erfüllbarkeitsprädikat* zwischen Ausgabewerten und Deskriptoren ersetzt.

Formalisiert man diese Vorgehensweise in einem einheitlichen Algorithmenschema, so erweist es sich als sinnvoll, Globalsuchalgorithmen durch mengenwertige Programme zu beschreiben, die versuchen *alle* möglichen Lösungen eines Problems zu berechnen. Auf diese Art kann tatsächlich jede der obengenannten Programmiertechniken als Spezialfall von Globalsuchalgorithmen dargestellt werden. Die allgemeine Grundstruktur von Globalsuchalgorithmen ist daher die folgende.

```
FUNCTION F(x:D):Set(R)  WHERE I(x)  RETURNS {z | O(x,z)}
= if Φ(x,s0(x))  then Fgs(x,s0(x))  else ∅
FUNCTION Fgs(x,s:D×S):Set(R)  WHERE I(x) ∧ J(x,s) ∧ Φ(x,s)
  RETURNS {z | O(x,z) ∧ sat(z,s)}
= {z | z∈ext(s) ∧ O(x,z)} ∪ ⋃{Fgs(x,t) | t∈split(x,s) ∧ Φ(x,t)}
```

Hierbei treten neben den Bestandteilen der eigentlichen Spezifikation die folgenden Komponenten auf.

- Ein Raum `S` von *Deskriptoren* für Mengen von Lösungskandidaten.
- Ein *Zugehörigkeitsprädikat* `sat` auf `R×S`, welches angibt, ob ein gegebener Wert `z∈R` zu der durch den Deskriptor `s∈S` beschriebenen Kandidatenmenge gehört.
- Ein Prädikat `J` auf `D×S`, welches angibt, ob ein gegebener Deskriptor `s∈S` für eine Eingabe `x∈D` überhaupt *sinnvoll*[23] ist.
- Ein *Initialdeskriptor* $\mathtt{s_0(x)}$ `∈S`, der den Ausgangspunkt des Suchverfahrens beschreibt und zwangsläufig alle alle korrekten Lösungen "enthalten" muß.
- Eine mengenwertige Funktion `ext`:`S→Set(R)`, welche die *direkte Extraktion* konkreter Lösungskandidaten aus einem Deskriptor durchführt.
- Eine mengenwertige Funktion `split`: `D×S → Set(S)`, welche die (rekursive) Aufspaltung von Deskriptoren in Mengen von Deskriptoren für "kleinere" Kandidatenmengen beschreibt.
- Ein Filter Φ: `D×S` $\to \mathbb{B}$, der dazu genutzt werden kann, unnötige Deskriptoren von der weiteren Betrachtung zu eliminieren. Diese dienen im wesentlichen der Effizienzsteigerung und können auch in die Funktion `split` integriert sein.

Ein Globalsuchalgorithmus `F` ruft also bei Eingabe eines Wertes `x` eine Hilfsfunktion F_{gs} mit `x` und dem Initialdeskriptor $\mathtt{s_0(x)}$ auf, sofern diese Werte den Filter Φ passieren. F_{gs} wiederum berechnet bei Eingabe von `x` und einem Deskriptor die Menge aller zulässigen Lösungen, die direkt aus `s` extrahiert werden können, und vereinigt diese mit der Menge aller Lösungen, die rekursiv durch Analyse aller Deskriptoren `t` bestimmt werden können, welche sich durch Aufspalten mit `split` und Filterung mit Φ ergeben.

Wir haben hier zur Beschreibung des Algorithmus eine funktionale Darstellung in der von uns angegebenen mathematischen Programmiersprache gewählt, da diese die kürzeste und eleganteste Darstellung des Verfahrens ermöglicht. Wir hätten ebenso aber auch ein Programmschema in einer beliebigen anderen Programmiersprache (sogar in einer parallelverarbeitenden Sprache) angeben können, denn hiervon hängt die Synthese nicht im geringsten ab. Wichtig ist nur, daß sich die obengenannten 7 zusätzlichen Komponenten in dem Schema wiederfinden lassen.

[23] Will man zum Beispiel Folgen über Elementen einer Eingabemenge `S⊆Set(ℤ)` aufzählen und Kandidatenmengen durch ihr größtes gemeinsames Präfix darstellen (siehe Beispiel 5.5.18), so muß dieser Deskriptor zwangsläufig eine Folge über `S` sein. Ansonsten würde keine sinnvolle Kandidatenmenge mehr dargestellt.

Nur 5 Axiome müssen erfüllt werden, um die Korrektheit des obengenannten Algorithmus sicherzustellen. Die ersten beiden davon sind nahezu trivial,denn sie verlangen nur, daß die Hilfsfunktion $\texttt{F}_{gs}$ ausschließlich mit sinnvollen Deskriptoren umgeht.

1. Für alle legalen Eingaben ist der Initialdeskriptor sinnvoll.
2. Splitting sinnvoller Deskriptoren liefert ausschließlich sinnvolle Deskriptoren

Das dritte und vierte Axiom garantiert die *Vollständigkeit* der Suche

3. Alle korrekten Lösungen sind in dem Initialdeskriptor enthalten
4. Alle korrekten Lösungen, die in einem Deskriptor `s` enthalten sind, lassen sich aus diesem auch nach endlich vielen `split`-Operationen extrahieren.

Das letzte Axiom garantiert schließlich Terminierung der Suche durch *Wohlfundiertheit* der `split`-Operation.

5. Jeder Deskriptor kann nur endlich oft in Deskriptorenmengen aufgespalten werden, bis keine aufspaltbaren Deskriptoren mehr vorhanden sind.

Um die letzten beiden Axiome formal präzise darstellen zu können, ist es nötig, die Menge aller Deskriptoren, die durch k-fache Interation der `split`-Operation entstehen, induktiv zu definieren.

Definition 5.5.15

Die k-fache Iteration $\underline{split^k}$ einer Operation split: $\texttt{D}\times\texttt{S}\rightarrow\texttt{Set(S)}$ *ist induktiv definiert durch*

$$split^k\texttt{(x,s)} \equiv \texttt{if } k\texttt{=0 then } \{\texttt{s}\} \texttt{ else } \bigcup\{\, split^{k-1}\texttt{(x,t)} \mid \texttt{t}\in split\texttt{(x,s)}\,\}$$

Mit dieser Definition können wir die obengenannten Axiome formalisieren und die Korrektheit des schematischen Globalsuchalgorithmus beweisen. Wir lassen dabei zunächst die Filter außer Betracht, da diese später gesondert betrachtet und bis dahin als Bestandteil der `split`-Operation aufgefaßt werden können.

Satz 5.5.16 (Korrektheit von Globalsuch-Algorithmen)

Es sei spec=(D, R, I, O) *eine beliebige Spezifikation,* S *ein Datentyp,* $\texttt{J}: \texttt{D}\times\texttt{S}\rightarrow\mathbb{B}$, $\texttt{s}_0\texttt{:D}\rightarrow\texttt{S}$, $\texttt{sat}: \texttt{R}\times\texttt{S}\rightarrow\mathbb{B}$, $\texttt{split}: \texttt{D}\times\texttt{S}\rightarrow\texttt{Set(S)}$ *und* $\texttt{ext:S}\rightarrow\texttt{Set(R)}$. *Das Programmpaar*

$$\texttt{FUNCTION F(x:D):Set(R) WHERE I(x) RETURNS } \{\texttt{z} \mid \texttt{O(x,z)}\} = \texttt{F}_{gs}\texttt{(x,s}_0\texttt{(x))}$$

$$\texttt{FUNCTION F}_{gs}\texttt{(x,s:D}\times\texttt{S):Set(R) WHERE I(x)} \wedge \texttt{J(x,s) RETURNS } \{\texttt{z} \mid \texttt{O(x,z)} \wedge \texttt{sat(z,s)}\}$$
$$= \{\texttt{z} \mid \texttt{z}\in\texttt{ext(s)} \wedge \texttt{O(x,z)}\} \cup \bigcup\{\texttt{F}_{gs}\texttt{(x,t)} \mid \texttt{t}\in\texttt{split(x,s)}\}$$

ist korrekt, wenn für alle $\texttt{x}\in\texttt{D}$, $\texttt{s}\in\texttt{S}$ *und* $\texttt{z}\in\texttt{R}$ *die folgenden fünf Axiome erfüllt sind*

1. $\texttt{I(x)} \Rightarrow \texttt{J(x,s}_0\texttt{(x))}$
2. $\texttt{I(x)}\wedge\texttt{J(x,s)} \Rightarrow \forall\texttt{t}\in\texttt{split(x,s).J(x,t)}$
3. $\texttt{I(x)}\wedge\texttt{O(x,z)} \Rightarrow \texttt{sat(z,s}_0\texttt{(x))}$
4. $\texttt{I(x)}\wedge\texttt{J(x,s)}\wedge\texttt{O(x,z)} \Rightarrow \texttt{sat(z,s)} \Leftrightarrow \exists\texttt{k:}\mathbb{N}.\exists\texttt{t}\in\texttt{split}^\texttt{k}\texttt{(x,s). z}\in\texttt{ext(t)}$
5. $\texttt{I(x)}\wedge\texttt{J(x,s)} \Rightarrow \exists\texttt{k:}\mathbb{N}.\ \texttt{split}^\texttt{k}\texttt{(x,s)}=\emptyset$

Beweis: Es sei D,R,I,O, S,J,$\texttt{s}_0$,sat,split,ext wie oben angegeben und die Axiome 1–5 seien erfüllt. Dann gilt

1. *Wenn für ein beliebiges* $\texttt{x}\in\texttt{D}$ *und* $\texttt{s}\in\texttt{S}$ *die Berechnung von* $\texttt{F}_{gs}\texttt{(x,s)}$ *nach* `i` *Schritten terminiert (d.h. wenn* $\texttt{split}^\texttt{i}\texttt{(x,s)}=\emptyset$ *ist), so ist das Ergebnis der Berechnung die Mennge aller Lösungen, die aus Deskriptoren extrahierbar sind, die zu einem* $\texttt{split}^\texttt{j}\texttt{(x,s)}$ *mit* `j<i` *gehören:*

$$\texttt{F}_{gs}\texttt{(x,s)} = \bigcup\{\ \{\texttt{z} \mid \texttt{z}\in\texttt{ext(t)} \wedge \texttt{O(x,z)}\} \mid \texttt{t}\in\bigcup\{\texttt{split}^\texttt{j}\texttt{(x,s)} \mid \texttt{j}\in\{\texttt{0..i-1}\}\}\ \}$$

Dies läßt sich durch Induktion über `i` und viele Detailberechnungen wie folgt beweisen.

- i=0 ist nach Definition $\texttt{split}^{\texttt{i}}\texttt{(x,s)} = \{\texttt{s}\} \neq \emptyset$, da die Berechnung von $\texttt{F}_{gs}\texttt{(x,s)}$ nicht anhalten kann, ohne mindestens einen Schritt auszuführen. Damit ist die Behauptung bewiesen, da die Voraussetzungen nicht erfüllt sind, und somit ist die Induktion verankert.
- Da die obige Beweisführung zwar logisch korrekt ist, den meisten Menschen jedoch suspekt erscheint, hier noch eine zweite Verankerung bei `i=1`.
  Es ist $\texttt{split}^1\texttt{(x,s)} = \bigcup\{\texttt{split}^0\texttt{(x,t)} \mid \texttt{t} \in \texttt{split(x,s)}\} = \bigcup\{\{\texttt{t}\} \mid \texttt{t} \in \texttt{split(x,s)}\} = \texttt{split(x,s)}$
  Ist also $\texttt{split}^1\texttt{(x,s)} = \emptyset$, so folgt
  $\texttt{F}_{gs}\texttt{(x,s)}$
  $= \{\texttt{z} \mid \texttt{z} \in \texttt{ext(s)} \wedge \texttt{O(x,z)}\} \cup \bigcup\{\texttt{F}_{gs}\texttt{(x,t)} \mid \texttt{t} \in \texttt{split(x,s)}\}$
  $= \{\texttt{z} \mid \texttt{z} \in \texttt{ext(s)} \wedge \texttt{O(x,z)}\} \cup \bigcup\{\texttt{F}_{gs}\texttt{(x,t)} \mid \texttt{t} \in \emptyset\}$
  $= \{\texttt{z} \mid \texttt{z} \in \texttt{ext(s)} \wedge \texttt{O(x,z)}\}$
  $= \bigcup\{\ \{\texttt{z} \mid \texttt{z} \in \texttt{ext(t)} \wedge \texttt{O(x,z)}\} \mid \texttt{t} \in \{\texttt{s}\}\ \}$
  $= \bigcup\{\ \{\texttt{z} \mid \texttt{z} \in \texttt{ext(t)} \wedge \texttt{O(x,z)}\} \mid \texttt{t} \in \texttt{split}^0\texttt{(x,s)}\ \}$
  $= \bigcup\{\ \{\texttt{z} \mid \texttt{z} \in \texttt{ext(t)} \wedge \texttt{O(x,z)}\} \mid \texttt{t} \in \bigcup\{\texttt{split}^{\texttt{j}}\texttt{(x,s)} \mid \texttt{j} \in \{\texttt{0..1-1}\}\}\ \}$
  Damit ist die Behauptung für `i=1` ebenfalls verankert.
- Es sei für ein `i>0` und alle $\texttt{x} \in \texttt{D}$ und $\texttt{s} \in \texttt{S}$ bewiesen
  $$\texttt{F}_{gs}\texttt{(x,s)} = \bigcup\{\ \{\texttt{z} \mid \texttt{z} \in \texttt{ext(t)} \wedge \texttt{O(x,z)}\} \mid \texttt{t} \in \bigcup\{\texttt{split}^{\texttt{j}}\texttt{(x,s)} \mid \texttt{j} \in \{\texttt{0..i-1}\}\}\ \},$$
  falls $\texttt{split}^{\texttt{i}}\texttt{(x,s)} = \emptyset$ ist.
- Es gelte $\texttt{split}^{\texttt{i+1}}\texttt{(x,s)} = \emptyset$. Nach Definition 5.5.15 ist $\texttt{split}^{\texttt{i+1}}\texttt{(x,s)} = \bigcup\{\texttt{split}^{\texttt{i}}\texttt{(x,t)} \mid \texttt{t} \in \texttt{split(x,s)}\}$ und somit folgt nach den üblichen Gesetzen über die Mengenvereinigung, daß $\texttt{split}^{\texttt{i}}\texttt{(x,t)} = \emptyset$ für alle $\texttt{t} \in \texttt{split(x,s)}$ gilt. Damit ist die Induktionsannahme für diese `t` anwendbar und es gilt
  $$\texttt{F}_{gs}\texttt{(x,t)} = \bigcup\{\ \{\texttt{z} \mid \texttt{z} \in \texttt{ext(u)} \wedge \texttt{O(x,z)}\} \mid \texttt{u} \in \bigcup\{\texttt{split}^{\texttt{j}}\texttt{(x,t)} \mid \texttt{j} \in \{\texttt{0..i-1}\}\}\ \}$$
  für alle $\texttt{t} \in \texttt{split(x,s)}$. Somit folgt
  $\texttt{F}_{gs}\texttt{(x,s)}$
  $= \{\texttt{z} \mid \texttt{z} \in \texttt{ext(s)} \wedge \texttt{O(x,z)}\} \cup \bigcup\{\texttt{F}_{gs}\texttt{(x,t)} \mid \texttt{t} \in \texttt{split(x,s)}\}$
  $= \{\texttt{z} \mid \texttt{z} \in \texttt{ext(s)} \wedge \texttt{O(x,z)}\}$
  $\quad \cup \bigcup\{\ \bigcup\{\ \{\texttt{z} \mid \texttt{z} \in \texttt{ext(u)} \wedge \texttt{O(x,z)}\} \mid \texttt{u} \in \bigcup\{\texttt{split}^{\texttt{j}}\texttt{(x,t)} \mid \texttt{j} \in \{\texttt{0..i-1}\}\}\ \} \mid \texttt{t} \in \texttt{split(x,s)}\}$
  $= \bigcup\{\{\texttt{z} \mid \texttt{z} \in \texttt{ext(u)} \wedge \texttt{O(x,z)}\} \mid \texttt{u} \in \{\texttt{s}\}\}$
  $\quad \cup \bigcup\{\ \{\texttt{z} \mid \texttt{z} \in \texttt{ext(u)} \wedge \texttt{O(x,z)}\} \mid \texttt{u} \in \bigcup\{\texttt{split}^{\texttt{j+1}}\texttt{(x,s)} \mid \texttt{j} \in \{\texttt{0..i-1}\}\}\ \}$
  $= \bigcup\{\{\texttt{z} \mid \texttt{z} \in \texttt{ext(u)} \wedge \texttt{O(x,z)}\} \mid \texttt{u} \in \texttt{split}^0\texttt{(x,s)}\}$
  $\quad \cup \bigcup\{\ \{\texttt{z} \mid \texttt{z} \in \texttt{ext(u)} \wedge \texttt{O(x,z)}\} \mid \texttt{u} \in \bigcup\{\texttt{split}^{\texttt{k}}\texttt{(x,s)} \mid \texttt{k} \in \{\texttt{1..i}\}\}\ \}$
  $= \bigcup\{\ \{\texttt{z} \mid \texttt{z} \in \texttt{ext(u)} \wedge \texttt{O(x,z)}\} \mid \texttt{u} \in \bigcup\{\texttt{split}^{\texttt{k}}\texttt{(x,s)} \mid \texttt{k} \in \{\texttt{0..i}\}\}\ \}$
  Damit ist die Behauptung auch für `i+1` beweisen.

Es sei angemerkt, daß durch das zweite Axiom sichergestellt ist, daß die Funktion $\texttt{F}_{gs}$ auf Eingaben `(x,t)` mit $\texttt{t} \in \texttt{split(x,s)}$ überhaupt anwendbar ist.

2. *Das Hilfsprogramm $\texttt{F}_{gs}$ ist für alle zulässigen Eingaben korrekt.*

Es sei $\texttt{x} \in \texttt{D}$ und $\texttt{s} \in \texttt{S}$ mit $\texttt{I(x)} \wedge \texttt{J(x,s)}$. Aufgrund von Axiom 5 (Wohlfundiertheit) gibt es dann eine Zahl $\texttt{k} \in \mathbb{N}$ mit der Eigenschaft $\texttt{split}^{\texttt{k}}\texttt{(x,s)} = \emptyset$. Für dieses `k` gilt nach (1.):

$$\texttt{F}_{gs}\texttt{(x,s)} = \bigcup\{\ \{\texttt{z} \mid \texttt{z} \in \texttt{ext(t)} \wedge \texttt{O(x,z)}\} \mid \texttt{t} \in \bigcup\{\texttt{split}^{\texttt{j}}\texttt{(x,s)} \mid \texttt{j} \in \{\texttt{0..k-1}\}\}\ \}$$

Für einen beliebigen Ausgabewert $\texttt{z} \in \texttt{R}$ gilt somit

$$\texttt{z} \in \texttt{F}_{gs}\texttt{(x,s)} \Leftrightarrow \exists \texttt{j} \in \{\texttt{0..k-1}\}.\ \exists \texttt{t} \in \texttt{split}^{\texttt{j}}\texttt{(x,s)}.\ \texttt{z} \in \texttt{ext(t)} \wedge \texttt{O(x,z)}$$

Da $\texttt{split}^{\texttt{k}}\texttt{(x,s)} = \emptyset$ ist, läßt sich dies verallgemeinern und umschreiben zu

$$\texttt{z} \in \texttt{F}_{gs}\texttt{(x,s)} \Leftrightarrow \texttt{O(x,z)} \wedge \exists \texttt{j}:\mathbb{N}.\ \exists \texttt{t} \in \texttt{split}^{\texttt{j}}\texttt{(x,s)}.\ \texttt{z} \in \texttt{ext(t)}$$

Mit Axiom 4 folgt nun $\texttt{z} \in \texttt{F}_{gs}\texttt{(x,s)} \Leftrightarrow \texttt{O(x,z)} \wedge \texttt{sat(z,s)}$, also $\texttt{F}_{gs}\texttt{(x,s)} = \{\texttt{z} \mid \texttt{O(x,z)} \wedge \texttt{sat(z,s)}\}$. Damit ist die Hilfsfunktion korrekt

$\texttt{FUNCTION F}_{gs}\texttt{(x,s:D}\times\texttt{S):Set(R) WHERE I(x)} \wedge \texttt{J(x,s) RETURNS } \{\texttt{z} \mid \texttt{O(x,z)} \wedge \texttt{sat(z,s)}\}$
$= \{\texttt{z} \mid \texttt{z} \in \texttt{ext(s)} \wedge \texttt{O(x,z)}\} \cup \bigcup\{\texttt{F}_{gs}\texttt{(x,t)} \mid \texttt{t} \in \texttt{split(x,s)}\}$

3. *Das Hauptprogramm* F *ist für alle zulässigen Eingaben korrekt.*

Es sei $x \in D$ mit I(x). Nach Axiom 1 folgt dann $J(x,s_0(x))$ und damit ist $(x,s_0(x))$ eine zulässige Eingabe für die Hilfsfunktion F_{gs}. Da F_{gs} auf solchen Eingaben korrekt ist, folgt

$$F(x) = F_{gs}(x,s_0(x)) = \{ z \mid O(x,z) \wedge sat(z,s_0(x)) \}$$

Aufgrund von Axiom 3 gilt somit für einen beliebigen Ausgabewert $z \in R$

$$z \in F(x) \Leftrightarrow O(x,z) \wedge sat(z,s_0(x)) \Leftrightarrow O(x,z)$$

Damit ist also $F(x) = \{ z \mid O(x,z) \}$ und somit ist das Hauptprogramm F ebenfalls korrekt.

FUNCTION F(x:D):Set(R) WHERE I(x) RETURNS $\{ z \mid O(x,z) \} = F_{gs}(x,s_0(x))$ □

Satz 5.5.16 gibt uns also eine präzise Beschreibung der Komponenten, die für die Erzeugung von Globalsuchalgorithmen nötig sind, und der Axiome, die erfüllt sein müssen, um die Korrektheit des schematisch erzeugten Algorithms zu garantieren. Darüber hinaus beschreibt er implizit eine Strategie zur Erzeugung von korrekten Globalsuchalgorithmen: es reicht, die zusätzlichen Komponenten S, J, s_0, sat, split und ext zu bestimmen, die fünf Axiome zu überprüfen und dann den schematischen Algorithmus zu instantiieren. Offen bleibt jedoch, *wie* diese 6 Komponenten effizient gefunden werden können, und wie man sich die Überprüfung der schwierigeren Axiome 4 und 5 zur Laufzeit des Syntheseprozesses erleichtern kann.

Die Antwort hierfür ist verblüffend einfach: *wir holen die wesentlichen Informationen direkt aus einer Wissensbank*. Dies heißt natürlich nicht, daß nun für jeden möglichen Algorithmus die 6 zusätzlichen Komponenten direkt in der Wissensbank zu finden sind,aber es bedeutet in der Tat, daß vorgefertigte Suchstrukturen – sogenannte *Globalsuchtheorien* – in der Wissensbank abgelegt werden. Diese beschreiben die Komponenten von allgemeinen Globalsuchalgorithmen, welche die grundsätzlichen Techniken zur Aufzählung der Elemente dieser Bereiche widerspiegeln und mit Hilfe des Spezialisierungsmechanismus aus dem vorigen Abschnitt (siehe Definition 5.5.10 und Strategie 5.5.14 auf Seite 255) auf eine konkrete Problemstellung angepaßt werden können. Die Erfahrung hat gezeigt, daß nur 6 dieser vorgefertigten Globalsuchtheorien ausreichen, um alle in der Programmsynthese jemals erforschten Probleme zu synthetisieren.

Es hat sich als sinnvoll herausgestellt, Globalsuchtheorien als ein Objekte zu definieren, die aus 10 Komponenten $(D, R, I, O, S, J, s_0, sat, split, ext)$ bestehen, welche die ersten 4 Axiome erfüllen. Dies liegt darin begründet, daß nicht alle allgemeinen Aufzählungsverfahren von sich aus wohlfundiert sind, sondern meist erst im Rahmen der Spezialisierung durch Hinzunahme sogenannter *notwendiger Filter* zu wohlfundierten Algorithmen verfeinert werden. Diese Filter müssen sich aus den Ausgabebedingungen des Problems (und dem Erfüllbarkeitsprädikat) ableiten lassen und modifizieren die split-Operation durch Elimination von Deskriptoren, die nicht mehr zu einer Lösung beitragen können, zu einer effizienteren und wohlfundierten Operation. Da Wohlfundiertheit zur Laufzeit eines Syntheseprozesses im allgemeinen nur sehr schwer zu beweisen ist – immerhin müssten hierzu Induktionsbeweise automatisch (!) geführt werden – lohnt es sich ebenfalls, in der Wissensbank zu jeder Globalsuchtheorie, die nicht bereits wohlfundiert ist, eine kleine Anzahl vorgefertigter Filter bereitzustellen, welche dafür sorgen, daß auch das fünfte Axiom erfüllt wird. Diese *Wohlfundiertheits-Filter* müssen zur Laufzeit dann nur noch daraufhin geprüft werden, ob sie zu der Ausgabebedingung des konkreten Problems passen.

Mit diesen vorgefertigten Informationen kann das Problem, passend zu einer gegebenen Spezifikation 6 neue Komponenten zu bestimmen und 5 Axiome zu überprüfen, darauf reduziert werden, eine geeignete Globalsuchtheorie und einen passenden Wohlfundiertheits-Filter aus der Wissensbank auszuwählen, die Generalisierungseigenschaft nachzuweisen, die Globalsuchtheorie samt Filter zu spezialisieren und – wenn dies *leicht* möglich ist – durch einen zusätzlichen notwendigen Filter weiter zu verfeinern. Anschließend kann dann mit Hilfe von Satz 5.5.16 ein korrekter Globalsuchalgorithmus für die Spezifikation erzeugt werden.

Um diese Strategie zur Konstruktion von Globalsuchalgorithmen genauer beschreiben zu können, müssen wir die Konzepte *Globalsuchtheorie*, *Filter* und *Spezialisierung* präze definieren und zeigen, wie sie dazu beitragen können, zu einer gegebenen Spezifikation die Voraussetzungen für die Anwendbarkeit von Satz 5.5.16 zu schaffen.

Definition 5.5.17 (Globalsuchtheorie)

Eine Globalsuchtheorie ist ein 10-Tupel G=((D,R,I,O),S,J,s_0,sat,split,ext)*, wobei* (D,R,I,O) *eine Spezifikation ist,* S *ein Datentyp,* J: D×S → 𝔹*,* s_0:D→S*,* sat: R×S → 𝔹*,* split: D×S → Set(S)*,* ext:S→Set(R) *und für alle* x∈D, s∈S *und* z∈R *die folgenden vier Bedingungen gelten.*

1. I(x) ⇒ J(x,s_0(x))
2. I(x) ∧ J(x,s) ⇒ ∀t∈split(x,s).J(x,t)
3. I(x) ∧ O(x,z) ⇒ sat(z,s_0(x))
4. I(x) ∧ J(x,s) ∧ O(x,z) ⇒ sat(z,s) ⇔ ∃k:ℕ.∃t∈$split^k$(x,s). z∈ext(t)

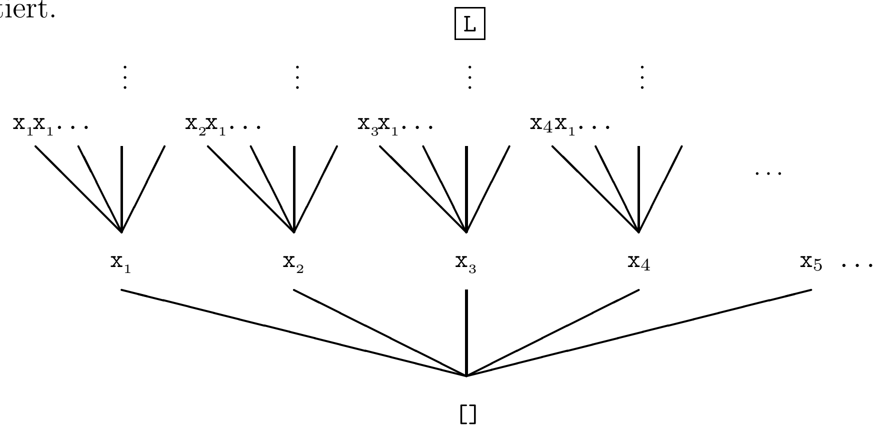
Eine Globalsuchtheorie G heißt wohlfundiert, wenn darüberhinaus für alle x∈D *und* s∈S *gilt:*

I(x) ∧ J(x,s) ⇒ ∃k:ℕ. $split^k$(x,s) = ∅

Wohlfundierte Globalsuchtheorien erfüllen also genau die Voraussetzungen von Satz 5.5.16 und das Ziel einer Synthese ist also, derartige wohlfundierte Globalsuchtheorien mit Hilfe vorgefertigter Informationen zu generieren. In der Wissensbank müssen daher verschiedene allgemeine Suchstrukturen für die wichtigsten *Ausgabebereiche* als Globalsuchtheorien abgelegt werden. Wir wollen hierfür ein Beispiel betrachten.

Beispiel 5.5.18 (Globalsuchtheorie für Folgen über einer endlichen Menge)

Die vorgefertigte Globalsuchtheorie gs_seq_set(α) beschreibt eine allgemeine Suchstruktur für Folgen L∈Seq(α), deren Elemente aus einer gegebenen endlichen Menge[24] S⊆α stammen müssen. Die Suche geschieht durch Aufzählung der Elemente von L, was bedeutet, daß schrittweise alle Präfixfolgen von L aufgebaut werden, bis die gesuchte Folge gefunden ist. Dieses Suchverfahren wird durch folgenden Suchbaum repräsentiert.



Dieser Baum zeigt, daß die jeweiligen Kandidatenmengen des Suchverfahrens Mengen von Folgen sind, welche die gleiche Präfixfolge V besitzen, also die Gestalt { L | VpreL } haben, Es bietet sich daher an die Präfixfolge V als Deskriptor und λL,V. VpreL als Erfüllbarkeitsprädikat zu verwenden. Sinnvoll sind natürlich nur solche Deskriptoren, die ausschließlich aus Elementen von S bestehen. Die Suche beginnt üblicherweise mit der Menge aller Folgen über S, also mit [] als Initialdeskriptor.

Da bei der Suche schrittweise die Elemente von L aufgezählt werden, ist das Aufspalten von Kandidatenmengen nichts anderes als eine Verlängerung der Deskriptorfolge V um ein weiteres Element der Menge S, was einer Auffächerung der Kandidatenmengen im Baum entspricht. Ist die gesamte Folge L aufgezählt, so ist die Suche beendet und L kann direkt extrahiert werden: Extraktion bedeutet somit, die gesamte Präfixfolge V auszuwählen.

Faßt man all dies in einer Globalsuchtheorie zusammen, so ergibt sich das folgende Objekt, das wir der Lesbarkeit halber komponentenweise beschreiben.

[24]Der Name S steht an dieser Stelle als Platzhalter für eine typische Variable vom Typ Set und darf nicht verwechselt werden mit dem Platzhalter S bei der Beschreibung der Komponenten von Globalsuchtheorien, der das Wort “Suchraum” abkürzt. Unglücklicherweise treten hier Konflikte bei der Vergabe von Standardnamen auf, die nur aus dem Kontext her auflösbar sind.

$$
\underline{\texttt{seq_set}(\alpha)} \equiv
\begin{array}{lll}
\texttt{D} & \mapsto & \texttt{Set}(\alpha) \\
\texttt{R} & \mapsto & \texttt{Seq}(\alpha) \\
\texttt{I} & \mapsto & \lambda\texttt{S. true} \\
\texttt{O} & \mapsto & \lambda\texttt{S,L. range(L)}\subseteq\texttt{S} \\
\texttt{S} & \mapsto & \texttt{Seq}(\alpha) \\
\texttt{J} & \mapsto & \lambda\texttt{S,V. range(V)}\subseteq\texttt{S} \\
\texttt{s}_0 & \mapsto & \lambda\texttt{S. []} \\
\texttt{sat} & \mapsto & \lambda\texttt{L,V. V}\sqsubseteq\texttt{L} \\
\texttt{split} & \mapsto & \lambda\texttt{S,V.} \{\texttt{V}\cdot\texttt{i} \mid \texttt{i}\in\texttt{S}\} \\
\texttt{ext} & \mapsto & \lambda\texttt{V.} \{\texttt{V}\}
\end{array}
$$

Man beachte, daß es sich hierbei um eine generische Globalsuchtheorie handelt, die auf beliebigen Datentypen α definiert ist und sehr vielen Suchalgorithmen, die auf endlichen Folgen operieren, zugrundeliegt.

Der Beweis, daß `gs_seq_set(`α`)` tatsächlich eine Globalsuchtheorie im Sinne von Definition 5.5.17 ist, ist eine einfache Übungsaufgabe. Man beachte jedoch, daß `gs_seq_set(`α`)` nicht wohlfundiert ist, da mit dieser Suchstruktur Folgen beliebiger Länge aufgezählt werden können.

Mit der Globalsuchtheorie `gs_seq_set(`α`)` haben wir eine allgemeine Suchstruktur für endliche Listen (bzw. Folgen) formalisiert. Wie können wir nun derartige Informationen dazu nutzen, um ein Suchverfahren für eine *konkrete* Problemstellung zu generieren?

Wie bereits angedeutet, wollen wir uns hierzu den Spezialisierungsmechanismus zunutze machen, den wir erstmals in Strategie 5.5.14 auf Seite 255 verwendet haben. Hierzu brauchen wir nur nachzuweisen, daß der Spezifikationsanteil einer Globalsuchtheorie wie `gs_seq_set(`α`)` eine gegebene Spzifikation im Sinne von Definition 5.5.10 generalisiert, aus dem Nachweis eine Transformation θ zu extrahieren, und die Globalsuchtheorie mit θ auf das Ausgangsproblem zu spezialisieren. Wir wollen dies zunächst an einem Beispiel erklären.

Beispiel 5.5.19 (Spezialisierung von `gs_seq_set(`$\mathbb{Z}$`)`)

In Beispiel 5.2.4 auf Seite 228 hatten wir eine Formalisierung des Costas-Arrays Problems vorgestellt, bei dem es um die Bestimmung aller Permutationen der Zahlen `{1..n}` ging, die in keiner Zeile ihrer Differenzentafel doppelt vorkommende Einträge besitzen. Wir hatten hierfür die folgende formale Spezifikation entwickelt.

```
FUNCTION Costas (n:ℤ):Set(Seq(ℤ))  WHERE n≥1
   RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
```

Die unterstrichenen Teile markieren dabei die Komponenten `D`, `R`, `I` und `O` der Problemstellung.

Da der Ausgabebereich der Funktion `Costas` die Menge der endlichen Folgen über den ganzen Zahlen ist, liegt der Versuch nahe, als Lösungsstruktur eine Spezialisierung der Globalsuchtheorie `gs_seq_set(`$\mathbb{Z}$`)` (d.h. der Typparameter α wurde mit $\mathbb{Z}$ instantiiert) zu verwenden. Wir müssen also zeigen

R' ist eine Obermenge von R und es gilt

$\forall$`x:D. I(x)` $\Rightarrow$ $\exists$`x':D'. I'(x')` $\wedge$ $\forall$`y:R. O(x,y)` $\Rightarrow$ `O'(x',y)`

wobei die gestrichenen Komponenten sich auf `gs_seq_set(`$\mathbb{Z}$`)` und die anderen auf das Costas-Arrays Problem beziehen.

1. Die Globalsuchtheorie `gs_seq_set(`$\mathbb{Z}$`)` ist genau so gewählt worden, daß die Ausgabebereiche `R=Seq(`$\mathbb{Z}$`)` und `R'` übereinstimmen. Damit ist die prinzipielle Generalisierbarkeit gewährleistet.
2. Die Eingabebereiche `D=`$\mathbb{Z}$ und `D'=Set(`$\mathbb{Z}$`)` stimmen nicht überein und müssen aneinander angepaßt werden. Auch dies ist prinzipiell möglich, da Mengen mehr Informationen enthalten als natürliche Zahlen, muß aber im Detail noch nachgewiesen werden.
3. Die Eingabebedingung `I'` von `gs_seq_set(`$\mathbb{Z}$`)` ist immer `true`, braucht also nicht weiter berücksichtigt zu werden.

4. Zu zeigen bleibt also
   $\forall$n:$\mathbb{Z}$.n$\geq$1 $\Rightarrow$ $\exists$S:Set($\mathbb{Z}$).
   $\forall$p:Seq($\mathbb{Z}$).perm(p,{1..n}) $\wedge$ $\forall$j$\in$domain(p).nodups(dtrow(p,j)) $\Rightarrow$ range(p)$\subseteq$S

   Für diesen Nachweis gibt es eine relativ einfache Heuristik, die fast immer in wenigen Schritten zum Ziel führt und nicht einmal einen vollständigen Theorembeweiser benötigt. Das Ziel der Inferenzen muß sein, durch Auffalten von Definitionen und Anwendung von Lemmata Folgerungen von

   perm(p,{1..n}) $\wedge$ $\forall$j$\in$domain(p).nodups(dtrow(p,j))

   zu bestimmen, in denen range(p) auf der linken Seite eines Teilmengenoperators vorkommt und auf der rechten Seite ein Ausdruck steht, der für S eingesetzt werden kann.

   Auffalten von perm(p,{1..n}) liefert perm(p,{1..n}) $\Leftrightarrow$ range(p)={1..n} $\wedge$ nodups(p), woraus wiederum durch Anwendung des Gesetzes über Mengengleichheit und Teilmengenbeziehungen (Lemma B.1.13.13, die Wahl ist eindeutig!) folgt perm(p,{1..n}) $\Rightarrow$ range(p)$\subseteq$ {1..n}.

   Damit ist die obige Behauptung gültig, wenn wir für S die Menge {1..n} wählen.

Aus dem Beweis ergibt sich eine Transformation $\theta=\lambda$n.{1..n}, die gemäß Theorem 5.5.13 zu einer Spezialisierung des generellen Suchalgorithmus für gs_seq_set($\mathbb{Z}$) zu einem Lösungsalgorithmus für das Costas-Arrays Problem[25] führt. Diese Spezialisierung führt, wie in Strategie 5.5.14 bereits ausgeführt, zu einer Umwandlung eines Eingabewertes n von Costas in einen Eingabewert θ(n) für den durch gs_seq_set($\mathbb{Z}$) charakterisierten Lösungsalgorithmus und der Elimination von Lösungen, die nicht die Ausgabebedingung O des Costas-Arrays Problems erfüllen.

Anstelle den Globalsuchalgorithmus zu einem Lösungsverfahren für das konkrete Syntheseproblem zu modifizieren, kann man auch die Globalsuchtheorie gs_seq_set($\mathbb{Z}$) selbst spezialisieren und in einer Globalsuchtheorie für das Costas-Arrays Problem umwandeln. Hierzu muß man nur den Spezifikationsanteil durch die speziellere konkrete Spezifikation austauscht und in den anderen Komponenten jedes Vorkommen der Menge S durch θ(n) ersetzt. Dies führt dann zu folgender spezialisierter Globalsuchtheorie, die in der Tat eine Globalsuchtheorie für das Costas-Arrays Problem ist.

D	$\mapsto$	Set($\mathbb{Z}$)
R	$\mapsto$	Seq($\mathbb{Z}$)
I	$\mapsto$	λn. n$\geq$1
O	$\mapsto$	λn, p. perm(p,{1..n}) $\wedge$ $\forall$j$\in$domain(p).nodups(dtrow(p,j))
S	$\mapsto$	Seq($\mathbb{Z}$)
J	$\mapsto$	λn, V. range(V)$\subseteq${1..n}
s_0	$\mapsto$	λn. []
sat	$\mapsto$	λp, V. V$\sqsubseteq$p
split	$\mapsto$	λn, V. {V·i|i$\in${1..n}}
ext	$\mapsto$	λV. {V}

Die in Beispiel 5.5.19 vorgenommene Spezialisierung von Globalsuchtheorien ist nur wegen des direkten Zusammenhangs zwischen Globalsuchtheorien und Globalsuchalgorithmen und einem expliziten Vorkommen der Ausgabebedingung innerhalb des schematischen Algorithmus möglich. Sie hat gegenüber der in Strategie 5.5.14 vorgestellten nachträglichen Modifikation des Algorithmus den Vorteil, daß die Spezialisierung gezielter in die Struktur des Algorithmus eingreift und somit zu effizienteren Programmen führt. Sie wird durch das folgende Theorem gerechtfertigt.

Satz 5.5.20 (Spezialisierung von Globalsuchtheorien)

Es sei spec = (D,R,I,O) *eine formale Spezifikation und* G=(D',R',I',O',S,J,s_0,sat,split,ext) *eine Globalsuchtheorie. Es gelte spec* $\ll$ spec$_G$=(D',R',I',O'). θ *sei die aus dem Spezialisierungsbeweis extrahierte Transformation.*

Dann ist G_θ(*spec*) = (D,R,I,O, S, λx,s.J(θ(x),s), λx.s_0(θ(x)), sat, λx,s.split(θ(x),s), ext) *eine Globalsuchtheorie.* G_θ(*spec*) *ist wohlfundiert, wenn dies auch für* G *gilt.*

[25] Man beachte, daß der spezialisierte Lösungsalgorithmus zwar partiell korrekt ist, aber – mangels Wohlfundiertheit der Globalsuchtheorie gs_seq_set($\mathbb{Z}$) – nicht terminiert, solange keine Filter zur Beschneidung des Suchraums verwendet werden.

Beweis: Siehe Übung 10.3.-a/b □

In den Arbeiten, in denen der systematische Entwurf von Globalsuchalgorithmen erstmals untersucht wurden [Smith, 1987b, Smith, 1991a], ist zugunsten iner einfachere Beschreibung von Satz 5.5.20 der Begriff der Generalisierung aus Definition 5.5.10 auf Globalsuchtheorien wie folgt fortgesetzt worden.

G generalisiert spec, falls spec $\ll$ $\texttt{spec}_G$ *gilt.*

G generalisiert spec mit θ, falls gilt $\texttt{R}\subseteq\texttt{R'}$ *und* $\forall \texttt{x:D. I(x)} \Rightarrow \texttt{I'}(\theta\texttt{(x))} \wedge \forall \texttt{z:R. O(x,z)} \Rightarrow \texttt{O'}(\theta\texttt{(x),z)}$

Mit diesen Begriffen läßt sich der Satz 5.5.20 dann wie folgt verkürzt ausdrücken

Generalisiert eine Globalsuchtheorie G eine Spezifikation spec mit θ, so ist G_θ*(spec) eine Globalsuchtheorie.*

Zusammen mit Satz 5.5.16 beschreibt dieser Satz die Grundlage einer mächtigen Strategie zur Synthese von Globalsuchalgorithmen aus formalen Spezifikationen. Man muß nur eine vorgefertigte Globalsuchtheorie der Wissensbank finden, welche die gegebene Spezifikation generalisiert, diese mit der aus dem Generalisierungsbeweis extrahierten Transformation spezialisieren und dann den schematischen Algorithmus instantiieren.

In manchen Fällen reicht diese Strategie jedoch nicht aus, da die gewählte Globalsuchtheorie – wie zum Beispiel gs_seq_set(α) – nicht notwendigerweise wohlfundiert sein muß und somit nicht zu terminierenden Algorithmen führt. In diesen Fällen kann die Terminierung durch Hinzunahme eines *Filters* Φ garantiert werden, welcher die Menge der zu betrachtenden Deskriptoren nach jeder split-Operation so einschränkt, daß jeder Deskriptor im Endeffekt nur endlich oft aufgespalten werden kann. Natürlich darf dieser Filter keine Deskriptoren der auf das Problem spezialisierten Globalsuchtheorie eliminieren, die noch mögliche Lösungen enthalten. Er muß also eine *notwendige* Bedingung für das Vorhandensein von Lösungen in einem Deskriptor beschreiben.

Für Globalsuchtheorien wie gs_seq_set(α) werden wir eine Reihe von Filtern angeben, welche die split-Operation in eine wohlfundierte Operation $\texttt{split}_\Phi$ verwandeln. Diese im allgemeinen jedoch zu stark, um notwendig für die allgemeine Globalsuchtheorie zu sein. Erst nach der Spezialisierung der Theorie auf eine gegebene Spezifikation, die üblicherweise mit einer deutlichen Verstärkung der Bedingung an die Lösungen verbunden ist, können manche Wohlfundiertheitsfilter auch zu notwendigen Filtern geworden sein.

Definition 5.5.21 (Filter für Globalsuchtheorien)

Es sei G=(D, R, I, O, S, J, s_0, sat, split, ext) *eine Globalsuchtheorie und* Φ: D×S → $\mathbb{B}$

1. *Φ ist notwendig für G, falls für alle* x∈D *und* s∈S *gilt* $\Phi\texttt{(x,s)} \Leftarrow \exists \texttt{z:R. sat(z,s)} \wedge \texttt{O(x,z)}$

2. *Φ ist ein Wohlfundiertheitsfilter für G, falls für alle* x∈D *und* s∈S *mit* I(x) *und* J(x,s) *eine Zahl* k∈$\mathbb{N}$ *existiert, so daß* $\texttt{split}_\Phi{}^{\texttt{k}}\texttt{(x,s)} = \emptyset$ *ist.*

   Dabei ist $\texttt{split}_\Phi$ *definiert durch* $\texttt{split}_\Phi\texttt{(x,s)} \equiv \{\texttt{t} \mid \texttt{t}\in\texttt{split(x,s)} \wedge \Phi\texttt{(x,t)}\}$

Offensichtlich ist ein notwendiger Filter Φ genau dann ein Wohlfundiertheitsfilter für die Globalsuchtheorie G, wenn G_Φ=(D, R, I, O, S, J, s_0, sat, $\texttt{split}_\Phi$, ext) eine wohlfundierte Globalsuchtheorie ist. Die Eigenschaft "notwendig" stellt dabei die Gultigkeit des vierten Axioms sicher (alle Lösungen sind im Endeffekt extrahierbar) während die Wohlfundiertheit genau dem fünften Axiom entspricht. Wohlfundiertheitsfilter müssen im allgemeinen separat von der Globalsuchtheorie betrachtet werden, da eine allgemeine Suchstruktur durch verschiedene Einschränkungen wohlfundiert gemacht werden kann, ohne daß sich hierdurch an dem eigentlichen Suchverfahren etwas ändert. Diese Einschränkungen sind jedoch nur selten notwendig für die allgemeine Globalsuchtheorie.

Da es für jede allgemeine Suchstruktur normalerweise nur sehr wenige Arten von Wohlfundiertheitsfiltern gibt und der Nachweis der Wohlfundiertheit meist einen komplexen Induktionsbeweis verlangt, der von einem Inferenzsystem kaum automatisch geführt werden kann, empfiehlt es sich, für jede nicht wohlfundierte Globalsuchtheorie eine Reihe vorgefertigter Wohlfundiertheitsfilter in der Wissensbank abzulegen. Wir wollen hierfür einige Beispiele geben.

Beispiel 5.5.22 (Wohlfundiertheitsfilter für gs_seq_set(α))

Die folgenden drei Filter sind Wohlfundiertheitsfilter für die Globalsuchtheorie gs_seq_set(α).

1. Φ_1(S,V) = |V|≤k: Begrenzung der Suche auf Folgen einer festen maximalen Länge k.

   Dieser Filter ist relativ leicht zu überprüfen, da nur die Länge eines Deskriptors überprüft werden muß, wird aber nur in den seltensten Fällen notwendig sein. Algorithmen mit diesem Filter terminieren (spätestens) nach k Schritten und erzeugen einen Suchbaum der Größenordnung $|S|^k$.

2. Φ_2(S,V) = |V|≤k*|S|: Begrenzung der Suche auf Folgen einer maximalen Länge, die von der Größe der Menge S abhängt.

   Auch dieser Filter, der für eine relativ große Anzahl von Problemen als notwendig nachgewiesen werden kann, ist leicht zu überprüfen. Algorithmen mit diesem Filter terminieren (spätestens) nach k*|S| Schritten und erzeugen einen Suchbaum der Größenordnung $|S|^{k*|S|}$.

3. Φ_3(S,V) = nodups(V): Begrenzung der Suche auf Folgen ohne doppelt vorkommende Elemente.

   Der Test ist etwas aufwendiger, wenn er nicht inkrementell durchgeführt wird. Algorithmen mit diesem Filter terminieren (spätestens) nach k Schritten und erzeugen aufgrund der immer stärker werdenen Einschränkungen an die noch möglichen Deskriptoren einen Suchbaum der Größenordnung |S| !.

Wohlfundiertheitsfilter werden im Rahmen eines Syntheseprozesses passend zu der Globalsuchtheorie ausgewählt, welche die Problemspezifikation generalisiert. Dementsprechend müssen sie zusammen mit dieser auch spezialisiert werden. Das folgende Lemma zeigt, daß die Wohlfundiertheit bei einer Spezialisierung erhalten bleibt und somit zur Laufzeit der Synthese nicht mehr überprüft werden muß.

Lemma 5.5.23 (Spezialisierung von Filtern)

Es sei spec eine Spezifikation, G eine Globalsuchtheorie und Φ ein Wohlfundiertheitsfilter für G. G generalisiere spec mit θ.

Dann ist Φ_θ=λx,s. Φ(θ(x),s) *ein Wohlfundiertheitsfilter für* G_θ(spec)

Während die Wohlfundiertheit eines Filters bei dem Aufbau der Wissensbank eines Programmsynthesesystems ein für alle Mal überprüft werden ann, muß die Notwendigkeit des spezialisierten Filters zur Laufzeit einer Synthese überprüft werden. Dies ist jedoch wesentlich einfacher, da hierzu nur ein zielgerichtetes Auffalten von Definitionen und Anwenden von Lemmata erforderlich ist.

Beispiel 5.5.24 (Wohlfundiertheitsfilter für das Costas-Arrays Problem)

In Beispiel 5.5.19 hatten wir die Globalsuchtheorie gs_seq_set($\mathbb{Z}$) mit Hilfe von θ = λn. {1..} zu einer Globalsuchtheorie G' für das Costas-Arrays Problem spezialisiert.

Wir wollen nun prüfen, welcher der Filter aus Beispiel 5.5.22 zu einem notwendigen Wohlfundiertheitsfilter für G' spezialisiert werden kann. Wir suchen also einen Filter Φ, für den gezeigt werden kann

perm(p,{1..n}) ∧ ∀j ∈domain(p).nodups(dtrow(p,j)), ∧ V⊑ p ⇒ Φ({1..n},V)

wobei n∈$\mathbb{Z}$ und V∈Seq($\mathbb{Z}$) beliebig sind. Hierzu könnte man alle vorgegebenen Filter einzeln durchtesten. Es lohnt sich jedoch auch, durch Auffalten von Definitionen und Anwenden von Lemmata nach *leicht abzuleitenden* Folgerungen von

perm(p,{1..n}) ∧ ∀j ∈domain(p).nodups(dtrow(p,j)), ∧ V⊑ p

zu suchen, in denen nur n und V vorkommen, und dann einen Filter zu wählen, der in den Folgerungen enthalten ist. Wir falten daher die Definition von perm(p,{1..n}) auf und erhalten wie zuvor

perm(p,{1..n}) ⇔ range(p) = {1..n} ∧ nodups(p)

woraus wiederum durch Anwendung des Gesetzes über Präfixfolgen und nodups (Lemma B.2.24.3, die Wahl ist eindeutig!) folgt

$\texttt{perm(p,\{1..n\})} \wedge \texttt{V} \sqsubseteq \texttt{p} \Rightarrow \texttt{nodups(V)}$.

Damit ist $\Phi_{3,\theta} = \lambda\texttt{n,V. nodups(V)}$ offensichtlich ein notwendiger Wohlfundiertheitsfilter für G'.

Aus der Ausgabebedingung für das Costas-Arrays Problem lassen sich zusammen mit dem Erfüllbarkeitsprädikat auch noch weitere notwenduge Filter ableiten. Diese können dazu benutzt werden, den soeben gefundenen notwendigen Wohlfundiertheitsfilter weiter zu verfeinern. Die Lemma über Präfixfolgen, begrenzte All-Quantoren, `domain`, `dtrow` und `nodups` führen schließlich zu der Folgerung

$\forall\texttt{j}\in\texttt{domain(p).nodups(dtrow(p,j))}, \wedge \texttt{V} \sqsubseteq \texttt{p} \Rightarrow \forall\texttt{j}\in\texttt{domain(V).nodups(dtrow(V,j))}$

so daß wir $\Psi = \lambda\texttt{n.V.}\ \forall\texttt{j}\in\texttt{domain(V).nodups(dtrow(V,j))}$ als zusätzlichen notwendigen Filter verwenden können.

Die Verfeinerung eines einmal gefundenen notwendigen Wohlfundiertheitsfilter Φ_θ für $G_\theta(spec)$ durch einen zusätzlichen notwendigen Filter Ψ für $G_\theta(spec)$ ist für die Erzeugung eines korrekten Globalsuchalgorithmus nicht unbedingt erforderlich, führt im allgemeinen jedoch zu einer nicht unerheblichen Effizienzsteigerung. Aus diesem Grunde beschränkt man sich auf zusätzliche Filter, die *leicht* aus der Bedingung $\texttt{sat(z,s)} \wedge \texttt{O(x,z)}$ abgeleitet werden können, wobei als Heuristik eine Beschränkung auf eine kleine Anzahl anzuwendender Lemmata sinnvoll erscheint. Es ist nicht schwer zu zeigen, die Kombination der beiden Filter zu einem neuen Filter $\underline{\Xi \equiv \lambda\texttt{x,s.}\Phi_\theta\texttt{(x,s)} \wedge \Psi\texttt{(x,s)}}$ einen notwendigen Wohlfundiertheitsfilter für $G_\theta(spec)$ liefert.

Faßt man nun die bisherigen Erkenntnisse (Satz 5.5.16, Satz 5.5.20, Lemma 5.5.23 und obige Anmerkung) zusammen, so ergibt sich die folgende effiziente Strategie, die in der Lage ist, eine Synthese von Globalsuchalgorithmen aus vorgegebenen formalen Spezifikationen mit Hilfe von einigen wenigen Zugriffen auf vorgefertigte Objekte der Wissensbank und verhältnismäßig simplen logischen Schlußfolgerungen erfolgreich durchzuführen.

Strategie 5.5.25 (Wissensbasierte Synthese von Globalsuchalgorithmen)

Gegeben sei eine Problemspezifikation $\texttt{FUNCTION F(x:}D\texttt{):}R\ \texttt{WHERE}\ I\texttt{(x) RETURNS \{z |}\ O\texttt{(x,z)\}}$

1. *Wähle eine Globalsuchtheorie G mit Ausgabetyp R (oder einer Obermenge) aus der Wissensbank.*

2. *Beweise, daß G die Spezifikation spec* $= (D,R,I,O)$ *generalisiert.*

   Extrahiere aus dem Beweis eine Transformation θ und bestimme die Globalsuchtheorie $G_\theta(spec)$.

3. *Wenn G nicht bereits wohlfundiert war, wähle einen Wohlfundiertheitsfilter Φ für G aus der Wissensbank, für den gezeigt werden kann, daß* Φ_θ *notwendig für* $G_\theta(spec)$ *ist.*

4. *Bestimme heuristisch einen zusätzlichen notwendigen Filter Ψ für* $G_\theta(spec)$.

5. *Instantiiere den folgenden schematischen Globalsuchalgorithmus*

   $\texttt{FUNCTION F(x:}D\texttt{):Set(}R\texttt{)}\ \ \texttt{WHERE}\ I\texttt{(x)}\ \ \texttt{RETURNS \{z |}\ O\texttt{(x,z)\}}$
   $= \texttt{if}\ \Phi(\theta\texttt{(x)},\texttt{s}_0(\theta\texttt{(x)})) \wedge \Psi\texttt{(x,s}_0(\theta\texttt{(x)}))\ \ \texttt{then F}_{gs}\texttt{(x,s}_0(\theta\texttt{(x)}))\ \ \texttt{else}\ \emptyset$

   $\texttt{FUNCTION F}_{gs}\texttt{(x,s:}D\times\texttt{S):Set(}R\texttt{)}\ \ \texttt{WHERE}\ I\texttt{(x)} \wedge \texttt{J(x,s)} \wedge \Phi(\theta\texttt{(x)},\texttt{s}) \wedge \Psi\texttt{(x,s)}$
   $\texttt{RETURNS \{z |}\ O\texttt{(x,z)} \wedge \texttt{sat(z,s)\}}$
   $= \texttt{\{z | z}\in\texttt{ext(s)} \wedge \texttt{O(x,z)\}} \cup \bigcup\texttt{\{F}_{gs}\texttt{(x,t) | t}\in\texttt{split(}\theta\texttt{(x),s)} \wedge \Phi(\theta\texttt{(x),t)} \wedge \Psi\texttt{(x,t)\}}$

   Das resultierende Programmpaar ist garantiert korrekt.

Wir wollen diese Strategie am Beispiel des Costas-Arrays Problems illustrieren

Beispiel 5.5.26 (Costas Arrays Synthese)

Das Costas Arrays Problem ist die Aufgabe, bei Eingabe einer natürlichen Zahl `n` alle sogenannten Costas-Arrays der Ordnung `n` zu berechnen. Dabei ist ein Costas Array der Größe `n` eine Permutation der Zahlen zwischen 1 und `n`, die in keiner Zeile ihrer Differenzentafel doppelt vorkommende Elemente besitzt. In Beispiel 5.2.4 auf Seite 228 hatten wir folgende formale Spezifikation des Problems aufgestellt.

$\texttt{FUNCTION Costas (n:}\mathbb{Z}\texttt{):Set(}\underline{\texttt{Seq(}\mathbb{Z}\texttt{)}}\texttt{)}\ \ \texttt{WHERE}\ \underline{\texttt{n}\geq\texttt{1}}$
$\texttt{RETURNS \{p |}\ \underline{\texttt{perm(p,\{1..n\})} \wedge \forall\texttt{j}\in\texttt{domain(p).nodups(dtrow(p,j))}}\texttt{\}}$

1. Der Ausgabetyp des Problems ist Seq(ℤ). Wir wählen daher eine Globalsuchtheorie für Folgen ganzer Zahlen, wobei im wesentlichen nur G = gs_seq_set(ℤ) aus Beispiel 5.5.18 (Seite 261) in Frage kommt.

2. Der Beweis dafür, daß G die Spezifikation *spec* des Costas-Arrays Problems generalisiert, ist in Beispiel 5.5.19 auf Seite 262 bereits ausführlich besprochen worden. Er liefert die Transformation θ = λn.{1..n} und führt zu der folgenden spezialisierten Globalsuchtheorie

```
Gθ(spec) ≡  D      ↦      Set(ℤ)
            R      ↦      Seq(ℤ)
            I      ↦    λn.n≥1
            O      ↦  λn,p.perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))
            S      ↦      Seq(ℤ)
            J      ↦  λn,V.range(V)⊆{1..n}
            s0     ↦    λn.[]
            sat    ↦  λp,V.V⊑p
            split  ↦  λn,V.{V·i|i∈{1..n}}
            ext    ↦    λV.{V}
```

3. Da G nicht wohlfundiert ist, wählen wir nun einen Wohlfundiertheitsfilter Φ für G aus der Wissensbank und versuchen zu zeigen, daß Φ_θ notwendig für $G_\theta(spec)$ ist.

   In Beispiel 5.5.22 haben wir drei mögliche Wohlfundiertheitsfilter für G angegeben. Von diesen läßt sich – wie in Beispiel 5.5.24 auf Seite 265 gezeigt – der Filter Φ_3 = λS,V.nodups(V) am schnellsten[26] als geeignet für die Theorie $G_\theta(spec)$ nachweisen.

4. Ebenso haben wir in Beispiel 5.5.22 den Filter Ψ = λn.V.∀j∈domain(V).nodups(dtrow(V,j)) als zusätzlichen notwendigen Filter für $G_\theta(spec)$ abgeleitet.

5. Die Instantiierung des Standard-Globalsuchalgorithmus mit den Parametern $G_\theta(spec)$, Φ_3 und Ψ liefert das folgende korrekte Programm zur Bestimmung aller Costas-Arrays der Ordnung n

```
FUNCTION Costas (n:ℤ):Set(Seq(ℤ))   WHERE n≥1
  RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
= if nodups([]) ∧ ∀j∈domain([]).nodups(dtrow([],j)) then Costas_gs(n,[]) else ∅

FUNCTION Costas_gs (n:ℤ,V:Seq(ℤ)):Set(Seq(ℤ))
  WHERE n≥1 ∧ range(V)⊆{1..n} ∧ nodups(V) ∧ ∀j∈domain(V).nodups(dtrow(V,j))
  RETURNS {p | perm(p,{1..n}) ∧ V⊑p ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
=     {p | p∈{V} ∧ perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
  ∪ ⋃{ Costas_gs(n,W) | W∈{V·i|i∈{1..n}} ∧ nodups(W) ∧ ∀j∈domain(W).nodups(dtrow(W,j))}
```

Natürlich kann man den so entstandenen Algorithmus noch in vielerlei Hinsicht vereinfachen und optimieren. So ist z.B. in der Hauptfunktion der Filter immer gültig und in der Hilfsfunktion wird manches zu kompliziert berechnet. Diese Optimierungen kann man jedoch besser im Nachhinein ausführen, da es bei der schematischen Synthese zunächst einmal darum geht, einen Algorithmus mit einer effizienten Berechnungsstruktur zu zu entwerfen, ohne auf alle optimierbaren Details einzugehen. Die Möglichkeiten einer nachträglichen Optimierung des in diesem Beispiel erzeugten Algorithmus werden wir in Abschnitt 5.6 ausführlicher betrachten.

5.5.3 Einbettung von Entwurfsstrategien in ein formales Fundament

Im vorghergehenden Abschnitt haben wir die theoretischen Grundlagen einer Strategie zur wissensbasierten Synthese von Globalsuchalgorithmen aus formalen Spezifikationen ausführlich untersucht und auf dieser eine sehr effiziente Entwurfsstrategie aufgestellt. Die bisherigen Untersuchungen waren allerdings losgelöst von

[26] Φ_2 wäre genauso gut geeignet, läßt sich aber schwerer überprüfen.

einem formalen logischen Fundament durchgeführt worden und stellen nicht sicher, daß eine *Implementierung* der Strategie tatsächlich nur korrekte Algorithmen generieren kann. Das Problem ist hierbei, daß eine "von-Hand" Codierung einer Strategie immer die Gefahr von Unterlassungsfehlern in sich birgt, die während des Syntheseprozesses nicht mehr auffallen, wenn sich diese Implementierung nicht auf Inferenzregeln eines formalen logischen Kalküls stützt. Daher ist es notwendig, Synthesestrategien in einen logischen Formalismus einzubetten, der die Korrektheit aller ausgeführten Syntheseschritte sicherstellt.

Wie können wir nun eine Entwurfsstrategie wie Strategie 5.5.25 in einen universellen logischen Formalismus wie den der Typentheorie integrieren, ohne dabei die Eleganz und Effizienz der informal beschriebenen Strategie zu verlieren? Wie können wir eine mit natürlichsprachigen Mitteln beschriebene Strategie in einer erheblich formaleren Sprache ausdrücken und dabei genauso natürlich und flexibel bleiben wie zuvor?

Für Strategien auf der Basis algorithmischer Theorien läßt sich diese Frage zum Glück relativ leicht beantworten, da hier auf der semi-formalen Ebene bereits eine Reihe von Vorarbeiten geleistet sind, welche die Grundlage des eigentlichen Syntheseverfahrens bilden. Es ist daher möglich, die entsprechenden Konzepte – wie zum Beispiel Globalsuchtheorien, Generalisierung und Filter – vollständig in der Typentheorie zu formalisieren und die Sätze über die Korrektheit schematischer Algorithmen als formale Theoreme der Typentheorie[27] zu beweisen. Ebenso kann man eine Wissensbank von vorgefertigten Teillösungen – zum Beispiel also konkrete Globalsuchtheorien und Wohlfundiertheitsfilter – dadurch aufstellen, daß man Lemmata über diese konkreten Objekte beweist und in der Library von NuPRL für eine spätere Verwendung während einer Synthese abspeichert. Mit Hilfe einer einfachen Taktik zur automatischen Anwendung von Theoremen kann man dann die Strategie innerhalb des formalen Fundaments (also z.B. in NuPRL) genauso elegant und effizient ablaufen lassen wie ihr informal entworfenes Gegenstück.

Gewonnen werden auf diese Art nicht nur eine Garantie für die Korrektheit aller Schritte sondern auch tiefere Einsichten über die von der Strategie beim Entwurf genutzten Zusammenhänge. Dies wollen wir im folgenden am Beispiel der im vorigen Abschnitt präsentierten Globalsuch-Strategie demonstrieren. Wir beginnen dazu mit einer Formalisierung aller relevanten Grundbegriffe, die bei der Synthese von Globalsuchalgorithmen eine Rolle spielen. Aufgrund der semi-formalen Definitionen des vorigen Abschnitts sind die meisten Definitionen sehr naheliegend und müssen nur präzise aufgeschrieben werden.

Definition 5.5.27 (Formalisierung der Grundkonzepte von Globalsuchalgorithmen)

$$\underline{\mathtt{GS}} \equiv (\mathtt{D{:}TYPES} \times \mathtt{R{:}TYPES} \times \mathtt{I{:}D}{\to}\mathbb{B} \times \mathtt{O{:}\,D}{\times}\mathtt{R}\to\mathbb{B}) \times \mathtt{S{:}TYPES} \times \mathtt{J{:}\,D}{\times}\mathtt{S}\to\mathbb{B} \times \mathtt{s_0{:}D}{\to}\mathtt{S} \times \mathtt{sat{:}\,R}{\times}\mathtt{S}\to\mathbb{B} \times \mathtt{split{:}\,D}{\times}\mathtt{S}\to\mathtt{Set(S)} \times \mathtt{ext{:}S}{\to}\mathtt{Set(R)}$$

$$\underline{split^k} \equiv \mathtt{natind}(k;\ \lambda\mathtt{x,s}.\{\mathtt{s}\};\ \mathtt{n,sp_n}.\ \lambda\mathtt{x,s}.\bigcup\{\mathtt{sp_n(x,t)} \mid \mathtt{t} \in split\mathtt{(x,s)}\})$$

$$\underline{G\ \mathtt{is\ a\ GS\text{-}theory}} \equiv \mathtt{let\ ((D,R,I,O),\ S,\ J,\ s_0,\ sat,\ split,\ ext)} = G\ \mathtt{in}$$
$$\forall\mathtt{x{:}D.\ I(x)} \Rightarrow \mathtt{s_0(x)\ hält} \wedge \mathtt{J(x,s_0(x))}$$
$$\wedge\ \forall\mathtt{x{:}D}.\forall\mathtt{s{:}S.\ I(x)} \wedge \mathtt{J(x,s)} \Rightarrow \forall\mathtt{t} \in \mathtt{split(x,s).J(x,t)}$$
$$\wedge\ \forall\mathtt{x{:}D}.\forall\mathtt{z{:}R.\ I(x)} \wedge \mathtt{O(x,z)} \Rightarrow \mathtt{sat(z,s_0(x))}$$
$$\wedge\ \forall\mathtt{x{:}D}.\forall\mathtt{s{:}S.\ I(x)} \wedge \mathtt{J(x,s)} \Rightarrow \forall\mathtt{z{:}R.O(x,z)} \Rightarrow \mathtt{sat(z,s)} \Leftrightarrow \exists\mathtt{k}{:}\mathbb{N}.\exists\mathtt{t} \in \mathtt{split^k(x,s).\ z} \in \mathtt{ext(t)}$$

$$\underline{\mathtt{Filters}(G)} \equiv \mathtt{let...}{=}G\ \mathtt{in\ D}{\times}\mathtt{S}{\to}\mathbb{B}$$

$$\underline{split_\Phi} \equiv \lambda\mathtt{x,s}.\ \{\ \mathtt{t} \mid \mathtt{t} \in split\mathtt{(x,s)} \wedge \Phi\mathtt{(x,t)}\ \}$$

$$\underline{\Phi\ \mathtt{necessary\ for}\ G} \equiv \mathtt{let...}{=}G\ \mathtt{in}\ \forall\mathtt{x{:}D}.\forall\mathtt{s{:}S}.\ \exists\mathtt{z{:}R.\ sat(z,s)} \wedge \mathtt{O(x,z)} \Rightarrow \Phi\mathtt{(x,s)}$$

$$\underline{\Phi\ \mathtt{wf\text{-}filter\ for}\ G} \equiv \mathtt{let...}{=}G\ \mathtt{in}\ \forall\mathtt{x{:}D}.\forall\mathtt{s{:}S.\ I(x)} \wedge \mathtt{J(x,s)} \Rightarrow \exists\mathtt{k}{:}\mathbb{N}.\ \mathtt{split_\Phi{}^k(x,s)} = \emptyset$$

$$\underline{G\ \mathtt{generalizes}\ spec\ \mathtt{with}\ \theta}$$
$$\equiv \mathtt{let...}{=}G\ \mathtt{and\ (D',R',I',O')}{=}spec\ \mathtt{in}$$
$$\mathtt{R'}{\subset}\mathtt{R} \wedge \forall\mathtt{x{:}D'.\ I'(x)} \Rightarrow \mathtt{I(}\theta\mathtt{(x))} \wedge \forall\mathtt{z{:}R'.\ O'(x,z)} \Rightarrow \mathtt{O(}\theta\mathtt{(x),z)}$$

$$\underline{\Phi_\theta} \equiv \lambda\mathtt{x,s}.\Phi(\theta\mathtt{(x),s)}$$

$$\underline{G_\theta(spec)} \equiv \mathtt{let...}{=}G\ \mathtt{in}\ (\ \mathtt{spec,\ S},\ \lambda\mathtt{x,s.J(}\theta\mathtt{(x),s)},\ \lambda\mathtt{x.s_0(}\theta\mathtt{(x))},\ \mathtt{sat},\ \lambda\mathtt{x,s.split(}\theta\mathtt{(x),s)},\ \mathtt{ext}\ )$$

[27] Hier erweist es sich als vorteilhaft, daß die Typentheorie ein Kalkül höherer Ordnung ist. Man kann daher ohne Bedenken Theoreme formulieren, in denen über Spezifikationen, Programme Prädikate, Globalsuchtheorien etc. quantifiziert wird.

Mit diesen formalisierten Begriffen kann man nun die Sätze und Lemmata des vorigen Abschnittes als formale Sätze der Typentheorie beweisen und dabei in den Beweisen nahezu das gleiche Abstraktionsniveau erreichen, wie dies bei den weniger formalen Gegenstücken der Fall ist. Diese Sätze und Lemmata lassen sich dann zu einem einzigen Theorem zusammenfassen, welches besagt, was genau die Voraussetzungen für die Erfüllbarkeit einer mengenwertigen Problemspezifikation sind. Dieses Theorem wird dann die Grundlage einer effizienten und verifizierten Realisierung der Strategie 5.5.25 innerhalb eines Beweissystems für einen logisch-formalen Kalkül[28] werden.

Satz 5.5.28 (Entwurf von Globalsuchalgorithmen: formale Voraussetzungen)

```
∀spec:SPECS. let (D,R,I,O) = spec
in
   FUNCTION F(x:D):Set(R) WHERE I(x) RETURNS {z | O(x,z)}  ist erfüllbar
   ⇐
       ∃G:GS. G is a GS-Theory
     ∧ ∃θ:D→D_G. G generalizes spec with θ
     ∧ ∃Φ:Filters(G). Φ wf-filter for G ∧ Φ_θ necessary for G_θ(spec)
     ∧ ∃Ψ:Filters(G_θ(spec)). Ψ necessary for G_θ(spec)
```

Beweis: Der formale Korrektheitsbeweis entspricht in seiner Struktur im wesentlichen dem Beweis von Satz 5.5.16, ist aber natürlich etwas aufwendiger, da hier keinerlei gedanklichen Abkürzungen zulässig sind. Auf dem Papier würde er mehr als 10 Seiten einnehmen, weil auch alle Zwischenlemmata über Generalisierung und Filter beweisen werden müssen (Details kann man in [Kreitz, 1992][Anhang C.5] finden). Wir beschränken uns hier daher auf eine grobe Beweisskizze.

Gemäß den Voraussetzungen (der rechten Seite von ⇐) haben wir als Hypothesen:
- G:GS mit G = ((D',R',I',O'), S, J, s_0, sat, split, ext) und G is a GS-Theory
- θ:D→D' mit G generalizes spec with θ
- Φ: D'×S →$\mathbb{B}$ mit Φ wf-filter for G und Φ_θ necessary for G_θ(spec)
- Ψ: D×S →$\mathbb{B}$ mit Ψ necessary for G_θ(spec)

Per Definition ist Erfüllbarkeit von Spezifikationen dasselbe wie die Existenz eines korrekten Lösungsprogramms. Wir geben als Lösung daher den Standard-Globalsuchalgorithmus in geschlossener Form an, wobei wir die Hilfsfunktion per Abstraktion einführen.

$$\begin{array}{lll} \text{letrec } f_{gs}(x,s) & = & \{ z \mid z \in ext(s) \wedge O(x,z) \} \\ & & \cup \bigcup \{ f_{gs}(x,t) \mid t \in split(\theta(x),s) \wedge \Phi(\theta(x),t) \wedge \Psi(x,t) \} \\ \text{in} & & \\ \quad \text{if } \Phi(\theta(x),s_0(\theta(x))) \wedge \Psi(x,s_0(\theta(x))) & \text{then } f_{gs}(x,s_0(\theta(x))) & \text{else } \emptyset \end{array}$$

Für den Nachweis der Korrektheit müssen wir nun alle Schritte des Beweises von Theorem 5.5.16 formal durchführen, wobei wir die Beweise von Satz 5.5.20 und Lemma 5.5.23 ebenfalls integrieren (bzw. separat ausführen) müssen. Bis darauf, daß der Beweis nun vollständig formal ist und nur mit Hilfe von Taktiken übersichtlich strukturiert werden kann, treten keine nennenswerten Unterschiede auf. □

Der Aufwand, der für den Beweis von Satz 5.5.28 getrieben werden muß, ist relativ hoch, wird sich aber bei der automatisierten Synthese von Programmen wieder auszahlen. Dadurch daß wir in der Lösung ein effizientes Algorithmenschema explizit als Extrakt-Term bereitgestellt und den Korrektheitsbeweis *ein für alle Mal* gegeben haben, können wir Globalsuchalgorithmen nun dadurch synthetisieren, daß wir im ersten Schritt den formalen Satz 5.5.28 mit Hilfe der Regeln `cut` und `lemma` einführen und auf das konkrete Programmierproblem anwenden. Als Teilziel bleibt dann nur noch der Nachweis der Existenz von G, θ, Φ und Ψ und ihrer Eigenschaften. Wir können also innerhalb des strengen formalen Inferenzsystems genauso effizient und elegant vorgehen, wie es in der informalen Strategie 5.5.25 beschrieben ist, ohne dabei einen zusätzlichen

[28]Auch wenn die Formalisierungen im wesentlichen auf der intuitionistischen Typentheorie des NuPRL Systems abgestützt werden, ist Theorem 5.5.28 weitestgehend unabhängig von der Typentheorie und könnte in ähnlicher Weise auch in anderen ähnlich mächtigen Systemen eingesetzt werden. Damit ist – zumindest im Prinzip – das Ziel erreicht, mit formalen logischen Kalkülen das natürliche logische Schließen auf verständliche Art widerspiegeln zu können.

Beweisaufwand[29] zu haben. Da zudem die wichtigsten allgemeinen GS-Theorien und ihre wf-Filter bereits als verifizierte Objekte der Wissensbank (d.h. als Lemmata der NuPRL-Bibliothek) vorliegen, wird auch der Nachweis der entstandenen Teilziele nicht mehr sehr schwierig ausfallen. Allein die Generalisierungseigenschaft und die Notwendigkeit von Filtern wird das Inferenzsystem belasten.

Eine Algorithmenentwurfstaktik, die auf der Basis von Satz 5.5.28 operiert, kann daher in wenigen großen Schritten zu einer vorgegebenen Spezifikation einen verifizierbar korrekten Globalsuchalgorithmus erstellen. Die folgende Beschreibung einer solchen Taktik ist das formale Gegenstück zu Strategie 5.5.25 und kann somit als verifizierte Implementierung dieser Strategie angesehen werden.

Man muß zunächst das Problem präzise formulieren und dann ein "Synthesetheorem" aufstellen, was nichts anderes besagt, als daß wir eine Lösung finden wollen. Nun wenden wir als erstes unser Theorem an und erhalten ein neues Unterziel, was sich aus den Voraussetzungen des Theorems ergibt. Konzeptionell ist nun der wichtigste Schritt vollzogen. Nun müssen noch die Parameter bestimmt werden und das kann entweder von Hand geschehen oder mit Unterstützung des Rechners.

Strategie 5.5.29 (Taktik zur formalen Synthese von Globalsuchalgorithmen)

Gegeben sei ein Syntheseproblem, formuliert als Beweisziel

$\vdash$ `FUNCTION F(x:`D`):`R `WHERE` I`(x) RETURNS {z |` O`(x,z)}` ist erfüllbar

1. *Wende das Synthesetheorem 5.5.28 wie eine Inferenzregel an.*

2. *Löse die entstandenen Unterziele – Bestimmung von* `G`*,* θ*,* Φ *und* Ψ *– wie folgt.*

   (a) *Wähle aus der Bibliothek eine GS-theorie* `G` *mit Ausgabetyp* `R` *oder einem Obertyp davon.*
   Beweise "`G is a GS-Theory`*" durch Aufruf des entsprechenden Lemmas.*

   (b) *Bestimme* θ *entweder durch Benutzerinteraktion oder durch einen Extraktion aus einem konstruktivem Beweis von "*`G generalizes (D,R,I,O)`*" (i.w. gezielte Suche nach anwendbaren Lemmata)*

   (c) *Wähle aus der Bibliothek eine Wohlfundiertheitsfilter* Φ *für* `G`*.*
   Beweise "Φ `wf-filter for G`*" durch Aufruf des entsprechenden Lemmas.*
   Beweise "Φ_θ `necessary for` G_θ`(spec)`*" durch gezielte Suche nach anwendbaren Lemmata*

   (d) *Bestimme heuristisch durch Suche nach anwendbaren Lemmata (mit Begrenzung der Suchtiefe) einen zusätzlichen notwendigen Filter* Ψ*, für den* Ψ `necessary for` G_θ`(spec)` *beweisbar ist.*

3. *Extrahiere eine Instanz des Standard-Globalsuchalgorithmus mit dem NuPRL-Extraktionsmechanismus. Da Taktiken nur auf verifizierten Theoreme und den Inferenzregeln der Typentheorie basieren, ist das generierte Programm korrekt.*

Am Beispiel des Costas-Arrays Problems (vergleiche Beispiel 5.5.26) wollen wir nun demonstrieren, wie die Synthese von Globalsuchalgorithmen innerhalb eines allgemeinen Beweisentwicklungssystems wie NuPRL durchgeführt werden kann. Nahezu alle der in Strategie 5.5.29 beschriebenen Teilschritte lassen sich durch den Einsatz von Taktiken automatisieren. Das hohe Abstraktionsniveau der Formalisierungen läßt es jedoch auch zu, die Synthese im wesentlichen interaktiv durchzuführen, wenn der Benutzer – wovon man ausgehen sollte – ein gewisses Verständnis für die zu lösenden Teilaufgaben[30] mitbringt.

[29]Taktiken, welche Globalsuchalgorithmen ohne Verwendung eines formalen Theorems wie Satz 5.5.28 herleiten, sind dagegen gezwungen, die Korrektheit des erzeugten Programms zur Laufzeit nachzuweisen, was in Anbetracht der komplizierten algorithmischen Struktur nahezu undurchführbar und zumindest sehr rechenaufwendig ist.

[30]Es ist sogar durchaus möglich, durch den eher interaktiven Umgang mit dem System ein wenig von der Methodik zu erlernen, die für einen systematischen Entwurf von Algorithmen erforderlich ist.

Es sei allerdings darauf hingewiesen, daß die Beispielsynthese vorläufig noch fiktiven Charakter hat, da die für die Synthese notwendigen Definitionen und Lemmata noch nicht in das NuPRL System eingebettet wurden und der hier beschriebene ZUstand noch nicht erreicht ist.

Beispiel 5.5.30 (Costas Arrays Synthese mit NuPRL)

Wir beginnen mit einer Formulierung des Problems und verwenden hierzu die in Beispiel 5.2.4 auf Seite 228 aufgestellte formale Spezifikation des Costas-Arrays Problems. Innerhalb von NuPRL wird diese Spezifikation zu dem Beweisziel eines Synthese-Theorems, welches mit Hilfe der in Strategie 5.5.29 beschriebenen Schritte beweisen werden soll.

```
⊢ FUNCTION Costas (n:ℤ):Set( Seq(ℤ) )   WHERE n≥1
    RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
  ist erfüllbar
```

Im Topknoten einer formalen Herleitung stehen wir nun vor der Aufgabe, eine Lösung für das Problem zu finden, und müssen hierzu – wie immer – eine Regel oder Taktik angeben. An dieser Stelle könnten wir viele verschiedene Wege einschlagen, entscheiden uns aber dafür, es mit Globalsuchalgorithmen zu versuchen. Daher geben wir eine Taktik an, welche das Theorem 5.5.28 instantiiert. Das System wendet diese Taktik an und hinterläßt als Teilziele die instantiierten Voraussetzungen des Theorems, wobei zugunsten der Lesbarkeit über die Spezifikation abstrahiert und Typinformation unterdrückt wird.

```
# top

 ⊢ FUNCTION Costas(n:ℤ):Set(Seq(ℤ))  WHERE n≥1
     RETURNS {p | perm(p,{1..n}) ∧ ∀j∈dom(p).nodups(dtrow(p,j))}
   ist erfüllbar

 BY  call 'Theorem_5.5.28'

 1. ⊢  ∃G. G is a GS-Theory
     ∧ ∃θ. G generalizes spec with θ
     ∧ ∃Φ. Φ wf-filter for G ∧ Φθ  necessary for Gθ(spec)
     ∧ ∃Ψ. Ψ necessary for Gθ(spec)
     where spec = (ℤ, Seq(ℤ), λn. n≥1,
                   λn,p. perm(p,{1..n}) ∧ ∀j∈dom(p).nodups(dtrow(p,j)) )
```

Aus Sicht eines formalen Inferenzsystems war dies der schwierigste Schritt der Synthese, der aber durch den Einsatz von Theorem 5.5.28 drastisch vereinfacht werden konnte. In den folgenden Schritten müssen nun die Komponenten G, θ, Φ und Ψ bestimmt und ihre Eigenschaften überprüft werden.

Im nächsttieferen Knoten der Herleitung (bezeichnet mit `top 1`) müssen wir als erstes eine Globalsuchtheorie G bestimmen, die das Costas-Arrays Problem generalisiert. Diese muß gemäß der Generalisierungsbedingung eine allgemeine Suchstruktur für endliche Folgen über ganzen Zahlen beschreiben. Zudem muß sie in der Bibliothek bereits vorhanden sein. Die Erfahrung hat gezeigt, daß zur Lösung der meisten Routineprobleme insgesamt etwa 6–10 vorgefertigte Globalsuchtheorien ausreichen, von denen maximal 2 oder 3 eine allgemeine Suchstruktur für endliche Folgen charakterisieren. Unter diesen entscheidet das Zusatzkriterium, daß die Ausgabebedingung schwächer sein muß als die des Costas-Arrays Problems. Sollten dann immer noch mehrere Möglichkeiten bestehen, so gibt es mehrere fundamental verschiedene Suchstrukturen, die in dem zu erzeugenden Algorithmus zum Tragen kommen können. Die Auswahl zwischen diesen ist eine Entwurfs-Entscheidung, die eigentlich nur ein Benutzer fällen darf.

Selbst dann, wenn keinerlei automatische Unterstützung bei der Auswahl der Globalsuchtheorie bereitsteht, wird es für einen Benutzer nicht sehr schwer sein, sich für eine passende Globalsuchtheorie zu entscheiden. In diesem Fall ist dies die Theorie `gs_seq_set(ℤ)`, die wir unter Verwendung der Regel `ex_i` [31] angeben. Nach Verarbeitung dieser Regel verbleiben zwei Teilziele. Zum einen müssen wir zeigen, daß `gs_seq_set(ℤ)` tatsächlich eine Globalsuchtheorie ist und zum zweiten müssen wir nun die anderen Komponenten passend hierzu bestimmen.

[31] Es wäre durchaus möglich, eine summarische Taktik mit Namen `INTRO` bereitzustellen, die anhand der äußeren Struktur der Konklusion bestimmt, daß in diesem Falle die Regel `ex_i` gemeint ist.

```
# top 1

⊢ ∃G. G is a GS-Theory
    ∧ ∃θ. G generalizes spec with θ
    ∧ ∃Φ. Φ wf-filter for G ∧ Φθ  necessary for Gθ(spec)
    ∧ ∃Ψ. Ψ necessary for Gθ(spec)
    where spec = (ℤ, Seq(ℤ),  λn. n≥1,
                  λn,p. perm(p,{1..n}) ∧ ∀j ∈dom(p).nodups(dtrow(p,j)))
BY  ex_i  gs_seq_set(ℤ)  THEN  unfold ‘perm‘
1. ⊢ gs_seq_set(ℤ) is a GS-Theory
2. ⊢ ∃θ. gs_seq_set(ℤ) generalizes spec with θ
     ∧ ∃Φ. Φ wf-filter for gs_seq_set(ℤ)
         ∧  Φθ necessary for gs_seq_set(ℤ)θ(spec)
     ∧ ∃Ψ. Ψ necessary for gs_seq_set(ℤ)θ(spec)
     where spec = (ℤ, Seq(ℤ), λn. n≥1,
                   λn,p. nodups(p) ∧ range(p)={1..n} ∧ ∀j ∈dom(p).nodups(dtrow(p,j)))
```

Das erste Teilziel lösen wir durch Instantiierung des entsprechenden Lemmas der NuPRL-Bibliothek, ohne das die Globalsuchtheorie `gs_seq(ℤ)` nicht für eine Synthese zur Verfügung stehen würde. Für das zweite werden wir die Definition des Begriffes Permutation auflösen müssen, da wir auf die Bestandteile immer wieder zugreifen werden. Der Übersichtlichkeit halber haben wir dies in diesen Schritt integriert, könnten es aber auch erst im nachfolgenden Schritt tun.

In diesem nächsten Schritt müssen wir eine Transformation θ bestimmen, mit der wir die Generalisierungseigenschaft nachweisen können. Eine automatische Unterstützung hierfür ist möglich, da man hierzu nur die in Beispiel 5.5.19 beschriebenen Schritte durch eine Taktik ausführen lassen muß. Bei einer interaktien Synthese kann ein Benutzer jedoch auch ein intuitives Verständnis von der Generalisierungseigenschaft benuzten, welches besagt, daß die Ausgabebedingung von `gs_seq_set(ℤ)` nach Transformation mit θ aus der Ausgabebedingung des Problems folgen muß.

Sieht man sich die entsprechenden Komponenten von `gs_seq_set(ℤ)` genauer an, so stellt man fest, daß θ eine Zahl in eine Menge umwandeln muß. Im Falle von `gs_seq_set(ℤ)` sind die Elemente der aufgezählten endlichen Folgen aus dieser Eingabemenge zu wählen. In der Spezifikation heißt es dagegen, daß die Elemente der Folgen genau die Menge `{1..n}` bilden müssen. Es liegt also nahe, die Zahl `n` in die Menge `{1..n}` zu transformieren. Genau das geben wir als Wert für θ an.

```
# top 1 2

⊢ ∃θ. gs_seq_set(ℤ) generalizes spec with θ
     ∧ ∃Φ. Φ wf-filter for gs_seq_set(ℤ)
          ∧ Φθ necessary for gs_seq_set(ℤ)θ(spec)
     ∧ ∃Ψ. Ψ necessary for gs_seq_set(ℤ)θ(spec)
     where spec = (ℤ, Seq(ℤ), λn. n≥1,
                   λn,p. nodups(p) ∧ range(p)={1..n} ∧ ∀j ∈dom(p).nodups(dtrow(p,j)))
BY  ex_i  λn.{1..n}
1. ⊢ gs_seq_set(ℤ) generalizes  spec with  λn.{1..n}
     where spec = ......
2. ⊢ ∃Φ. Φ wf-filter for gs_seq_set(ℤ)
        ∧ Φθ necessary for gs_seq_set(ℤ)λn.{1..n}(spec)
     ∧ ∃Ψ. Ψ necessary for gs_seq_set(ℤ)λn.{1..n}(spec)
     where spec = ......
```

Von den beiden Teilzielen kann das erste durch eine einfache Beweistaktik gelöst werden, welche die Definition von `generalizes` auffaltet und zielgerichtet nach Lemmata über die vorkommenden Begriffe `range` und `sub` sucht.

Im zweiten Teilziel ist ein Filter Φ zu bestimmen, der die spezialisierte Globalsuchtheorie wohlfundiert machen kann. Da der Nachweis von Wohlfundiertheit üblicherweise einen komplizierten Induktionsbeweis benötigt, darf man diesen auf keinen Fall zur Laufzeit des Syntheseprozesses durchführen, sondern

muß vorgefertigte Filter in der Wissensbank ablegen. Erfahrungsgemäß reichen hierfür 3–4 Filter (siehe Beispiel 5.5.22) aus, um die meisten Routineprobleme zu behandeln. Unter diesen müssen wir einen auswählen, für den wir nachher Φ_θ `necessary for gs_seq_set(`$\mathbb{Z}$`)`$_{\lambda n.\{1..n\}}$`(spec)` beweisen können. Eine automatische Unterstützung hierfür ist möglich, da man hierzu nur die in Beispiel 5.5.24 beschriebenen Schritte durch eine Taktik ausführen lassen muß. Es reicht jedoch auch aus, ein intuitives Verständnis des Begriffs "notwendig" zu verwenden: aus der Ausgabebedingung des Costas Arrays Problems und dem Erfüllbarkeitsprädikat zwischen Ausgabewerten und Deskriptoren muß die Gültigkeit des gewählten Filters folgen.

Da ein Präfix `V` einer Folge `p`, die keine doppelten Elemente hat, selbst wieder keine doppelten Elemente hat, liegt es nahe, den Filter Φ_3`=λS,V.nodups(V)` auszuwählen.

```
# top 1 2 2

⊢ ∃Φ. Φ wf-filter for gs_seq_set(ℤ)
     ∧ Φθ necessary for gs_seq_set(ℤ)λn.{1..n}(spec)
     ∧ ∃Ψ. Ψ necessary for gs_seq_set(ℤ)λn.{1..n}(spec)
     where spec = ( ℤ, Seq(ℤ), λn. n≥1,
                    λn,p. nodups(p) ∧ range(p)={1..n} ∧ ∀j ∈dom(p).nodups(dtrow(p,j)) )
BY  ex_i λS,V.nodups(V)
1. ⊢ λS,V.nodups(V) wf-filter for gs_seq_set(ℤ)
2. ⊢ λn,V.nodups(V) necessary for gs_seq_set(ℤ)λn.{1..n}(spec)
     where spec = ......
3. ⊢ ∃Ψ. Ψ necessary for gs_seq_set(ℤ)λn.{1..n}(spec)
     where spec = ......
```

Von den drei entstandenen Teilzielen folgt das erste wiederum aus einem Lemma der NuPRL-Bibliothek. Für den Beweis des zweiten Ziels lösen wir alle Definitionen auf, um eine einfache Beweistaktik anwendbar zu machen. Diese Beweistaktik – wir haben sie `Prover` genannt – muß nur in der Lage sein, zielgerichtet nach Lemmata über die vorkommenden Begriffe zu suchen und diese schrittweise (wie z.B. in Beispiel 5.5.24) anzuwenden.

```
# top 1 2 2 2

⊢ λn,V.nodups(V) necessary for gs_seq_set(ℤ)λn.{1..n}(spec)
     where spec = ( ℤ, Seq(ℤ), λn. n≥1,
                    λn,p. nodups(p) ∧ range(p)={1..n} ∧ ∀j ∈dom(p).nodups(dtrow(p,j)) )
BY  undo_abstractions THEN unfold ‘specialize‘ THEN unfold ‘necessary‘
1. ⊢ ∀n.∀V. ∃p. nodups(p) ∧ range(p)={1..n} ∧ ∀j ∈dom(p).nodups(dtrow(p,j)) ∧ V⊑p
              ⇒ nodups(V)
     * BY  Prover
```

Damit fehlt uns nur noch der zusätzliche Filter Ψ, der eigentlich nur aus Effizienzgründen benötigt wird, um den Suchraum des entstehenden Algorithmus frühzeitig zu begrenzen. Wie eben lösen wir dazu erst einmal die Definitionen auf und konzentrieren uns darauf, was unmittelbar aus den Ausgabebedingung des Costas Arrays Problems und dem Erfüllbarkeitsprädikat folgt.

Da wir die Eigenschaften `nodups(p)` und `range(p)={1..n}` bereits ausgenutzt haben, müssen wir nun Folgerungen aus der dritten Bedingung ziehen. Auch hier kann man sowohl eine maschinelle Unterstützung (mit Suche nach passenden Lemmata wie in Beispiel 5.5.24) als auch ein intuitives Verständnis des Problems einsetzen. An einer Skizze macht man sich schnell klar, daß jede Zeile in der Differenzentafel einer Folge `V⊑p` ein Präfix der entsprechenden Zeile in der Differenzentafel von `p` ist und dementsprechend auch die Eigenschaft übernimmt, keine doppelten Vorkommen zu haben.

2	4	1	6	5	3	p
-2	3	-5	1	2		Zeile 1
1	-2	-4	3			Zeile 2
-4	-1	-2				Zeile 3
-3	1					Zeile 4
-1						Zeile 5
						Zeile 6

2	4	1	6			V
-2	3	-5				Zeile 1
1	-2					Zeile 2
-4						Zeile 3
						Zeile 4
						Zeile 5
						Zeile 6

Die Deskriptorfolge muß also selbst ein Costas Array einer kleineren Ordnung sein. Wir wählen also Ψ entsprechend und überlassen den Rest des Beweises wieder der Beweisstrategie `Prover`. Diese wird wie im Falle von Φ_θ durch Suche nach geeigneten Lemmata in der Lage sein, die Notwendigkeit von Ψ nachzuweisen – allerdings mit einer deutlich größeren Anzahl von Inferenzen.

```
# top  1 2 2 3

 ⊢ ∃Ψ. Ψ necessary for gs_seq_set(ℤ)λn.{1..n}(spec)
      where spec = ......
 BY  undo_abstractions  THEN unfold ‘specialize‘  THEN unfold ‘necessary‘
 1. ⊢ ∃Ψ.∀n.∀V. ∃p. nodups(p) ∧ range(p)={1..n} ∧ ∀j∈dom(p).nodups(dtrow(p,j)) ∧ V⊑p
                 ⇒  Ψ(n,V)
 * BY  ex_i  λn,V. ∀j∈dom(V).nodups(dtrow(V,j)  THEN Prover
```

Die formale Synthese ist nun beendet: alle Komponenten sind bestimmt und das Synthesetheorem über die Lösbarkeit des Costas Arrays Problems ist bewiesen. Nun kann aus dem Beweis eine Instanz des schematischen Algorithmus extrahiert werde, den wir in Satz 5.5.28 angegeben hatten. Diese beschreibt einen ersten lauffähigen, vor allem aber nachgewiesenermaßen korrekten Algorithmus, denn wir z.B. auch im NuPRL System auswerten könnten.

```
FUNCTION Costas(n:ℤ):Set(Seq(ℤ))  WHERE n≥1
  RETURNS {p | perm(p,{1..n}) ∧ ∀j∈dom(p).nodups(dtrow(p,j))}
=
   letrec Costas_gs(n,V) =
       { p | p∈{V} ∧ perm(p,{1..n}) ∧  ∀j∈dom(p).nodups(dtrow(p,j))}
     ∪⋃{ Costas_gs(n,W) | W∈{V·i|i∈{1..n}}  ∧  nodups(W)  ∧  ∀j∈dom(W).nodups(dtrow(W,j))}
  in
     if nodups([]) ∧ ∀j∈dom([]).nodups(dtrow([],j))
        then Costas_gs(n,[])   else ∅
```

Dieser Algorithmus entspricht im wesentlichen demjenigen, den wir bereits in Beispiel 5.5.26 auf informale Weise hergeleitet hatten.

Das Beispiel zeigt, daß es auch in einem vollständig formalen Rahmen möglich ist, gut strukturierte Algorithmen in wenigen großen Schritten zu entwerfen – selbst dann, wenn das Ausmaß der automatischen Unterstützung eher gering ist und ein Mensch diese Herleitung alleine steuern müsste. Der Vorteil gegenüber den Verfahren, die dasselbe ohne Abstützung auf einen formalen Kalkül erreichen, ist ein größtmögliches Maß an Sicherheit gegenüber logischen Fehlern bei der Erstellung von Software bei gleichzeitiger Erhaltung einer Kooperation zwischen Mensch und Maschine bei der Softwareentwicklung.

5.5.4 Divide & Conquer Algorithmen

Divide & Conquer ist eine der einfachsten und zugleich gebräuchlichsten Programmiertechniken bei der Verarbeitung strukturierter Datentypen wie Felder, Liste, Bäume, etc. Wie Globalsuchalgorithmen lösen Divide & Conquer Algorithmen ein Programmierproblem durch eine rekursive Reduktion der Problemstellung in

unabhängige Teilprobleme, die im Prinzip parallel weiterverarbeitet und dann zu einer Lösung des Gesamtproblems zusammengesetzt werden können. Während bei Globalsuchalgorithmen jedoch jedes dieser Teilprobleme zu einer eigenen Lösung führt ($\vee$-Reduktion, Reduktion des Ausgabebereichs), werden bei Divide & Conquer Algorithmen *alle* Teillösungen benötigt, um eine einzelne Gesamtlösung zusammenzusetzen ($\wedge$-Reduktion, Reduktion des Eingabebereichs).

Die Grundstruktur, die all diese Algorithmen gemeinsam haben, ist eine rekursive *Dekomposition* der Eingaben in "kleinere" Eingabewerte, auf denen der Algorithmus *rekursiv* aufgerufen werden kann, und ggf. weitere Werte, die durch eine Hilfsfunktion weiterverarbeitet werden. Die so erzielten Werte werden dann wieder zu einer Lösung des *Originalproblems* zusammengesetzt. Eingaben, die sich nicht zerlegen lassen, gelten als *primitiv* und müssen *direkt gelöst* werden. Formalisiert man diese Vorgehensweise in einem einheitlichen Algorithmenschema, so ergibt sich folgende Grundstruktur von Divide & Conquer Algorithmen.

```
FUNCTION F(x:D):R   WHERE I(x) RETURNS y  SUCH THAT O(x,y)
= if primitive(x)  then Directly-solve(x)
                   else (Compose ∘ G×F ∘ Decompose)(x)
```

Hierbei treten neben den Bestandteilen der eigentlichen Spezifikation die folgenden Komponenten auf.

- Ein Funktion `Decompose`:`D → D'×D` zur *Dekomposition* von Eingaben in Teilprobleme
- Eine *Hilfsfunktion* `G`:`D'×R'`, die zusammen[32] mit einem rekursiven Aufruf von `F` auf die Dekomposition der Eingabewerte angewandt wird.
- Eine *Kompositions*funktion `Compose`: `R'×R → R`, welche rekursiv erzeugte Teillösungen (und die von `G` erzeugten Zusatzwerte) zu einer Lösung des Gesamtproblemms zusammensetzt.
- Ein *Kontrollprädikat* `primitive` auf `D`, welches Eingaben beschreibt, die sich nicht zerlegen lassen.
- Eine Funktion `Directly-solve`:`D→R`, welche für primitive Eingaben eine *direkte Lösung* berechnet.

Die Vielseitigkeit dieses Schemas entsteht vor allem durch die Hilfsfunktion `G` und ihre Ein- und Ausgabebereiche `D'` und `R'`. Sie kann im einfachsten Fall die Identitätsfunktion sein, welche die bei der Dekomposition erzeugten Hilfswerte – wie das erste Element einer endlichen Folge bei einer First/Rest-Zerlegung – überhaupt nicht weiterverarbeitet. Sie kann identisch mit der Funktion `F` sein, wenn Dekomposition einen Eingabewert in zwei gleichartige Werte zerlegt – wie etwa in linke und rechte Hälfte einer endlichen Folge bei einer `ListSplit`-Zerlegung. Aber auch kompliziertere Zerlegungen und komplexere Hilfsfunktionen sind denkbar.

6 Axiome mussen erfüllt sein, um die Korrektheit des obigen Algorithmenschemas sicherzustellen. Die meisten davon beschreiben die Spezifikationen der vorkommenden Teilfunktionen. Hinzu kommt ein Axiom über rekursive Zerlegbarkeit (*Strong Problem Reduction Principle* – kurz SPRP) und die Wohlfundiertheit der durch die Dekomposition induzierten Ordnung auf den Eingabewerten.

1. Die direkte Lösung ist korrekt für primitive Eingabwerte.
2. Die Ausgabebedingung ist rekursiv zerlegbar in Ausgabebedingungen für die Dekomposition, die Hilfsfunktion und die Komposition (*Strong Problem Reduction Principle*).
3. Die Dekompositionsfunktion erfüllt ihre Ausgabebedingung und verkleinert dabei die Eingabewerte.
4. Die Kompositionsfunktion erfüllt ihre Ausgabebedingung.
5. Die Hilfsfunktion erfüllt ihre Ausgabebedingung.
6. Die durch die Dekomposition induzierte Verkleinerungsrelation ist eine wohlfundierte Ordnung.

[32]Die Schreibweise G×F ist an dieser Stelle eine Abkürzung für `λy',y.(G(y'),F(y))` und f∘g kürzt `λx.f(g(x))` ab.

Präzisiert man die obengenannten Axiome durch logische Formeln, so läßt sich die Korrektheit des obengenannten Schemas für Divide & Conquer Algorithmen relativ leicht nachweisen.

Satz 5.5.31 (Korrektheit von Divide & Conquer Algorithmen)

Es seien spec=(D, R, I, O) *eine beliebige Spezifikation,* D' *und* R' *beliebige Datentypen,* Decompose:D $\rightarrow$ D'×D*,* G:D'×R'*,* Compose: R'×R $\rightarrow$ R*,* primitive:D$\rightarrow\mathbb{B}$ *und* Directly-solve:D$\rightarrow$R*. Das Programmschema*

```
FUNCTION F(x:D):R   WHERE I(x) RETURNS y  SUCH THAT O(x,y)
= if primitive(x)  then Directly-solve(x)  else (Compose ∘ G×F ∘ Decompose)(x)
```

ist korrekt, wenn es Prädikate O_D : D×D'×D $\rightarrow\mathbb{B}$, I':D'$\rightarrow\mathbb{B}$, O': D'×R' $\rightarrow\mathbb{B}$, O_C: R'×R×R $\rightarrow\mathbb{B}$ *und eine Relation* $\succ$: D×D $\rightarrow\mathbb{B}$ *gibt, so daß die folgenden sechs Axiome erfüllt sind*

1. Directly-solve *erfüllt die Spezifikation*
   FUNCTION F_p(x:D):R WHERE I(x) $\wedge$ primitive(x) RETURNS y SUCH THAT O(x,y)
2. *(SPRP:) Für alle* x,y$\in$D, y'$\in$D', z,t$\in$R *und* z'$\in$R' *gilt*
   O_D(x,y',y) $\wedge$ O(y,z) $\wedge$ O'(y',z') $\wedge$ O_C(z,z',t) $\Rightarrow$ O(x,t)
3. Decompose *erfüllt die Spezifikation*
   FUNCTION F_d(x:D):D'×D WHERE I(x) $\wedge$ $\neg$primitive(x)
   RETURNS y',y SUCH THAT I'(y') $\wedge$ I(y) $\wedge$ x$\succ$y $\wedge$ O_D(x,y',y)
4. Compose *erfüllt die Spezifikation*
   FUNCTION F_c(z,z':R×R'):R RETURNS t SUCH THAT O_C(z,z',t)
5. G *erfüllt die Spezifikation*
   FUNCTION F_G(y':D'):R' WHERE I'(y') RETURNS z' SUCH THAT O'(y',z')
6. $\succ$ *ist eine wohlfundierte Ordnung auf* D

Beweis: Es seien D,R,I,O,D',R',Decompose,G,Compose,primitive,Directly-solve sowie O_D, I',O',O_C,$\succ$ wie oben angegeben. Die Axiome 1–6 seien erfüllt und F sei definiert durch

F(x) = if primitive(x) then Directly-solve(x) else (Compose ∘ G×F ∘ Decompose)(x)

Da $\succ$ nach *Axiom 6* wohlfundiert ist, können wir durch strukturelle Induktion über (D,$\succ$)[33] zeigen, daß für alle x$\in$D mit I(x) die Ausgabebedingung O(x, F(x)) erfüllt ist.

Es sei also x$\in$D mit I(x) und es gelte O(y, F(y)) für alle y$\in$D mit x$\succ$y. Wir unterscheiden zwei Fälle.

- x ist 'primitiv'. Dann gilt F(x) = Directly-solve(x).
  Aus *Axiom 1* folgt dann unmittelbar O(x, F(x)).
- Es gilt $\neg$primitive(x). Dann gilt F(x) = (Compose ∘ G×F ∘ Decompose)(x).
  Nach *Axiom 3* berechnet Decompose(x) dann zwei Werte y'$\in$D' und y$\in$D mit den Eigenschaften I'(y'), O_D(x,y',y) und x$\succ$y.
  Für y' ist *Axiom 5* anwendbar und die Berechnung von G liefert einen Wert z'$\in$R' mit O'(y',z')
  Für y ist die *Induktionsannahme* anwendbar und die Berechnung von F liefert einen Wert z$\in$R mit O(y,z).
  Auf die entstandenen Werte z und z' kann die Funktion Compose angewandt werden. Nach Axiom 4 liefert die Berechnung von Compose(z,z') einen Wert t mit O_C(z,z',t).
  Insgesamt gilt für die Werte x,y',y,z',z und t also O_D(x,y',y) $\wedge$ O(y,z) $\wedge$ O'(y',z') $\wedge$ O_C(z,z',t)
  Damit ist Axiom 2 anwendbar und es folgt O(x,t)
  Da t identisch mit (Compose ∘ G×F ∘ Decompose)(x) ist, folgt t=F(x), also O(x,F(x))

Damit ist die Ausgabebedingung O(x, F(x)) für alle legalen Eingaben erfüllt.

Aufgrund des Prinzips der strukturellen Induktion über Wohlordnungen ist die Ausgabebedingung O(x, F(x)) somit für alle legalen Eingaben erfüllt. Hieraus folgt per Definition die Korrektheit der Funktion F. □

[33] Diese Form der Induktion, welche manchmal auf *wohlfundierte Ordnungsinduktion* genannt wird, basiert auf der Erkenntnis, daß eine Eigenschaft $Q[x]$ für alle x$\in$D gilt wenn man zeigen kann, daß $Q[x]$ aus der Annahme $\forall$y:D. x$\succ$y $\Rightarrow Q[y]$ folgt.

Satz 5.5.31 behandelt eigentlich nur den Fall, daß die Hilfsfunktion `G` verschieden von `F` ist. Andernfalls müsste die Eingabebedingung `I'` für `G` die Ordnungsbeziehung `x`$\succ$`y'` enthalten, was formal nicht mehr ganz dem obigen Schema entspricht. Wir wollen dies an dieser Stelle jedoch nicht weiter vertiefen[34] sondern stattdessen einige Entwurfsstrategien ansprechen, welche auf Satz 5.5.31 und auf ähnlichen Sätzen über Divide & Conquer Algorithmen beruhen. Im Gegensatz zu der Strategie zur Entwicklung von Globalsuchalgorithmen enthalten diese Strategien jedoch deutlich größere Freiheitsgrade, die einer heuristischen Steuerung bedürfen, da man auf ein wesentlich geringeres Maß von vorgefertigten Informationen stützen muß. Sie wirken daher nicht ganz so zielgerichtet wie Strategie 5.5.25.

Alle Strategien haben gemeinsam, daß man zunächst mindestens eine der notwendigen Komponenten aus einer Wissensbank auswählt und dann daraus Spezifikationen und Lösungen der jeweils anderen Komponenten bestimmt. Dabei basiert die Herleitung der Spezifikationen meist auf einer Zerlegung des Problems in Teilprobleme mit dem Problemreduktionsprinzip (SPRP), während ihre Lösungen meist eine weitere Verwendung von Teilinformationen aus der Wissensbank oder einen erneuten Syntheseprozeß (der nicht nur auf Divide & Conquer beruht) verlangen. Die Erzeugung *abgeleiteter Vorbedingungen* (wie z.B. des Prädikats `primitive`) spielt hierbei eine wichtige Rolle. Wir wollen exemplarisch eine der Strategien im Detail beschreiben.

Strategie 5.5.32 (Synthese von Divide & Conquer Algorithmen)

Gegeben sei eine Problemspezifikation `FUNCTION F(x:D):R WHERE I(x) RETURNS y SUCH THAT O(x,y)`.

1. *Wähle eine einfache Dekompositionsfunktion* `Decompose` *samt ihrer Spezifikation aus der Wissensbank und hierzu passend eine wohlfundierte Ordnungsrelation* $\succeq$ *auf* `D`.
   Hierdurch wird der Datentyp `D` und das Prädikat O_D ebenfalls festgelegt. Axiom 6 ist erfüllt.

2. *Bestimme die Hilfsfunktion* `G` *und ihre Spezifikation heuristisch. Wähle hierfür die Funktion* `F`*, falls* `D'=D` *gilt und sonst die Identitätsfunktion.*
   Hierdurch werden die Prädikate `O'` und `I'` ebenfalls festgelegt. Axiom 5 ist erfüllt.

3. *Verifiziere die Korrektheit von* `Decompose`*. Leite die zusätzlichen Vorbedingungen an Eingabewerte* `x`$\in$`D` *ab, unter denen die Spezifikation in Axiom 3 von* `Decompose` *erfüllt wird.*
   Hierdurch wird die Negation des Prädikates `primitive` heuristisch festgelegt.

4. *Konstruiere die Ausgabebedingung* O_C *des Kompositionsoperators durch Bestimmung der Vorbedingungen für die Gültigkeit der Formel* O_D`(x,y',y)` $\land$ `O(y,z)` $\land$ `O'(y',z')` $\Rightarrow$ `O(x,t)` *und erfülle so Axiom 2.*
   Synthetisiere eine Funktion `Compose`*, welche Axiom 4 erfüllt.*
   Sollte `Compose` nicht in der Wissensbank zu finden sein, muß das Verfahren rekursiv aufgerufen werden.

5. *Synthetisiere eine Funktion* `Directly-solve`*, welche Axiom 1 erfüllt.*
   Die Funktion muß über einen einfachen Nachweis von $\forall$`x:D. I(x)` $\land$ `primitive(x)` $\Rightarrow$ $\exists$`y:R. O(x,y)` konstruiert werden können. Ist dies nicht möglich, leite eine zusätzliche Vorbedingung für die Gültigkeit dieser Formel ab und starte das Verfahren erneut.

6. *Instantiiere das Divide & Conquer Schema mit den gefundenen Parametern.*
```
FUNCTION F(x:D):R   WHERE I(x) RETURNS y  SUCH THAT O(x,y)
= if primitive(x)  then Directly-solve(x)  else (Compose ∘ G×F ∘ Decompose)(x)
```

Viele dieser Schritte können mit Hilfe einer Technik zur Bestimmung von *Vorbedingungen* für die Gültigkeit einer logischen Formel, die in [Smith, 1985b, Section 3.2] ausführlich beschrieben ist, weitestgehend automatisiert werden. Die Synthese von `Compose` kann einen weiteren Aufruf des Verfahrens erfordern, bei dem eine andere Reihenfolge bei der Bestimmung der Komponenten von Divide & Conquer Algorithmen erfolgreicher sein mag als Strategie 5.5.32. Als alternative Vorgehensweisen werden hierfür in [Smith, 1985b, Section 7] vorgeschlagen.

[34] Ein etwas allgemeineres, allerdings nicht mehr ganz so einfach strukturiertes Schema für Divide & Conquer Algorithmen und den dazugehörigen Korrektheitsbeweis findet man in [Smith, 1982].

- Wähle einen einfachen Kompositionsoperator aus der Wissensbank. Konstruieren dann sukzessive die Hilfsfunktion, die Ordnungsrelation, den Dekompositionsoperator und zum Schluß die direkte Lösung für primitive Eingabewerte.
- Wähle einen einfachen Dekompositionsoperator und die Ordnung $\succ$ aus der Wissensbank. Konstruieren dann sukzessive den Kompositionsoperator, die Hilfsfunktion und zum Schluß die direkte Lösung für primitive Eingabewerte.

Wir wollend die Strategie 5.5.32 am Beispiel der Synthese eines Sortieralgorithmus illustrieren.

Beispiel 5.5.33 (Synthese eines Sortieralgorithmus)

Das Sortierproblem wird durch die folgenden formale Spezifikation beschrieben.

```
FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) RETURNS S  SUCH THAT SORT(L,S)
```

wobei `SORT(L,S)` eine Abkürzung für `rearranges(L,S)` $\wedge$ `ordered(S)` ist.

1. Listen über ganzen Zahlen lassen sich auf mehrere Arten zerlegen. Wir könnten uns für eine `FirstRest`-Zerlegung, für eine `ListSplit`-Zerlegung in zwei (nahezu) gleiche Hälften, oder eine `SplitVal`-Zerlegung in Werte unter- bzw. oberhalb eines Schwellwertes entscheiden. In diesem Falle wählen wir die Funktion

   `ListSplit` $\equiv$ `λL.([L[i]|i∈[1..|L|÷2]], [L[i]|i∈[1+|L|÷2..|L|]])`

   Die zugehörige Ausgabebedingung O_D`(L,L₁,L₂)` lautet `L₁∘L₂=L ∧ |L₁|=|L|÷2 ∧ |L₂|=(1+|L|)÷2`.

   Eine der wichtigsten wohlfundierten Ordnungen auf Listen besteht in einem Vergleich der Längen. Diese Ordnung ist nicht total, aber darauf kommt es ja nicht an. Wir wählen also `λL,L'. |L|>|L'|`.

2. Entsprechend der Heuristik in Strategie 5.5.32 wählen wir die Funktion `sort` selbst als Hilfsfunktion. Damit werden die Ein- und Ausgabebedingungen `I` und `O` entsprechend übernommen.

3. Wir stellen nun die vollständige Spezifikation des Dekompositionsoperators – einschließlich der Vorbedingungen an `sort` und der *beiden* Ordnungsbedingungen $L \succ L_1 \wedge L \succ L_2$ – auf und leiten die Vorbedingungen für die Korrektheit des Programms

   ```
   FUNCTION Fd(L:Seq(ℤ):Seq(ℤ)×Seq(ℤ)  RETURNS L₁,L₂
     SUCH THAT |L|>|L₁| ∧ |L|>|L₂| ∧ L₁∘L₂=L ∧ |L₁|=|L|÷2 ∧ |L₂|=(1+|L|)÷2
   =  ListSplit(L)
   ```

   her. Da `|L|>|L₁|` nur gelten kann, wenn `|L|>0`, also `L≠[]` ist, erhalten wir als Beschreibung primitiver Eingaben das Prädikat `λL. L=[]`.

4. Wir leiten nun die Vorbedingungen für die Gültigkeit von

   `L₁∘L₂=L ∧ |L₁|=|L|÷2 ∧ |L₂|=(1+|L|)÷2 ∧ SORT(L₁,S₁), ∧ SORT(L₂,S₂)  ⇒  SORT(L,S)`

   ab. Dies liefert als Kompositionsbedingung `ordered(S) ∧ range(S)=range(S₁)∪range(S₂)`

   Die Synthese der Kompositionsfunktion

   ```
   FUNCTION Fc(S₁,S₂:Seq(ℤ)×Seq(ℤ)):Seq(ℤ)  WHERE ordered(S₁) ∧ordered(S₂)
     RETURNS S  SUCH THAT ordered(S) ∧ range(S)=range(S₁)∪range(S₂)
   ```

   benötigt einen rekursiven Aufruf des Verfahrens mit einer anderen Reihenfolge der Einzelschritte (siehe [Smith, 1985b, Section 7.2]) und liefert den `merge`-Operator als Kompositionsfunktion.

5. Nun müssen wir nur noch die direkte Lösung für primitive Eingaben bestimmen und überprüfen hierzu `∀L:Seq(ℤ). L=[]  ⇒  ∃S:Seq(ℤ). SORT(L,S)`. Der Beweis ist nicht schwer zu führen und liefert `S=L`.

6. Wir instantiieren nun den schematischen Divide & Conquer Algorithmus, abstrahieren dabei über die beiden Resultate von `ListSplit(L)` und erhalten folgenden Sortieralgorithmus.

   ```
   FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) RETURNS S  SUCH THAT rearranges(L,S) ∧ ordered(S)
   =  if L=[] then L
               else let L₁,L₂=ListSplit(L)  in  merge(sort(L₁),sort(L₂))
   ```

Weitere Anwendungsbeispiele für eine Synthese von Divide & Conquer Algorithmen kann man in [Smith, 1983a, Smith, 1985a, Smith, 1987a] finden.

5.5.5 Lokalsuch-Algorithmen

Bei der Lösung von Optimierungsproblemen sind lokale Suchverfahren, oft auch als *Hillclimbing-Algorithmen* bezeichnet, eine beliebte Vorgehensweise. Hierbei wird ausgehend eine optimale Lösung eines Problems dadurch bestimmt, daß man eine beliebige Anfangslösung schrittweise verbessert, indem man die unmittelbare (lokale) Nachbarschaft einer Lösung nach besseren Lösungen absucht. Dabei hat die Nachbarschaftsstruktur des Suchraumes einen großen Einfluß auf das Verfahren. Kleine Nachbarschaften beschleunigen die Suche, können aber dazu führen, daß unbedeutende lokale Extremwerte als globale Optima verstanden werden. Dieses Problem tritt bei größeren Nachbarschaften nicht mehr auf, aber dafür muß man wiederum eine erheblich längere Zeit für die Suche in Kauf nehmen.

Die Konstruktion einer guten Nachbarschaftsstruktur spielt daher eine zentrale Rolle für den Erfolg von Lokalsuchalgorithmen. Hierbei sind zwei wichtige Eigenschaften zu berücksichtigen. Zum einen muß der Suchraum vollständig miteinander verbunden sein (*Konnektivität*), damit eine optimale Lösung von jeder beliebigen Lösung auch erreicht werden kann. Zum anderen müssen lokale Optima auch globale Optima sein (*Exaktheit*), damit lokale Suche nicht bei den falschen Lösungen steckenbleibt. Formalisiert man all diese Voraussetzuneg, so kann man Lokalsuchalgorithmen ebenfalls als algorithmische Theorie darstellen und ein schematisches Verfahren global verifizieren.

Die allgemeine Grundstruktur von Lokalsuchalgorithmen läßt sich durch folgendes Schema beschreiben.

```
FUNCTION F_opt(x:D):R   WHERE I(x)  RETURNS y
  SUCH THAT O(x,y) ∧ ∀t:R. O(x,t) ⇒ cost(x,y)≤cost(x,t)
= F_LS(x,F(x))

FUNCTION F_LS(x:D,z:R):R  WHERE I(x) ∧ O(x,y)  RETURNS y
  SUCH THAT O(x,y) ∧ ∀t∈N(x,y). O(x,t) ⇒ cost(x,y)≤cost(x,t)
= if ∀t∈N(x,z). O(x,t)  ⇒  cost(x,z)≤cost(x,t)
   then z   else F_LS(x, arb({t | t∈N(x,z) ∧ O(x,t) ∧ cost(x,z)>cost(x,t)}) )
```

Dabei treten neben den üblichen Bestandteilen D, R, I und O einer Spezifikation folgende Komponenten auf

- Eine totale *Ordnung* $\boxed{\leq}$ auf einem Kostenraum $\mathcal{R}$.
- Eine *Kostenfunktion* $\boxed{\texttt{cost}}$:R→$\mathcal{R}$, die jeder Lösung eine Bewertung zurodnet.
- Eine *Nachbarschaftsstruktur*funktion $\boxed{\texttt{N}}$: D×R → Set(R), die jedem Element des Lösungsraumes eine Menge benachbarter Elemente zuordnet.
- Eine *Initiallösung* $\boxed{\texttt{F(x)}}$, die korrekt im Sinne der Ausgabebedingung O ist.

Wie bei den anderen Algorithmentheorien kann man nun Axiome für die Korrektheit von Lokalsuchalgorithmen aufstellen und durch logische Formeln präzisieren. Dabei kann die Konnektivität der Nachbarschaftsstruktur nur über eine Iteration der – durch O modifizierten – Nachbarschaftsstrukturfunktion erklärt werden, die analog zur Interation von split in Definition 5.5.15 wie folgt definiert ist.

Definition 5.5.34

Die k-fache Iteration N^k_O einer Nachbarschaftsstruktur N: D×R → Set(R) *unter der Bedingung O*: D×R → $\mathbb{B}$ *ist definiert durch*

$$N^k_O\texttt{(x,y)} \equiv \texttt{if } k\texttt{=0 then } \{\texttt{y}\} \texttt{ else } \bigcup\{\, N^{k-1}\texttt{(x,t)} \mid \texttt{t}\in\texttt{N(x,y)} \wedge \texttt{O(x,t)}\,\}$$

Mit dieser Definition können wir nun einen Satz über die Korrektheit von Lokalsuchalgorithmen aufstellen.

Satz 5.5.35 (Korrektheit von Lokalsuchalgorithmen)

Es seien spec=(D, R, I, O) *eine beliebige Spezifikation,* $(\mathcal{R},\leq)$ *ein total geordneter (Kosten-)Raum,* cost:R→$\mathcal{R}$*,* N:D×R→Set(R) *und* F:D→R*. Das Programmpaar*

```
FUNCTION F_opt(x:D):R   WHERE I(x)  RETURNS y
  SUCH THAT O(x,y)  ∧  ∀t:R. O(x,t) ⇒ cost(x,y)≤cost(x,t)
= F_LS(x,F(x))

FUNCTION F_LS(x:D,z:R):R  WHERE I(x) ∧ O(x,y)  RETURNS y
  SUCH THAT O(x,y)  ∧  ∀t∈N(x,y). O(x,t) ⇒ cost(x,y)≤cost(x,t)
= if ∀t∈N(x,z). O(x,t)  ⇒  cost(x,z)≤cost(x,t)
     then z  else F_LS(x, arb({t | t∈N(x,z) ∧ O(x,t) ∧ cost(x,z)>cost(x,t)}) )
```

ist korrekt, wenn für alle x∈D *und* y,t∈R *die folgenden vier Axiome erfüllt sind*

1. *Korrektheit der Initiallösung:*
   F *erfüllt die Spezifikation* FUNCTION F(x:D):R WHERE I(x) RETURNS y SUCH THAT O(x,y)
2. *Reflexivität der Nachbarschaftsstruktur:* I(x) ∧ O(x,y) ⇒ y∈N(x,y)
3. *Exaktheit lokaler Optima:*
   I(x) ∧ O(x,y) ⇒ ∀t∈N(x,y). O(x,t) ⇒ cost(x,y)≤cost(x,t)
   ⇒ ∀t:R. O(x,t) ⇒ cost(x,y)≤cost(x,t)
4. *Erreichbarkeit gültiger Lösungen:* I(x) ∧ O(x,y) ∧ O(x,t) ⇒ ∃k:IN. t∈N_O^k(x,y)

Auf einen Beweis und die Beschreibung einer konkreten Strategie zur Erzeugung von Lokalsuchalgorithmen wollen wir an dieser Stelle verzichten. Interessierte Leser seien hierfür auf [Lowry, 1988, Lowry, 1991] verwiesen.

5.5.6 Vorteile algorithmischer Theorien für Syntheseverfahren

Praktische Erfahrungen mit bisherigen Ansätzen zur Programmsynthese und vor allem der große Erfolg des KIDS Systems haben gezeigt, daß ein Verfahren zur Synthese von Programmen mit Hilfe von Algorithmenschemata eine Reihe praktischer Vorteile gegenüber anderen Syntheseverfahren hat. Diese liegen vor allem darin begründet, daß formale Schlüssen auf einem sehr *hohen Abstraktionsniveau* stattfinden und auf die Lösung von Teilproblemen abzielen, welche aus Erkenntnissen über Programmiertechniken erklärbar sind. Die Hauptvorteile sind ein *sehr effizientes Syntheseverfahren*, die Erzeugung *effizienter Algorithmenstrukturen*, ein *wissensbasiertes Vorgehen* und ein sehr starkes *formales theoretisches Fundament*.

Das Syntheseverfahren ist vor allem deshalb so effizient, weil der eigentliche Syntheseprozeß durch aufwendige theoretische Voruntersuchungen erheblich entlastet wird. Die tatsächliche Beweislast wird vom Ablauf des Syntheseverfahrens in seine Entwicklung – also in den Entwurf der Algorithmentheorien und ihren Korrektheitsbeweis – verlagert, wo sie nur ein einziges Mal durchgeführt werden muß. Zudem kann die Überprüfung der schwierigeren Axiome einer Algorithmentheorie oft dadurch entfallen, daß Techniken zur Verfeinerung vorgefertigter Teillösungen zur Verfügung stehen. Insgesamt ist ein solches Verfahren erheblich zielgerichteter als andere und somit auch leichter zu verstehen und leichter zu steuern – selbst dann, wenn nur wenig automatische Unterstützung bereitsteht. Die gewünschte *Kooperation zwischen Mensch und Computer* ist somit erstmalig erreichbar: der Mensch kann sich tatsächlich auf Entwurfsentscheidungen konzentrieren während die formalen Details durch den Computer (bzw. die Vorarbeiten) abgehandelt werden.

Auch die erzeugten Algorithmen können erheblich effizienter sein als solche, die zum Beispiel durch (reine) Extraktion aus konstruktiven Beweisen gewonnen werden können. Der Grund hierfür ist einfach: es wird eine sehr *effiziente Grundstruktur vorgegeben* und nach Überprüfung der Axiome instantiiert. Für die Erzeugung der Algorithmen können somit alle bekannten Erkenntnisse über gute Programmstrukturen eingesetzt werden. Ineffiziente Bestandteile, die sich in speziellen Fällen ergeben, bestehen im wesentlichen nur aus Redundanzen und können durch achträgliche Optimierungen beseitigt werden. Zudem ist es ohne weiteres möglich, die Schemata in nahezu jeder beliebigen Programmiersprache zu formulieren – also zum Beispiel das Schema für

Globalsuchalgorithmen aus Satz 5.5.16 auch in C, Pascal, Eiffel, C++, LISP, ML, Prolog oder einer Parallelverarbeitungssprache auszudrücken. An dem entsprechenden Satz und seinem Korrektheitsbeweis würde sich nur wenig ändern, da zusätzlich nur die konkrete Semantik der Programmiersprache in den Beweis integriert werden muß, die zentrale Beweisidee aber nur von der algorithmischen Struktur abhängt. In den anderen Syntheseparadigmen ist der Übergang auf eine andere Programmiersprache verhältnismäßig kompliziert, da hier beliebige Programme übersetzt werden müssen, ohne daß man von Erkenntnissen über deren Struktur Gebrauch machen kann.

Syntheseverfahren auf der Basis algorithmischer Theorien sind die einzigen Verfahren, bei denen man wirklich von einer Verarbeitung von *Wissen* sprechen kann, da hier Erkenntnisse über Algorithmen als Theoreme vorliegen. Während alle Syntheseverfahren in der Lage sind, Lemmata über verschiedene Anwendungsbereiche mehr oder weniger explizit zu verarbeiten, findet die Verarbeitung von Programmierwissen eigentlich nur bei der Anwendung algorithmischer Theorien statt. Bei andersartigen Verfahren ist Programmierwissen bestenfalls zu einem geringen Anteil – und auch nur implizit – in der Codierung der Synthesestrategie enthalten, welche die Anwendung elementarer Inferenz- bzw. Transformationsregeln steuert.

Das formale theoretische Fundament ist die Basis all dieser guten Eigenschaften. Es sichert die Korrektheit der erzeugten Algorithmen auf eine leicht zu verstehende Art und Weise und weist auch den Weg für eine unkomplizierte realisierung von Synthesestrategien mit einem Programm- und Beweisentwicklungssystem für einen universellen logischen Formalismus wie die intuitionistische Typentheorie.

5.6 Nachträgliche Optimierung von Algorithmen

Algorithmen, die durch automatisierte Syntheseverfahren aus formalen Spezifikationen erzeugt wurden, sind nur selten optimal, da ein allgemeines Syntheseverfahren nicht auf alle Details des Problems eingehen kann, welches durch den zu erzeugenden Algorithmus gelöst werden soll. Zwangsläufig bleiben nach der Synthese gewisse Redundanzen und ineffiziente Teilberechnungen zurück, die ein geschulter menschlicher Programmierer schnell als solche erkennen würde. Sie im Voraus zu vermeiden würde aber den eigentlichen Syntheseprozeß unnötig belasten, da die Suche nach anwendbaren Umformungen erheblich ausgedehnt und durch Heuristiken, welche die Effizienz bestimmter Teilkonstrukte bewerten, gesteuert werden müsste. Stattdessen bietet es sich an, *nachträgliche* Optimierungen vorzunehmen, die von einem Menschen gesteuert werden, und hierzu eine Reihe von Grundtechniken bereitzustellen, welche die Korrektheit der Optimierungsschritte garantieren.

Programmtransformationen zur Optimierung von Algorithmen sind ein großes Forschungsgebiet für sich, das an dieser Stelle nicht erschöpfend behandelt werden kann. Wir wollen uns stattdessen auf die Beschreibung einiger einfacher Optimierungstechniken konzentrieren, die sich gut in ein formales Synthesesystem integrieren lassen. Die meisten dieser Techniken sind bereits in dem KIDS System [Smith, 1990, Smith, 1991a] enthalten und haben sich als sehr wirksame Hilfsmittel bei der wissensbasierten Erzeugung effizienter Algorithmen aus formalen Spezifikationen erwiesen. Als Leitbeispiel verwenden wir eine nachträgliche Optimierung des Lösungsalgorithmus für das Costas Arrays Problem, den wir in Beispiel 5.5.26 auf Seite 266 synthetisch erzeugt hatten.

5.6.1 Simplifikation von Teilausdrücken

Die einfachste aller Optimierungstechniken ist die Simplifikation von Teilausdrücken des Programmtextes. Hierdurch sollen unnötig komplexe Ausdrücke durch logisch äquivalente, aber einfachere Ausdrücke ersetzt werden, wobei die Bedeutung dieser Ausdrucke und die algorithmische Struktur des Programms keine Rolle spielen darf. Aus logischer Sicht bestehen Simplifikationen aus einer Anwendung gerichteter Gleichungen als *Rewrite*-Regeln (z.B. durch Substitution oder Lemma-Anwendung). Dabei unterscheidet man *kontextunabhängige* und *kontextabhängige* Simplifikationen.

- Bei kontextunabhängigen (CI-) Simplifikationen wird ausschließlich der zu vereinfachende Teilausdruck

betrachtet und mit Hilfe von Rewrite-Regeln *ohne Vorbedingungen* so lange umgeschrieben, bis keine Vereinfachungsregel mehr angewandt werden kann. So wird zum Beispiel der Ausdruck `a∈(x.[])∘(b.L)` mit den Lemmata aus Appendix B.2 wie folgt schrittweise vereinfacht werden.

```
a∈(x.[])∘(b.L) ↦ a∈x.([]∘(b.L)) ↦ a∈x.(b.L) ↦ a=x ∨ a∈b.L ↦ a=x ∨ a=b ∨ a∈L
```

- Bei kontextabhängigen (CD-) Simplifikationen werden *bedingte Rewrite-Regeln* bzw. bedingte Gleichungen für die Vereinfachung eines Teilausdrucks eingesetzt. Hierbei spielt – wie der Name bereits andeutet – der Kontext des zu vereinfachenden Teilausdrucks eine wesentliche Rolle, denn hieraus ergibt sich, ob eine bedingte Gleichung wie `a∈S ∧ p(a) ⇒ ∀x∈S. p(x) ⇔ ∀x∈S-a. p(x)` auf einen gegebenen Ausdruck überhaupt anwendbar ist.

  Der Kontext wird dabei im Voraus durch eine Analyse des abstrakten Syntaxbaumes bestimmt, in dem sich der Teilausdruck befindet, wobei Ausdrücke wie z.B. der Test einer Fallunterscheidung, benachbarte Konjunkte in einem generellen Set-Former `{ f(x) | x∈S ∧ p(x) ∧ q(x)}` oder die Vorbedingungen der `WHERE`-Klausel zum Kontext gehören.

  Eine der wesentlichen Aspekte der kontextabhängigen Simplifikation ist die Elimination redundanter Teilausdrücke im Programmcode wie zum Beispiel die Überprüfung einer Bedingung, die bereits in den Vorbedingungen des Programms auftaucht.

Üblicherweise wird eine Simplifikation in Kooperation mit einem Benutzer des Systems durchgeführt. Dieser gibt an, welche konkreten Teilausdrücke vereinfacht werden sollen und welcher Art die Vereinfachung sein soll. Hierdurch erspart man sich die aufwendige Suche nach simplifizierbaren Teilausdrücken, die einem Menschen beim ersten Hinsehen sofort auffallen. Wir wollen dies an unserem Leitbeispiel illustrieren.

Beispiel 5.6.1 (Costas Arrays Algorithmus: Simplifikationen)

In Beispiel 5.5.26 hatten wir den folgenden Algorithmus zur Lösung des Costas Arrays Problems durch Instantiierung eines Globalsuch-Schemas erzeugt.

```
FUNCTION Costas (n:ℤ):Set(Seq(ℤ))  WHERE n≥1
  RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}

=  if [nodups([]) ∧ ∀j∈domain([]).nodups(dtrow([],j))]
     then Costas_gs(n,[])  else ∅

FUNCTION Costas_gs (n:ℤ,V:Seq(ℤ)):Set(Seq(ℤ))
  WHERE n≥1 ∧ range(V)⊆{1..n} ∧ nodups(V) ∧ ∀j∈domain(V).nodups(dtrow(V,j))
  RETURNS {p | perm(p,{1..n}) ∧ V⊑p ∧ ∀j∈domain(p).nodups(dtrow(p,j))}

=    {p | [p∈{V}] ∧ perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
   ∪ ⋃{ Costas_gs(n,W) | [W∈{V·i|i∈{1..n}}] ∧ nodups(W) ∧ ∀j∈domain(W).nodups(dtrow(W,j)) }
```

In diesem Algorithmenpaar gibt es eine Reihe offensichtlicher Vereinfachungsmöglichkeiten, die im Programmtext umrandet wurden. So ist nach Lemma B.2.24.1 `nodups([])` immer gültig[35] und ebenso `∀j∈domain([]).nodups(dtrow([],j))` nach Lemma B.2.21.1 und B.1.11.1. der gesamte umrandete Ausdruck vereinfacht sich somit zu `true`, was zur Folge hat, daß das Konditional

```
if nodups([]) ∧ ∀j∈domain([]).nodups(dtrow([],j)) then Costas_gs(n,[]) else ∅
```

[35]Dieses Lemma zu finden ist nicht schwer, da der äußere Operator `nodup` und der innere Operator `[]` zusammen eindeutig bestimmen, welches Lemma überhaupt anwendbar ist. Bei einer entsprechenden Strukturierung der Wissensbank kann dieses Lemma in konstanter Zeit gefunden werden.

insgesamt zu $\texttt{Costas}_{gs}$`(n,[])` vereinfacht werden kann. Auf ähnliche Weise lassen sich auch die anderen umrandeten Ausdrücke mit den Gesetzen endlicher Mengen und Folgen vereinfachen. Die Anwendung kontextunabhängiger Vereinfachungen führt somit zu dem folgenden vereinfachten Programmpaar.

```
FUNCTION Costas (n:ℤ):Set(Seq(ℤ))  WHERE n≥1
  RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
=  Costas_gs(n,[])

FUNCTION Costas_gs (n:ℤ,V:Seq(ℤ)):Set(Seq(ℤ))
  WHERE n≥1 ∧ range(V)⊆{1..n} ∧ nodups(V) ∧ ∀j∈domain(V).nodups(dtrow(V,j))
  RETURNS {p | perm(p,{1..n}) ∧ V⊑p ∧ ∀j∈domain(p).nodups(dtrow(p,j)) }
=    (if perm(V,{1..n}) ∧ ∀j∈domain(V).nodups(dtrow(V,j))  then {V}  else ∅)
 ∪ ⋃{ Costas_gs(n,V·i) |i∈{1..n} ∧ nodups(V·i)  ∧ ∀j∈domain(V·i).nodups(dtrow(V·i,j))}
```

Das aufrufende Hauptprogramm kann nun nicht mehr weiter vereinfacht werden. In der Hilfsfunktion $\texttt{Costas}_{gs}$ bietet sich an vielen Stellen jedoch eine kontextabhängige Vereinfachung an, da im Programmcode eine Reihe von Bedingungen geprüft werden, die bereits in den Vorbedingungen genannt sind. Die zusammengehörigen Ausdrücke und ihre relevanten Kontexte sind durch gleichartige Markierungen gekennzichnet, wobei ein Kontext durchaus mehrfach Verwendung finden kann. So ist zum Beispiel der Ausdruck `perm(V,{1..n})` äquivalent zu `range(V)⊆{1..n} ∧ {1..n}⊆range(V) ∧ nodups(V)`. Da zwei dieser drei Klauseln bereits in den Vorbedingungen genannt sind, kann `perm(V,{1..n})` im Kontext der Vorbedingungen zu dem Ausdruck `{1..n}⊆range(V)` vereinfacht werden. Auf ähnliche Art können wir auch die anderen markierten Teilausdrücke vereinfachen und erhalten als Ergebnis kontextabhängiger Simplifikationen.

```
FUNCTION Costas (n:ℤ):Set(Seq(ℤ))  WHERE n≥1
  RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
=  Costas_gs(n,[])

FUNCTION Costas_gs (n:ℤ,V:Seq(ℤ)):Set(Seq(ℤ))
  WHERE n≥1 ∧ range(V)⊆{1..n} ∧ nodups(V) ∧ ∀j∈domain(V).nodups(dtrow(V,j))
  RETURNS {p | perm(p,{1..n}) ∧ V⊑p ∧ ∀j∈domain(p).nodups(dtrow(p,j)) }
=    (if {1..n}⊆range(V) then {V} else ∅)
 ∪ ⋃{ Costas_gs(n,V·i) | i∈{1..n}\range(V) ∧ ∀j∈domain(V).(V[|V|+1-j]-i)∉dtrow(V,j) }
```

5.6.2 Partielle Auswertung

Eine besondere Art der Vereinfachung bietet sich immer dann an, wenn in einem Ausdruck Funktionen mit einem konstantem (Teil-)Argument vorkommen. Dies legt den Versuch nahe, diese Funktion so weit wie möglich im voraus auszuwerten, also eine sogenannte *Partielle Auswertung* [Bjorner *et.al.*, 1988] oder *Spezialisierung* [Scherlis, 1981] des Programms bereits zur Entwurfszeit vorzunehmen. Dies erspart die entsprechenden Berechnungen zur Laufzeit des Programms.

Bei funktionalen Programmen besteht partielle Auswertung[36] aus dem simplen *Auffalten* einer Funktionsdefinition, gefolgt von einer Simplifikation. Dies wird so lange durchgeführt, wie konstante Teilausdrück vorhanden sind. Wir wollen dies an einem einfachen Beispiel illustrieren.

[36]Dies deckt natürlich nicht alle Möglichkeiten ab, bei denen eine partielle Auswertung möglich ist. Auch Operationen mit mehreren Argumenten, von denen nur eines konstant ist, können mit den aus der Literatur bekannten Techniken behandelt werden. Zudem ist bei imperativen Programmen die partielle Auswertung deutlich aufwendiger, da Zustände von Variablen berücksichtigt werden müssen.

Beispiel 5.6.2

Ein Teilausdruck der Form `|[4;5] ∘ V|`, wobei `V` ein beliebiger Programmausdruck – zum Beispiel eine Variable – ist, kann wie folgt partiell ausgewertet werden.

```
  |[4;5] ∘ V|                  Auffalten der Definition von concat
= |4.([5] ∘ V)|                Auffalten der Definition von concat
= |4.(5.([] ∘ V))|             Auffalten der Definition von concat
= |4.(5.V)|                    Auffalten der Definition von length
= 1 + |5.V|                    Auffalten der Definition von length
= 1 + 1 + |V|                  arithmetische Simplifikation
= 2 + |V|
```

Wie bei den Simplifikationen muß der auszuwertende Teilausdruck durch den Benutzer ausgewählt werden. In einfachen Fällen deckt sich die partielle Auswertung mit der kontextunabhängigen Simplifikation, da hier das Auffalten der Funktionsdefinition einer Lemma-Anwendung gleichkommt. Die wirklichen Stärken der partiellen Auswertung kommen daher erst bei der Behandlung neu erzeugter Funktionen zum Tragen.

5.6.3 Endliche Differenzierung

In manchen Programmen werden innerhalb einer Schleife oder einer Rekursion Teilausdrücke berechnet, deren Hauptargument die Schleifenvariable (bzw. ein Parameter des rekursiven Aufrufs) ist. Diese Berechnung kann vereinfacht werden, wenn ein Zusammenhang zwischen der Änderung der Schleifenvariablen und dem berechneten Teilausdruck festgestellt werden kann. In diesem Fall lohnt es sich nämlich, den Teilausdruck *inkrementell* zu berechnen, anstatt die Berechnung in jedem Schleifendurchlauf neu zu starten. Dies kann zu einer signifikanten Beschleunigung der Berechnung führen.

So verbraucht zum Beispiel die Berechnung von `{1..n}\range(V)` in der Hilfsfunktion $\texttt{Costas}_{gs}$ aus Beispiel 5.6.1 in etwa `|V|*n` Schritte. Berücksichtigt man jedoch, daß bei jedem rekursiven Aufruf von $\texttt{Costas}_{gs}$ die Variable `V` durch `V·i` ersetzt wird, so ließe sich die Berechnung dadurch vereinfachen, daß man eine neue Variable `Pool` einführt, welche die Menge `{1..n}\range(V)` inkrementell verwaltet. Bei einem rekursiven Aufruf braucht `Pool` dann nur entsprechend der Änderung von `V` in `Pool-i` umgewandelt werden, was erheblich schneller zu berechnen ist.

Die soeben angedeutete Technik, durch Einführung einer zusätzlichen Variablen eine inkrementelle Berechnung wiederkehrender Teilausdrücke zu ermöglichen, heißt in der Literatur *endliche Differenzierung* [Paige, 1981, Paige & Koenig, 1982], da eben nur noch die differentiellen Veränderungen während eines Rekursionsaufrufes berechnet werden müssen. Im Kontext funktionaler Programme läßt sich endliche Differenzierung in zwei elementarere Operationen zerlegen, nämlich in einer *Abstraktion über einen Teilausdruck* und eine anschließende Simplifikation.

- Die *Abstraktion eines Algorithmus* f mit der Variablen x *über einen Teilausdruck* $E[x]$ erzeugt einen neuen Algorithmus f', der zusätzlich zu den Variablen von f eine neue Variable c besitzt. Die Vorbedingung von f wird um $c{=}E[x]$ erweitert und im Funktionskörper wird jeder Aufruf der Gestalt $f(t[x])$ durch $f'(t[x],E[t[x]])$ ersetzt. In ähnlicher Weise wird auch jeder externe Aufruf der Gestalt $f(u)$ durch $f'(u,E[u])$ ersetzt.

  Eine Abstraktion transformiert also eine Menge von Programmen der Gestalt

  FUNCTION g... WHERE ... RETURNS ... SUCH THAT... $=$ .... $f(u)$ ....

  FUNCTION $f(x{:}D){:}R$ WHERE $I[x]$ RETURNS y SUCH THAT $O[x,y]$
  $=$ .... $E[x]$ ... $f(t[x])$ ....

  in die folgende Menge von Programmen

  FUNCTION g... WHERE ... RETURNS ... SUCH THAT... $=$ .... $f'(u,E[u])$ ....

  FUNCTION $f'(x{:}D,c{:}D'){:}R$ WHERE $I[x] \wedge c{=}E[x]$ RETURNS y SUCH THAT $O[x,y]$
  $=$ .... $E[x]$ ... $f'(t[x],E[t[x]])$ ....

- Die anschließende Simplifikation beschränkt sich auf die Berücksichtigung der neu entstandenen Gleichung c=$E[x]$, die als Kontext der Teilausdrücke des Programmkörpers von f' hinzukommt.

  Kontextabhängige Simplifikation wird nun gezielt auf alle Teilausdrücke der Form $E[t[x]]$ angewandt, wobei vor allem versucht wird diese Teilausdrücke in die Form $t'[E[x]]$ umzuwandeln. Anschließend wird jedes Vorkommen von $E[x]$ zu c vereinfacht.

Wieder ist es der Benutzer, der den konkreten Teilausdruck $E[x]$ auswählt, mit dem endliche Differenzierung durchgeführt werden soll. Wir wollen dies an unserem Leitbeispiel illustrieren.

Beispiel 5.6.3 (Costas Arrays Algorithmus: Endliche Differenzierung)

Mit den in Beispiel 5.6.1 durchgeführten Vereinfachungen hatten wir folgende Version des Costas Arrays Algorithmus erzeugt.

```
FUNCTION Costas (n:ℤ):Set(Seq(ℤ))  WHERE n≥1
  RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
=  Costas_gs(n,[])

FUNCTION Costas_gs (n:ℤ,V:Seq(ℤ)):Set(Seq(ℤ))
  WHERE n≥1 ∧ range(V)⊆{1..n} ∧ nodups(V) ∧ ∀j∈domain(V).nodups(dtrow(V,j))
  RETURNS {p | perm(p,{1..n}) ∧ V⊑p ∧ ∀j∈domain(p).nodups(dtrow(p,j)) }
=    (if {1..n}⊆range(V) then {V} else ∅)
 ∪ ⋃{ Costas_gs(n,V·i) | i∈[{1..n}\range(V)] ∧ ∀j∈domain(V).(V[[|V|+1]-j]-i)∉dtrow(V,j) }
```

Ein geübter Programmierer erkennt relativ schnell, daß die ständige Neuberechnung von `{1..n}\range(V)` den Algorithmus unnötig ineffizient werden läßt. Ebenso überflüssig ist die Berechnung von `|V|+1`, die sich ebenfalls nur inkrementell ändert. Beide Ausdrücke werden vom Benutzer gekennzeichnet. Nach (zweimaliger) Anwendung der endlichen Differenzierung ergibt sich dann folgendes Programmpaar.

```
FUNCTION Costas (n:ℤ):Set(Seq(ℤ))  WHERE n≥1
  RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
=  Costas_gs2(n,[],{1..n},1)

FUNCTION Costas_gs2 (n:ℤ,V:Seq(ℤ), Pool:Set(ℤ),Vsize:ℤ):Set(Seq(ℤ))
  WHERE n≥1 ∧ range(V)⊆{1..n} ∧ nodups(V) ∧ ∀j∈domain(V).nodups(dtrow(V,j))
           ∧ Pool = {1..n}\range(V)      ∧ Vsize = |V|+1
  RETURNS {p | perm(p,{1..n}) ∧ V⊑p ∧ ∀j∈domain(p).nodups(dtrow(p,j)) }
= (if Pool=∅ then {V} else ∅)
 ∪ ⋃{ Costas_gs2(n,V·i,Pool−i,Vsize+1) | i∈Pool ∧ ∀j∈domain(V).(V[Vsize-j]-i)∉dtrow(V,j) }
```

Man beachte, daß hierbei `{1..n}⊆range(V)` in `{1..n}\range(V)=∅` umgewandelt und dann durch `Pool=∅` ersetzt wurde. Genauso könnte man in dem neuen Kontext auch `domain(V)` zu `{1..Vsize-1}` vereinfachen, was aber gezielt geschehen müsste und keinen großen Gewinn bringt.

Durch die endliche Differenzierung werden übrigens auch sinnvolle neue Datenstrukturen und Konzepte eingeführt, die im Algorithmus bisher nicht verarbeitet wurden. So ist zum Beispiel die Verwaltung eines Pools von Werten, die noch als mögliche Ergänzung der Folge `V` benutzt werden können, ein wichtiges Konzept, das auch einem menschlichen Programmierer schnell in den Sinn kommen könnte. Hier wurde es nun durch eine allgemeine Optimierungstechnik nachträglich in den Algorithmus eingefügt.

5.6.4 Fallanalyse

In Programmen, die Konditionale der Form `if` P `then` e_1 `else` e_2 vorkommen, mag es sinnvoll sein, die Fallunterscheidung mit P auf den gesamten Programmkörper auszuweiten, um so weitere kontextabhängige

Simplifikationen durchführen zu können. Durch eine solche Art der *Fallanalyse* könnte man zum Beispiel herausbekommen, daß die Funktion $\texttt{Costas}_{gs2}$ niemals aufgerufen wird, wenn `Pool=∅` ist, und die beiden mit ∪ verbundenen Teilausdrücke somit disjunkte Fälle beschreiben.

Die einfachste Technik der *Fallanalyse über einer Bedingung P* besteht darin, einen Teilausdruck E durch `if` P `then` E `else` E zu ersetzen und anschließend kontextabhängige Simplifikationen auf beiden Instanzen von E auszuführen. Auch dies wollen wir an unserem Leitbesipiel illustrieren.

Beispiel 5.6.4 (Costas Arrays Algorithmus: Fallanalyse)

Nach der endlichen Differenzierung hatte der Costas Arrays Algorithmus die folgende Form.

```
FUNCTION Costas (n:ℤ):Set(Seq(ℤ))  WHERE n≥1
  RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
=  Costas_gs2(n,[],{1..n},1)

FUNCTION Costas_gs2 (n:ℤ,V:Seq(ℤ), Pool:Set(ℤ),Vsize:ℤ):Set(Seq(ℤ))
  WHERE n≥1 ∧ range(V)⊆{1..n} ∧ nodups(V) ∧ ∀j∈domain(V).nodups(dtrow(V,j))
            ∧ Pool = {1..n}\range(V)       ∧ Vsize = |V|+1
  RETURNS {p | perm(p,{1..n}) ∧ V⊑p ∧ ∀j∈domain(p).nodups(dtrow(p,j)) }
= (if [Pool=∅] then {V} else ∅)
 ∪ ⋃{ Costas_gs2(n,V·i,Pool−i,Vsize+1) | i∈Pool ∧ ∀j∈domain(V).(V[Vsize-j]-i)∉dtrow(V,j) }
```

Eine Fallanalyse des Programmkörpers von $\texttt{Costas}_{gs2}$ über der markierten Bedingung `Pool=∅` führt dazu, daß im positiven Fall der Ausdruck $\bigcup\{$ $\texttt{Costas}_{gs2}$`(....) | i∈∅ ∧ ...`$\}$ entsteht, der wiederum zu ∅ vereinfacht werden kann. Damit haben wir in beiden Fällen eine Vereinigung mit einer leeren Menge, was zu der folgenden vereinfachten Version des Algorithmus führt.

```
FUNCTION Costas (n:ℤ):Set(Seq(ℤ))  WHERE n≥1
  RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
=  Costas_gs2(n,[],{1..n},1)

FUNCTION Costas_gs2 (n:ℤ,V:Seq(ℤ), Pool:Set(ℤ),Vsize:ℤ):Set(Seq(ℤ))
  WHERE n≥1 ∧ range(V)⊆{1..n} ∧ nodups(V) ∧ ∀j∈domain(V).nodups(dtrow(V,j))
            ∧ Pool = {1..n}\range(V)       ∧ Vsize = |V|+1
  RETURNS {p | perm(p,{1..n}) ∧ V⊑p ∧ ∀j∈domain(p).nodups(dtrow(p,j)) }
= if Pool=∅
     then {V}
     else ⋃{ Costas_gs2(n,V·i,Pool−i,Vsize+1) | i∈Pool ∧ ∀j∈domain(V).(V[Vsize-j]-i)∉dtrow(V,j)}
```

5.6.5 Datentypverfeinerung

Mit den bisherigen Techniken haben wir vor allem die Möglichkeiten einer algorithmischen Optimierung synthetisierter Programme beschrieben. Darüberhinaus kann man aber noch weitere Effizienzsteigerungen dadurch erhalten, daß man für abstrakt definierte Datentypen wie Listen, Mengen, Graphen etc. und ihre Operationen eine Implementierung auswählt, die für den konkreten Algorithmus besonders geeignet ist.

So können endliche Mengen zum Beispiel durch Listen, Felder (Bitvektoren), Bäume, charakteristische Funktionen etc. implementiert werden. Listen wiederum können in vowärts oder rückwärts verketteter Form dargestellt werden. Welche dieser Implementierungen für eine konkrete Variable des Algorithmus gewählt werden sollte, hängt dabei im wesentlichen von den damit verbundenen Operationen ab. So empfiehlt sich eine rückwärts verkettete Liste immer dann, wenn viele Append-Operationen im Algorithmus vorkommen, was innerhalb von $\texttt{Costas}_{gs2}$ zum Beispiel für die Variable `V` gilt. Felder für endliche Mengen sind empfehlenswert, wenn diese – wie die Variable `Pool` – eine Maximalgröße nicht überschreiten und häufig die ∈-Relation getestet wird.

Die Wahl einer konkreten Implementierung für ein abstrakt definiertes Konstrukt nennt man *Datentypverfeinerung* [Schonberg *et.al.*, 1981, Blaine & Goldberg, 1990a, Blaine & Goldberg, 1990b]. Diese Technik ist sehr sicher, wenn das Syntheseystem eine Reihe von Implementierungen für einen abstrakten Datentyp – sowie die eventuell nötigen Konversonsoperationen – bereithält, aus denen ein Benutzer dann separat für jede einzelne Variable eine Auswahl treffen kann. Da die richtige Wahl einer Implementierung sehr stark von einer Einschätzung der Häufigkeit bestimmter Operationen abhängt, kann sie bisher nur beschränkt automatisiert[37] werden.

5.6.6 Compilierung und sprachabhängige Optimierung

Die letzte Möglichkeit zu einer Optimierung von synthetisch erzeugten Algorithmen besteht in einer Übersetzung[38] der Algorithmen in eine konkrete Programmiersprache, die effizienter zu verarbeiten ist als die abstrakte mathematische Sprache, in der alle anderen Schritte stattgefunden haben. Dieser Schritt sollte erst dann ausgeführt werden, wenn alle anderen Möglichkeiten ausgeschöpft wurden, da eine algorithmische Optimierung nach einer Übersetzung erheblich schwerer ist. Sprachabhängige Optimierungen können im Anschluß daran durchgeführt werden, wobei man sich allerdings auf die in einen Compiler der Zielsprache eingebauten Möglichkeiten beschränken sollte.

5.7 Diskussion

Wir haben in diesem Kapitel die verschiedenen Techniken zur Erzeugung effizienter Algorithmen aus ihren formalen Spezifikationen vorgestellt und aufgezeigt, wie diese Methoden in einen allgemeinen logischen Formalismus wie der Typentheorie des NuPRL Systems integriert werden können. Damit sind wir unserem usprünglichem Ziel, durch Verfahren zur rechnergestützten Softwareentwicklung eine Antwort auf die Softwarekrise zu finden, einen deutlichen Schritt nähergekommen und mehr oder weniger bis an den derzeitigen Stand der Forschungen vorgedrungen.

Es hat sich herausgestellt, daß Programmsyntheseverfahren prinzipiell notwendig sind, um die derzeitige Praxis der Programmierung zu verbessern und im Hinblick auf die Erzeugung korrekter und wiederverwendbarer Software zu unterstützen. Trotz einiger spektakulärer Teilerfolge[39] sind Programmsynthesesysteme jedoch noch weit davon entfernt, Marktreife zu erlangen. Zu viele Fragen – vor allem im Hinblick au die Erzeugung formaler Spezifikationen und die Synthese komplerer Systeme – sind noch ungeklärt und das Gebiet des *wissensbasierten Software-Eingineering* ist mehr denn je ein zukunftsträchtiges Forschungsgebiet, das von fundamentaler Bedeutung für die Informatik ist.

Unter den bisherigen Ansätzen erscheint der Weg über algorithmische Theorien der sinnvollste zu sein, vor allem weil der die Stärken der Paradigmen "Beweise als Programme" und "Synthese durch Transformationen" auf einem hohen Abstraktionsniveau miteinander zu verbinden weiß und dem Ideal der *wissensbasierten* Programmsynthese am nächsten kommt. Bei diesem Ansatz muß allerdings der Gefahr begegnet werden,

[37]Dadurch, daß die Implementierung jedoch automatisch erstellt wird, bietet sich die Möglichkeit eines experimentellen Vorgehens an, bei dem man das Laufzeitverhalten verschiedener Implementierungen einer Variablen im konkreten Fall austestet und anhand dieser Erkenntnisse seine Wahl trifft. Die Testergebnisse können hierbei jedoch nur einen Hinweischarakter haben und eine gründliche Analyse des entstandenen Algorithmus nicht ersetzen.

[38]Die Sprache LISP ist hierfür am besten geeignet, da unsere abstrakte mathematische Sprache ebenfalls funktional ist und somit kein Paradigmenwechsel stattfinden müsste. Da LISP auf größeren Computern mittlerweile genauso schnell ist wie andere gängige Programmiersprachen, besteht auch kein Grund, eine andere Zielsprache zu wählen, solange der generierte Algorithmus nicht in ein festes Softwarepaket eingebaut werden soll. Die Übersetzung in andere Zielsprachen wird daher erst bei einem kommerziellen Einsatz von Programmsynthesesystemen relevant werden.

[39]Eine weitestgehend vom KIDS-System geleistete Neuentwicklung des Programms, mit dem z.B. die amerikanische Armee ihre Transporte bei Einsätzen in Übersee plant, konnte den bisher hierzu verwendeten Algorithmus bei realistischen Testdaten um den Faktor 2000 (!) beschleunigen (siehe [Smith & Parra, 1993]). Man vermutet, daß alleine durch diesen Einzelerfolg soviele Millionen von Dollar eingespart werden können, wie die Entwicklung von KIDS gekostet hat.

durch eine von logischen Kalkülen losgelöste "von Hand Codierung" der entsprechenden Spezialstrategien die Vorteile wieder aufs Spiel zu setzen. Im Endeffekt kann nur eine Integration dieses Ansatzes in ein formales Beweisentwicklungssystem wie NuPRL die notwendige Sicherheit und Flexibilität garantieren, die man sich von der rechnergestützten Softwareentwicklung erhofft.

Hierzu ist allerdings noch eine Menge Arbeit zu leisten. So muß zum Beispiel eine große *Wissensbank* aufgebaut werden, um eine Synthese von Algorithmen aus verschiedensten Anwendungsbereichen zu ermöglichen. Dies bedeutet nicht nur die Formalisierung und Verifikation von Wissen über die Standard-Anwendungsbereiche, mit denen sich die Informatik-Literatur beschäftigt,[40] sondern auch Techniken zur effizenten Verwendung dieses Wissens als Rewrite-Regeln und Methoden zur *effizienten Strukturierung der Wissensbank* in Teiltheorien, die einen schnellen und gezielten Zugriff auf dieses Wissen unterstützen. Auf der anderen Seite sind neben den einfachen Rewrite-Techniken, die an sich schon sehr viel zum praktischen Erfolg von Synthesesystemen beitragen, eine Reihe von *Beweisverfahren* in das allgemeine System zu integrieren. Hierzu gehören vor allem ein konstruktives prädikatenlogisches Beweisverfahren, allgemeine Induktionstechniken, und natürlich auch anwendungsspezifische Beweismethoden, welche die besonderen Denkweisen bestimmter Theorien widerspiegeln. All diese Techniken sind mehr oder weniger als Taktiken zu modellieren, wenn sie in ein einziges System integriert werden sollen. Mit einem derartigen Fundament können dann die in diesem Kapitel besprochenen *Syntheseverfahren und Optimierungstechniken* in das allgemeine System eingebettet werden. Mit der in Abschnitt 5.5.3 vorgestellten Vorgehensweise ist dies auf elegante und natürliche Art möglich: die Formalisierung der Strategien entspricht dann im wesentlichen ihrer Beschreibung auf dem Papier und benötigt keine gesonderte Codierung.

Im Hinblick auf *praktische Verwendbarkeit* müssen Programmsynthesesysteme viele einander scheinbar widersprechende Bedingungen erfüllen. So muß die interne Verarbeitung von Wissen natürlich formal korrekt sein, um die geforderte Sicherheit der erzeugten Software zu garantieren. Nach außen hin aber sollte so wenig formal wie möglich gearbeitet werden, um einem "normalen" benuzter den Umgang mit dem System zu ermöglichen. Der in Abschnitt 4.1.7.2 angesprochene Display-Mechanismus ist ein erster Ansatz, die Brücke zwischen diesen beiden Anforderungen zu bauen. Dies muß jedoch ergänzt werden um eine anschauliche graphische Unterstützung bei der Steuerung eines Syntheseprozesses, damit sich auch ungeübte Benutzer zügig in den Umgang mit einem Softwarentwicklungssystem einarbeiten können. In diesem Punkte sind wissenschaftliche Programme leider weit von dem entfernt, was man von einem modernen Benutzerinterface erwarten kann.

Möglichkeiten zur Mitarbeit an Systemen für eine rechnergestützte Softwareentwicklung gibt es also mehr als genug. Dies setzt allerdings ein gewisses Interesse für eine Verknüpfung von theoretische Grundlagen und praktischen Programmierarbeiten voraus, also Formale Denkweise, Kenntnis logischer Kalküle und Abstraktionsvermögen auf der einen Seite sowie Kreativität, Experimentierfreudigkeit und Ausdauer auf der anderen. Da Forschungstätigkeiten immer auch an die Grenzen des derzeit Machbaren und der eigenen Fähigkeiten heranführen, Unzulänglichkeiten der zur Verfügung stehenden Hilfsmittel schnell offenbar werden und sich Fortschritte nicht immer so schnell einstellen, wie man dies gerne hätte, ist auch ein gehöriges Maß von Frustrationstoleranz eine wesentliche Voraussetzung für einen Erfolg.

[40]Man müsste mindestens 500 Definitionen und 5000 Lemmata aufstellen, um ein Wissen darzustellen, was dem eines Informatik-Studenten nach dem Vordiplom gleichkommt.