FernUniversität - Gesamthochschule Hagen
Fachbereich Informatik
Lehrgebiet Praktische Informatik IV
Prof. Dr. Güting

Partielle Auswertung für eine einfache imperative Sprache

Seminar 1902: Programmtransformation
Betreuer: Dr. Martin Erwig
Wintersemester 1997/98

Basis-Literatur: N.D. Jones, C.K. Gomard, P. Sestoft: *Partial Evaluation and Automatic Program Transformation*; Kapitel 4 (bis einschließlich 4.5), pp. 67-86, einzelne Seiten aus Kapitel 3; Prentice-Hall International, 1993 [1]

Ulrich Telle
Matrikel-Nr. 1471341
Wolfskaul 12
51061 Köln

mailto:Ulrich.Telle@FernUni-Hagen.de
http://www.stud.fernuni-hagen.de/q1471341

Februar 1998

Inhaltsverzeichnis

1 Einleitung und Grundbegriffe

1.1 Historie

Die *Partielle Auswertung* oder *Programmspezialisierung* als Methode der Programmtransformation ist schon ein vergleichsweise altes Forschungsgebiet der Informatik. Bereits 1964 wurde eine Arbeit [4] veröffentlicht, in der vermutlich zum ersten Mal der Begriff *partielle Auswertung* verwendet wurde. Seit 1971 ist bekannt, daß die *Partielle Auswertung* eng mit der Compilierung verbunden ist. Insbesondere kann ein Programmspezialisierer zum Compilieren verwendet werden, wenn eine interpretative Definition einer Programmiersprache als Eingabe vorliegt. Ein Programmspezialisierer kann, falls er auf sich selbst anwendbar ist, sogar einen Compiler oder auch einen Compiler-Generator erzeugen.

Die meisten veröffentlichten Arbeiten stammen aus dem Bereich der funktionalen Sprachen, mit Schwerpunkt *Lisp*. Erstmals wurden in den späten 70er und frühen 80er Jahren Ideen [5] entwickelt, die Methoden der *partiellen Auswertung* auch auf imperative Sprachen anzuwenden. Da fast alle gängigen Maschinenarchitekturen auf dem von-Neumann-Konzept, das zu imperativen Maschinensprachen führt, aufbauen, partielle Auswertung für funktionale Sprachen jedoch wiederum zu funktionalem Code führt, ist die partielle Auswertung für imperative Sprachen für praktische Anwendungen besonders interessant.

1.2 Partielle Auswertung

Die theoretische Grundlage für die *partielle Auswertung* bildet das aus der Berechenbarkeitstheorie bekannte *s-m-n-Theorem*, welches besagt, daß zu jeder partiell rekursiven Funktion $g(x,y)$ eine total rekursive Funktion $\sigma(x)$ existiert, so daß $f_{\sigma(x)}(y) = g(x,y)$ für alle (x,y) gilt.

In der mathematischen Logik findet sich eine gleichwertige Aussage: Jede Funktion von n Variablen läßt sich in eine äquivalente einstellige Funktion umwandeln. Das Ergebnis dieser neuen Funktion ist eine Funktion von (n-1) Variablen. Diese Umwandlung wird nach H.B. Curry auch *Curryfizierung* genannt.

Bei der *partiellen Auswertung* eines in einer Quellsprache L geschriebenen Programms p bezüglich bekannter Eingabedaten d_1, d_2, ... d_n und unbekannter Eingabedaten u_1, ... u_m versucht man durch teilweise Abarbeitung von p ein neues Programm p' zu konstruieren, so daß $p'(u_1, \dots u_m) = p(d_1, \dots d_n, u_1, \dots u_m)$ gilt. Alle Befehle, die aufgrund der Abhängigkeit von unbekannten Eingabedaten noch nicht ausgeführt werden können, werden zurückgestellt und erst in einer späteren Phase abgearbeitet. Das entstandene Programm p' heißt *residuales Programm* zu p. Ein Programm, das aus einem gegebenen Programm p zu bekannten Eingabedaten ein *residuales Programm* p' erzeugt, heißt *partieller Auswerter* oder *Spezialisierer*.

1.3 Zielsetzung

Ziel des Kapitels aus [1], das dieser Ausarbeitung zugrunde liegt, ist es, an einem konkreten Beispiel zu zeigen, daß es möglich ist, mit relativ einfachen Techniken einen nicht-trivialen Programmspezialisierer für eine einfache imperative Sprache zu konstruieren.

Von anderen Arbeiten auf dem Gebiet der Programmtransformation unterscheidet sich der hier vorgestellte Ansatz durch drei grundlegende Forderungen:

1. Die Methoden sollen vollautomatisch sein, ohne jegliche Notwendigkeit eines Benutzereingriffs während der Programmtransformation.
2. Die Methoden müssen ausreichend mächtig sein, um zu compilieren, indem ein Interpreter in Bezug auf ein festes Quellprogramm spezialisiert wird.
3. Es soll möglich sein, automatisch eigenständige Compiler zu erzeugen. Um dahin zu gelangen, muß der Spezialisierer auf sich selbst angewendet werden können.

Zunächst werden die wesentlichen Prinzipien der partiellen Auswertung, die weitgehend unabhängig von der speziellen Programmiersprache sind, eingeführt. Aufbauende Konzepte und Techniken werden nur bei Bedarf verwendet. Im Anschluß daran wird der Zusammenhang von partieller Auswertung und Compilierung dargestellt, der im wesentlichen auf den Ideen von *Futamura*, den sogenannten *Futamura-Projektionen*, beruht.

1.4 Schreibweisen

Um *Programmspezialisierung* genauer definieren zu können, müssen die Effekte einer Programmausführung formal beschrieben werden. Wenn mit *p* ein Programm in Sprache *L* und mit *d* die Eingabedaten des Programms bezeichnet werden, so wird das Ergebnis einer Programmausführung mit

$$||p||_L \; d = \text{result}$$

bezeichnet (falls das Programm terminiert). $|| \cdot ||_L$ bezeichnet man auch als Semantik-Funktion.

Da *Programmspezialisierer* sowohl Programme als auch Daten als Eingabe akzeptieren, müssen sowohl das Programm p als auch die Eingabedaten d aus einer gemeinsamen Menge D von Datenwerten stammen. Eine Notation, die es erlaubt, Programme wie Daten zu behandeln, ist die *Listen-Notation* von Lisp mit Hilfe von sogenannten *S-Ausdrücken*.

Wenn $\boldsymbol{A}$ die Menge der Atome und $\boldsymbol{S}$ die Menge der S-Ausdrücke ist, kann die Menge $\boldsymbol{S}$ wie folgt rekursiv definiert werden:

1. $\boldsymbol{A} \subset \boldsymbol{S}$
2. $\alpha, \beta \in \boldsymbol{S} \Rightarrow (\alpha \,.\, \beta) \in \boldsymbol{S}$
3. andere S-Ausdrücke gibt es nicht

Für spezielle S-Ausdrücke benutzt man die Listenschreibweise:

4. $(\,) \equiv \text{nil}$
5. $s_1, \ldots s_n \in \boldsymbol{S} \Rightarrow (s_1\ s_2 \ldots s_n) \equiv (s_1 \,.\, (s_2 \,.\, (\ldots (s_{n-1} \,.\, (s_n \,.\, \text{nil})) \ldots)))$

Ein Konstruktor C mit Parametern $p_1, \ldots p_n$ wird dann beispielsweise als $(C\ p_1 \ldots p_n)$ dargestellt.

1.5 Die Flußdiagrammsprache

Um die grundlegenden Prinzipien der partiellen Auswertung einzuführen, wird eine einfache imperative Sprache verwendet:

⟨Program⟩	::=	**read** ⟨Var⟩, ... ⟨Var⟩; ⟨BasicBlock⟩$^+$
⟨BasicBlock⟩	::=	⟨Label⟩: ⟨Assignment⟩* ⟨Jump⟩
⟨Assignment⟩	::=	⟨Var⟩ := ⟨Expr⟩ ;
⟨Jump⟩	::=	**goto** ⟨Label⟩ ;
	|	**if** ⟨Expr⟩ **goto** ⟨Label⟩ **else** ⟨Label⟩ ;
	|	**return** ⟨Expr⟩ ;
⟨Expr⟩	::=	⟨Constant⟩
	|	⟨Var⟩
	|	⟨Op⟩ ⟨Expr⟩ ... ⟨Expr⟩
⟨Constant⟩	::=	**quote** ⟨Val⟩
		(Abkürzend kann der konstante Ausdruck (quote value) *auch als* 'value *geschrieben werden.)*
⟨Op⟩	::=	**hd | tl | cons |** ...
		(und weitere für Interpreter und Programmspezialisierer benötigte)
⟨Label⟩	::=	*Identifizierer oder Zahl*

Im Vergleich zu einer echten Programmiersprache ist diese Sprache lächerlich klein, aber aus folgenden Gründen ist es dennoch sinnvoll, die partielle Auswertung an einer solchen Sprache zu untersuchen:

- Die Semantik der Sprache ist so einfach zu verstehen, daß nicht vom Kernproblem der partiellen Auswertung abgelenkt wird.
- Diese Sprache war die erste imperative Sprache, für die erfolgreich selbstanwendbare partielle Auswertung implementiert wurde. Ein Schlüssel zum Erfolg war in der Tat die Einfachheit der Sprache. Nachfolgende Experimente mit der partiellen Auswertung stärkerer imperativer Sprachen haben sämtlich die hier gezeigten Techniken verwendet.
- Partielle Auswertung einer Flußdiagramm-Sprache dient auch als Ausgangspunkt für die partielle Auswertung funktionaler Sprachen wie z.B. *Scheme* und logischer Sprachen wie z.B. *Prolog*. Trotz der unterschiedlichen Natur der Sprachen sind die Kerntechniken übertragbar.
- Viele der üblichen Komplikationen bei partieller Auswertung treten zwar auf, aber bedingt durch die Einfachheit der Sprache in einer sehr reinen Form. Es hat sich als fruchtbar erwiesen, die Probleme zunächst für eine minimale imperative Sprache zu lösen und dann diese Lösungen auf komplexere Situationen zu übertragen.

Konstanten sind Lisp-S-Ausdrücke und Operationen werden auf S-Ausdrücke angewendet. Der Programmspeicher ist eine Abbildung der Programmvariablen in ihre aktuellen Werte. Die Eingabe eines Programms p ist eine Werteliste d = (v_1, ... v_n), die zu Beginn an die Variablen var_1, ... var_n gebunden ist. Alle nicht eingegebenen Variablen var_{n+1} haben als Anfangswert die leere Liste ( ), so daß der Anfangszustand des Speichers eine endliche Funktion ist:

$$[var_1 \mapsto v_1, \dots var_n \mapsto v_n, var_{n+1} \mapsto (\,), \dots var_{n+k} \mapsto (\,)]$$

Die Basisfunktionen werden als frei von Seiteneffekten auf die Werte der Variablen angenommen. Zuweisungen, Bedingungen und `goto`'s werden in der üblichen Weise ausgeführt und ein `return`-Ausdruck beendet die Ausführung, mit dem Wert des Ausdrucks als Ergebnis der Programmausführung $\|p\|_L$ d.

Wegen der besseren Lesbarkeit werden Programme frei unter Verwendung Pascal-ähnlicher Konstrukte wie ***begin...end***, ***while...do*** und ***repeat...until*** geschrieben, da diese Konstrukte problemlos mit Hilfe von Basisblöcken, Labels und `if`-Anweisungen dargestellt werden können.

1.6 Residuale Programme und Programmspezialisierung

Im weiteren bezeichnet L die Flußdiagrammsprache. Wenn ***p*** ein L-Programm, das als Eingabe eine zwei-elementige Liste erwartet, und d_1 ein Element der Datenmenge ***D*** ist, dann wird ein L-Programm ***r*** ein *residuales Programm* für ***p*** bezüglich d_1 genannt, wenn die Gleichung

$$||p||_L\ [d_1, d_2] = ||r||_L\ d_2$$

für alle d_2 aus ***D*** erfüllt ist.

Die Gleichung drückt aus, daß die Ausführung von ***r*** mit Eingabe d_2 das gleiche Resultat liefert wie die Ausführung von ***p*** mit der Eingabe d_1 und d_2. Die Eingabe d_1 ist also in das residuale Programm ***r*** integriert worden. Ein L-Programm ***mix***, das bei gegebenem ***p*** und d_1 ein residuales Programm ***r*** so erzeugt, daß obige Gleichung gilt, wird als *Programmspezialisierer* bezeichnet. Kurz ausgedrückt muß die Gleichung

$$||p||_L\ [d_1, d_2] = ||(||mix||_L\ [p, d_1])||_L\ d_2$$

für alle d_2 aus ***D*** erfüllt sein.

Das Programm ***p*** wird *Subjektprogramm* genannt. Die rechte Seite der Gleichung beinhaltet zwei Programmausführungen: Zuerst wird ein residuales Programm $r = \|mix\|_L\ [p, d_1]$ erzeugt; dann wird das residuale Programm ***r*** auf den Daten d_2 ausgeführt.

Die Gleichungen können dahingehend verallgemeinert werden, daß ***p*** eine feste Anzahl **n** von Eingabedaten erwartet, von denen **m** durch den Programmspezialisierer ***mix*** in das residuale Programm ***r*** integriert werden. Es gilt stets $0 \le \mathbf{m} \le \mathbf{n}$. In obigen Gleichungen gilt **n**=2 und **m**=1.

An einem Beispiel soll die Wirkung der Gleichungen verdeutlicht werden. Sei also folgendes Programmfragment, das etwa innerhalb eines Interpreters auftreten könnte, der den Speicher eines interpretierten Programms mit Hilfe einer Namensliste `namelist` und einer parallelen Werteliste `valuelist` darstellt, gegeben:

```
while name ≠ hd(namelist) do
begin
  valuelist := tl(valuelist);
  namelist := tl(namelist)
end;
value := hd(valuelist);
```

Die *Eingabedaten* für das Programmfragment sind die drei Variablen `name`, `namelist` und `valuelist`, d.h., **n**=3. Angenommen, dem Programmspezialisierer werden die Anfangswerte der Variablen `name` und `namelist` gegeben, z.B. `name = z` und `namelist = (x y z)`, aber die Werteliste `valuelist` ist unbekannt, d.h., **m**=2. Da die Programmkontrolle vollständig durch `name` und `namelist` festgelegt ist, kann es symbolisch ausgeführt werden mit folgendem residualem Programm als Ergebnis:

```
valuelist := tl(valuelist);
valuelist := tl(valuelist);
value := hd(valuelist);
```

Die ***while***-Schleife ist verschwunden und das residuale Programmfragment enthält nur noch die Befehle des Subjektprogramms, deren Effekte nicht während der Programmspezialisierung berechnet werden können. Weiterhin taucht der Befehl

```
valuelist := tl(valuelist);
```

zweimal im residualen Programm auf, einmal für jede Iteration der Schleife, obwohl er im Subjektprogramm nur ein Mal vorkommt. Die zwei Iterationen rühren daher, daß zweimal ***tl*** von `namelist` zu bestimmen war, bevor ***hd*** gleich `name` war.

Da die Variable `valuelist` nach dem Programmfragment nicht mehr benötigt wird, wäre denkbar, das resultierende Programm zu

```
value := hd(tl(tl(valuelist)));
```

zu optimieren. Für die Programmspezialisierung sind solche Optimierungen jedoch nicht entscheidend und werden daher im folgenden außer acht gelassen.

In diesem Beispiel wird angenommen, daß der Programmspezialisierer residuale Programme erzeugt, die in der gleichen Sprache wie die Subjektprogramme geschrieben sind. Das vereinfacht natürlich das Verständnis der Transformationen, die ***mix*** ausführt. In der praktischen Anwendung braucht das nicht der Fall zu sein. Um maximale Effizienz zu erreichen, würden die residualen Programme stattdessen vielleicht in einer Maschinensprache erzeugt.

2 Programmspezialisierung

2.1 Allgemeines

Eine Programmspezialisierung kann in drei Schritten vollzogen werden. Alle drei Schritte sind im wesentlichen unabhängig von der speziellen Programmiersprache, die transformiert wird. Es wird davon ausgegangen, daß jedes Programm eine Menge von *Programmpunkten* besitzt. Damit sind zum Beispiel Label in einer Flußdiagrammsprache, Funktionsnamen in einer funktionalen Sprache oder Prozedurnamen (Prädikate) in einer logischen Programmiersprache gemeint. Die drei Schritte im Algorithmus der Programmspezialisierung sind:

1. Bei gegebenen Werten eines Teils der Eingabe eines Programms wird eine Beschreibung aller Berechnungszustände, die erreichbar sind, wenn das Programm für alle möglichen Eingabewerte läuft, bestimmt.
2. Die Programmkontrolle wird redefiniert, indem Teile der Datenzustandsinformation in den Kontrollzustand eingearbeitet werden. Unter Umständen ergeben sich daraus keine, eine oder mehrere spezialisierte Versionen jedes Kontrollpunkts des Programms. Daher wird dieser Programmspezialisierungsansatz auch *polyvariante Spezialisierung* bzw. *polyvariante gemischte Berechnung* genannt. Daneben gibt es auch andere Ansätze wie z.B. *Supercompilierung*.
3. Das resultierende Programm enthält üblicherweise viele triviale Übergänge. Es wird daher mit herkömmlichen Techniken optimiert. Daraus ergibt sich das spezialisierte bzw. residuale Programm.

Damit der Programmspezialisierer ***mix*** so einfach und lesbar wie möglich bleibt, wird die Existenz von Bibliotheksfunktionen vorausgesetzt, die lästige Arbeiten erledigen, die für die Konzepte der partiellen Auswertung jedoch nicht entscheidend sind.

2.2 Zustände, Programmpunkte und Einteilungen

Für jeden beliebigen Zeitpunkt während der Ausführung eines Programms wird ein *Berechnungszustand* als Paar (`pp`, `store`) geschrieben. Dabei bezeichnet `pp` den *Programmpunkt*, der den aktuellen Kontrollpunkt markiert, und `store` den aktuellen Inhalt aller Programmvariablen. Ein Speicher, der die aktuellen Werte der Variablen X_1, ... X_n enthält, wird durch eine Liste der Form (v_1, ... v_n) dargestellt. Wenn eine Zuweisung ausgeführt wird, wird der Speicher `store` aktualisiert und `pp` auf den nächsten Programmpunkt gesetzt; wenn eine Bedingung ausgewertet oder ein **goto** ausgeführt wird, wird nur der Kontrollpunkt aktualisiert.

Ist nur ein Teil der Eingabedaten gegeben, so kann das Programm zwar nicht ausgeführt werden, aber es kann in Bezug auf die bekannten Eingabedaten spezialisiert werden. In diesem Fall sind zu Beginn und während der Spezialisierung die Werte einiger Variablen unbekannt; Der Speicherzustand kann daher zu Beginn und später nur unvollständig beschrieben werden. Daraus ergibt sich, daß zur Spezialisierungszeit in der Regel nicht alle Ausdrücke des Subjektprogramms ausgewertet werden können.

Eine einfache Methode, einen solchen unvollständigen Speicher zu beschreiben, ist, jede Variable als *statisch* zu klassifizieren, wenn ihre Werte zur Programmspezialisierungszeit bestimmt werden können, und als *dynamisch*, wenn sie nicht statisch ist. Ein partieller Berechnungszustand kann dann als Paar der Form (`pp`, `vs`) notiert werden, wobei `vs` eine Liste der Werte der statischen Variablen ist. Die Werte der dynamischen Variablen sind unbekannt.

Eine solche Klassifikation der Variablen als statisch oder dynamisch wird *Einteilung* (englisch *division*) genannt. Sofern nicht explizit etwas anderes angegeben wird, wird im folgenden angenommen, daß dieselbe Einteilung für **alle** Programmpunkte Gültigkeit hat. Eine solche Einteilung wird *uniform* genannt. Diese Annahme vereinfacht die Betrachtungen und trifft in der Praxis häufig zu.

Bei der Programmspezialisierung ist eine *(uniform) kongruente* Einteilung eine wesentliche Voraussetzung. Damit ist gemeint, daß bei jedem Übergang von einem Zustand (`pp`, `store`) zu einem Zustand (`pp'`, `store'`) die Werte der statischen Variablen im Programmpunkt `pp'` aus den Werten der statischen Variablen im Programmpunkt `pp` berechenbar sein müssen. Anders ausgedrückt muß jede Variable, die von einer dynamischen Variablen abhängt, selbst dynamisch sein.

Ein Ausdruck wird statisch genannt, wenn er ausschließlich Konstanten und statische Variablen enthält, und dynamisch, wenn er dynamische Variablen enthält. Ist in folgender Zuweisung eines Subjektprogramms

```
X := exp
```

der Ausdruck `exp` dynamisch, dann muß auch die Variable `X` wegen der Kongruenzbedingung dynamisch sein.

Unter der Annahme, daß in dem Programmfragment aus Abschnitt 1.6 die Variablen `name` und `namelist` statisch und die Variablen `value` und `valuelist` dynamisch sind, wird in keiner der Zuweisungen die Kongruenzbedingung verletzt:

```
while name ≠ hd(namelist) do
begin
  valuelist := tl(valuelist);
  namelist := tl(namelist)
end;
value := hd(valuelist);
```

2.3 Programmpunktspezialisierung

Wird zur Programmspezialisierungszeit während der Berechnung ein Programmpunkt `pp` mit Werten der statischen Variablen `vs` erreicht, werden die Werte der statischen Variablen in den Programmpunkt eingearbeitet, indem das Paar (`pp`, `vs`) zu einem Programmpunkt des residualen Programms gemacht wird. Das Paar (`pp`, `vs`) wird auch als *spezialisierter Programmpunkt* bezeichnet. Der Code, den (`pp`, `vs`) im residualen Programm markiert, ist eine optimierte Version des Codes an der Stelle `pp` des Subjektprogramms, da die Werte der statischen Variablen bekannt sind. Ein und derselbe Programmpunkt `pp` kann in unterschiedlichen Varianten mit jeweils anderen Werten für die statischen Variablen im residualen Programm auftreten.

Diese Vorgehensweise wird *Programmpunktspezialisierung* genannt und soll an dem aus den vorangegangenen Abschnitten bekannten Programmfragment verdeutlicht werden. Allerdings muß nun auf die Verwendung Pascal-ähnlicher Kontrollstrukturen verzichtet werden, um zu explizit benannten Programmpunkten zu gelangen:

```
search: if name = hd(namelist) goto found else cont;
cont:   valuelist := tl(valuelist);
        namelist := tl(namelist);
        goto search;
found:  value := hd(valuelist);
```

Zu Beginn habe die Variable `name` den Wert `z` und die Variable `namelist` den Wert `(x y z)`. Wird nun die Spezialisierung im Programmpunkt `search` begonnen, dann ist der Anfangswert von `vs` im Spezialisierer das Paar:

```
vs = (z,(x y z))
```

Bei der Ausführung des Programms kann die Liste `vs` im Programmpunkt `search` drei verschiedene Wert annehmen, nämlich `(z,(x y z))`, `(z,(y z))` und `(z,(z))`. Im Programmpunkt `cont` kann `vs` die ersten beiden der aufgeführten Werte annehmen und im Programmpunkt `found` hat `vs` als einzigen Wert `(z,(z))`.

Im residualen Programm ist der Wert von `vs` in den spezialisierten Programmpunkt (`pp`, `vs`) eingebaut. Die Menge aller spezialisierten Programmpunkte (`pp`, `vs`), die während der Ausführung des Subjektprogramms erreichbar sind, werden ***poly*** genannt. ***Poly*** repräsentiert daher die Menge der Programmpunkte im residualen Programm. In obigem Beispiel ergibt sich:

```
poly = { (search, (z,(x y z))), (search, (z,(y z))), (search, (z,(z))),
         (cont,   (z,(x y z))), (cont,   (z,(y z))),
         (found,  (z,(z)))
       }
```

Bevor gezeigt wird, wie die Berechnung von ***poly*** erfolgen kann, wird zunächst die Codeerzeugung für das residuale Programm beschrieben.

2.4 Codeerzeugung

Im Subjektprogramm markiert der Label `pp` des betrachteten, spezialisierten Programmpunkts (`pp`, `vs`) einen Basisblock, der aus einer Folge von Befehlen besteht. Der generierte Code für einen Basisblock ist die Aneinanderreihung des Codes, der für diese Befehle generiert wird.

Für die eigentliche Codeerzeugung wird die Existenz von zwei Bibliotheksfunktionen vorausgesetzt:

1. `eval(exp,vs)`
2. `reduce(exp,vs)`

wobei `exp` ein Ausdruck und `vs` eine Werteliste der statischen Variablen des Programms ist. Die Funktion `eval` gibt den Wert des statischen Ausdrucks `exp` zurück, während die Funktion `reduce` eine Konstantenfaltung des statischen Teils des dynamischen Ausdrucks durchführt. Wenn `vs` z.B. besagt, daß `b = 2` gilt, dann kann der Ausdruck `b * b + a` ersetzt werden durch `4 + a`. Für einen statischen Ausdruck `exp` liefert die Funktion `reduce` den *Wert* des Ausdrucks, `eval(exp,vs)`, als konstanten Ausdruck zurück.

In der nachfolgenden Tabelle wird für jede gültige Anweisung der imperativen Sprache die zur Spezialisierungszeit durchzuführende Berechnung sowie der daraus resultierende erzeugte Code angegeben:

Befehl im Subjektprogramm	*Zur Spezialisierungszeit*	*Erzeugter Code*
`X := exp` (`X` dynamisch)	`reduced_exp := reduce(exp,vs)`	`X := reduced_exp`
`X := exp` (`X` statisch)	`val := eval(exp,vs);` `vs := vs [X ↦ val]`	
`return` `exp`	`reduced_exp := reduce(exp,vs)`	**`return`** `reduced_exp`
`goto` `pp'`	**`goto`** `(pp',vs)`	
Code für `if exp goto pp' else pp''`		
(wenn `exp` dynamisch)	`reduced_exp := reduce(exp,vs)`	**`if`** `reduced_exp` **`goto`** `(pp', vs)` **`else`** `(pp'', vs)`
(wenn `exp` statisch und `val = true`)	`val := eval (exp, vs)`	**`goto`** `(pp', vs)`
(wenn `exp` statisch und `val = false`)	`val := eval (exp, vs)`	**`goto`** `(pp'',vs)`

2.5 Berechnung von *poly*

Wenn der erste Label im Programm mit pp_0 und die Anfangswerte der statischen Daten mit vs_0 bezeichnet werden, dann muß selbstverständlich (pp_0, vs_0) in ***poly*** enthalten sein. Ist darüber hinaus irgendein Programmpunkt (`pp`, `vs`) in ***poly*** enthalten, so müssen alle spezialisierten Programmpunkte, die von (`pp`, `vs`) erreichbar sind, ebenfalls enthalten sein. Die Menge ***poly*** ist somit der *Abschluß* von vs_0 unter der Relation ***'erreichbar von'***.

Der Label `pp` eines spezialisierten Programmpunkts (`pp`, `vs`) markiert einen Basisblock des Subjektprogramms. Ein Basisblock besteht aus einer Sequenz von Zuweisungen und wird durch eine 'Kontrollübergabe' beendet. Einige Zuweisungen mögen statische Variablen verändern. Das zieht zwangsläufig eine Aktualisierung der Einträge in der Werteliste `vs` der statischen Variablen nach sich, so daß ein neuer spezialisierter Programmpunkt die Form (`pp'`, `vs'`) hat. Die Menge der Nachfolger hängt natürlich von der Form der Kontrollübergabe ab. Die Menge der möglichen Nachfolger wird mit `successors(pp,vs)` bezeichnet. Sie hat zwei Elemente für eine dynamische Bedingung, keines für ein `return` und sonst eines. Für das Beispiel aus Abschnitt 1.6 ergibt sich daraus:

```
successors (search, (z,(x y z))) = { (cont,   (z,(x y z))) }
successors (search, (z,(z)))     = { (found,  (z,(z))) }
successors (cont,   (z,(x y z))) = { (search, (z,(y z))) }
```

usw.

Aus der Kombination der Codeerzeugung mit der Berechnung von ***poly*** ergibt sich eine erste Annäherung an die allgemeine Struktur des Programmspezialisierers ***mix***:

```
poly := { (pp0,vs0) };
while poly enthält ein nicht markiertes Paar (pp,vs) do
begin
  markiere (pp,vs);
  generiere Code für den Basisblock, der mit pp beginnt
    (unter Benutzung der Werte in vs)
  poly := poly ∪ successors (pp,vs)
end
```

In der nachfolgenden Tabelle sind die Regeln für die Berechnung der Nachfolger angegeben:

Kontrollkomponente von pp	*successors (pp, vs)*	
return	{ }	
goto pp′	{ (pp′, vs′) }	
if exp **goto** pp′ **else** pp′′	{ (pp′, vs′) }	wenn exp true ist
	{ (pp′′, vs′) }	wenn exp false ist
	{ (pp′, vs′), (pp′′, vs′) }	wenn exp dynamisch ist

2.6 Übergangskomprimierung

Wenn das Beispielsubjektprogramm ganz genau nach dem oben gegebenen Algorithmus spezialisiert wird, ergibt sich folgendes residuales Programm:

```
(search, (z,(x y z))):  goto (cont, (z,(x y z)));
(cont,   (z,(x y z))):  valuelist := tl(valuelist);
                        goto (search, (z,(y z)));
(search, (z,(y z))):    goto (cont, (z,(y z)));
(cont,   (z,(y z))):    valuelist := tl(valuelist);
                        goto (search, (z,(z)));
(search, (z,(z))):      goto (found, (z,(z)));
(found,  (z,(z))):      value := hd(valuelist);
```

Das Ergebnis ist zwar korrekt, aber nicht sehr befriedigend, da es eine Reihe überflüssiger `goto`'s enthält. Eine Möglichkeit, überflüssige `goto`'s zu beseitigen, besteht darin, Anweisungen der Form `goto pp` durch Kopien des Basisblocks mit dem Label `pp` zu ersetzen. Diese Technik wird *Übergangskomprimierung* genannt.

Mit Hilfe der Übergangskomprimierung ergibt sich für das Beispiel das folgende residuale Programm:

```
(search, (z,(x y z))):  valuelist := tl(valuelist);
                        valuelist := tl(valuelist);
                        value     := hd(valuelist);
```

Die Vorteile der Komprimierung sind offensichtlich: der Code wird klarer und effizienter. Zusammengesetzte Label sollten noch durch einen einfachen Identifizierer, eine Zahl oder einen Namen, ersetzt werden.

Die unkritische Verwendung der Übergangskomprimierung birgt allerdings zwei Fallen: *Codeduplizierung* und *unendliche Komprimierung*.

Codeduplizierung kann auftreten, wenn mehrere unterschiedliche Übergänge zum gleichen Programmpunkt komprimiert werden. Enthält das residuale Programm eine Schleife, kann die Komprimierung sogar unendlich sein.

Führt man die Übergangskomprimierung als separate Phase nach der Generierung des residualen Programms durch, können die oben genannten Probleme leicht vermieden werden. Es kann ein Programmflußplan erzeugt und analysiert werden, um festzustellen, welche Übergänge sicher komprimiert werden können. In der Praxis ist es allerdings wünschenswert, die Komprimierung während der Codeerzeugung durchzuführen, da es effizienter ist, erst gar keine überflüssigen `goto`'s, die anschließend nur wieder entfernt werden müssen, zu erzeugen. Die begleitende Komprimierung erschwert allerdings die Gewährleistung einer sicheren Komprimierung. In Kapitel 5 von [1] wird diese Problematik genauer untersucht.

An dieser Stelle wird eine einfache Strategie verwendet: *Alle* Übergänge, die nicht Teil einer residualen Bedingung sind, werden komprimiert. Residuale Bedingungen sind Bedingungen, die dynamische Variablen enthalten. Diese Strategie bewirkt einige Codeduplizierungen, aber die Erfahrung zeigt, daß dies kaum ein Problem darstellt. Wichtiger ist, daß diese Strategie keine Endlosschleifen verursacht, außer wenn das Subjektprogramm selbst eine potentielle Endlosschleife enthält, d.h., daß allein die statischen Daten, unabhängig von den dynamsichen Daten, eine Endlosschleife bewirken können.

2.7 Einteilung (der Variablen)

Die Aufgabe, Variablen als statisch oder dynamisch zu klassifizieren, ist schwieriger als es zunächst scheint. Eine naheliegende Strategie wäre es, alle Variablen, denen aus Konstanten und statischen Eingabedaten berechnete Werte zugewiesen werden, als statisch zu bezeichnen. Wie sich jedoch an einem einfachen Beispiel zeigen läßt, kann diese Strategie dazu führen, daß der Programmspezialisierer in eine Endlosschleife gerät:

```
Iterate:  if Y ≠ 0 then begin
            X := X + 1; Y := Y - 1; goto iterate;
          end;
```

Dieses unscheinbare Programm enthält zwei Variablen, `X` und `Y`. Wenn der Anfangswert von `X` bekannt - z.B. 0 - und der von `Y` unbekannt ist, erscheint es plausibel, `X` als statisch und `Y` als dynamisch zu klassifizieren, da der Wert des Ausdrucks `X+1` zur Spezialisierungszeit berechnet werden kann. Leider funktioniert das nicht.

Zu Beginn enthält ***poly*** den spezialisierten Programmpunkt (`iterate`, `0`). Mischt man wie in der Praxis üblich die Berechnung von ***poly*** mit der Codeerzeugung, so ergibt sich:

```
(iterate, 0): if Y ≠ 0 then begin
                Y := Y - 1; goto (iterate, 1);
              end;
```

Daraus folgt unmittelbar, daß (`iterate`, `1`) zu ***poly*** hinzugefügt werden muß. Also muß dem residualen Programm der Code

```
(iterate, 1): if Y ≠ 0 then begin
                Y := Y - 1; goto (iterate, 2);
              end;
```

hinzugefügt werden. Und so geht es bis in alle Ewigkeit weiter. Die Menge ***poly*** wird also unendlich:

poly = { (`iterate`, `0`), (`iterate`, `1`), (`iterate`, `2`), ... }

Das Problem entsteht dadurch, daß der Wert der Variablen `x` zwar bekannt ist, aber unter dynamischer Kontrolle berechnet wird. Daher müssen solche Variablen ebenfalls als dynamisch klassifiziert werden. Eine Einteilung, die die Endlichkeit von ***poly*** garantiert, wird selbst endlich genannt.

Der Prozeß, eine Variable `x` als dynamisch zu klassifizieren, auch wenn die Kongruenz eine Klassifizierung als statisch zugelassen hätte, wird *Generalisierung* genannt. Generalisierung kann zur Verhinderung nicht-terminierender partieller Auswertung erforderlich sein.

2.8 Bindezeitanalyse

Läßt man die Endlichkeitsproblematik außer acht und nimmt zudem an, daß dieselbe Einteilung der Variablen für jeden Programmpunkt anwendbar ist, so läßt sich die Einteilung *aller* Programmvariablen bei gegebener Einteilung der Eingabevariablen auf einfache Weise berechnen. Dieser Prozeß wird *Bindezeitanalyse* genannt, da er feststellt, zu welcher Zeit der Wert einer Variablen berechnet werden kann, d.h., wann der Wert an die Variable gebunden werden kann.

Seien $\mathtt{X}_1$, $\mathtt{X}_2$, ... $\mathtt{X}_\mathtt{N}$ alle Programmvariablen und $\mathtt{X}_1$, $\mathtt{X}_2$, ... $\mathtt{X}_\mathtt{n}$ die Eingabevariablen, wobei $0 \leq \mathtt{n} \leq \mathtt{N}$ gilt, und sei eine Anfangseinteilung $b_1, \ldots b_n$ der Eingabevariablen gegeben, wobei jedes b_j entweder den Wert S (statisch) oder D (dynamisch) hat.

Eine kongruente Einteilung $B = (b_1, b_2, \ldots b_N)$ aller Programmvariablen läßt sich dann wie folgt berechnen:

1. Konstruktion der anfänglichen Einteilung $B = (b_1, \ldots b_n, S, \ldots S)$
2. Enthält das Programm eine Zuweisung
   $\mathtt{X}_\mathtt{k}$ `:= exp`
   wobei Variable $\mathtt{X}_\mathtt{j}$ in `exp` vorkommt und $b_j = D$ in B gilt, dann wird $b_k = D$ in B gesetzt.
3. Wiederholung von Schritt 2, solange bis sich B nicht mehr ändert. Der Algorithmus endet dann mit einer kongruenten Einteilung B.

2.9 Algorithmus für den Programmspezialisierer *mix*

Nachdem nun alle für die Programmspezialisierung benötigten Techniken beschrieben wurden, kann eine mögliche Struktur eines Algorithmus für einen Programmspezialisierer, der diese Techniken aufeinanderfolgend anwendet, angegeben werden:

Eingabe Ein Subjektprogramm, eine Einteilung seiner Variablen in statische und dynamische und ein Teil der Eingabedaten

Ausgabe Ein residuales Programm

Algorithmus
- Berechnung von ***poly*** und Codeerzeugung für alle spezialisierten Programmpunkte in ***poly***
- Übergangskomprimierung des Codes
- Umbenennung der spezialisierten Programmpunkte

Diese Struktur spiegelt die Prinzipien der Programmspezialisierung gut wider, ist aber in der Praxis ineffizient, da zunächst ein großes residuales Programm erzeugt und erst in einem folgenden Schritt die reduzierte endgültige Fassung erzeugt wird.

Einen effizienteren Algorithmus für den Programmspezialisierer ***mix*** erhält man, wenn die Phasen vermischt werden:

```
(* program   - Subjektprogramm *)
(* division  - Einteilung der Variablen *)
(* vs0       - Anfangswert des Variablenspeichers *)
read(program,division,vs0);
  pending := { (pp0,vs0) };  (* pp0 ist der Anfangsprogrammpunkt *)
  (* pending = Menge spez. Programmpunkte, Code noch nicht erzeugt *)
  marked := { };
  (* marked  = Menge spez. Programmpunkte, Code schon erzeugt *)
  while pending ≠ { } do begin
    Wähle Element (pp,vs) aus pending und entferne es;
    marked := marked ∪ { (pp,vs) };
    bb := lookup(pp,program);     (* Mit pp markierten Basisblock in *)
                                  (* program finden *)
    code := initial_code(pp,vs); (* Leerer Basisblock mit Markierung *)
                                  (* (pp,vs) *)
    while bb is not empty do begin
      command := first_command(bb);
      bb := rest(bb);
      case command of
        X :=exp:
          if X is classified as static by division
            then vs := vs [X ↦ eval(exp,vs)]; (* Statische Zuweisung *)
            else code := extend(code, X := reduce(exp,vs));
                                           (* Dynamische Zuweisung *)
        goto pp':
          bb := lookup(pp',program);         (* Übergangskomprimierung *)
        if exp then goto pp' else pp'':
          if exp is static by division
            then begin                       (* Statische Bedingung *)
              if eval(exp,vs) = true
                then bb := lookup(pp',program);  (* Komprimierung *)
                else bb := lookup(pp'',program); (* Komprimierung *)
            end
            else begin                       (* Dynamische Bedingung *)
              pending := pending ∪ ( { (pp',vs) } \ marked );
              pending := pending ∪ ( { (pp'',vs) } \ marked );
              code := extend(code, if reduce(exp,vs)
                                   goto (pp',vs)
                                   else (pp'',vs));
            end;
        return exp:
          code := extend(code, return reduce(exp,vs));
        otherwise error;
    end; (* while bb is not empty *)
    residual := extend(residual,code); (* Neuen Basisblock anfügen *)
  end (* while pending ≠ { }  *)
```

Der besseren Übersichtlichkeit halber fehlt die Umbenennung der Label der spezialisierten Programmpunkte (`pp`, `vs`).

3 Partielle Auswertung und Compilierung

3.1 Futamura-Projektionen

Futamura erkannte als erster Forscher, daß mit Hilfe der Selbstanwendung partieller Auswertung grundsätzlich Compiler generiert werden können. Daher werden die Gleichungen, die Compilierung, Compiler und Compilergenerierung beschreiben, *Futamura-Projektionen* genannt:

1. Futamura-Projektion (Compilierung): **target = $||$mix$||_L$ [int, source program]**
2. Futamura-Projektion (Compiler): **compiler = $||$mix$||_L$ [mix, int]**
3. Futamura-Projektion (Compilergenerator): **cogen = $||$mix$||_L$ [mix, mix]**

Obwohl die Gleichungen einfach zu überprüfen sind, ist ihre intuitive Bedeutung ungleich schwerer nachzuvollziehen. Daher soll anhand eines konkreten Beispiels gezeigt werden, daß, als Folge der 1. Futamura-Projektion, ein partieller Auswerter zum Compilieren verwendet werden kann, wenn ein Interpreter und ein Quellprogramm gegeben sind.

Zunächst wird der Begriff des Interpreters genauer definiert und ein konkreter Interpreter angegeben. Anschließend wird die 1. Futamura-Projektion verifiziert.

In [1] wird darüber hinaus die Generierung eines Compilers als Folge der Anwendung der 2. Futamura-Projektion dargestellt.

3.2 Ein Interpreter für Turingmaschinenprogramme

Ein Interpreter für eine Quellsprache ***S***, der selbst in der Sprache ***L*** implementiert ist, hat zwei Eingaben: ein auszuführendes Quellprogramm ***p*** aus der Menge der *S*-Programme und die Eingabedaten ***d*** aus der Datenmenge ***D***. Die Ausführung des Interpreters mit den Eingaben ***p*** und ***d*** auf einer *L*-Maschine muß die gleichen Ergebnisse liefern wie die Ausführung des Programms ***p*** mit Eingabe ***d*** auf einer *S*-Maschine:

$$||p||_S\ d = ||int||_L\ [p,d]$$

Als konkreter Interpreter soll nun ein Interpreter für eine Turing-Maschine vorgestellt werden. Ein Turing-Programm Q ist eine Liste (I_0, I_1, ... I_n) von Instruktionen der Form

`right,  left,  write a,  goto i` oder `if a goto i`

Ein Berechnungszustand besteht aus der aktuellen Instruktion I, die als nächstes auszuführen ist, und einem unendlichen Band mit Zellen a_i: ... a_{-2} a_{-1} a_0 a_1 a_2 ... Das Bandalphabet A ist gegeben durch A = { 0, 1, B }, wobei B für 'blank' (Leerzeichen) steht. Nur eine endliche Anzahl von Zellen enthalten Symbole a_i ungleich B; a_0 wird aktuelle Position genannt. Die Instruktionen haben nachfolgend beschriebene Wirkungen:

- `write a` ändert a_0 auf `a`
- `right` und `left` ändern die Position
- `if a goto i` setzt den nächsten Kontrollpunkt auf I_i, falls a_0 = `a`; in allen anderen Fällen ist die nächste Instruktion der nächste Kontrollpunkt.

Ein Beispielprogramm Q in der Turing-Sprache lautet:

```
0: if 0 goto 3
1: right
2: goto 0
3: write 1
```

Die Eingabe für dieses Programm ist $a_0a_1...a_n \in \{ 0, 1 \}^*$. In allen anderen Zellen enthält das Band zu Beginn ein B. Die Programmausgabe ist der Wert der Zellen a_0a_1... (bis zur letzten Zelle ungleich B) und wird erzeugt, wenn keine weitere Instruktion mehr auszuführen ist. Die Bandposition kann sich dabei von der am Anfang unterscheiden.

Das Beispielprogramm findet die erste 0 rechts von der Anfangsposition auf dem Eingangsband und wandelt sie in eine 1 um. Falls keine 0 gefunden wird, läuft es in einer Endlosschleife. Bei 110101 als Eingabe für Q, wird die Ausgabe 1101 erzeugt. Das ergibt sich daraus, daß die Position, an der die in eine 1 umgewandelte 0 gefunden wurde, nun die aktuelle Position a_0 ist. Das Band enthält $(a_{-2}a_{-1}\underline{a_0}a_1a_2a_3) = (11\underline{1}101)$.

Der Quellcode des Turing-Interpreters in der Flußdiagrammsprache lautet:

```
Read (Q, Right);
init:      Qtail := Q;
           Left := `();

loop:      if Qtail = `() goto stop else cont;
cont:      Instruction := first_instruction(Qtail);
           Qtail       := rest(Qtail);
           Operator    := hd(tl(Instruction));
           if Operator = `right goto do-right else cont1;
cont1:     if Operator = `left  goto do-left  else cont2;
cont2:     if Operator = `write goto do-write else cont3;
cont3:     if Operator = `goto  goto do-goto  else cont4;
cont4:     if Operator = `if    goto do-if    else error;

do-right:  Left      := cons(firstsym(Right), Left);
           Right     := tl(Right);
           goto loop;
do-left:   Right     := cons(firstsym(Left), Right);
           Left      := tl(Left);
           goto loop;
do-write:  Symbol    := hd(tl(tl(Instruction)));
           Right     := cons(Symbol,tl(Right));
           goto loop;
do-goto:   NextLabel := hd(tl(tl(Instruction)));
           Qtail     := new_tail(NextLabel, Q);
           goto loop;
do-if:     Symbol    := hd(tl(tl(Instruction)));
           NextLabel := hd(tl(tl(tl(tl(Instruction)))));
           if Symbol = firstsym(Right) goto jump else loop;

jump:      Qtail     := new_tail(NextLabel,Q);

error:     return(`syntax-error: Instruction);

stop:      return Right;
```

Das gesamte zu interpretierende Programm wird in der Variablen `Q` gespeichert. Ein Programmkontrollpunkt wird durch ein Suffix `Qtail` von `Q` repräsentiert, entspricht also der Liste der noch auszuführenden Instruktionen. Das Band wird durch die Variablen `Left` und `Right` mit Werten aus A* repräsentiert, wobei `Right` $a_0a_1a_2$... (bis zur ersten Zelle ungleich B) und `Left` $a_{-1}a_{-2}a_{-3}$... entspricht. Zu beachten istt, daß für `Left` die Reihenfolge umgekehrt wurde.

Der Interpreter verwendet einige spezielle Basisfunktionen:

`new_tail`	bekommt als Argumente einen Label `lab` und das Programm `Q` und gibt den Teil (Suffix) des Programms zurück, der mit Label `lab` beginnt.
`first_instruction`	gibt die erste Instruktion aus einer Instruktionenfolge zurück.
`rest`	gibt alles bis auf die erste Instruktion aus einer Instruktionenfolge zurück.
`firstsym`	ist eine spezielle Version von `hd`, für die `firstsym () = B` gilt. Es wird `tl () = ()` angenommen.

Für das Beispielprogramm gilt `Q = (0: if 0 goto 3 1: right 2: goto 0 3: write 1)`. Einige typische Werte für diese Hilfsfunktionen sind dann:

```
new_tail(2, Q)        = (2: goto 0  3: write 1)
first_instruction(Q)  = (0: if 0 goto 3)
rest(Q)               = (1: right  2: goto 0  3: write 1)
```

Für jede ausgeführte Anweisung von Q muß der Turing-Interpreter zwischen 15 und 28 Operationen ausführen, wobei eine Operation für jede Zuweisung, jedes `goto` und jeden Aufruf einer Basisfunktion gezählt wird.

3.3 Compilierung durch die erste Futamura-Projektion

Die Gültigkeit der ersten Futamura-Projektion, die besagt, daß die Spezialisierung eines Interpreters in Bezug auf ein Quellprogramm den Effekt der Compilierung hat, kann leicht formal bewiesen werden:

Sei int ein *S*-Interpreter geschrieben in *L*, sei s ein *S*-Programm und d seine Eingabedaten, dann gilt:

$\|\|s\|\|_S\ d$	$= \|\|int\|\|_L\ [s, d]$	nach Definition des Interpreters
	$= \|\|(\|\|mix\|\|_L\ [int, s]\ )\|\|_L\ d$	nach der mix Gleichung
	$= \|\|target\|\|_L\ d$	nach Benennung des residualen Programms als target

Die Gleichungen sagen nichts über die Qualität des Zielprogramms aus, aber in der Praxis kann sie recht hoch sein.

Die erste Aufgabe auf dem Weg zur Spezialisierung des Turing-Interpreters in Bezug auf das Beispielprogramm ist es, eine Einteilung der Variablen des Interpreters unter der Annahme zu ermitteln, daß das zu interpretierende Programm Q statisch und sein anfängliches Eingabeband (`Right`) dynamisch ist. Relativ leicht ist zu sehen, daß dann folgende Variablen statisch sind: `Q`, `Qtail`, `Instruction`, `Operator`, `Symbol`, `NextLabel`. Dagegen sind `Right` und `Left` dynamisch. Diese Information wird dem Programmspezialisierer zusammen mit dem Text des Interpreters und dem Quellprogramm eingegeben.

Als Endergebnis ergibt sich aus dem obigen Interpreter und dem Quellprogramm `s =(0: if 0 goto 3 1: right 2: goto 0 3: write 1)` folgendes Zielprogramm:

```
read (Right);
lab0:  Left := '();
       if '0 = firstsym (Right) goto lab2 else lab1;
lab1:  Left := cons (firstsym (Right), Left);
       Right := tl (Right);
       if '0 = firstsym (Right) goto lab2 else lab1;
lab2:  Right := cons ('1, tl (Right));
       return (Right);
```

Alle Zuweisungen `X := exp` und alle Tests `if exp ...`, bei denen `X` statisch ist, werden eliminiert, da sie zur Programmspezialisierungszeit ausgeführt werden. Die Label `lab0`, `lab1` und `lab2` sind Aliasnamen für spezialisierte Programmpunkte (`pp`, `vs`), wobei `pp` ein Interpreter-Label ist und `vs` die Werte der statischen Interpreter-Variablen enthält.

In der nachfolgenden Tabelle wird die Zuordnung der Label im Zielprogramm zu spezialisierten Programmpunkten aufgelistet. Da die Variable `Q` an jedem Programmpunkt das gesamte Turing-Programm als Wert enthält, wurde sie weggelassen. Die Variable `Operator` wurde aus Platzgründen weggelassen. Mit `()` sind die Werte der nicht initialisierten Variablen bezeichnet.

Ziel-Label	*Interpreter-Label*	*Statische Interpreter-Variablen* (`vs`)			
		Instruction	Qtail	Symbol	NextLabel
lab0	init	( )	( )	( )	( )
lab1	cont	right	(2: goto 0 3: write 1)	0	3
lab2	jump	if 0 goto 3	(1: right 2: goto 0 3: write 1)	0	3

Aus den Gleichungen der Futamura-Projektion folgt direkt, daß das Zielprogramm in derselben Sprache wie der Interpreter geschrieben ist. Die Struktur des Zielprogramms erinnert jedoch mehr an das Quellprogramm, von dem es abgeleitet wurde, als an die des Interpreters. Das Zielprogramm ist aus Teilen des Interpreters zusammengesetzt, z.B. `Left := cons (firstsym(Right), Left)`, wobei einige Teile in Bezug auf Daten des Quellprogramms spezialisiert sind, z.B. `if '0 = firstsym(Right) goto lab2 else lab1`. Das ist typisch für Zielprogramme, die mit dem Programmspezialisierer ***mix*** erzeugt wurden.

Eine Laufzeitanalyse des Zielprogramms ergibt, daß die Hauptschleife im Zielprogramm 8 Operationen erfordert, während der Interpreter dafür 61 Operationen benötigt. Das Zielprogramm läuft also fast 8 mal schneller als der Interpreter.

4 Offene Fragestellungen

4.1 Terminierung

Probleme bei der partiellen Auswertung im Hinblick auf die Terminierungseigenschaften sind bei den bisherigen Ausführungen zumeist nur am Rande erwähnt und ansonsten ignoriert worden, da das Ziel die Vermittlung von Programmierkonzepten war. Eine formalere Darstellung hätte das Verständnis der Basisalgorithmen erschwert. Es wurde nur sichergestellt, daß die behandelten Programme *häufig genug* terminieren. In Kapitel 14 von [1] werden die Probleme und Lösungen bezüglich der Terminierung ausführlich diskutiert.

4.2 Endlichkeit der Menge der spezialisierten Programmpunkte

Wie in Abschnitt 2.7 an einem Beispiel gezeigt, ist es keine triviale Aufgabe, die Endlichkeit von ***poly*** zu garantieren. Bei genauerer Untersuchung zeigt sich, daß das Problem, eine genügende Anzahl von Variablen als dynamisch zu klassifizieren ohne zu einer unnötig großen Anzahl zu gelangen, nicht berechenbar ist. Diese Problemstellung wird ebenfalls in Kapitel 14 von [1] behandelt.

4.3 Online/Offline-Strategien

Die partielle Auswertung wurde als ein Prozeß beschrieben, der zwei (oder mehr) Stufen hat. *Zuerst* wird eine Einteilung B aus dem Programm und der anfänglichen Einteilung B' berechnet, *ohne* Gebrauch von den konkreten Werten der statischen Eingabevariablen zu machen. *Danach* findet die eigentliche Programmspezialisierung statt, wobei die statischen Variablen entsprechend der Einteilung ausgenutzt werden, nicht jedoch die konkreten, während der Spezialisierung berechneten Werte. Diese Vorgehensweise wird partielle *Offline*-Auswertung genannt. Wenn jedoch die konkreten Werte, die während der Programmspezialisierung berechnet werden, die ausgeführten Aktionen beeinflussen können, spricht man von partieller *Online*-Auswertung.

Fast alle *Offline*-Strategien, einschließlich der hier vorgestellten, stützen ihre Entscheidungen auf die Ergebnisse eines Vorprozesses, der *Bindezeitanalyse*.

Viele partielle Auswerter mischen *Online*- und *Offline*-Methoden, da beide Strategien ihre Vorteile haben. Der Hauptvorteil der partiellen *Online*-Auswertung ist, daß sie während der Spezialisierung manchmal mehr statische Informationen herausfinden kann. Daraus ergeben sich in der Regel bessere residuale Programme. *Offline*-Techniken dagegen machen die Generierung von Compilern usw. durch Selbstanwendung handhabbarer und führen durch die Verwendung einfacherer Spezialisierungsalgorithmen zu schnelleren Systemen.

Kapitel 7 von [1] enthält eine detaillierteren Vergleich von partieller *Online*- und *Offline*-Auswertung.

5 Zusammenfassung und Ausblick

Partielle Auswertung kann durchaus mit einem temperamentvollen Pferd verglichen werden: Richtig behandelt, können eindrucksvolle Ergebnisse erzielt werden. Es wurde aufgezeigt, daß die Programmtransformation vollautomatisch, ohne Eingriff des Anwenders, durchgeführt werden kann und daß die Methoden sich auch für die partielle Auswertung imperativer Sprachen eignen.

Vergleicht man die Flußdiagrammsprache jedoch mit einer Programmiersprache wie etwa *Pascal*, fällt auf, daß wichtige Sprachbestandteile wie z.B. ein Typkonzept oder Funktionsaufrufe fehlen. Arbeiten, die sich mit der partiellen Auswertung gebräuchlicher imperativer Sprachen wie *C*, *Fortran* oder *Pascal* beschäftigen (u.a. [4], [5], [6], [7], [8], [9]), belegen, daß sich diese Bestandteile relativ leicht in das Konzept der partiellen Auswertung integrieren lassen.

Mit Erfolg konnten die Techniken der Programmspezialisierung bereits in ganz unterschiedlichen Anwendungsfeldern eingesetzt werden:

- **Computer-Grafik mit Ray-Tracing**: ein in effizientem *C* geschriebener Ray-Tracer lief nach der Spezialisierung in Bezug auf die zu zeichnende Szene 1,5 bis 3 mal schneller. Bei den oft stundenlangen Berechnungen ist das eine durchaus bedeutsame Verbesserung.
- **Schnelle Fourier-Transformation**: ein in *Fortran* geschriebenes FFT Programm wurde in Bezug auf eine feste Funktion und Anzahl von Punkten spezialisiert. Die spezialisierte Programmversion lief 1,5 bis 4 mal schneller als das Originalprogramm.

Näheres hierzu findet man im Internet [14].

Bislang wurden solche Erfolge fast ausschließlich in akademischen Projekten erzielt, aber insbesondere für die imperativen Sprachen *C* und *Fortran* existieren bereits Systeme, die recht vielversprechend für den industriellen Einsatz erscheinen.

A Literaturverzeichnis

A.1 Basisliteratur

[1] N.D. Jones, C.K. Gomard, P. Sestoft: Partial Evaluation and Automatic Program Transformation; Prentice-Hall International, 1993

[2] R.H. Güting, M. Erwig: Übersetzerbau, Kurs 1810, Fernuniversität Hagen, 1997

[3] U. Meyer: Partielle Auswertung imperativer Sprachen; Bericht 9206, Arbeitsgruppe Informatik, Justus-Liebig-Universität Giessen, 1992

A.2 Weiterführende Literatur

[4] *Partielle Auswertung für eine Untermenge von Lisp:*
L. Lombardi, B. Raphael: Lisp as the language for an incremental computer; in: D.G. Bobrow, E.C. Bekeley, Editor, The programming Language Lisp, Ist Operation and Applications, MIT Press, 1964

[5] *Partielle Auswertung für imperative Sprachen:*
A.P. Ershov: On mixed computation: Informal account of the strict and polyvariant computational schemes; in: M. Broy, Editor, Control Flow and Data Flow: Concepts of Distributed Programming; Springer Verlag, 1984

[6] *Partielle Auswertung für eine Untermenge von C:*
L.O. Andersen: Self-applicable C program specialization; in: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Transformation, pp. 54-61, San Francisco, 1992, Yale University, Dept. Of Computer Science (YALEU/DCS/RR-909)

[7] *Partielle Auswertung für eine C-ähnliche Sprache:*
V. Nirkhe, W. Pugh: Partial evaluation of high-level imperative programming languages with applications in hard real-time systems; in: Nineteenth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, Januar 1992, ACM 1992

[8] *Partielle Auswertung für Fortran:*
P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, R. Glück: Fortran Program Specialization; in: Semantikgestützte Analyse, Entwicklung und Generierung von Programmen, Bericht 9402, pp. 45-54, Arbeitsgruppe Informatik, Justus-Liebig-Universität Giessen, 1994

[9] *Partielle Auswertung für eine große Untermenge von Pascal:*
U. Meyer: Report on a partial evaluator for a large subset of Pascal; Bericht 9201, Arbeitsgruppe Informatik, Justus-Liebig-Universität Giessen, 1992

A.3 Links im Internet

[10] Forschungsberichte der Arbeitsgruppe Informatik der Justus-Liebig-Universität Giessen
`http://www.informatik.uni-giessen.de/reports.html`

[11] Forschungsprojekt *Partielle Auswertung imperativer Programme*
`http://www.informatik.uni-giessen.de/forschung/imp.html`

[12] Forschungsprojekt *Meta-Programmierung, Semantik, Interpretierer, Übersetzer ...*
`http://www.informatik.uni-giessen.de/forschung/meta.html`

[13] P. Sestoft *Partielle Auswertung und automatische Programmgenerierung*
`http://www.dina.kvl.dk/~sestoft/pebook/pebook.html`

[14] Überblick über partielle Auswertung
`http://www.diku.dk/research-groups/topps/activities/PartialEvaluation.html`