Visuelle Datenflusssprache mit Kombinatoren höherer Ordnung

Arne Bayer

Universität der Bundeswehr München
Fakultät für Informatik

Visuelle Datenflusssprache mit Kombinatoren höherer Ordnung

Arne Bayer

Dissertation
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)

1. Gutachter: Prof. Dr. Gunther Schmidt
2. Gutachter: Prof. Dr. Andy Schürr

Kolloquium: 26. November 2004

Zusammenfassung

Moderne Softwaretechnik (Software Engineering) befasst sich mit der professionellen Entwicklung von hochqualitativen Softwaresystemen. Softwarequalität und insbesondere die relative Korrektheit eines Programms kann mit Hilfe von formalen Verfahren auf der Basis einer Spezifikation bestimmt werden. Im besten Falle kann a posteriori auf der Grundlage eines fertigen Programms eine formale Verifikation durchgeführt werden.

Diese Arbeit untersucht eine konstruktive Herangehensweise, die Programme a priori korrekt erstellt, indem eine abstrakte Spezifikation sukzessive in ein ausführbares Programm überführt wird. Es wird eine graphische Notation und zugleich ein methodischer Ansatz zur Softwareentwicklung definiert, der die transformative Programmierung in den Vordergrund stellt. Die behandelte Thematik liegt dabei im Spannungsfeld mehrerer Aspekte wie Datenflussprogrammierung, Techniken des λ-Kalküls und des gegebenen Transformations-Rahmenwerks HOPS. Letzteres ist ein Entwicklungssystem zur visuellen Programmierung, das Gegenstand mehrerer Forschungsprojekte an der Universität der Bundeswehr München war.

Ziel der Arbeit ist es, einen nahtlosen Übergang zwischen den einzelnen Phasen der Softwareentwicklung wie Analyse, Entwurf und Implementierung mittels eines transformativen Ansatzes zu erreichen. Dies wird an einem umfangreichen Beispiel aus dem Gebiet der digitalen Signalverarbeitung dargestellt.

Danksagung

Viele Personen haben mich beim Verfassen dieser Arbeit unterstützt. In erster Linie möchte ich Prof. Dr. Gunther Schmidt für die Betreuung der Dissertation danken, der mir immer wieder wichtige Anregungen geliefert und Mut zugesprochen hat. Außerdem möchte ich meinen besonderen Dank Franz Schmalhofer aussprechen, der in einer altruistischen Weise die Arbeit genauestens gelesen hat und zu einem der wenigen Kenner der selbigen wurde.

Nicht zuletzt möchte ich meiner Familie herzlich danken, insbesondere meiner Mutter, die mir von Kindesalter an ideelle Werte vermittelt hat und mir während meiner Ausbildung jederzeit Rückhalt gab, meinem Vater, der mich in meiner Zielstrebigkeit prägte, meinem Bruder, der mich bei den letzten Schritten vor der Drucklegung unterstützt hat, sowie meiner Lebensgefährtin, die mir stets motivierend ihre Anerkennung versicherte.

Schließlich möchte ich Padma Eck meinen Dank aussprechen, die einer Dissertation mit eher trockener Thematik ein ansprechendes Äußeres zu verleihen wusste.

Abstract

Modern software engineering deals with professional development of high quality software systems. Software quality and especially relative correctness of programs based on a specification can be determined by formal methods. At best a formal verification can be performed a posteriori on the basis of a complete program.

This thesis investigates a constructive approach which generates programs correctly a priori by transforming an abstract specification successively into an executable program. A graphical notation together with a methodical approach for software development will be defined emphasizing transformational programming. The outlined topic covers areas of several aspects like data flow programming, techniques of the λ-calculus and the given transformation framework HOPS. The latter is a development system for visual programming which was subject of several research projects at the University of the Federal Armed Forces.

The aim of the thesis is to obtain a transformational approach for the seamless transition between the specific phases of software development like analysis, design, and implementation. This will be demonstrated by a substantial example in the field of digital signal processing.

INHALTSVERZEICHNIS

KAPITEL 1

Einführung

Programmiersprachen dienen zur Beschreibung von Programmen. Im engeren Sinne definieren sie eine Syntax, also die zulässigen Texte für Programme, die meist in einer (erweiterten) Backus-Naur-Form beschrieben wird. Doch oft ist mit einer Sprache auch deren Semantik verbunden, d. h. deren Bedeutung, wie Programme auf einer Rechenanlage interpretiert und ausgeführt werden.

Angefangen bei einfachen, maschinennahen Sprachen, die mehr oder weniger direkt die Rechenstruktur erkennen ließen, entwickelten sie sich zu höheren, problemorientierten Sprachen, die durch geeignete Abstraktionen eine bestimmte Programmiermethodik unterstützten. Ein gemeinsames Merkmal fast all dieser Sprachen war, dass sie eine textuelle Notation besaßen.

In der Vergangenheit bereits aufgezeigte Alternativen zur textuellen Darstellung versuchen, die hervorragende bildhafte Wahrnehmung des Menschen zu nutzen. Graphische Darstellungen erlauben, die Linearität und Sequenzialität der textuellen Notation aufzubrechen und Strukturen und Bezugnahmen explizit auszudrücken.

Diese Arbeit untersucht graphische Notationen für Programme und zugleich einen methodischen Ansatz zur Softwareentwicklung, der die transformative Programmierung in den Vordergrund stellt. Die untersuchten Themenkomplexe umfassen Aspekte wie

- Datenflussprogrammierung,
- Techniken des λ-Kalküls
- und das gegebene Transformations-Rahmenwerk HOPS.

Die transformative Herangehensweise zur Software-Entwicklung erfährt zur Zeit nicht zuletzt durch die Object Management Group (OMG) neue Impulse, die durch ihren Ansatz der *Model Driven Architecture* (MDA) auf eine schrittweise Verfeinerung einer abstrakten Spezifikation

setzt. Nicht ohne Grund wird MDA von PricewaterhouseCooper (PwC) als eine Schlüsseltechnologie bezeichnet, die geeignet ist, den Softwareentwurf und den Implementierungsprozess in den nächsten Jahren zu revolutionieren[1].

Die Arbeit setzt grundlegende Kenntnisse in den genannten drei Gebieten voraus. Es gibt umfangreiche Literatur zur Datenflussprogrammierung. Das HOPS-System ist in mehreren Diplom- und Doktor-Arbeiten ausführlich beschrieben worden. Schließlich sind dem Logiker viele Elemente des λ-Kalküls und die Wirksamkeit von Kombinatoren im Prinzip bekannt. Anhand von Beispielen wird knapp an die Fakten erinnert.

Ziel der Arbeit ist es, ein Zusammenwirken der drei Bereiche zu untersuchen, insbesondere die Einbettung des Paradigmas der Datenflussprogrammierung in HOPS. In dem Entwicklungssystem sollen neue Bausteine definiert und neue Regelsätze aufgenommen werden, die für den jeweiligen Anwendungsfall adäquat erscheinen.

Die Darstellung der erzielten Resultate auf Papier kann verständlicherweise nicht an allen Stellen die Dynamik und Interaktivität des Systems widerspiegeln. Aus Gründen einer kompakten Darstellung werden Programme entlang der Linien des λ-Kalküls abgebildet, wo dies angemessen erscheint. An anderer Stelle werden auseinander hervorgehende HOPS-Graphen gezeigt, jedoch aus Platzgründen nicht mit allen Zwischenschritten. Im Prinzip liegt dort jeweils ein weiträumiger Übergang anhand der Regelsysteme vor, der in seiner Schematizität oft automatisch erfolgen kann und auch soll.

Hinzuweisen ist darauf, dass es zwei Linien von Fallbeispielen gibt, die das exemplarische Vorgehen durchgehend stützen sollen.

- Die häufig genutzte Fakultätsfunktion veranschaulicht wie stets in der Literatur einfachere Wirkungen von λ-Kalkül oder Kombinatoren — sie ist keinesfalls selbst Gegenstand der Untersuchung.

- Im Bereich der digitalen Signalverarbeitung (DSP) wird ein großes running example vorgeführt. Erkenntnisse und Ergebnisse werden aus der Sicht des Informatikers unter Berücksichtigung von dessen Werkzeugen und Notationen präsentiert. Das Beispiel ist komplex genug, um ein Prüfstein für die Methoden zu sein. Auch ist die Umsetzung von der Darstellungsweise der Elektrotechnik in diejenige der Informatik neu.

Die Untersuchung des Zusammenwirkens der drei Kernbereiche wird mit der Darstellung verschiedener Programmierparadigmen in Kapitel 2 vorbereitet. Dabei werden neben visuellen Datenflusssprachen auch funktionale Sprachen erörtert, die zumindest auf akademischem Gebiet Bedeutung erlangt haben.

Kapitel 3 beschreibt das Programm- und Transformationssystem HOPS, das seit mehr als zehn Jahren am Institut für Softwaretechnologie der Universität der Bundeswehr München Gegenstand der Forschung war. HOPS ist eine graphische Software-Entwicklungsumgebung, die es

[1] aus *Technology Forecast: 2002–2004, Band 1: Navigating the Future of Software*

dem Benutzer ermöglicht, eine abstrakte Spezifikation einer Anforderung mittels semantikerhaltender Regeln zu transformieren. Dazu gehört auch die Erzeugung von ausführbarem Code. Der Abschnitt geht ein in die Programmiermethodik mit HOPS, die die Definition einer dem Anwendungsbereich geeigneten Sprache nebst einem dazu passenden Regelsatz umfasst. Das System wurde um eine Datenflusssicht erweitert, die eine alternative Darstellung eines Programmgraphen repräsentiert. Sie visualisiert anstatt der Struktur eines Graphen dessen Fluss der Daten.

Das HOPS-System erreicht seine Flexibilität durch einen regelbasierten Ansatz der Semantikzuordnung. Bausteine besitzen nicht a priori eine Bedeutung, sondern erlangen sie erst im Zusammenspiel mit definierenden Regeln. Der eingebaute Transformationsmechanismus kann nicht nur zur Definition von Bausteinen verwendet werden, sondern macht Programme semantikerhaltend veränderbar. Programme sind nicht weiter starr, sondern werden 'knetbar' und können beliebigen Optimierungen unterzogen werden.

Im Laufe der Arbeit stellte sich heraus, dass Transformationsregeln für Programme ohne Variablenbindungen leichter zu formulieren sind als für solche mit Bindungen. Einen Weg, um aus einem beliebigen Programm zu einem Programm ohne Bindungen zu gelangen, beschreibt Kapitel 4. Hier wird auf bekannte Kombinatoren wie **S**, **K** und **I** eingegangen, mit deren Hilfe ein Abstraktionsalgorithmus Programme bindungsfrei machen kann. In der Literatur sind darüberhinaus einige Optimierungen für diesen Algorithmus zu finden, die in dieser Arbeit erstmals in einen geschlossenen Transformationsalgorithmus Eingang finden. Ein weiterer Abschnitt beschäftigt sich mit der Frage, wie unter Ausnutzung einer alternativen Kombinatormenge, die für das Programmieren in HOPS typisch ist, ein bindungsfreies Programm erstellt werden kann. Programme ohne Variablenbindungen besitzen eine kanonische Datenflussdarstellung, bei der Funktionen kompositionell zusammengefügt werden können, ohne einen bestimmten Kontext, hier die Bindungen, berücksichtigen zu müssen. Einige Beispieltransformationen verdeutlichen diese Vorgehensweise.

Die vorgestellten Werkzeuge und Methoden sollen nun im Software-Entwicklungsprozess Verwendung finden. Die Erstellung von Programmen durchläuft in der Regel mehrere Phasen, die u. a. *Analyse*, *Entwurf* und *Codierung* umfasst (Abbildung 1.1). Heutige Entwicklungswerkzeuge unterstützen die verschiedenen Phasen unterschiedlich gut. Einen weiten Raum zur Verbesserung dieses Software-Zyklus bieten *rechnergestützte Phasenübergänge*, bei denen Brüche zwischen den einzelnen Phasendokumenten vermieden werden.

Abbildung 1.1: Phasenmodell mit inhomogenen Phasenübergängen

HOPS mit seinem Transformationsmechanismus bietet dem Entwickler die Möglichkeit, den Prozess durchgehend zu unterstützen. Dazu wird eine abstrakte Problembeschreibung in die HOPS-eigene interne Notation gebracht. Aufbauend auf dieser Notation können nachfolgend

automatische oder interaktive Transformationen angewandt werden, die eine Spezifikation in ein ausführbares Programm überführen. Anschließend emittieren spezielle Durchlaufalgorithmen einen Zielcode, der mit entsprechenden Compilern in ein ablauffähiges Programm übersetzt werden kann.

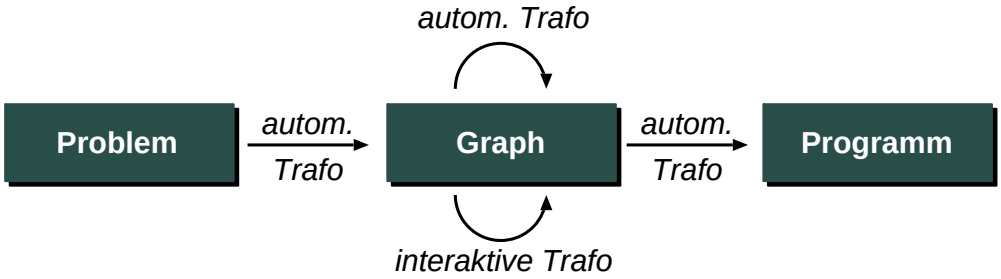


Abbildung 1.2: Entwicklungsmodell mit homogenen Phasenübergängen

Der Vorteil dieses Verfahrens liegt in den homogenen Übergängen von der Spezifikation des Problems bis hin zum fertigen Programm, wie dies in Abbildung 1.2 veranschaulicht ist.

Kapitel 5 verfolgt den beschriebenen Ansatz mit einem Beispiel aus der digitalen Signalverarbeitung. Hier wird eine kurze Einführung in den Anwendungsbereich gegeben und eine an den λ-Kalkül angelehnte Notation vorgeschlagen, die auf einen Zugang für Software-Entwicklung abgestimmt ist. Die Problemstellung des verwendeten Beispiels besteht darin, eine Differenzengleichung, die mathematisch das Verhalten eines Signalverarbeitungsoperators beschreibt, sukzessive in ein Blockdiagramm zu transformieren, das einem Datenflussgraphen einer Programmiersprache gleicht und zur Visualisierung von Operatoren der Signalverarbeitung besonders geeignet ist. Ein Abschnitt beschäftigt sich mit der Modellierung von Differenzengleichungen und Blockdiagrammen im System HOPS. Es wird eine domänenspezifische Beschreibungssprache für digitale Operatoren eingeführt und eine Regelmenge angegeben, welche eine abstrakte Spezifikation eines Signalverarbeitungsoperators in eine ausführbare Architektur überführt. Daraufhin erläutert ein Abschnitt, wie aus einem HOPS-Modell in der erwähnten Beschreibungssprache Programmtext in unterschiedlichen Zielsprachen erzeugt werden kann.

Den Abschluss der Arbeit bildet Kapitel 6 mit einem Vergleich mit bestehenden Ansätzen und einer Bewertung der vorgestellten Vorgehensweise.

KAPITEL 2

Ausgewählte Paradigmen von Programmiermodellen

Die Mehrzahl der kommerziellen Softwaresysteme sind in einer imperativen, textuellen Programmiersprache erstellt. Alternative Beschreibungssprachen und Programmiermodelle finden jedoch heute immer mehr Eingang in spezielle Anwendungsgebiete.

Dieses Kapitel umfasst einen kurzen Abriss ausgewählter Programmiermodelle und -paradigmen, auf denen die vorliegende Arbeit aufbaut bzw. zu denen die Arbeit thematisch in einem engen Zusammenhang steht. Darunter befinden sich Ansätze der visuellen Programmierung, der Datenflussprogrammierung sowie der funktionalen Programmierung.

Inhaltsverzeichnis

Die meisten (kommerziell) erfolgreichen Sprachen wie FORTRAN, COBOL, Ada, C oder Pascal gehören zu der *imperativen* Gattung: Ein Programm besteht aus einer Sequenz von Anweisungen, die hintereinander ausgeführt werden[1]. Zu Grunde liegt immer ein impliziter globaler Zustand zur Ausgabe auf einem Bildschirm oder Drucker bzw. zum Ändern von — nicht notwendigerweise globalen — Variablen. Sequenzielle Anweisungen können diesen Zustand ändern. An diesem Paradigma haben auch objektorientierte Sprachen wie C++, Java oder Smalltalk nichts grundlegend geändert. Hier kapseln nun einzelne Objekte ihren Zustand; Funktionen werden Klassen zugeordnet und heißen fortan Methoden.

Einen anderen Weg zeigt die *Logikprogrammierung* auf. Hier wird ein Problem durch eine Menge von Formeln beschrieben, und eine Inferenzmaschine sucht nach erfüllenden Belegungen der Formeln und somit nach möglichen Lösungen. Prominentester Vertreter dieser Sprachklasse ist Prolog (Programming in Logic).

Eine dritte Herangehensweise stellen *funktionale Sprachen* dar, die wie logikorientierte Sprachen zu der Klasse der deklarativen Sprachen zählen. Hier ist ein Programm aus Funktionsdefinitionen gestützt auf Ausdrücke aufgebaut. Im Gegensatz zu Logiksprachen, deren zu Grunde liegendes Modell die Relation ist, steht hier die Funktion im Vordergrund ([Hudak 1989, S. 361]). Ein herausragendes Merkmal funktionaler Sprachen ist die *referenzielle Transparenz*, die oft auch beschrieben wird mit: „Gleiches kann durch Gleiches ersetzt werden“.

In einem (Haskell-)Ausdruck

```
... x + x ...
  where x = f(a)
```

kann die Funktionsanwendung „f(a)“ für jedes freie Vorkommen von x substituiert werden, ohne möglicherweise das Resultat zu ändern. Der Unterschied liegt lediglich in unterschiedlicher Effizienz, wie oft ein gemeinsamer Teilausdruck ausgewertet wird. Dies ist in imperativen Sprachen weitaus schwieriger zu handhaben, da Funktionen (hier: f) Nebeneffekte bewirken können. Erst eine aufwändige statische Analyse kann manchmal zusichern, dass eine Funktion ohne Nebeneffekte ist. Rein funktionale Sprachen besitzen die Eigenschaft der referenziellen Transparenz a priori.

[1] Wir vernachlässigen hierbei hochoptimierende Compiler, die in der Lage sind, semantikerhaltende Umsortierungen von Anweisungen vorzunehmen.

2.1 Parallelverarbeitung

Parallelverarbeitung verwendende Algorithmen haben ihren Ursprung in den 60er Jahren, als erste pseudoparallele Systeme verfügbar wurden ([Louden 1994]). Bei diesen Verfahren werden mehrere Prozesse scheinbar simultan von einem Prozessor ausgeführt. Später kamen parallelverarbeitende Systeme auf den Markt, die echte Parallelität auf mehreren Prozessoren zur Verfügung stellten.

In der Folge entstanden Programmiersprachen, die nebenläufige Programmierung erlaubten, um die Möglichkeit der Parallelität auszudrücken. Ob ein nebenläufiges Programm tatsächlich parallel ausgeführt wird, liegt letztendlich in der Verantwortung des zugrunde liegenden Betriebssystems.

Die Unterstützung von Parallelität kann auf mehreren Ebenen erfolgen. So können hochoptimierende Compiler Programmstücke entdecken, die prinzipiell parallel ausgeführt werden können, ohne ihre Bedeutung zu ändern. Andererseits wurden bestehende Sprachen um parallele Konstrukte erweitert (z. B. ParC) oder Parallelität wurde in neu entworfenen Sprachen bereits im Entwurf berücksichtigt (z. B. Occam oder Java). In beiden Fällen hat der Programmierer explizit die Möglichkeit, in Programmen Parallelität zu modellieren. Die Parallelverarbeitung kann auf unterschiedlichen Ebenen, wie Anweisungen, Prozeduren oder ganzen Programmen, ablaufen.

Das Programmiermodell hat einen wesentlichen Einfluss auf Handhabung und Darstellbarkeit von Parallelverarbeitung. Wird eine imperative Sprache eingesetzt, die eine sequenzielle Abarbeitung der Anweisungen erfordert, liegt es in der Hand des Entwicklers, diese strenge Sequenzialisierung zu lockern, um Parallelität zu ermöglichen. Bei dem Modell der Datenflussberechnung hingegen spezifiziert der Programmierer nur die Datenabhängigkeiten, so dass viele parallele Abarbeitungen möglich sind, solange etwaige Abhängigkeiten berücksichtigt werden.

2.2 Datenfluss

Das Erstellen von Software beruht heute noch immer auf der Grundlage von formalen Programmiersprachen, da die Informatik noch nicht in der Lage ist, dass Computer die Bedeutung von natürlichsprachlichen Texten erkennen. Der vorherrschende Typ von Programmiersprachen sind textuelle Sprachen, die mittlerweile vielfach entworfen wurden ([Landin 1966]). Obwohl eine der ersten Programmsprachen, der Plankalkül von Zuse, Datenflusscharakter hatte, konnte sich dieser Ansatz bisher nicht durchsetzen.

Erst mit dem Beginn der Softwarekrise und der Steigerung der Rechenleistung von Computeranlagen suchte man intensiver nach alternativen Programmiermodellen.

Ein anderer Punkt, der in der Vergangenheit maßgeblich die Programmiermodelle beeinflusst hat, ist die zu Grunde liegende Architektur der Computer. Auch heute noch beruhen praktisch alle digitalen Rechenanlagen, die am Markt von Bedeutung sind, auf dem von-Neumann-Prinzip. So ist es nicht verwunderlich, dass dieses Prinzip Einfluss auf viele Programmiersprachen hatte.

Viele neuentworfene Sprachen verwenden noch immer die sequenzielle Abarbeitung von Anweisungen. Das Programm kennt einen globalen Zustand, auf dem Operationen in einer fest definierten Reihenfolge abgearbeitet werden.

Nachdem jedoch vermehrt Prozessoren mit Parallelverarbeitung mittels VLIW (Very Large Instruction Word) oder MIMD (Multiple Instruction Multiple Data) auf den Markt drängen, ist es an der Zeit, sich neuen Programmiermodellen zuzuwenden, die der neuen Hardwaregeneration gerecht werden. Das Modell des *Datenflusses* ist ein Weg zu mehr Parallelität, eine Datenflusssprache ist dessen Repräsentant.

2.2.1 Datenflussgraphen

Berechnungen wie

```
e := a + b
f := c − d
g := e ∗ f
h := (a + b) / g
```

können auf natürliche Weise in einen gerichteten Datenflussgraphen (Abbildung 2.1) überführt werden. Solche Graphen können als Beschreibungsmittel zur Modellierung von Datenabhängigkeiten aufgefasst werden, die Freiraum zur parallelen Ausführung lassen.

Der Graph repräsentiert lediglich eine partielle Ordnung bzgl. der Ausführungsreihenfolge im Gegensatz zu von-Neumann-Sprachen, die dem Programm typischerweise eine totale Ordnung auferlegen, die erst durch eine aufwändige Datenflussanalyse wieder „aufgeweicht“ werden kann.

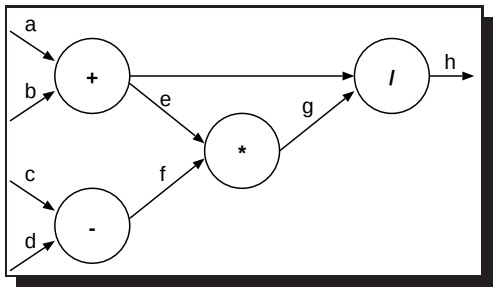


Abbildung 2.1: Einfacher Datenflussgraph

Ein allgemeiner Datenflussgraph umfasst Knoten, die Funktionen darstellen und Kanten, die den Datenfluss repräsentieren. Auf eingehenden Kanten zu einem Knoten fließen Eingabedaten, auf ausgehenden fließt das Funktionsergebnis. Oft werden die Dateneinheiten, die sich auf den Kanten befinden, Token genannt; der Knoten wird häufig mit einer Funktion oder Prozedur aus herkömmlichen Programmiersprachen gleichgesetzt.

2.2.2 Modelle der Datenflussberechnung

Datenflussgraphen bezeichnen zunächst einmal nur eine Syntax für bestimmte Ausdrücke. Von besonderem Interesse ist aber vor allem deren Semantik, d. h. was sie ausdrücken. In diesem Abschnitt stellen wir einige Modelle der Datenflussberechnung vor, die sich in den letzten Jah-

ren herauskristallisiert haben. Dazu zählen insbesondere *statische Datenflussgraphen*, *rekursive Programmgraphen* und das *Tagged Token Modell*, die in [Dennis 1984] kategorisiert sind.

Im Gegensatz zum von-Neumann-Modell existieren im Datenflussmodell mitunter viele Stellen, an denen die Abarbeitung gerade stattfindet oder stattfinden könnte. Es fehlt dementsprechend auch ein globaler Programmzähler. Stattdessen gibt es eine Menge von *Aktoren*, die **lokal** *Ereignisse* auslösen können. Da mehrere Aktoren zugleich feuern können, d. h. bereit sind, Teilergebnisse weiterzuleiten, entsteht eine asynchrone, parallele Berechnungsweise. Das Berechnungsergebnis ist dennoch determiniert, da das Ein-/Ausgabeverhalten unabhängig von der Reihenfolge der Aktivierungen der Aktoren ist.

Zunächst werden wir einige grundlegende Eigenschaften angeben, die allen Datenflussmodellen gemein sind.

Ein *Programmmodul* entspricht einer Funktion bzw. Prozedur in herkömmlichen Programmiersprachen. Dieses Modul wird durch einen kreisfreien, gerichteten Graphen repräsentiert, dessen Knoten *Aktoren* und dessen Kanten *Links* genannt werden. Auf Links werden Daten zwischen zwei Aktoren übermittelt. Eingehende und ausgehende Links auf Aktoren respektieren eine totale Ordnung. Erstere stellen Eingabelinks dar. Links, die bei Aktoren beginnen, sind Ausgabelinks.

Datenflussmodelle stellen im Gegensatz zu imperativen Sprachen nicht nur eine Definition der Berechnungsvorschrift dar, sondern beinhalten zusätzlich einen Zustand der aktuellen Auswertung, der durch *Token*, die sich auf den Links befinden, spezifiziert ist. So kann die Auswertung eines Datenflussgraphen durch eine Abfolge solcher Graphen beschrieben werden, auf denen an bestimmten Links Token eingefügt bzw. gelöscht werden. Siehe dazu Abbildung 2.2, welche einen Datenflussgraphen zeigt, in dem das Programm

$(x + y) * (y - z)$ *mit* x=7, y=4, z=2

sukzessive abgearbeitet wird[2].

Dabei gelten folgende Regeln zur Auswertung:

- Ein Aktor ist aktiviert, wenn auf jedem von ihm benötigten Eingabelink ein Token sitzt.
- Jeder beliebige aktivierte Aktor kann gefeuert werden, um den nächsten Zustand zu definieren.
- Ein Aktor wird gefeuert, indem von jedem Eingabelink ein Token entfernt und auf jedem Ausgabelink ein Token platziert wird.

Diese einfachen Graphen können nun durch bedingte Anweisungen erweitert werden, indem spezielle Aktoren wie ein *T-Gate* und eine *Weiche* (switch) eingeführt werden. Dazu ist es erforderlich, Datenströme zu partitionieren in Datenwerte und Kontrollwerte.

[2]Es sei hier darauf hingewiesen, dass Abbildung 2.1 lediglich ein Beschreibungsmittel ist, während Abbildung 2.2 bereits auf einer bestimmten Semantik beruht.

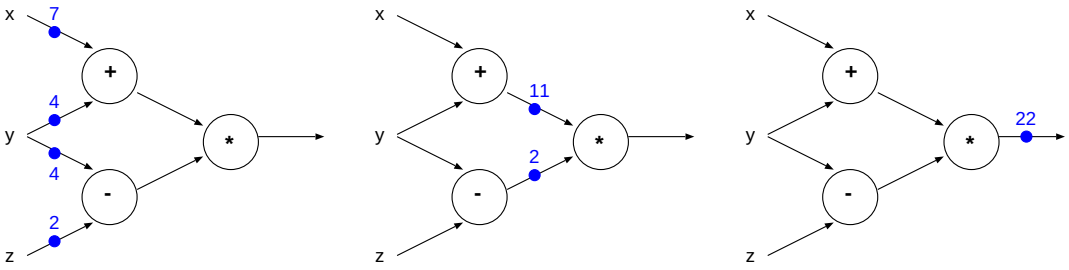


Abbildung 2.2: Mögliche sukzessive Auswertung eines Datenflussgraphen

Eine Weiche entspricht einem **if then else**-Konstrukt, das in Abhängigkeit von seinem Kontrollwert (**true** oder **false**) nur seinen linken bzw. rechten Datenstrom hindurchlässt. Das T-Gate hingegen befördert nur seinen Datenwert weiter, wenn der anliegende Kontrollwert **true** ist, sonst wird der Datenwert absorbiert.

Um nun ein Programmfragment mit Bedingung in einen Datenflussgraphen abzubilden, schauen wir uns Abbildung 2.3 näher an. Hier wird das Programm

```
if p(y) then f(x,y) else g(y) endif
```

in einem äquivalenten Datenflussgraphen dargestellt. Zu beachten ist der Einsatz des T-Gates, das nur den Wert von x weitertransportiert, wenn dieser auch von dem Aktor f benötigt wird.

Ein weiteres Konzept von Datenflussgraphen ist das der Applikation, welche durch einen speziellen Aktor implementiert wird. Operationell ist die Applikation definiert durch die Substitution des mit der Funktion parametrierten Aktors durch den Datenflussgraphen der ‘gerufenen’ Funktion. Dabei stehen die Ein- und Ausgabelinks in einer totalen Ordnung, die während der Substitution berücksichtigt wird. Wichtiges Merkmal ist die Tatsache, dass hierbei eine Art nicht-strikte Semantik eingeführt wird, da Werte von Funktionen berechnet werden können, ohne dass sämtliche Eingabeparameter bereits definiert sind.

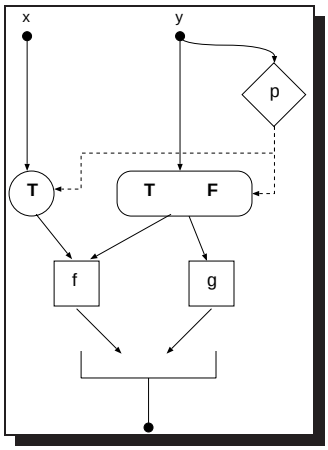


Abbildung 2.3: Datenflussgraph mit bedingter Anweisung

Die bisher beschriebenen Mechanismen weisen jedoch zwei Defizite auf: Zum einen werden Programmgraphen von praktischer Bedeutung unverhältnismäßig groß und zum anderen lassen sie keine Rekursion zu. Im Folgenden wollen wir nun auf drei spezifische Modelle der Datenflussberechnung eingehen, die die genannten Einschränkungen aufheben.

Statischer Datenflussgraph

Im Gegensatz zum Basismodell, in dem während der gesamten Berechnung auf einem Link höchstens ein Token existieren darf, können in einem statischen Datenflussgraphen **zu unterschiedlichen Zeiten** mehrere Token auf einem Link residieren, jedoch zu einem bestimmten Zeitpunkt höchstens eines. Dazu muss das Regelsystem zum Feuern eines Aktors um folgende Regel erweitert werden:

- Ein Aktor ist nur dann aktiviert, wenn auf keinem seiner Ausgabelinks ein Token sitzt.

Dieser Mechanismus ist eine wichtige Voraussetzung für Pipeline-Verarbeitung, da nun voneinander unabhängige Berechnungen einander folgen können und dadurch potenzielle Parallelität ausgenutzt werden kann.

Durch diese Parallelverarbeitung entsteht ein zusätzlicher Bedarf an Ablaufkontrolle, da sich Datenwerte nun überholen können. Eine weitere Möglichkeit der Koordination besteht in einem Verschmelzungsaktor (merge actor), der nur dann einen Datenwert weiterleitet, wenn auf dem Ausgabelink kein Token ist. Welcher Wert der beiden Eingabelinks transportiert werden soll, steuert ein zusätzlicher Kontrollwert.

Rekursiver Programmgraph

Rekursive Programmgraphen erlauben im Gegensatz zu statischen Datenflussgraphen keine Zyklen. Sie beschränken sich auf gerichtete, kreisfreie Graphen, bei denen auf einem Link höchstens ein Token während der gesamten Lebenszeit platziert sein darf.

Um nun dennoch Repetitionen ausdrücken zu können, die bei statischen Datenflussgraphen durch Zyklen beschrieben werden, führt man eine Endrekursion ein. Bei jeder Inkarnation einer Rekursion bzw. ganz allgemein bei jedem Aufruf einer Funktion fügt ein Applikationsaktor eine Kopie des Programmgraphen der Funktion in den rufenden Graphen ein.

Tagged-Token-Modell

Das Tagged-Token-Modell hebt eine wesentliche Einschränkung des rekursiven Programmgraphen wieder auf, demzufolge nun mehrere Token zu gleicher Zeit auf einem Link zugelassen sind. Damit ist es möglich, sowohl Rekursion wie auch Iteration mittels dieses erweiterten Modells auszudrücken.

Um verschiedene Token auf einem Link zu unterscheiden, werden sie mit einem fortlaufenden Index ausgestattet, der dem jeweiligen Iterationszyklus entspricht. Letztendlich wird das Regelsystem ersetzt durch folgende Bedingungen:

- Ein Aktor ist genau dann aktiviert, wenn es einen Index i gibt, so dass auf jedem Eingabelink ein Token mit Index i sitzt.
- Jeder aktivierte Aktor kann feuern.

- Während des Feuerns wird von jedem Eingabelink ein Token mit Index *i* entfernt und auf den Ausgabelinks jeweils ein Token mit einem Wert und Tag entsprechend des Aktortyps gesetzt.

2.2.3 Berechnungsmodelle

Wir haben im vorhergehenden Abschnitt verschiedene Graphtypen kennengelernt, die eine prinzipielle Aussage darüber machen, wann ein Knoten aktiviert ist. Nun wenden wir uns unterschiedlichen *Berechnungsmodellen* (computational models) zu. Bei einem spezifischen Modell steht die Frage im Vordergrund, an welchem Knoten ein Berechnungsschritt ausgeführt wird, vorausgesetzt, es sind mehrere Knoten aktiv. Wir werden zwei verschiedene Auswertungsstrategien vorstellen.

Applicative-Order Datenfluss

Eine typische Ausführungsreihenfolge für funktionale Sprachen ist die der applikativen Reihenfolge (engl. *applicative order*), die oft auch als datengetrieben (engl. *data driven*) bezeichnet wird. Hier werden nur Operationen respektive Knoten ausgeführt, deren Argumente alle bereitstehen.

Entscheidend bei der Semantik von Datenflussgraphen ist die Behandlung von (auch rekursiven) Funktionsaufrufen. Dazu führt [Field und Harrison 1988, S. 348] an, dass Instruktionen bzw. Knoten, die bereits ausgeführt wurden, gelöscht werden können, da ausschließlich DAGs behandelt werden.

Die Ausführung eines Applikationsknotens ist abhängig von seiner Umgebung (closure), die auf der ersten Eingabekante ansteht:

- Wenn die Applikation ausreicht, um die Umgebung zu vervollständigen, d. h. alle freien Variablen zu versorgen, wird eine **Kopie** des Funktionsrumpfes erzeugt, und die Applikation vollzogen.

- Wenn hingegen die Arität der Umgebung größer als eins ist, wird in der Umgebung die nächste freie Variable entsprechend mit dem Argument aus der Eingabekante besetzt. Das Ergebnis ist eine Kopie der alten Umgebung mit einer erweiterten Bindung.

Das Problem bei dieser Art der Implementierung ist das Einfügen einer neuen Instanz des Funktionskörpers in den bestehenden Graphen, welches eine sehr teure Operation darstellt. Abhilfe schafft hier eine stapelorientierte Implementierung, die in [Glaser und Hayes 1986] vorgeschlagen wird.

Normal-Order Datenfluss

Im Falle der applikativen Ausführung eines Datenflussgraphen werden nur Knoten ausgeführt, bei denen alle Operanden bereits ausgewertet sind, unabhängig von der Tatsache, ob diese auch benötigt werden. So können Berechnungen angestoßen werden, deren Ergebnisse gar nicht verwendet werden.

Diesem Umstand trägt der Normal-Order Datenfluss Rechnung. Hier werden Funktionen nur appliziert, wenn deren Resultat auch angefordert wird (engl. *demand driven*). Diese Anforderung wird im Falle des Datenflusses entgegen der Richtung der Datenflusskanten propagiert, in Abbildung 2.2 also von rechts nach links. Ausgehend von einer Funktion, deren Ergebnis verwendet wird, pflanzt sich eine Auswerteanforderung fort zu deren Operanden, welche wiederum Funktionsergebnisse darstellen können.

2.3 Visuelle Datenflusssprachen

Die Entwicklung von Programmiersprachen, die von Assembler über COBOL bis hin zu Smalltalk, Java und weiter reicht, hatte stets das Bestreben, das Programmieren zu vereinfachen. Fast allen Sprachen ist jedoch gemein, dass sie eine lineare Repräsentation haben. Anweisungen werden aneinander gereiht, und die Struktur ist eindimensional und textuell ([Shu 1988]).

Visuelle Programmierung versucht mit dieser Tradition zu brechen und einen völlig neuen Weg zu gehen. Chang unterscheidet in [Chang 1986] zwei Typen:

- Eine Sprache zur Verarbeitung visueller Informationen.
- Eine Sprache zur Programmierung mit visuellen Ausdrücken.

Während die erste Klasse Objekte verarbeitet, die inhärent eine visuelle Repräsentation besitzen, die Sprache selbst aber nicht unbedingt visuell sein muss [3], sind wir an der zweiten Klasse interessiert. Die Programmiersprache ist graphisch orientiert und verarbeitet Objekte, die nicht notwendigerweise eine kanonische graphische Darstellung haben.

Kombiniert man das Konzept der visuellen Sprachen mit der Datenflussprogrammierung, so erhält man eine graphische Darstellung von Datenflussgraphen. Einige typische Vertreter dieser *visuellen Datenflusssprachen* werden im Folgenden beschrieben.

2.3.1 Extended Show and Tell (ESTL)

Show and Tell wurde in seiner ersten Fassung von Kimura als Allzweck-Programmiersprache für Kinder im Schulalter entwickelt und dient heute noch als Vorbild für viele visuelle Datenflusssprachen ([Kimura u. a. 1986; Kimura u. a. 1990]). Die Syntax umfasst Kästchen, die Funktionen darstellen und Verbindungen, auf denen Integer-, Real-Werte sowie Dateien fließen

[3] Hierzu kann man im weiteren Sinne auch Sprachen wie Visual Basic u. ä. zählen, bei denen sich das Visuelle auf die graphische Repräsentation der Benutzeroberfläche beschränkt.

können. Eine Variable wird als leeres Kästchen dargestellt, eine Konstante ist eine Box, die beim Programmstart einen Wert bekommt und unveränderbar ist.

Ein herausragendes Konzept der Sprache ist *Inkonsistenz*. Inkonsistente Kästchen lassen keinen Datenfluss zu, insofern ist dieser Mechanismus ein Ersatz für Fallunterscheidungen resp. den Datentyp Boolean. Ein Kästchen kann inkonsistent werden, wenn entweder zwei verschiedene Werte in ein Variablenkästchen fließen oder wenn ein offenes Kästchen, das ein Prädikat enthält, zu false evaluiert.

Najork und Golin erweiterten die Programmiersprache und entwickelten daraus Extended Show and Tell (ESTL, siehe [Najork und Golin 1990]). Die Autoren fügten ein polymorphes Typsystem hinzu, das eine statische Typprüfung erlaubte. Nun war es z. B. möglich, einen generischen Stapeltyp

stack of T

zu modellieren und ihn gemäß Abbildung 2.4 zu visualisieren.

Der obere Teil definiert den Typ mit einem variablen Teil '**T**' (Typvariable), der untere Teil gibt dessen Implementierung an. Dabei besteht der Stapeltyp aus einem Aufzählungstyp (union type): Ein Wert ist entweder vom Typ *void* (im Bild ■) oder ein strukturiertes Element, das einen Wert vom Typ **T** und rekursiv einen weiteren Stapel umfasst.

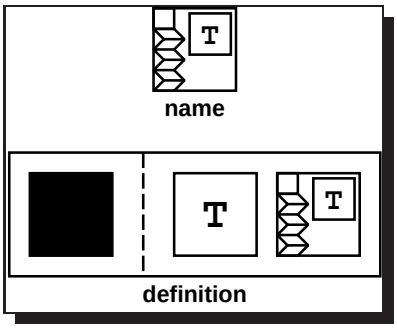


Abbildung 2.4: Generischer Stapeltyp

Außerdem wurden in ESTL Funktionen höherer Ordnung eingeführt, wobei die Funktionenslots, die als Parameter für Funktionen dienen, bereits zum Übersetzungszeitpunkt feststehen müssen. Als Beispielprogramm sei die Fakultät in Abbildung 2.5 gezeigt, die wir noch an anderen Stellen aufgreifen werden.

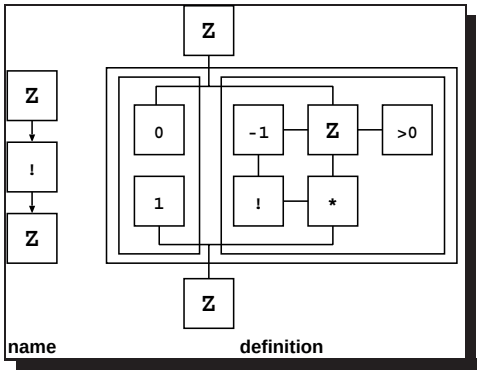


Abbildung 2.5: Fakultät

Auffallend an dem Programmbeispiel ist, dass die Daten von oben nach unten fließen. Die linke Seite führt den Funktionsnamen ('**!**') mit Typisierung ein, die rechte Seite ist dessen expandierte Form. Auch ist der Mechanismus der Inkonsistenz gut erkennbar. Fließt eine '**0**' in das linke Kästchen der expandierten Form, wird eine '**1**' weiter geleitet, andernfalls wird das Kästchen inkonsistent, und die rechte Teildefinition wird aktiv. Dieses Kästchen der Funktionsdefinition umfasst einen Vergleich ('**>0**'). Trifft die Bedingung nicht zu, wird das Kästchen inkonsistent, trifft sie zu, wird über die Vorgängerfunktion ('**-1**') rekursiv die Fakultät berechnet. Durch den Vergleich auf '**0**' bzw. '**>0**' ist also höchstens

ein Kästchen konsistent, sodass beim Zusammenführen der Ergebnisse der beiden Teilkästchen immer nur höchstens ein Wert ansteht.[4]

2.3.2 Cube

Die Entwicklung von Cube war im Wesentlichen geleitet von zwei Grundgedanken:

- einer **visuellen Syntax für Logikprogramme** und
- dem Anspruch, bestehende Logikprogramme, wie z. B. Prolog, **sicherer** und zugleich **ausdrucksstärker** zu gestalten.

Cube entsprang aus den Arbeiten an ESTL (Abschnitt 2.3.1) und bediente sich zweier Schlüsselkonzepte daraus, die, wie sich erst später herausstellen sollte, eng mit der Logikprogrammierung verknüpft sind. Zum einen ist hier der Begriff der *Konsistenz* zu nennen: während der Auswertung eines Graphikkästchens (engl. boxgraph) können Teile desselben inkonsistent werden und leisten somit keinen Beitrag zu einer Berechnung. Zum anderen spielt der Begriff der *Vervollständigung* (engl. completion) eine bedeutende Rolle. Kästchen können zu Beginn einer Berechnung durchaus leer sein. Sobald sie jedoch einen Wert erhalten, ist dieser während der gesamten Lebensdauer des Kästchens unveränderbar. Zusätzliche Eigenschaften wie eine schrittweise Unifikation und ein Backtracking-Mechanismus verhelfen der Sprache zu einer ähnlichen Mächtigkeit wie Prolog.

Die Grundbausteine in Cube sind Kästchen und Kanten. Kästchen werden *Würfel* (engl. cube) genannt, *Leitungen* (engl. pipe) sind die Synonyme für Kanten. Transparente Würfel sind Platzhalterwürfel, die sehr stark an Variablen in herkömmlichen, auch textuellen, Sprachen erinnern. Werte können über Leitungen fließen, welchen keine explizite Richtung zugeordnet ist. Fließt ein Wert in einen Würfel, der bereits seinerseits einen Wert besitzt, so müssen diese beiden Werte unifizierbar sein, anderenfalls schlägt der Datenfluss fehl. Auf diese Weise können sich Werte einzelner Würfel nie grundlegend ändern, sondern werden immer nur verfeinert.

Typisierung

Die Typisierung von Cube orientiert sich stark an der polymorpher, funktionaler Sprachen, als deren prominentester Vertreter wohl ML ([Milner 1978]) zu nennen ist. Cube setzt ein statisches Typsystem ein, durch das bereits zur Übersetzungszeit (sämtliche) Typfehler erkannt werden können.

[4] Steht ein negativer Wert an, so ist kein Kästchen konsistent, und die Funktion besitzt kein definiertes Ergebnis.

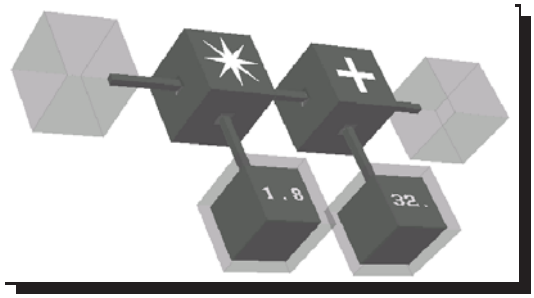
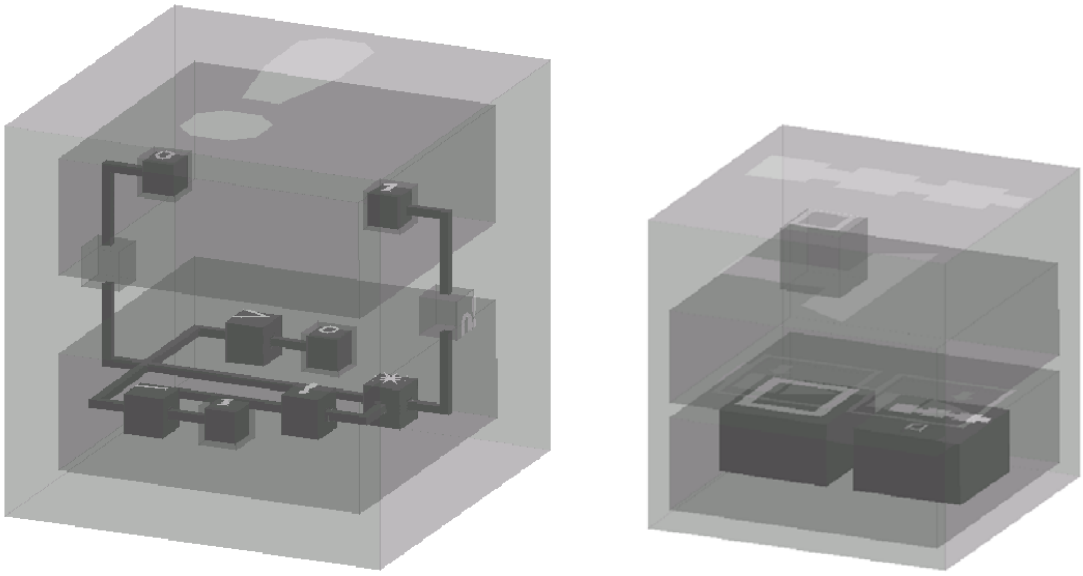


Abbildung 2.6: conv(C, F) $\Leftarrow$ times(C, 1.8, X), plus(X, 32, F).

Prädikate sind in Cube nichts anderes als Funktionen, die auf *propositions* abgebildet wurden. Abbildung 2.6 zeigt ein Beispiel für ein Programm, das die Umrechnung einer Temperatur von Fahrenheit nach Celsius bzw. umgekehrt vornimmt. Bemerkenswert ist in diesem Zusammenhang die Tatsache, dass die Temperatur sowohl in Fahrenheit als auch in Celsius als Eingabeparameter angegeben werden kann; die Leitungen und somit auch der Datenfluss sind ungerichtet. Dabei ist festzustellen, dass die Multiplikation und Addition nur im Falle des Datenflusses von links nach rechts vorgenommen wird. In der Umkehrrichtung findet die Division und Subtraktion durch die jeweils inverse Operation Anwendung.

Die dritte Dimension

Während in Abbildung 2.6 nur zwei Dimensionen genutzt werden, unterstützt Cube indes auch die dritte Dimension zur Darstellung der Disjunktion von logischen Formeln. Dies sei an dem Beispiel der Fakultät (siehe Abbildung 2.7(a)) gezeigt.



(a) Fakultät

(b) Listentyp: List α = nil $+$ cons α (List α)

Abbildung 2.7: Programmgraphen in Cube

Hier sind die einzelnen Ebenen (engl. plane) übereinander angeordnet, die die beiden disjunkten Fälle ($x = 0$ bzw. $x > 0$) beschreiben. Jede Ebene entspricht einer Klausel im logischen Sinne und einem der beiden Kästchen aus der Definition der Fakultät in Abbildung 2.5.

Des Weiteren unterstützt Cube, wie schon erwähnt, ein polymorphes Typsystem mit generischen Typvariablen. So stellt z. B. Abbildung 2.7(b) die polymorphe Liste dar. In Typdefinitionen werden die verschiedenen Dimensionen etwas anders genutzt: Verschiedene Ebenen drücken einen Summentyp aus, wogegen eine horizontale Anordnung dem Produkt entspricht.

2.4 Funktionale Sprachen

Programme in imperativen Sprachen wie C, Pascal und Smalltalk bestehen aus einer Folge von Anweisungen, die sequenziell auszuführen sind. Im Gegensatz dazu wird ein Programm in einer *funktionalen Sprache* durch eine Menge von Funktionsdefinitionen charakterisiert. Die Reihenfolge der Auswertung der Teilfunktionen obliegt dem Compiler bzw. dem Laufzeitsystem. Einige bekannte funktionale Sprachen sind nachfolgend aufgeführt.

2.4.1 Haskell

Haskell (siehe [Peyton Jones 1998; Peyton Jones 1999]) ist eine funktionale Programmiersprache, die seit etwa zehn Jahren von einer Forschergruppe entwickelt wird und viele innovative Sprachelemente eingeführt hat. Die Sprache, die nach dem Logiker Haskell B. Curry benannt wurde, verfügt über Funktionen höherer Ordnung und eine Strategie der *Bedarfsauswertung*[5]. Ein mächtiges Typkonzept mit parametrischem Polymorphismus[6] versucht möglichst viele Fehler bereits zum Übersetzungszeitpunkt zu entdecken. Algebraische Datentypen, Patternmatching, Listenkomprehension und ein monadisches Ein-/Ausgabesystem ergänzen das Bild einer modernen Sprache.

Ohne hier näher auf die sprachlichen Konzepte einzugehen, sei ein kleines Programm angegeben, das den Quicksort-Algorithmus implementiert:

Listing 2.1: Quicksort in Haskell

```
qsort []     = []
qsort (x:xs) = qsort elts_lt_x ++ [x] ++ qsort elts_greq_x
               where
                 elts_lt_x  = [y | y ←xs, y < x]
                 elts_greq_x = [y | y ←xs, y >= x]
```

[5] engl. lazy evaluation: Ausdrücke werden nur dann ausgewertet, wenn sie zum Gesamtergebnis einen Beitrag leisten.

[6] Eine Variante des parametrischen Polymorphismus findet in Form von *generischen Datentypen* Eingang in die Sprache Java (J2SE 1.5), das dieses Jahr die Betaphase verlassen wird. An diesem Konzept waren maßgeblich Forscher beteiligt, die bereits bei verschiedenen Versionen von Haskell mitgewirkt haben.

Ergänzend soll hinzugefügt werden, dass oben stehendes Programm zwar eine Liste korrekt sortiert, dabei aber nicht die Komplexität $O(n * \log(n))$ des Original-Algorithmus von Hoare erzielt. Die Funktion hat stattdessen eine quadratische Komplexität ($O(n^2)$), da die Konkatenation von Listen (++) linear bzgl. der Länge der Liste ist.

2.4.2 FP

John Backus, der in [Backus 1978] die mathematischen Grundlagen der funktionalen Programmierung legte[7], entwickelte ein funktionales Programmiersystem mit dem Namen FP (Functional Programming). Ein Programm der zugrunde liegenden Sprache ist ein Ausdruck, der Objekte[8] auf Objekte abbildet.

Ein FP-System besteht aus drei Komponenten: Objekte, Primitive und zusammengesetzte Konstrukte (*combining forms*).

Objekte

Im Gegensatz zu vielen anderen Sprachen sind Objekte im Quelltext nicht durch Variablen repräsentiert. Sie existieren nur während der Laufzeit zur Parameterübergabe bzw. als Ergebnis von Funktionen. Dieser Mechanismus steht in engem Bezug zu Kombinatoren und der variablenfreien Programmierung aus Kapitel 4. Drei Arten von Objekten sind vordefiniert: das Bottom-Element $\perp$, Atome und Sequenzen. Während $\perp$ das undefinierte Element repräsentiert, gibt es Atome wie Ganzzahlen (-1, 0, 1 ...), Zeichen ('a', 'b', 'c' ...) und boolesche Konstanten (true, false). Sequenzen (<>, < 1, 'a', <> >) bilden das Äquivalent zu Listen aus anderen Sprachen, sind beliebig schachtelbar und können Elemente unterschiedlichen Typs aufnehmen. Bei FP handelt es sich um keine statisch typisierte Sprache, ja sie ist sogar untypisiert.

Primitive

Primitive sind vordefinierte Funktionen, die direkt vom System bereitgestellt werden und darüberhinaus strikt sind. Die Applikation auf eine Funktion wird durch den Operator ':' notiert. Per Definition hat eine Funktion höchstens ein Argument, sollen mehrere übergeben werden, so müssen diese in eine Sequenz verpackt werden[9]:

Listing 2.2: Nachbildung einer mehrstelligen Funktion in FP

```
f : <x1 , x2 , ... , xn>
```

Zusammengesetzte Konstrukte

In FP stehen sechs Bausteine (*program-forming operations*) zur Verfügung, um komplexere Funktionen zusammenzusetzen. Diese sind Konstante, Konditional, Komposition, Konstrukti-

[7] Für diesen Aufsatz bekam Backus den Turing Award.

[8] Nicht zu verwechseln mit Objekten in objektorientierten Sprachen.

[9] Vgl. auch den λ-Kalkül, in dem ebenfalls eine Funktion nur einen Parameter besitzt und mehrere Argumente in einem Tupel, sofern der Kalkül dieses unterstützt, übergeben werden.

on, Apply-to-all und Einfügen. Ohne hier auf die jeweiligen Formen einzugehen, erklären wir die wichtigsten an einer Funktion, die die Fakultät berechnet (Listing 2.3). Man beachte v. a. die variablenfreie Formulierung und den Umstand, dass ausschließlich Funktionen verwendet werden.

Listing 2.3: Fakultät in FP (rekursiv)

```
def fac = eq0 → 1 ; ∗ ∘ [ id , fac ∘ (− ∘ [ id , 1 ])]
```

Das Schema $X \rightarrow Y\,;\,Z$ beschreibt eine Fallunterscheidung, bei der ein Parameter auf X angewandt wird und davon abhängig entweder Y oder Z das Ergebnis des Gesamtausdrucks ist. $\underline{1}$ wiederum bezeichnet eine Funktion, die zu einem beliebigen Parameter den Wert 1 liefert, und [ ] ist eine Tupelbildung, die für mehrstellige Funktionen (z. B. *, -) benötigt wird.

Die Programmzeile definiert durch **def** eine Funktion fac, deren Argument auf 0 getestet wird (eq0). Ist der Vergleich wahr, so wird das Argument auf die Konstantenfunktion $\underline{1}$ appliziert, und das Ergebnis ist 1.

$$\begin{aligned}(\text{eq0} \rightarrow \underline{1}\,;\, * \circ [\text{id}, \text{fac} \circ (- \circ [\text{id}, \underline{1}])]) : 0 &\Rightarrow \underline{1} : 0 \qquad \text{appliziert auf 0}\\ &\Rightarrow 1\end{aligned}$$

Ist das Argument >0, so bilden wir den Vorgänger mit

$$\begin{aligned}- \circ [\text{id}, \underline{1}] & \qquad \text{Vorgängerfunktion}\\ (- \circ [\text{id}, \underline{1}]) : n &\Rightarrow - : ([\text{id}, \underline{1}] : n) \qquad \text{appliziert auf n>0}\\ &\Rightarrow - : \langle n, 1\rangle\\ &\Rightarrow n-1\end{aligned}$$

so dass wir insgesamt

$$\begin{aligned}(\text{eq0} \rightarrow \underline{1}\,;\, * \circ [\text{id}, \text{fac} \circ (- \circ [\text{id}, \underline{1}])]) : n &\Rightarrow (* \circ [\text{id}, \text{fac} \circ (- \circ [\text{id}, \underline{1}])] : n \qquad \text{appliziert auf n>0}\\ &\Rightarrow * : ([\text{id}, \text{fac} \circ (- \circ [\text{id}, \underline{1}])] : n) \qquad \text{appliziert auf n>0}\\ &\Rightarrow * : \langle \text{id} : n, (\text{fac} \circ (- \circ [\text{id}, \underline{1}])) : n\rangle\\ &\Rightarrow * : \langle n, \text{fac} : ((- \circ [\text{id}, \underline{1}]) : n)\rangle\\ &\Rightarrow * : \langle n, \text{fac} : n-1\rangle\\ &\Rightarrow n * (\text{fac} : n-1)\end{aligned}$$

erhalten.

Die Eleganz und Mächtigkeit der Sprache FP zeigt sich, wenn wir die Fakultät in einer nichtrekursiven Form wie in Listing 2.4 notieren.

Listing 2.4: Fakultät in FP (nichtrekursiv)

```
fac = /1 * ι
```

Diese Art der variablenfreien Programmierung beeinflusste in starkem Maße die datenflussartige Herangehensweise in HOPS. Vgl. dazu Kapitel 4 und insbesondere den HOPS-Graphen der Fakultät in Abbildung 3.14(a).

2.4.3 pH

pH ([Nikhil u. a. 1995]) ist eine parallele Variante der funktionalen Sprache Haskell (parallel Haskell) mit einer *strikten Auswertungsstrategie* [10]. Deswegen ist es auch nicht verwunderlich, dass die Entwickler die Semantik in [Aditya u. a. 1995] relativ zur Basis Haskell angeben. Außerdem deckt pH einige Merkmale von datenflussorientierten Sprachen wie ID und SISAL ab.

Es gibt verschiedene Ansätze zur Implementierung von Parallelität in funktionalen Sprachen. So liegt es auf der Hand, die Abwesenheit einer totalen Ordnung der Ausführungsreihenfolge von Funktionsaufrufen auszunutzen und Argumente von strikten Funktionen bzw. Operatoren parallel auszuwerten. Dies wird durch die referenzielle Transparenz einerseits und die aufwändigen Striktheitsanalysen andererseits unterstützt.

Ein konkurrierender Ansatz dazu ist die parallele Auswertung aller Redexe, nicht nur jener, deren Werte benötigt werden. Diese Strategie verfolgt ebenfalls eine nicht-strikte Semantik unter der Voraussetzung, dass kein Redex seine Auswertung unendlich verzögert. So erfuhr pH Erweiterungen im Bereich von Schleifen, synchronisierten Seiteneffekten und expliziten Sequenzialisierungen.

Ein Programm zu Berechnung der Summe der ganzen Zahlen von 1 bis *n* sieht mit Hilfe des Schleifenkonstrukts folgendermaßen aus:

Listing 2.5: Programm in pH zur Berechnung von $\sum_{i=1}^{n} i$

```
let sum = 0
in for i ←[1..n] do
   next sum = sum + i
   finally sum
```

Man sieht hier zum Einen den Zugriff auf die folgende Schleifeninkarnation (**next** sum), zum Anderen ist die syntaktische Nähe zu der Muttersprache unverkennbar. Schleifen sind rein funktional und können leicht in endrekursive Funktionen übersetzt werden. Zusammen mit der Eager-Semantik entsteht daraus eine besonders effiziente Implementierung.

[10] engl. eager evaluation: Ausdrücke werden ausgewertet, auch wenn sie zu dem Gesamtergebnis keinen Beitrag leisten.

2.4.4 SISAL

SISAL ist eine funktionale Sprache für parallele numerische Berechnungen. Sie wurde von Digital Equipment Corporation in Zusammenarbeit mit der University of Manchester, dem Lawrence Livermore National Laboratory und der Colorado State University entwickelt (siehe [Cann 1992; Böhm u. a. 1992]). Das Hauptziel war es, sequenziellen wie auch parallelen Programmcode zu erzeugen, der schneller war als Programme, die in bisher üblichen Sprachen geschrieben wurden. SISAL ist eine applikative Sprache, die mit Elementen aus Datenflusssprachen angereichert ist und als eine Weiterentwicklung von VAL ([Ackerman und Dennis 1979]) bzw. ID ([Arvind u. a. 1978]) angesehen werden kann.

Der zum damaligen Zeitpunkt neuartige Ansatz zeichnet sich durch Merkmale aus wie strenge Typisierung und Single-Assignment[11]. Das Haupteinsatzgebiet ist der naturwissenschaftliche Bereich. Mit Hilfe mannigfaltiger Funktionen zur Manipulation von Arrays und der Entdeckung und Behandlung von möglicher Parallelität erreicht sie eine ähnliche Performanz wie optimierter FORTRAN-Code.

Die Entwickler legten damals Wert auf höchste Effizienz. So gingen sie einen bewussten Kompromiss ein, indem sie sich auf Funktionen erster Ordnung beschränkten. Um parallele Prozessoren bzw. Betriebssysteme besser zu unterstützen, führten sie in pragmatischer Weise sowohl parallele Schleifen ein, wie auch eine sequenzielle Version mit Zugriff auf die vorhergehende Inkarnation der Variablenbelegung.

Nachfolgend sehen wir als kleines Beispielprogramm die Fakultät, deren Programmtext sich kaum von anderen funktionalen oder imperativen Sprachen unterscheidet.

Listing 2.6: Fakultät in SISAL

```
function Factorial ( n:integer returns integer )
  if ( n <= 0 ) then 1 else n*Factorial(n-1) end if
end function
```

Etwas ungewöhnlicher ist das Konzept der Schleife, die eine syntaktische Ähnlichkeit zu prozeduralen Sprachen aufweist. Im Gegensatz hierzu existiert jedoch in SISAL kein veränderbarer Zustand, so dass Variablen, denen einmal ein Wert zugewiesen wurde, diesen bis zu ihrem Lebensende bewahren. Insofern kann es auch keinen Schleifenzähler geben. Im Folgenden geben wir ein Programm zur Berechnung der Matrixmultiplikation an, das zwei parallele Schleifen benutzt. Die äußere Schleife liefert parallel die Elemente des Ergebnisarrays, die innere sequenziell jeweils ein Skalarprodukt. Durch die Einführung von Schleifen steht die Verteilung der Berechnung auf mehrere Prozessoren offen.

[11] Single-Assignment bedeutet, dass eine Variable unveränderlich ist, sobald ihr ein Wert zugewiesen wurde.

Listing 2.7: Matrixmultiplikation in SISAL

```
function MatMult( A,B:TwoDim; M,N,L:integer returns TwoDim )
  for i in 1, M cross j in 1, L
    S := for k in 1, N
      returns value of sum A[i,k]*B[k,j]
      end for
  returns array of S
  end for
end function
```

Eine Einschränkung, die SISAL gegenüber anderen funktionalen Sprachen aufweist, ist die eingeschränkte Polymorphie. Die Sprache unterstützt keine Typvariablen, sondern der Programmierer gibt eine Menge von monomorphen Typen an, die in etwa mit dem Klassenkonzept von Haskell vergleichbar sind. So würde ein numerischer Typ numeric in SISAL wie folgt aussehen:

```
type numeric = integer + real + double ...
function f (x: numeric returns numeric)
```

während Haskell parametrischen Polymorphismus (durch die Typvariable a) unterstützt:

```
f :: Num a => a →a
```

2.5 Regelbasierte Sprachdefinition

Die bisher vorgestellten Paradigmen und Systeme bauen auf einer abgeschlossenen Menge von Sprachbausteinen auf. Die Konstrukte sind fest vorgegeben. Erweiterungen sind nur durch Definition von Funktionen oder ähnlichen Konstrukten möglich. Zudem herrscht eine strikte Trennung zwischen einem vordefinierten Grundbaustein (z. B. einem for-Konstrukt) und einer vom Benutzer definierten Funktion (z. B. Fakultät). So gibt es z. B. bei textuellen Sprachen in der Art der Aufschreibung einen begrenzten Vorrat an Schlüsselwörtern, allein die Menge der Variablen- und Funktionsbezeichner ist unbeschränkt.

Ähnlich ist es bei der Definition der Semantik. Die von der Sprache vorgegebenen Bausteine besitzen eine inhärente, vom Benutzer nicht änderbare Bedeutung. Allein selbst definierte Funktionen sind in ihrer Definition frei wählbar.

Diese strikte Trennung zwischen Systembausteinen mit eingebauter Semantik und benutzerdefinierten Funktionen weichen Systeme mit einer regelbasierten Sprachdefinition auf. Eine Sprache wird vorgegeben durch eine Menge von Bausteinen, die zwar aus einer allgemeinen

Bibliothek stammen können, sich jedoch nicht von benutzerdefinierten Erweiterungen unterscheiden. Eine syntaktische Unterscheidung gibt es nicht.

Die Semantik von Bausteinen ist allein durch eine Menge von Regeln bestimmt, die für verschiedene Anwendungskontexte definiert sein können. Auch hier unterbleibt eine künstliche Trennung der Semantikvorgabe von Systemkonstrukten und Benutzerfunktionen.

HOPS – ein Programmier- und Transformationssystem

Ein Vertreter der Idee einer regelbasierten Sprachdefinition ist HOPS. Es ist ein generisches, sprachunabhängiges Programm- und Transformationssystem, das sich zum interaktiven Entwickeln von Programmen eignet. Der Benutzer entwirft ein Programm mit Unterstützung des Entwicklungssystems, das Vorschläge und Hinweise zur weiteren Vorgehensweise unterbreitet. Eine ausführlichere Beschreibung zu HOPS findet sich in Kapitel 3; hier seien nur einige Merkmale angeführt, die im Vergleich zu den oben beschriebenen Programmiermodellen stehen.

HOPS baut auf der visuellen Programmierung auf, bei der eine Funktion bzw. ein Programm durch einen Graphen repräsentiert und graphisch angezeigt wird. Dem System ist jedoch keine feste Sprache immanent; diese ist vielmehr durch die Angabe von Basisbausteinen vorgegeben. Daher ist es möglich, für individuelle Domänen spezifische Sprachen zu definieren im Gegensatz zu den oben erwähnten Sprachen, bei denen der Sprachschatz fest definiert ist.

Des Weiteren umfasst HOPS ein Transformationssystem, durch das ein gegebenes Programm nachträglich semantikerhaltend gemäß eines Regelsatzes transformiert werden kann.

Die meisten Sprachen bauen auf einer mehr oder weniger formal spezifizierten Semantik auf. Hier verfolgt HOPS eine allgemeinere Herangehensweise, bei der keine feste Abarbeitungsstrategie in das System eingebaut ist. Erst die benutzerdefinierte Angabe einer Regelmenge, die auf einem mächtigen Regelmechanismus fußt, beschreibt — möglicherweise auch nur partiell — die Semantik einer Sprache. Dies bedeutet, dass für partielle Auswertungen von Termen oder das Ausführen eines Programms immer auch Regeln spezifiziert werden, die in den Kontext von Regelstrategien eingebettet sind.

Selbst die Auswertung eines Programms ist allein durch die Menge der Regeln und einer darauf basierenden Regelstrategie bestimmt. So kann eine Regelmenge sowohl für eine Strategie der Bedarfsauswertung als auch für eine strikte Auswertung appliziert werden, möglicherweise mit unterschiedlichen Ergebnissen.

Wir haben zunächst einige, für diese Arbeit wichtige Programmiermodelle aufgezeigt und deren Merkmale diskutiert. In den folgenden Kapiteln werden wir sehen, wie sich die Paradigmen der Datenflusssprachen und der funktionalen Sprachen ergänzen und mit einer regelbasierten Sprachdefinition kombinieren lassen.

Dazu stellen wir zunächst in Kapitel 3 ein geeignetes Werkzeug vor und untersuchen in Kapitel 4 kombinatorisches Programmieren, das eine Ausprägung des funktionalen Programmierens darstellt, im Kontext einer regelbasierten Sprachdefinition. Ein ausführliches Beispiel zeigt dann

in Kapitel 5 datenflussorientiertes Programmieren mit Unterstützung eines mächtigen Transformationswerkzeugs.

KAPITEL 3

Einführung in HOPS

Das folgende Kapitel gibt einen Einblick in das Programmier- und Transformationssystem HOPS. Dazu stellen wir dessen wichtigste Charakteristika vor und beschreiben durch die Definition einer Kernsprache eine typische Art seiner Programmierung.

Inhaltsverzeichnis

HOPS — ein Akronym für Higher Object Programming System[1] — ist ein visuelles Pro-

[1] Nicht zu verwechseln mit dem Heineken Operational Planning System (HOPS), siehe auch http://www.internetworld.com/print/current/webenterprise/19991201-beer.html

gramm- und Transformationsystem, das einige neue Ansätze im Bereich der Erstellung und Pflege von Programmen verwirklicht.

Während die meisten Software-Entwickler daran gewöhnt sind, ihr Programm in textueller Form zu erfassen, wird dieses in HOPS durch einen Graphen repräsentiert, der in graphischer Weise angezeigt wird. Dieser auf den ersten Blick ungewöhnliche Ansatz verschafft dem Benutzer direkten Zugriff auf die Struktur eines Programms, die bei textorientierten Sprachen erst mühsam durch Einrücken und Pretty Printing offensichtlich wird.

Im Zusammenspiel mit einem eingebauten graphischen Editor werden so Syntaxfehler bereits bei der Programmerstellung bemerkt; die Umgebung erlaubt dann auch nur die Eingabe syntaktisch korrekter Programme. Darüber hinaus arbeitet im Hintergrund ein Typisierungsmechanismus, der die Eingabe nicht typisierbarer Graphen unterbindet. Nicht zuletzt bietet ein mächtiger Transformationsmechanismus das Ausführen einzelner Regeln sowie ganzer Regelsequenzen an. Somit können Programme semantikerhaltend transformiert werden, um z. B. effizienteren oder verteilbaren Code zu erhalten. Schließlich kann Code für unterschiedliche Zielsprachen erzeugt werden mit Hilfe von generischen Durchlaufalgorithmen, die sich an der Graphstruktur eines Programms orientieren.

3.1 Historie

Seit Mitte der 80er Jahre führt Gunther Schmidt eine kleine Forschungsgruppe an, die für das HOPS-Projekt verantwortlich zeichnet. Eine erste Beschreibung über dieses Vorhaben findet sich in [Zierer u. a. 1986]. Hierauf folgten zwei Diplomarbeiten an der TU München ([Libera 1987], [Bayer 1987]), die gemeinsam den Grundstein zu der Urversion HOPS 1 legten.

Nach dem Wechsel von Schmidt und seinem Team an die Universität der Bundeswehr München wurde dieses System zunächst weiterentwickelt ([Bayer u. a. 1990] und v. a. [Kahl 1991]). Aus dem Wunsch, einen anderen Ansatz zu verfolgen, nämlich aus einem Programmgraphen imperativen Code zu erzeugen ([Schmidt u. a. 1990]), entsprang HOPS 2 ([Held und Zimmermann 1990]). Parallel dazu wurde im Rahmen mehrerer Diplomarbeiten (u. a. [Held 1991] und [Zimmermann 1991]) ein C-Programm erstellt, das den Anforderungen moderner, graphischer Benutzungsschnittstellen entsprach. Diese Plattform war auch die erste Version, die second-order Typisierung und eine eingeschränkte Form von Polytypic Programming umfasste.

Im Rahmen mehrerer Diplomarbeiten ([Endres und Müller 1991], [Stender 1992], [Derichsweiler 1996] und [Christ 1996]) wurde das System einer Umstrukturierung unterzogen und von Grund auf neu entwickelt. Eine kleine Entwicklergruppe, zu der auch der Autor gehört, ist ständig bestrebt, das Programm weiterzuentwickeln. So wurde HOPS u. a. um eine Datenflusskomponente und ein Transformationsstrategie-Modul ergänzt. Nicht zuletzt in Sachen Benutzerfreundlichkeit erfuhr HOPS massive Verbesserungen, so dass das System in der Folge zu einem stabilen System heranwuchs und auf Workshops mit Erfolg demonstriert wurde.

Kahl wandte sich in seiner Dissertation ([Kahl 1996]) theoretischen Problemen bzgl. Term-

graphersetzung zu, die in Zusammenhang mit HOPS standen. Die Erkenntnisse seiner Arbeit resultierten in einer HOPS-Version, die er in OLabl entwickelte, einer Variante der funktionalen Sprache OCaml, welche wiederum aus ML entstand. Mit seiner auf Tk basierenden, graphischen Benutzungsoberfläche ist ein System entstanden, das innerhalb der theoretischen Forschungsgemeinde Beachtung gefunden hat.

3.2 Beispiel

Programmieren in HOPS heißt Programmieren mit Termgraphen. Während in herkömmlichen Sprachen ein Programm in einer textuellen Zeichenkette verfasst ist, besteht ein Programm in HOPS aus einem azyklischen Graphen.

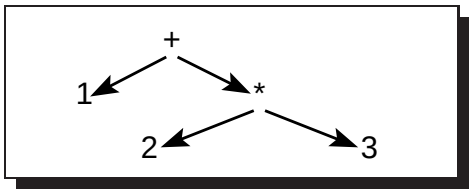


Abbildung 3.1: Einfacher Termgraph

Zunächst erläutern wir an einem kleinen Beispiel, dargestellt in Abbildung 3.1, die Syntax eines einfachen Termgraphen. Knoten eines Graphen sind mit Markierungen versehen (+, *, 1, 2, 3), die Sprachbausteine notieren. Kanten sind gerichtet und weisen eine Ordnung bezüglich des Anfangsknotens auf.

Der Leser erkennt unschwer, dass der abgebildete Graph Ähnlichkeiten zu einem Parsebaum des Ausdrucks

```
1 + (2 * 3)
```

aufweist.

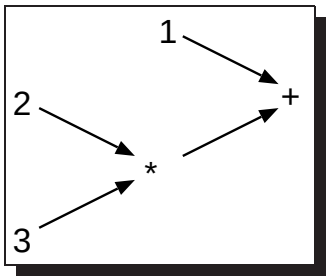


Abbildung 3.2: Einfacher Datenflussgraph

Zusätzlich zu einem Termgraphen besitzt ein Baustein einen parallelen Datenflussgraph, der eine datenflussartige Darstellung des Programms repräsentiert. Der zugehörige Datenfluss zu unserem Beispiel ist in Abbildung 3.2 zu sehen, bei dem der Termgraph um 90° gedreht und die Richtung der Kanten umgekehrt ist. Datenflussgraphen anderer Termgraphen können jedoch eine völlig andere Darstellung haben.

Aus dem Beispiel sehen wir, dass Kanten in den beiden Graphen eine unterschiedliche Bedeutung haben. Während Kanten im Termgraph die Struktur eines Terms beschreiben, bezeichnen Kanten im Datenflussgraph den Datenfluss des Programms und geben Einblick in dessen Ablauf.

3.3 Kernsprache

Von einer bestimmten Sprache in HOPS zu sprechen, ist nicht ganz korrekt, da das System eher ein *Rahmenwerk* zur Programmentwicklung und -transformation darstellt. In der Tat stellt das Kernsystem „lediglich“ eine Reihe von syntaktischen Mechanismen zur Verfügung, mit deren Hilfe der Benutzer Termgraphen zweiter Ordnung mit einer Typisierung erster Ordnung definieren kann. Aber gerade darin steckt auch das Potenzial. Die Programmenwicklung wird geleitet und unterstützt durch Sprachen, die der Problemstruktur angepasst sind und können ggf. weiter angepasst werden.

Motiviert durch den Hintergrund, einfache Beweise und Transformationen der erstellten Programme durchzuführen, bietet es sich an, eine Sprache zu verwenden, die nahe an mathematischen Definitionen ist. So wollen wir eine Kernsprache angeben, die eng an den λ-Kalkül resp. die funktionale Programmierung angelehnt ist.

Doch selbst innerhalb der funktionalen Gemeinde kann man noch Feinheiten in der Programmierung ausmachen. Während die einen eher eine *applikative Programmierung* bevorzugen, in der hauptsächlich **Funktionen auf Argumente angewandt** werden, wollen wir uns der wahrhaft funktionalen Programmierung ([Kahl 1994]) zuwenden. Dieser Stil propagiert die Komposition von Funktionen mittels **Kombinatoren höherer Ordnung**.

Funktionen werden aus Funktionen zusammengesetzt und reduzieren somit die Anzahl der Terme erster Ordnung. Damit geht auch einher, dass Variablenbindungen entfallen und somit eine einfachere Struktur des Programms erzielt wird.

3.3.1 Bausteine

Bausteine sind die Grundelemente in HOPS, aus denen komplexe Programme zusammengesetzt sind. Jeder Baustein verfügt über zwei Graphen in Form eines Termgraphen (DAG) und eines Datenflussgraphen (DFG).

An dem Beispiel der Funktionskomposition (comp bzw. in Infixschreibweise $\circ$) mit der allgemeinen Definition

Listing 3.1: Baustein comp

```
(∘) :: (α → β) × (β → γ) → (α → γ)
f ∘ g = λx. g(f(x))
```

erklären wir die Syntax der beiden Graphausprägungen eines Bausteins. Die Funktionskomposition ist aus der Mathematik als ‘$\circ$’, jedoch mit vertauschten Parametern, bekannt,

Termgraph

Der Termgraph repräsentiert in Form eines DAG (*directed acylic graph*) die strukturelle Ansicht eines Bausteins. Abbildung 3.3 zeigt den entsprechenden Graphen für das Element der Funkti-

onskomposition[2].

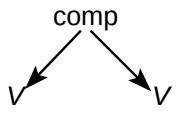


Abbildung 3.3: Funktionskomposition (comp) als Termgraph

comp ist ein zweistelliger Konstruktor, der genau zwei Nachfolger (V) erfordert, hier die beiden Funktionen, die komponiert werden sollen. Die Nachfolger sind Platzhalter für beliebige, komplexe Graphen. Da explizite Kanten auf die Nachfolgerknoten (V) verweisen, können wir auf eine eindeutige Namensgebung der Variablen verzichten.

Datenflussgraph

Die Datenflussansicht ist eine alternative Darstellung der strukturellen Sicht eines Termgraphen. Der dazugehörige Datenflussgraph (DFG, *data flow graph*) des Bausteins comp ist in Abbildung 3.4 abgebildet.

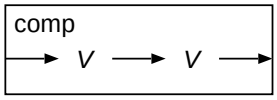


Abbildung 3.4: Funktionskomposition (comp) als Datenflussgraph

Hier sehen wir die anschaulich sequenzielle Datenverarbeitung der beiden Funktionen. Der Zusammenhang zwischen den beiden Graphen ist in Abschnitt 3.3.3 erläutert.

3.3.2 Typisierung

Zu jedem Baustein gehört eine Typisierung in Form eines Typgraphen. Dazu ist jeder Knoten des Bausteins mit einem Verweis versehen, der auf einen Einstieg in den Typgraphen zeigt (grüne Pfeile, →)

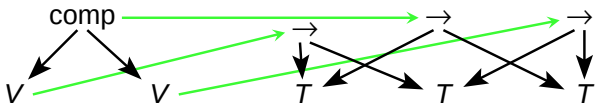


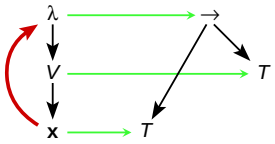
Abbildung 3.5: Funktionskomposition (comp) mit Typisierung

[2]Es sei an dieser Stelle angemerkt, dass sämtliche HOPS-Graphen aus dem System generiert wurden und nicht etwa durch ein Graphikwerkzeug händisch erstellt sind. Daher kann es vorkommen, dass das Layout mancher Graphen für das menschliche Auge nicht perfekt erscheint.

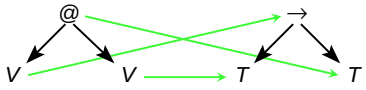
So hat comp den Funktionstyp, dessen erstes Argument den Typ des ersten Arguments der ersten Funktion besitzt, das zweite Argument hat denselben Typ wie das zweite Argument der zweiten Funktion.

Abbildung 3.6 enthält eine Übersicht über fundamentale Bausteine in HOPS. Hier sind neben grundlegenden Funktionen wie Applikation (@) und Abstraktion (λ) solche zur Behandlung von Paar und Summe abgebildet.

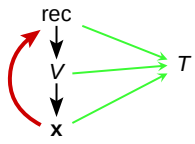
Grundlegende Funktionen



(a) Abstraktion

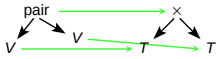


(b) Applikation



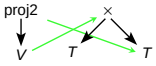
(c) Rekursion

Generisches Produkt

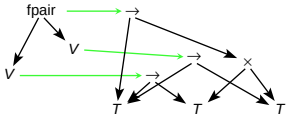


(d) Paarbildung

(e) Linke Projektion

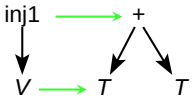


(f) Rechte Projektion

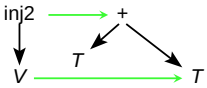


(g) Funktionspaarung

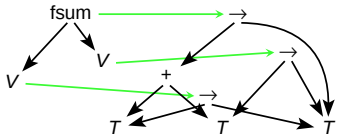
Generische Summe



(h) Linke Injektion



(i) Rechte Injektion



(j) Funktionssumme

Abbildung 3.6: Bausteine in HOPS

Darüber hinaus ist das HOPS-System jederzeit durch neue Bausteine erweiterbar. Hier hat der Benutzer die Möglichkeit, für einen Baustein neben seinem Typelement auch seine Daten-

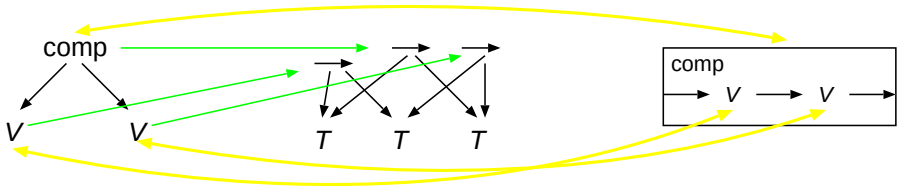


Abbildung 3.7: Korrespondenz zwischen DAG und DFG

flussdarstellung anzugeben und in einem Modul abzulegen.

Typinferenzmechanismus

Während der Benutzer seinen Graphen editiert, läuft im Hintergrund stets ein Typisierungsalgorithmus, der für jeden Teilausdruck bzw. Teil-DAG die allgemeinste Typisierung errechnet und nicht korrekt typisierbare Terme zurückweist. Auf diese Weise ist es auch unmöglich, nicht typisierbare Programme zu erstellen. Die Typinformation ist also bereits zum Zeitpunkt des Editierens verfügbar. Dies steht im Gegensatz zu (herkömmlichen) textorientierten Sprachen, bei denen erst der Compiler zu einem späteren Zeitpunkt eine Typüberprüfung durchführt.

3.3.3 Korrespondenz zwischen Termgraph und Datenflussgraph

Termgraphen und Datenflussgraphen stellen hier zwei Alternativen in der Darstellung eines Terms dar. Diesem Zusammenhang wird durch eine enge Kopplung zwischen den beiden Graphtypen auf Bausteinebene Rechnung getragen. In den Bausteinen ist dazu eine Korrespondenzabbildung zwischen einzelnen Knoten der Termgraph- und Datenflussansicht definiert.

Die Beziehung illustriert Abbildung 3.7 anhand des Bausteins comp. Jeder Knoten des Termgraph ist über eine Kopplungskante (gelber Pfeil, ↔) mit einem Knoten im Datenflussgraphen verbunden. Dadurch erhält der Datenflussknoten seine Typisierung.

3.3.4 Regeln

Ziel von HOPS ist nicht allein die Programmierung mit Graphen, sondern auch deren (semantikerhaltende) Transformation. Die Idee, dass ein Quelltext nicht geronnen, d. h. starr ist, sondern weiteren Änderungen offen steht, wurde u. a. in dem CIP-Projekt („Computer-Aided Intuition-Guided Programming“) verfolgt ([Bauer u. a. 1985]). HOPS führt diesen Ansatz mit seinem graphischen Zugang zu Programmen weiter und bietet dem Benutzer an, zum Einen sein Programm performanter zu gestalten — sei es durch partielle Auswertung oder andere Umformungen. Durch eine iterative Vorgehensweise ermöglicht es andererseits, aus einer Anforderung einen fertigen, ausführbaren Algorithmus zu entwickeln, wie dies in Kapitel 5 durchgeführt wird.

Um einen Eindruck von den Regeln zu geben, sei hier lediglich die β-Reduktion (Abbildung 3.8) angegeben, die zur Auswertung von Teilausdrücken zuständig ist. Der orangefarbene Pfeil spezifiziert die Ersetzungsrichtung, d. h. in diesem Fall wird statt einer λ-Abstraktion appliziert auf einen Wert dessen Rumpf eingesetzt, wobei die darin vorkommende Variable mit dem Parameter ersetzt wird. Nebenbei sei bemerkt, dass Regeln durchaus invers angewandt werden dürfen. Sind sie nicht eindeutig wie in unserem Beispiel, muss eine Benutzerinteraktion diese Mehrdeutigkeit auflösen.

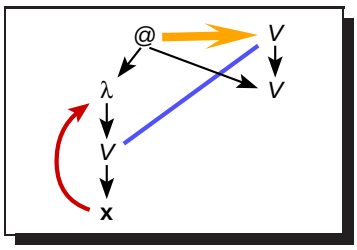
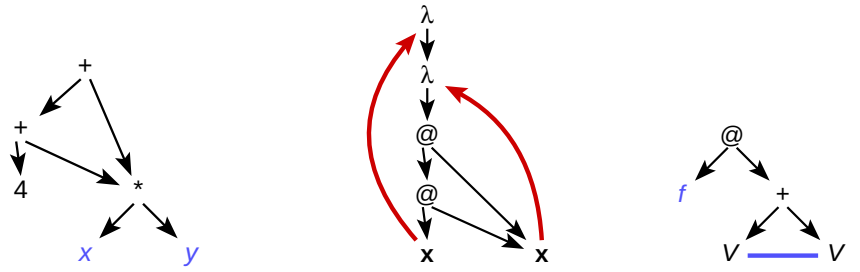


Abbildung 3.8: β-Reduktion

3.3.5 Besondere Grapheigenschaften

HOPS gründet von Anfang an auf der Idee, dem Benutzer die Struktur eines Programmes, v. a. unter dem Aspekt eines Transformationssystems, so plastisch wie möglich anzubieten. Aus diesem Grund haben sich die Entwickler zur Repräsentation für zweidimensionale Graphen entschieden.



(a) Sharing (b) Variablenbindung (c) Variablenidentität

Abbildung 3.9: Merkmale von Termgraphen

Sharing

Zunächst kann jede Funktion als Syntaxbaum dargestellt werden. Nichts anderes verfolgen Strukturediteren für herkömmliche Sprachen wie Pascal oder Java, indem sie einen Quelltext parsen und die daraus gewonnene Struktur anzeigen. HOPS geht in diesem Zusammenhang einen Schritt weiter und verwendet zur Darstellung Graphen statt Bäume, indem gemeinsame Teilausdrücke durch *Sharing*, also durch Knoten, die mehr als eine einlaufende Kante haben, spezifiziert werden, wie in Abbildung 3.9(a) zu sehen ist.

Bindbare Variable und Variablenbindung

Das Konzept des Sharings findet auch bei bindbaren Variablen Anwendung. Bindbare Variable entsprechen den Variablen des λ-Kalküls und bezeichnen formale Parameter einer Funktionsdefinition. Sie werden bei einem Funktionsaufruf zur Laufzeit mit den aktuellen Parametern instanziert.

Wo in textuellen Sprachen der Weg über Namensgleichheiten gegangen wird, wenn die gleiche bindbare Variable gemeint ist, wird diese Beziehung explizit durch eine *Bindungskante* gekennzeichnet (rote Kante vom Knoten x zum Knoten λ in Abbildung 3.9(b)). Mehrfaches Auftreten einer Variable in einem Ausdruck wird durch Referenzierung genau eines Variablenknotens ausgedrückt. Hier ist der Benutzer davon entlastet, im Gedanken einen Ausdruck zu parsen und evtl. Verschattungen zu berücksichtigen. Die Struktur ist im Graphen bereits explizit präsent.

Metavariable und Variablenidentität

Ein weiteres Konzept, das in Verbindung mit Variablen steht, ist die *Variablenidentität*. Sie zeigt an, dass zwei Variablenknoten dieselbe Metavariable referenzieren. Metavariable sind Platzhalter in Regeldefinition, die zur Laufzeit eines Programms nicht existent sind, sondern nur während der Applikation einer Regel benötigt werden.

Die Variablenidentität ist eine partielle Äquivalenzrelation auf Metavariablen, wie sie in Abbildung 3.9(c) als blaue Kante zwischen den Knoten V zu sehen ist[3]. Weitere Aspekte zu dem Thema Termgraphen in HOPS sind in [Bayer u. a. 1996] und [Kahl 1998] nachzulesen.

3.4 Funktionale Programmierung mit Graphen

In dem Entwicklungssystem HOPS werden zweierlei Arten von Graphen angeboten. Dies sind einerseits *Termgraphen*, die die **Struktur** eines Programms charakterisieren. Dem gegenüber stehen *Datenflussgraphen*, die den **Datenfluss** eines Programms betonen, vorausgesetzt, es besitzt einen datenflussartigen Charakter. Näheres zu dem Zusammenhang zwischen den beiden Grapharten findet sich in [Bayer 1995] und [Bayer und Derichsweiler 1997].

3.4.1 Termgraph

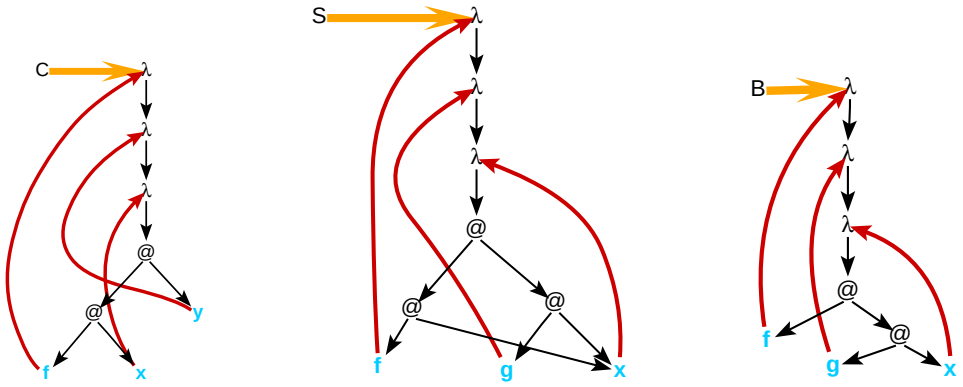
Termgraphen für einige bekannte Kombinatoren aus dem λ-Kalkül sollen an dieser Stelle die Notation von HOPS verdeutlichen. So sehen wir in Abbildung 3.10(a)[4] den C-Kombinator

$$\mathbf{C} \equiv \lambda f.\lambda y.\lambda x.\ f\ x\ y$$

der in Haskell auch unter dem Namen flip bekannt ist und, angewandt auf eine Funktion, selbige mit vertauschten Parametern zurückliefert.

[3] Hier böte sich zwar die Möglichkeit an, einen einzigen Knoten mit zwei einlaufenden Kanten zu verwenden, doch ist dies bereits bei mehrstelligen Metavariablen, d. h. Variablen mit Nachfolgern, nicht mehr möglich.

[4] Der Graph ist eine Regel, die den Baustein C definiert.



(a) C-Kombinator (b) S-Kombinator (c) B-Kombinator

Abbildung 3.10: Bekannte Kombinatoren aus dem Lambda-Kalkül

Der S-Kombinator

$$\mathbf{S} \equiv \lambda f.\lambda g.\lambda x.\ f\ x\ (g\ x)$$

hat seine Entsprechung in Abbildung 3.10(b), der B-Kombinator

$$\mathbf{B} \equiv \lambda f.\lambda g.\lambda x.\ f\ (g\ x)$$

ist in Abbildung 3.10(c) zu finden. In HOPS ist dieser unter comp, jedoch mit vertauschten Parametern, zu finden; Haskell-Programmierer kennen die Funktionskomposition unter '.'. Wir verwenden in diesem Abschnitt Kombinatoren lediglich als Repräsentanten von Programmgraphen. Auf deren Bedeutung und Verwendung werden wir in Kapitel 4 näher eingehen.

Generische Summe und Rekursion

Um auf einige Merkmale der Programmierung hinzuweisen, bemühen wir im Folgenden die allseits bekannte Fakultätsfunktion:

Listing 3.2: Fakultät als Funktional

```
factorial = rec f . λx . if x==0 then 1 else ∗ (x, f(x−1))
```

Diese Funktion transformieren wir in eine Version, die eher der Programmierphilosophie von HOPS entspricht. Dazu fassen wir die natürlichen Zahlen $\mathbb{N}$ als Lösungen der Gleichung

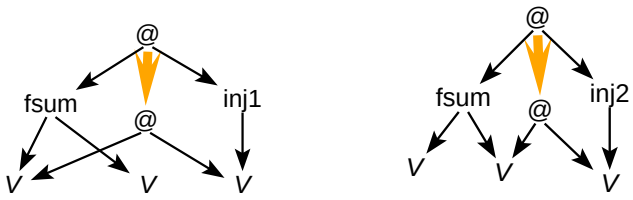
$$\mathbb{N} = \tau(\lambda x.\ \mathbb{1} + x)$$

auf. Hier bezeichnet τ den Fixpunktoperator, $\mathbb{1}$ ist eine einelementige Menge mit dem gleichnamigen Element $\mathbb{1}$. Wir erhalten mit dieser Gleichung also so genannte „Strichzahlen“. Als generische, zueinander inverse Zugriffsfunktionen zwischen dem Urbild des Fixpunktes und dem Bild des Fixpunktes bieten wir die Funktionen

```
abstr :: 𝟙 + ℕ→ℕ
repr  :: ℕ→𝟙 + ℕ
```

an. Diese erscheinen zwar für diese einfache Datenstruktur etwas überzogen, doch bietet dieser allgemeine und homogene Ansatz Vorteile bei der Verwendung komplexer Datenstrukturen wie Listen und Bäumen, wenn wir auch dort mit den oben genannten Zugriffsfunktionen arbeiten. Im Kontext der natürlichen Zahlen bedeuten repr und abstr nichts anderes als die Vorgänger- bzw. die Nachfolgerfunktion.

Die generische Summe wiederum ist eine Datenstruktur, die Varianten bzw. Alternativen ausdrückt[5]. Im Zusammenhang mit der generischen Summe steht die Funktionssumme fsum (Abbildung 3.6(j)). Sie verarbeitet Varianten je nach Bildung durch linke oder rechte Injektion mit einer entsprechenden Funktion: Wird auf eine Variante appliziert, die durch eine linke Injektion inj1 gebildet wurde, so wird der linke Nachfolger der Funktionssumme angewandt (Abbildung 3.11(a)), andernfalls der rechte (Abbildung 3.11(b)).



(a) fsum angewandt auf linke Injektion

(b) fsum angewandt auf rechte Injektion

Abbildung 3.11: Semantik der Funktionssumme definiert über ihren Kontext

Mit diesem Rüstzeug können wir eine HOPS-konforme Version der Fakultät entwickeln, die die oben erwähnten generischen Zugriffsfunktionen verwendet. Die Funktionen fsum und Either sind austauschbar und bezeichnen die gleiche Funktion. fsum ist die historisch gewachsene Bezeichnung, Either verwenden wir zunehmend wegen des gleichnamigen Haskell-Konstruktors. Ebenso sind **rec** und Y unterschiedliche Namen für den gleichen Operator.

[5] Die Programmiersprache Pascal verwendet dazu variante Records.

Listing 3.3: Fakultät als Funktional (in HOPS)

```
factorial = rec f . λx . Either (Const 1) (λy . ∗ (x, f y)) (repr x)
```

Die Version aus Listing 3.3 ist in einer applikativen Weise verfasst. In Kapitel 4 werden wir sie in eine funktionale Form überführen, in der einige Bindungen herausfallen, die durch Terme mit **Kombinatoren höherer Ordnung** ersetzt werden.

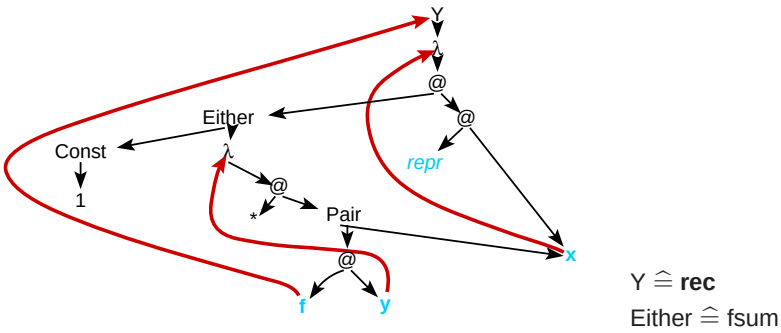


Abbildung 3.12: Fakultät in HOPS

Mit dem Ausdruck repr x ermitteln wir den Vorgänger des Parameters. Dieser besitzt eine variante Datenstruktur und weist entweder eine linke Injektion auf und repräsentiert damit die Zahl 0 oder ist eine rekursive Schachtelung von rechten Injektionen. Ist der Parameter nun die linke Injektion für die 0, so kommt die Konstantenfunktion Const 1 zum Tragen, andernfalls wird der rechte Nachfolger von fsum angewandt, der die rechte Injektion, also den Vorgänger, als Parameter erhält und im Rumpf x * f(x-1) berechnet[6]. Der zugehörige Graph ist in Abbildung 3.12 abgebildet.

Termkonstruktoren

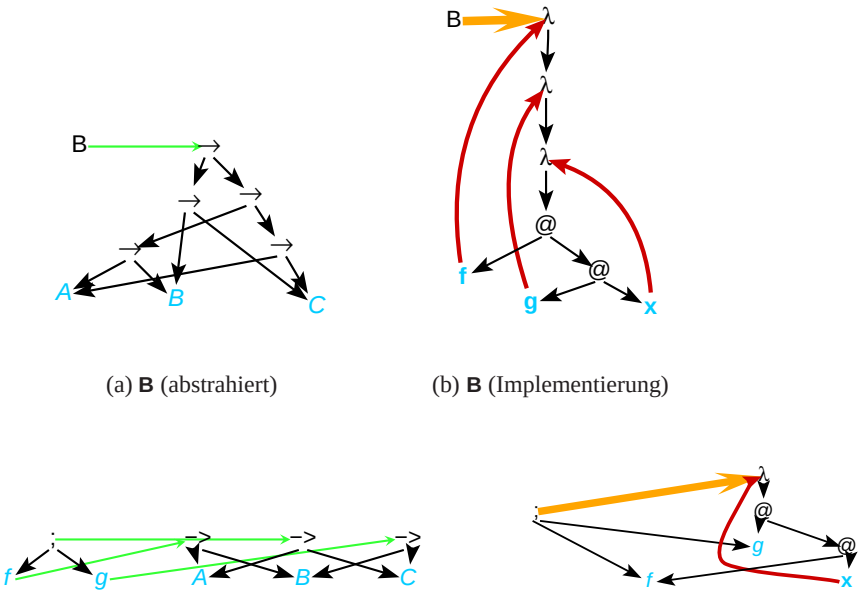
Häufig existieren in der HOPS-Bibliothek zwei Varianten eines Bausteins. Ein nullstelliger Termkonstruktor erzeugt durch Applikation auf einen Parameter ein Objekt von entsprechendem Typ und wird auch abstrahierte Version genannt, da der Knoten keine Nachfolger hat. Demgegenüber steht ein mehrstelliger Termkonstruktor, der ein Knoten mit mehreren Nachfolgern ist und nur im Kontext dieser Nachfolger definiert ist.[7] Ein mehrstelliger Konstruktor ist also nicht für sich allein existenzberechtigt.

@ inj1 *x*	inj1 als nullstelliger Konstruktor
inj1 *x*	inj1 als einstelliger Konstruktor

[6] Genauer: x-1 ergibt sich durch repr x, welches der Funktion von Either als Parameter x übergeben wird.
[7] In Abbildung 3.11 ist inj1 und inj2 jeweils als einstelliger Konstruktor (Funktionsobjekt) zu sehen.

Unter diesen beiden Typen von Konstruktoren (nullstellig, mehrstellig) kann der Entwickler die für die Anwendung adäquate Version wählen. Die Erfahrung in der Arbeit mit HOPS zeigt, dass die abstrahierte Version eines Bausteins oft vorteilhaft bei der Transformation ist, da man häufig auf explizite Regeln für diesen Baustein verzichten kann. Andererseits ist die Variante mit Nachfolgern beim Editieren eines Programms bequemer, da man sich hierdurch einige Knoten (insbesondere Applikationsknoten) erspart und somit der Graph kompakter bleibt.

In Abbildung 3.13 ist der **B**-Kombinator als nullstelliger Konstruktor abgebildet. In HOPS ist der Baustein unter '∘' bekannt und wird auch Funktionskomposition genannt. '∘' ist ein zweistelliger Konstruktor, also eine Version mit Nachfolgern.



(a) **B** (abstrahiert) (b) **B** (Implementierung)

(c) ; (d) ; (Implementierung)

Abbildung 3.13: Funktionskomposition in HOPS

Man sieht auf der linken Seite jeweils die Definition des Bausteins mit seiner Typisierung, auf der rechten Seite eine Regel, die den Baustein auf einfachere Bausteine zurückführt.[8]

[8] Bei dem Baustein '∘' ist zu beachten, dass der Parameterfluss umgekehrt zu dem des Kombinators **B** ist und folgendes gilt:

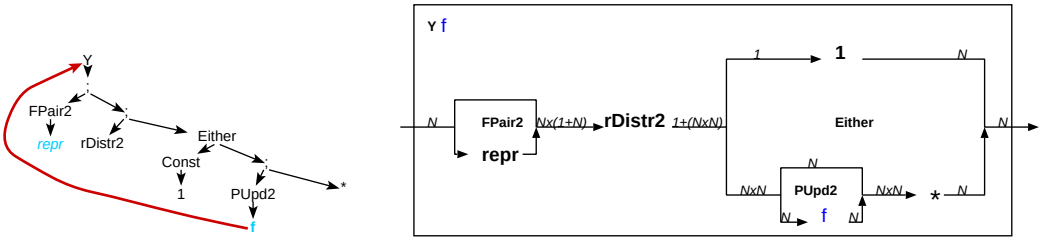
$$f \circ g = g \; \mathbf{B} \; f \qquad (\mathbf{B} \text{ als zweistelliger Konstruktor}) \quad \text{bzw.}$$
$$= @ \; (@ \; \mathbf{B} \; g) \; f \qquad (\mathbf{B} \text{ als nullstelliger Konstruktor})$$

Wir sind der Meinung, dass die Leserichtung von links nach rechts — auch im Hinblick auf einen Datenfluss –

3.4.2 Datenflussgraph

Hatten wir es in Abschnitt 3.4.1 mit Graphen zu tun, welche die Struktur eines Programms hervorheben, so wenden wir uns nun den Datenflussgraphen zu, die den Ablauf, respektive den Datenfluss, betonen. Diese sind nicht gesondert zu den DAGs zu sehen, sondern vielmehr ein weiterer Zugang, also eine alternative Visualisierung eines Programms.

Der Anwender entwickelt seinen Graphen in der ihm naheliegenden Sicht (DAG oder DFG); der korrespondierende Graph wird im Hintergrund jeweils stets neu berechnet und aktuell gehalten.



(a) Fakultät (DAG)

(b) Fakultät (DFG)

Abbildung 3.14: Verschiedene Sichtweisen eines Programms

Ein wesentlicher Vorteil von DFGs ist es, dass die Typisierungen der Werte an den Datenflusskanten mit abzulesen sind; in der DAG-Darstellung ist die Typisierungsinformation in einen zweiten Graphen ausgelagert.

Sehen wir uns eine veränderte Version der Fakultät Abbildung 3.14(a) an, die mit den in Kapitel 4 vorgestellten Werkzeugen in eine Form gebracht wurde, in der Variablenbindungen entfernt wurden. Hier sind nur noch Funktionen und keine Konstanten vorhanden. Diese funktionale Form besitzt eine Datenflussvisualisierung in Abbildung 3.14(b), in der ein Datenfluss von links nach rechts ersichtlich ist. Daten werden durch einzelne Funktionen geleitet und von diesen transformiert. Kein Zugriff auf eine gebundene bzw. freie Variable[9] ist nötig; der Datenfluss wird im Besonderen durch die Funktionskomposition ‘;’ erreicht.

Auf den ersten Blick mag es so aussehen, dass der DFG wesentlich mehr Platz konsumiert, doch liegt dies zu einem großen Teil an der zusätzlichen Typinformation, die man ohne weiteres unterdrücken kann.

vorteilhafter ist.

[9] Sehen wir einmal von der durch den Fixpunktoperator Y gebundenen Funktion f ab.

3.5 Weitere Merkmale

Im Folgenden gehen wir auf einige weitere Merkmale der HOPS-Umgebung kurz ein.

Generische Durchlaufalgorithmen

HOPS verfügt im Gegensatz zu vielen anderen Programmiersystemen über keine eingebaute Semantik. Vielmehr steht es dem Benutzer frei, eine oder auch mehrere Semantiken für einen Sprachschatz durch die Definition von unterschiedlichen Durchlaufungen zu definieren. Dadurch gewinnt der Benutzer eine Flexibilität, Programme nach unterschiedlichen Berechnungsmodellen wie z. B. einer Eager-Auswertung oder einer Lazy-Auswertung abzuarbeiten.

Eine Durchlaufung definiert im Kontext einer Transformationsstrategie, also der Anwendung mehrerer Regeln hintereinander, die Reihenfolge der anwendbaren Regeln. So kann eine Strategie lauten, dass Graphen von unten nach oben ausgewertet werden, eine andere beschreibt die Auswertung von oben nach unten. HOPS verfügt also nicht nur über die Möglichkeit, eine eigene Sprache zu definieren, sondern auch die Fähigkeit, deren Semantik anzugeben.

Andererseits können Durchlaufungen für eine flexible Codegenerierung verwendet werden. So sind in HOPS Durchläufe definiert, die aus einem Programmgraphen entsprechenden Haskell-Code erzeugen, der ohne weitere Nachbearbeitung in einem Interpreter sofort ablauffähig ist.

Visuelle Programmierung

Nachdem wir es in HOPS mit verschiedenen Arten von Graphen zu tun haben, liegt es natürlich nahe, dem Benutzer eine graphische Benutzeroberfläche bereitzustellen, die es erlaubt, durch Interaktion mit einer Maus Programme zu erstellen. Dazu stehen umfangreiche Graphbearbeitungsfunktionen wie Einfüge- und Löschoperationen bereit, die ein komfortables Arbeiten mit Programmobjekten erlauben.

Modulsystem

Um innerhalb einer Entwicklungsumgebung Programmpakete vernünftig strukturieren zu können, wurde ein Modulsystem entwickelt, das zu bekannten Modulsystemen vergleichbare Merkmale wie Importabhängigkeiten, Namensräume und Ähnliches aufweist. Darüber hinaus verfolgt HOPS den Ansatz des *Literate Programming*, in dem Programmelemente und dazu gehörige Dokumentation bzw. Kommentare zu einem Dokument verschmolzen werden. Letztgenanntes enthält neben den Typelementen (Bauvorschriften für Funktionen) auch Regeln, Texte und nicht zuletzt Beispielgraphen, die die Idee hinter definierten Sprachelementen oder Einzelalgorithmen näher erläutern.

Im Gegensatz zum Ansatz von Knuth, der Literate Programming auf eine bestehende Programmiersprache[10] aufsetzte, ist ein formatiertes Dokument in HOPS ein integrierter und zentraler Bestandteil der Programmierumgebung mit einem eigenen Editor.

[10] Knuth wählte Pascal, doch gibt es zu vielen anderen textorientierten Sprachen vergleichbare Implementierungen.

KAPITEL 4

Kombinatoren, Transformationen und Datenfluss

Kombinatoren sind ein bekanntes Konzept in der Logik und Informatik. Sie werden häufig zur Codeerzeugung für funktionale Sprachen verwendet. Wir werden in diesem Kapitel zeigen, dass Kombinatoren nicht nur in Form einer Implementierungstechnik in einem Compiler von Nutzen sein können, sondern auch zur Formulierung von Programmen auf Quellcodeebene von Vorteil sind.

Eine Variablenbindung, wie sie aus praktisch allen gängigen Programmiersprachen bekannt ist, hat die Eigenschaft, dass sie einen Ausdruck abhängig von einem Kontext macht, nämlich von der bindenden Stelle. Daher erfordert die Auswertung eines Ausdrucks auch die Berücksichtigung seines Kontexts. Der Kontext hingegen erschwert die Definition von Regeln, die mögliche Bindungen berücksichtigen muss. Zudem vereinfacht ein Ausdruck ohne Kontext den Übergang zu einem Datenfluss. Eine Verwendung von Kombinatoren hilft, Programme von Variablenbindungen zu befreien.

Wir geben zunächst einen Überblick über bekannte Kombinatoren. Anschließend beschreiben wir einen verbreiteten Abstraktionsalgorithmus, der einen beliebigen Term zu einem semantisch äquivalenten Term ohne Bindungen umformt, indem er geeignete Kombinatoren einführt. Wir werden dann in diesen Algorithmus Optimierungen integrieren, die bisher in der Literatur lediglich als separate Regeln formuliert waren. Damit steht uns ein geschlossener Abstraktionsalgorithmus inklusive Optimierungen zur Verfügung.

In einem weiteren Abschnitt zeigen wir, dass mit der Kombinatorform eines Ausdrucks die Visualisierung mittels eines Datenflusses besonders einfach möglich ist. Wir definieren für einen Datenfluss geeignete Randbedingungen, die insbesondere eine kompositionale Hintereinanderschaltung von Funktionen unterstützen. Schließlich geben wir einen Satz von Kombinatoren an,

der diesen Bedingungen genügen wird.

Inhaltsverzeichnis

4.1 Motivation

In diesem Kapitel wollen wir eine Art der Programmierung angeben, die auf den Einsatz von freien Variablen völlig verzichtet. Dieses Programmierparadigma hat Schönfinkel bereits in [Schönfinkel 1924] vorgestellt. Dazu führte er das Konstrukt des *Kombinators* ein, welches eine Funktion **ohne freie Variable** darstellt. Damit ist eine applikative Verknüpfung möglich, die ohne den Abstraktionsoperator λ auskommt und sich nur auf die Anwendung der Applikation und von Kombinatoren stützt.

Die Verwendung von Kombinatoren kommt heutzutage vor allem bei der Implementierung funktionaler Sprachen zum Tragen. Hier versucht der Compilerbauer, durch eine extensive Programmanalyse und eine Reihe von Programmtransformationen effizienten Code zu erzeugen. Wir sind der Meinung, dass die Technologie der Kombinatoren nicht nur als Hintergrundprozess eines Programms wie das eines Compilers einsetzbar ist, sondern auch ein geeignetes Werkzeug im Bereich der interaktiven, computergestützten Programmtransformation ist.

Durch die Einführung von geeigneten Kombinatoren können wir das Auftreten von freien Variablen in geschachtelten λ-Ausdrücken ([Field und Harrison 1988, S. 267]) eliminieren. Bei der Formulierung und Anwendung von Transformationsregeln zeigt sich, dass Programme bzw. Graphen ohne freie Variable leichter zu behandeln sind. So kann es durch das Vorhandensein freier Variablen, die an einer äußeren Stelle gebunden sind, vorkommen, dass beim Anwenden von Regeln Teile eines Programms dupliziert werden müssen[1]. Daher ist es wünschenswert, ein Programm derart umzuformen, dass Bindungen eliminiert werden und dadurch in den inneren Teilen eines Programms keine freien Variablen mehr auftreten.

Dadurch gewinnen wir eine Gestalt, die Transformationen besser zugänglich ist, als dies Ausdrücke mit gebundenen Variablen sind. Dieses Ziel erreichen wir durch die Einführung spezieller Kombinatoren.

Unser Ziel muss also die Erstellung eines Programms sein, das frei von Variablen ist und somit auf Bausteinen mit Bindungen weitgehend verzichtet. Eine ähnliche Form weisen auch Terme in FP (Abschnitt 2.4.2) auf. Diese sind ebenfalls frei von Objekten und Variablen und nur aus Funktionen zusammengesetzt.

4.2 Beispiel

Variablenfreie Programmierung unter Verwendung von Kombinatoren zeigen wir exemplarisch an zwei kleinen Beispielen.

Shellskript unter UNIX

Im Umfeld des Betriebssystems UNIX ist es Tradition, kleine Shellskripten zu schreiben, die auf dem Werkzeugkasten der vielen kleinen Systemprogramme wie *grep* und *awk* aufbauen. Diese Idee verfolgen wir nun bei der Erstellung eines kleinen Programms.

Wir wollen zu Beginn eine einfache Funktion entwickeln, die eine Datei einliest, diese nach einem Muster durchsucht und die Anzahl der Vorkommnisse eines bestimmten Musters in unterschiedlichen Zeilen ermittelt.

Wir formulieren diese Funktion zuerst in der gewohnten mathematischen Weise unter Zuhilfenahme UNIX-typischer Funktionen:

$$f = g \circ h \circ i \qquad :: \text{Datei} \rightarrow \text{Integer}$$

[1] Voraussetzung für dieses Phänomen ist, dass auf einen Teil eines Graphen mehrere Eingangskanten zeigen.

mit

$g =$ **wc** -l	:: String → Integer	Zählen der Zeilen
$h =$ **grep** *Muster*	:: String → String	Filtern der Zeilen, die *Muster* enthalten
$i =$ **cat**	:: Datei → String	Einlesen einer Datei

Wenn wir die Funktion f auf einen Parameter — hier eine *Datei* — anwenden, so gilt:

$$f\ Datei = (\ g \circ h \circ i\ )\ Datei = g(\ h(\ i(Datei)\ )\ )$$

Die bindungsfreie Programmierung in $(g \circ h \circ i)$ fällt hier sofort auf. Hier haben wir es ausschließlich mit einer Komposition von Funktionen zu tun, die sequenziell — datenflussartig — mit dem Kombinator '$\circ$' hintereinander geschaltet sind.

Diese Art der variablenfreien Programmierung bzw. der Hintereinanderschaltung von Funktionen besitzt nicht nur theoretischen Wert, sondern kann direkt in eine Skriptsprache von UNIX umgesetzt werden.

Wir benutzen dazu den Mechanismus von Pipes zum Aufbau einer Interprozesskommunikation wie folgt:

```
cat | grep Muster | wc -l
```

Diese Anweisung entspricht der mathematischen Version wie vorstehend mit dem Unterschied, dass die Komposition '|' von links nach rechts — vergleichbar dem relationalen Operator '$\fatsemi$' — zu lesen ist. So könnte eine Instanz des folgenden Anfragemusters

```
(cat | grep Muster | wc -l) Datei
```

das Ergebnis

```
2
```

liefern.

Nachdem wir gesehen haben, wie ein variablenfreies Programm aussehen kann, stellt sich die Frage, ob es nicht eine Möglichkeit gibt, ein bereits vorhandendes Programm mit Bindungen in ein äquivalentes ohne Bindungen umzuwandeln. Ohne auf weitere Details des Algorithmus einzugehen, führen wir dies an einer kleinen Funktion vor, die in der Notation des λ-Kalküls beschrieben ist.

Funktion im λ-Kalkül

Zunächst beschreiben wir eine Funktion im λ-Kalkül

$$\lambda y.(\ \lambda x.(+\ x)\ 3\ )\ y \tag{4.1}$$

die lediglich den Wert 3 zu einem übergebenen Parameter addiert. Übertragen wir diesen Term in die Notation von HOPS, so liest sich der Ausdruck wie in Abbildung 4.1.

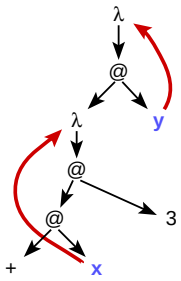


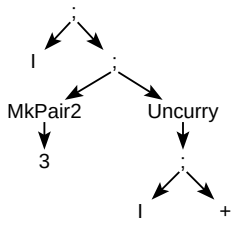
Abbildung 4.1: '$\lambda y.(\ \lambda x.(+\ x)\ 3\ )\ y$' in HOPS-Notation

Wenn wir in dieser Funktion nach einem allgemeinen Verfahren mittels der Kombinatoren **S**, **K** und **I** alle gebundenen Variablen eliminieren, so erhalten wir den folgenden, semantisch äquivalenten Ausdruck:

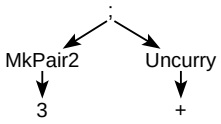
$$\lambda y.(\ \lambda x.(+\ x)\ 3\ )\ y \quad \Rightarrow \quad \mathbf{S}\ (\mathbf{S}\ (\mathbf{S}\ (\mathbf{KS})\ (\mathbf{S}\ (\mathbf{S}\ (\mathbf{KS})(\mathbf{S}(\mathbf{KK})(\mathbf{K}+))))(\mathbf{KI}))\ (\mathbf{S}\ (\mathbf{KK})\ (\mathbf{K}3)))\ \mathbf{I}$$

Ohne auf die Bedeutung der einzelnen Kombinatoren näher einzugehen, die im nächsten Abschnitt definiert werden, ist an dem vorstehenden Ausdruck sofort ersichtlich, dass das Resultat bedeutend komplexer und unübersichtlicher geworden ist. Der Grund liegt vor allem an dem verwendeten Satz an Standardkombinatoren.

Wir suchen nun eine geeignete Menge von Kombinatoren, die die Basismenge von HOPS-Bausteinen wie **FPair** und **FSum** umfasst und zudem die Codeexplosion vermeidet. Dazu werden wir geeignete Kombinatoren vorstellen. Das Resultat der Transformation des Graphen aus Abbildung 4.1 ist in Abbildung 4.2 zu sehen, bei dem neue, adäquate Kombinatoren verwendet werden.



(a) Programmgraph ohne Bindungen



(b) Programmgraph ohne Bindungen (optimiert)

Abbildung 4.2: Transformation mit HOPS-Kombinatoren

Hier sehen wir auf der linken Seite den entstandenen Programmgraph ohne gebundene Variablen. Die rechte Seite zeigt den Term nach einer weiteren Optimierung, die darin besteht, dass die Identitätsfunktion **I** bei der Hintereinanderschaltung ‘$\frac{9}{9}$’ herausfällt[2].

Interessant ist nicht nur der einfachere Graph, sondern auch die Nähe zu einer Datenflussdarstellung. Im Laufe des Kapitels werden wir eine Übersetzung eines Programmgraphen in einen entsprechenden Datenflussgraphen angeben. Dessen Ergebnis ist in Abbildung 4.3 zu sehen.

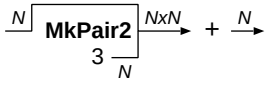


Abbildung 4.3: Programmgraph als Datenfluss

An dieser Stelle sei darauf hingewiesen, dass sämtliche Übersetzungen und Verfahren, die in diesem Kapitel entwickelt wurden, in dem System HOPS gerechnet werden. Wir haben lediglich an einigen Stellen eine textuelle Syntax gewählt, da diese in der einschlägigen Literatur gebräuchlich ist und so für den Leser, der weniger an das Lesen von Graphen gewöhnt ist, auf den ersten Blick zugänglicher sein mag.

4.3 Standardkombinatoren

Kombinatoren sind in der Mathematik seit Schönfinkel ([Schönfinkel 1924]) und Curry ([Curry 1930]) bekannt. Gemäß deren fundamentalen Theorem kann der gesamte λ-Kalkül in eine Theorie von Kombinatoren umgewandelt werden, die sich ausschließlich auf eine Basisoperation, die Applikation, stützt (vgl. [Révész 1988]).

Turner gelang der Bogen zur angewandten Informatik. Er wandte Ergebnisse aus der kombinatorischen Logik auf applikative Sprachen wie LISP an und verfeinerte in [Turner 1979] das Verfahren der *Variablenabstraktion* von Schönfinkel und Curry, in dem sukzessive alle gebundenen Variablen entfernt werden und der resultierende Ausdruck von einer Maschine in effizienter Weise abgearbeitet werden kann. Diese Technik wird im folgenden Abschnitt näher erläutert. Sie ist Voraussetzung für eine mechanische Übersetzung in einen datenflussartigen Ausdruck, der frei von gebundenen Variablen ist.

Freie Variablen in einem λ-Ausdruck entsprechen globalen Variablen, die in einem äußeren Block definiert sind. Gebundene Variablen entsprechen formalen Parametern oder lokalen Variablen. λ-Ausdrücke, die keine freien Variablen enthalten, werden *Kombinatoren* genannt. Sie sind

[2] Sehen wir uns noch einmal den Ausgangsausdruck mit Bindungen aus Gleichung (4.1) an, so erkennen wir eine mögliche η-Reduktion mit

$$\lambda x.M\,x \Rightarrow M \qquad \text{wenn } x \notin FV(M)$$

als Optimierungsschritt.

von ihrem Kontext unabhängig. Damit können sehr viel leichter Codeumformungen vorgenommen werden, ohne dass stets der Kontext auf mögliche Abhängigkeiten berücksichtigt werden muss.

Zudem lässt ein Ausdruck mit Kombinatoren eine einfachere Datenflussdarstellung zu. Die Bedeutung eines Datenfluss-Bausteins ist nur von dem eingehenden Datenstrom abhängig und wird nicht zusätzlich durch einen Variablen-Kontext beeinflusst.

Die folgenden Abschnitte beschreiben ein Verfahren, wie aus einem beliebigen λ-Ausdruck ein semantisch äquivalenter Ausdruck entsteht, der ausschließlich aus Kombinatoren aufgebaut ist und damit die gewünschte Eigenschaft der Kontextunabhängigkeit erreicht wird.

4.3.1 Die allgemeine Übersetzungsfunktion

Wir geben nun die allgemeine Übersetzungsfunktion *comb* an, die einen λ-Ausdruck mit beliebig vielen Abstraktionen in sein Kombinator-Äquivalent konvertiert, in dem der Abstraktionsoperator λ nicht mehr auftaucht. Die Funktion lautet:

$comb(v) = v$	für v Variable
$comb(c) = c$	für c Konstante
$comb(\lambda x.E) = [x]\ comb(E)$	für E Ausdruck
$comb(E_1\ E_2) = comb(E_1)\ comb(E_2)$	für E_1, E_2 Ausdruck

Tabelle 4.1: Übersetzungsfunktion *comb*

Die rekursiv definierte Funktion *comb* stützt sich wiederum auf einen Abstraktionsalgorithmus $[x]$, der bei jedem Auftreten des Abstraktionsoperators λ aufgerufen wird.

Der Abstraktionsalgorithmus kann unterschiedlich definiert werden. Jede Variante baut auf einem Satz von Kombinatoren auf. In den nächsten Abschnitten definieren wir eine minimale Menge an Kombinatoren mit einem entsprechenden Abstraktionsalgorithmus und erweitern sukzessive den Algorithmus wie auch den dazugehörigen Kombinatorsatz.

4.3.2 Kombinatoren S, K und I

Zunächst führen wir drei Kombinatoren **S** (*distributor*), **K** (*cancellator*) und **I** (*identity*) ein, die durch die Gleichungen in Tabelle 4.2 definiert sind.

$$\begin{aligned} \mathbf{S}\ f\ g\ x &= f\ x\ (g\ x) \\ \mathbf{K}\ x\ y &= x \\ \mathbf{I}\ x &= x \end{aligned}$$

Tabelle 4.2: Definition von **S**, **K** und **I**

Auf diese Kombinatoren wird sich der Abstraktionsalgorithmus $[x]$ stützen. Man kann sogar zeigen, dass die ausschließliche Verwendung der Kombinatoren **S** und **K** ausreichend ist, um einen beliebigen λ-Ausdruck zu transformieren, da der Kombinator **I** auf die beiden anderen mit folgender Gleichung zurückgeführt werden kann:

$$\mathbf{I} = \mathbf{S}\ \mathbf{K}\ \mathbf{K}$$

Jedoch ist es oft sehr mühsam und ineffizient, den Identitätskombinator **I** wegzulassen. Daher nehmen wir ihn zu unseren Basiskombinatoren hinzu.

Nun geben wir in Tabelle 4.3 den Abstraktionsalgorithmus, also die Übersetzung einer Kombination E an, in der die Variable x eliminiert werden soll.

$[x]\ v \Rightarrow \mathbf{I}$	wenn $v \equiv x$ Variable
$[x]\ c \Rightarrow \mathbf{K}\ c$	wenn c Konstante
$[x]\ v \Rightarrow \mathbf{K}\ v$	wenn $v \not\equiv x$ Variable
$[x]\ (E_1\ E_2) \Rightarrow \mathbf{S}\ ([x]\ E_1)\ ([x]\ E_2)$	sonst

Tabelle 4.3: Abstraktionsalgorithmus mittels **S**, **K** und **I**

4.3.3 Kombinatoren B und C

Eine große Bedeutung der genannten Kombinatoren ergibt sich aus ihrer mathematischen Eleganz und Einfachheit. Da jedoch die Länge der mittels dieser Kombinatoren gebildeten Ausdrucksäquivalente mit exponentieller Komplexität wächst, ist es angebracht, einige Vereinfachungen einzuführen, die das Größenwachstum eindämmen. Dazu geben wir mit Hilfe der Gleichungen aus Tabelle 4.4 zwei weitere Kombinatoren **B** (*compositor*) und **C** (*permutator*) an, deren Definition sich auf Curry ([Curry und Feys 1958]) stützt.

$$\mathbf{B}\ f\ g\ x = f\ (g\ x)$$
$$\mathbf{C}\ f\ x\ y = f\ y\ x$$

Tabelle 4.4: Definition von **B** und **C**

Darauf aufbauend lassen sich die Optimierungen aus Tabelle 4.5 angeben, die ebenfalls auf [Curry und Feys 1958] zurück gehen.

$$\mathbf{S}\ (\mathbf{K}\ E_1)\ (\mathbf{K}\ E_2) \Rightarrow \mathbf{K}\ (E_1\ E_2) \tag{4.2}$$
$$\mathbf{S}\ (\mathbf{K}\ E)\ \mathbf{I} \Rightarrow E \tag{4.3}$$
$$\mathbf{S}\ (\mathbf{K}\ E_1)\ E_2 \Rightarrow \mathbf{B}\ E_1\ E_2 \tag{4.4}$$
$$\mathbf{S}\ E_1\ (\mathbf{K}\ E_2) \Rightarrow \mathbf{C}\ E_1\ E_2 \tag{4.5}$$

Tabelle 4.5: Optimierung u. a. durch **B** und **C**

Vor allem die Vereinfachung in Gleichung (4.2) führt dazu, dass applikative Teilausdrücke — das sind Teilausdrücke, in denen nicht diejenige Variable enthalten ist, von der abstrahiert wird — erhalten bleiben und nicht auf mehrere zusätzliche Kombinatoren aufgespalten werden. Es werden dadurch Mehrfachberechnungen von Teilausdrücken vermieden. Die Gleichungen 4.4 und 4.5 führen die zusätzlichen Kombinatoren **B** und **C** ein und greifen, wenn einer der beiden Operanden von **S** konstant ist.

In Lehrbüchern werden diese Optimierungen stets separat zum Übersetzungsalgorithmus angegeben. Um diese Regeln nicht erst nachträglich anzuwenden, können wir sie auch direkt in den Transformationsalgorithmus integrieren. Tabelle 4.6 führt einen geschlossenen Algorithmus an, den der Autor bisher in keiner Veröffentlichung entdecken konnte.

$[x]\ v \Rightarrow \mathbf{I}$	$v \equiv x$ Variable
$[x]\ E \Rightarrow \mathbf{K}\ E$	wenn $x \notin FV(E)$
$[x]\ (E_1\ E_2) \Rightarrow \mathbf{B}\ E_1\ ([x]\ E_2)$	wenn $x \notin FV(E_1)$
$[x]\ (E_1\ E_2) \Rightarrow \mathbf{C}\ ([x]\ E_1)\ E_2$	wenn $x \notin FV(E_2)$
$[x]\ (E_1\ E_2) \Rightarrow \mathbf{S}\ ([x]\ E_1)\ ([x]\ E_2)$	sonst

Tabelle 4.6: Abstraktionsalgorithmus mittels **S**, **K**, **I**, **B** und **C**

4.3.4 Kombinatoren S', B' und C'

Turner führte in [Turner 1979] drei weitere Kombinatoren **S'**, **B'** und **C'** ein, die sich wie **S**, **B** und **C** verhalten mit der Ausnahme, dass sie noch über einen weiteren Parameter bei der Anwendung hinausreichen. Diese sind in Tabelle 4.7 definiert.

$$\begin{aligned}
\mathbf{S'}\ k\ E_1\ E_2\ x &= k\ (E_1\ x)\ (E_2\ x)\\
\mathbf{B'}\ k\ E_1\ E_2\ x &= k\ (E_1\ (E_2\ x))\\
\mathbf{C'}\ k\ E_1\ E_2\ x &= k\ (E_1\ x)\ E_2
\end{aligned}$$

Tabelle 4.7: Definition von **S'**, **B'** und **C'**

Der Abstraktionsalgorithmus aus Tabelle 4.6 wird nun um die Zeilen

$$\begin{aligned}
[x]\ (E_1\ (E_2\ E_3)) &\Rightarrow \mathbf{B'}\ E_1\ E_2\ ([x]\ E_3) && \text{wenn } x \notin FV(E_1) \cup FV(E_2)\\
[x]\ (E_1\ E_2\ E_3) &\Rightarrow \mathbf{C'}\ E_1\ ([x]\ E_2)\ E_3 && \text{wenn } x \notin FV(E_1) \cup FV(E_3)\\
[x]\ (E_1\ E_2\ E_3) &\Rightarrow \mathbf{S'}\ E_1\ ([x]\ E_2)\ ([x]\ E_3) && \text{wenn } x \notin FV(E_1)
\end{aligned}$$

erweitert.

Das modifizierte Regelwerk für Vereinfachungen ist in Tabelle 4.8 abgebildet, dessen Regeln an allen möglichen Stellen anzuwenden sind.

$$\begin{aligned}
\mathbf{S}\ (\mathbf{B}\ E_1\ E_2)\ E_3 &\Rightarrow \mathbf{S'}\ E_1\ E_2\ E_3\\
\mathbf{B}\ E_1\ (\mathbf{B}\ E_2\ E_3) &\Rightarrow \mathbf{B'}\ E_1\ E_2\ E_3\\
\mathbf{C}\ (\mathbf{B}\ E_1\ E_2)\ E_3 &\Rightarrow \mathbf{C'}\ E_1\ E_2\ E_3
\end{aligned}$$

Tabelle 4.8: Optimierung durch **S'**, **B'** und **C'**

Der Trick der zusätzlichen Kombinatoren besteht darin, dass sehr häufig im ersten Parameter konstante (nicht von der zu abstrahierenden Variablen abhängige) Ausdrücke vorkommen, die die neu eingeführten Kombinatoren unverändert lassen und damit der Codeaufblähung vorbeugen. Dadurch konnte Turner die Übersetzung des Ausdrucks von einer quadratischen auf eine lineare Komplexität drücken.

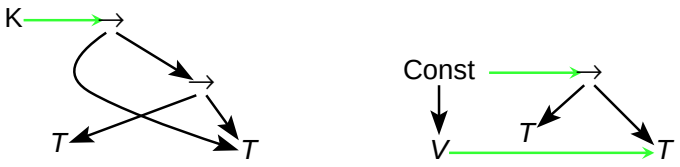
Der vollständige Abstraktionsalgorithmus ist noch einmal in Tabelle 4.9 zusammengefasst, wobei sämtliche Optimierungen — also Einführung der erweiterten Kombinatoren — bereits während der Transformation ausgeführt werden.

$[x]\ v \Rightarrow \mathbf{I}$	$v \equiv x$ Variable
$[x]\ E \Rightarrow \mathbf{K}\ E$	wenn $x \notin FV(E)$
$[x]\ (E_1\ (E_2\ E_3)) \Rightarrow \mathbf{B'}\ E_1\ E_2\ ([x]\ E_3)$	wenn $x \notin FV(E_1) \cup FV(E_2)$
$[x]\ (E_1\ E_2\ E_3) \Rightarrow \mathbf{C'}\ E_1\ ([x]\ E_2)\ E_3$	wenn $x \notin FV(E_1) \cup FV(E_3)$
$[x]\ (E_1\ E_2\ E_3) \Rightarrow \mathbf{S'}\ E_1\ ([x]\ E_2)\ ([x]\ E_3)$	wenn $x \notin FV(E_1)$
$[x]\ (E_1\ E_2) \Rightarrow \mathbf{B}\ E_1\ ([x]\ E_2)$	wenn $x \notin FV(E_1)$
$[x]\ (E_1\ E_2) \Rightarrow \mathbf{C}\ ([x]\ E_1)\ E_2$	wenn $x \notin FV(E_2)$
$[x]\ (E_1\ E_2) \Rightarrow \mathbf{S}\ ([x]\ E_1)\ ([x]\ E_2)$	sonst

Tabelle 4.9: Vollständiger Abstraktionsalgorithmus mit Optimierungen

4.3.5 Beziehung zu HOPS-Kombinatoren

In der Arbeit mit HOPS hat sich eine Reihe wichtiger und grundlegender Bausteine herauskristallisiert. Einige der im letzten Abschnitt auftretenden Kombinatoren finden sich, wenn auch mit teilweise abweichender Bezeichnung, darunter wieder. Das typische Entwickeln mit HOPS ist so durch Nutzung allgemeiner, leistungsfähiger Kombinatoren geprägt.



(a) **K** (abstrahiert) (b) **Const**

Abbildung 4.4: Konstantenfunktion in HOPS

Zwei exemplarische Kombinatoren mögen dies veranschaulichen. In der Abbildung 4.4 geben wir die Konstantenfunktion in zwei Ausprägungen an. Zum einen die abstrahierte Version, die wir unter dem Namen **K** kennen (**const** in HOPS), daneben **Const**, die Konstantenfunktion mit einem Nachfolger. Die Identitätsfunktion **I** (**id** in HOPS) ist in Abbildung 4.5 zu sehen.

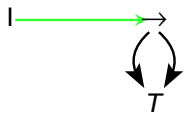


Abbildung 4.5: Identitätsfunktion in HOPS

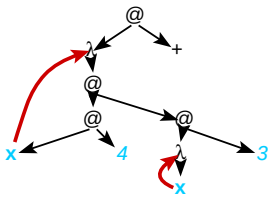
Die Abbildungen der Kombinatoren umfassen jeweils ihre Typisierungen, charakterisiert durch die Typisierungseinstiege (grüne Pfeile, →).

4.3.6 Beispiel

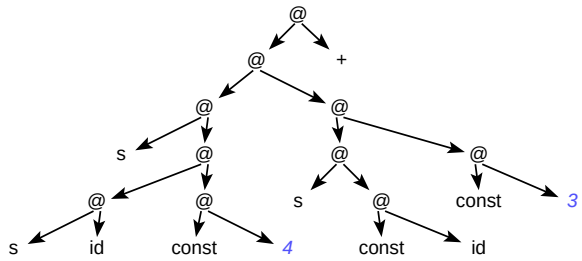
Betrachten wir nun die Kombinatoren in der Anwendung. In dem Ausdruck

$$(\lambda x.(x\ 4)\ ((\lambda x.x)\ 3))\ +$$

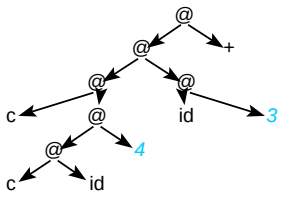
wollen wir die gebundene Variable x eliminieren. Der Term ist in der entsprechenden Notation von HOPS in Abbildung 4.6(a) abgebildet.



(a) Ausgangsausdruck



(b) Ausdruck in Kombinatorfassung (**S**, **K**, **I**)



(c) Ausdruck in erweiterter Kombinatorfassung (**S**, **K**, **I**, **B**, **C**)

Abbildung 4.6: Transformation in Kombinatorfassung

Abbildung 4.6(b) zeigt den mit Hilfe des Algorithmus aus Tabelle 4.3 transformierten Ausdruck[3]. Hier ist gut ersichtlich, wie sich der Term — vor allem bei der Elimination mehrerer geschachtelter λ-Ausdrücke — aufbläht. Durch die Einführung der erweiterten Kombinatorenmenge lässt sich der Ausdruck nun mittels des Algorithmus aus Tabelle 4.6 erheblich komprimieren (Abbildung 4.6(c)).

4.3.7 Datenfluss

Wir haben in den voran gegangenen Abschnitten einen Weg aufgezeigt, wie wir aus einem beliebigen Ausdruck einen äquivalenten Ausdruck gewinnen, der keinen Abstraktionsoperator λ mehr aufweist. Stattdessen wurden je nach Ausprägung des Abstraktionsalgorithmus $[x]$ unterschiedliche Kombinatoren **S**, **K** usw. eingeführt.

Wir zeigen nun, dass es für diese Standardkombinatoren besonders einfache und kanonische Datenflussdarstellungen gibt und erleichtern damit den Übergang von einer Termgraphdarstellung zu einer Datenflussdarstellung. Die Visualisierungen geben anschaulich den Fluss des gebundenen Parameters der ursprünglichen Abstraktion wieder.

Standardkombinatoren S, K und I

In Abbildung 4.7 sind die Kombinatoren **S**, **K** und **I** in einer Datenflussversion angegeben. Hier sehen wir, dass die Terme **I** und **K** *y* Funktionen erster Ordnung[4] entsprechen, also einem Datenfluss, dessen Bausteine Ein- und Ausgabe von Daten nullter Ordnung darstellen.

Im Gegensatz dazu steht der **S**-Kombinator, der auf einer nachgeschalteten Applikation aufbaut (Abbildung 4.7(c)). Der Datenfluss zwischen den beiden Knoten **f** und @ ist nun höherer Ordnung, da auf der Datenleitung eine Funktion fließt, die durch die abstrahierte Applikation (@) zur Anwendung gebracht wird. Der Datenfluss höherer Ordnung kann wiederum beseitigt werden, indem wir eine äquivalente Darstellung einer Funktionsapplikation wählen, in der einlaufende Kanten auf eine currysierte Funktion nacheinander zu applizieren sind (Abbildung 4.7(d)). Der Knoten **f** bezeichnet in beiden Abbildungen die gleiche, currysierte Funktion. Mehrere einlaufende Kanten beschreiben nun aufeinander folgende Applikationen auf eine currysierte Funktion.

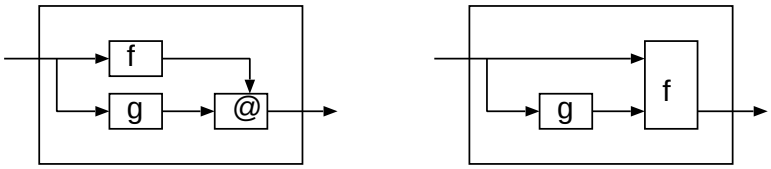
Standardkombinatoren B und C

Die Kombinatoren **B** und **C** sind in Abbildung 4.8 dargestellt. Hier sehen wir, dass der **B**-Kombinator (Abbildung 4.8(a)), der der Funktionskomposition entspricht, ein Spezialfall des **S**-Kombinators ist, in dem das erste Argument *f* nicht mit dem eingehenden Datenstrom versorgt wird. Das Konstrukt **B** *f g* ist ebenfalls in der wünschenswerten Datenflussform erster Ordnung, in der keine Funktionen auf Kanten fließen.

[3] Der Knoten const entspricht dem Kombinator **K**, der Knoten id ist äquivalent zu **I**.

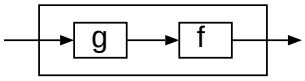
[4] Die Ordnung einer Funktion beschreibt die 'Komplexität' ihrer Parameter. Eine Funktion nullter Ordnung ist eine Funktion ohne Parameter, also gleichbedeutend mit einer Konstante. Eine Funktion erster Ordnung besitzt Parameter mit nullter Ordnung; eine Funktion zweiter Ordnung besitzt Parameter erster Ordnung und wird auch als Funktional bezeichnet.

(a) **I** (b) **K** *y*

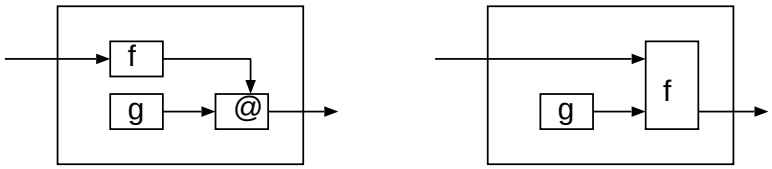


(c) **S** *f g* (d) **S** *f g* (currysiert)

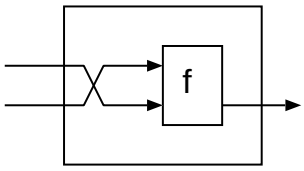
Abbildung 4.7: Kombinatoren **S**, **K** und **I**



(a) **B** *f g*



(b) **C** *f g* (c) **C** *f g* (currysiert)



(d) **C** *f* (teilappliziert, currysiert)

Abbildung 4.8: Kombinatoren **B** und **C**

Der **C**-Kombinator in der Form **C** f g hingegen, der sein zweites Funktionsargument g nicht mit dem Datenstrom verbindet, verlangt zwar nach einer nachgeschalteten Applikation (Abbildung 4.8(b)), die jedoch wieder eliminiert werden kann (Abbildung 4.8(c)). Auch dem Term **C** f lässt sich eine natürliche Datenflussdarstellung zuordnen. Ist der **C**-Kombinator nur auf ein Argument f angewandt, so erzeugt der Kombinator eine neue, semantisch identische Funktion, in der lediglich die Argumente vertauscht sind (Abbildung 4.8(d)).

Die Optimierung aus Gleichung (4.4) wird aus der Sicht des Datenflusses besonders offensichtlich. Die Vereinfachung

$$\mathbf{S}\ (\mathbf{K}\ f)\ g \Rightarrow \mathbf{B}\ f\ g$$

ist in Abbildung 4.9 zu sehen. In der Datenflussdarstellung erkennt man, wie der Kombinator **K** den eingehenden Datenfluss ignoriert und so eine Versorgung hin zu f überflüssig macht. Daraus ergibt sich eine einfache Hintereinanderschaltung der beiden Funktionen f und g, die durch den **B**-Kombinator repräsentiert wird.

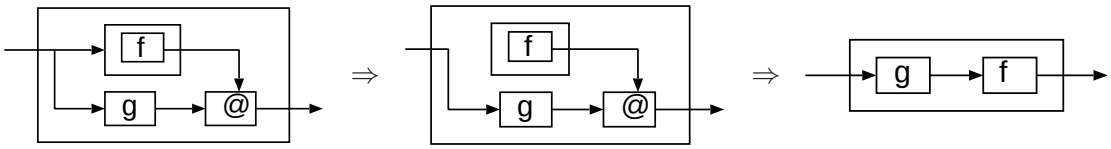
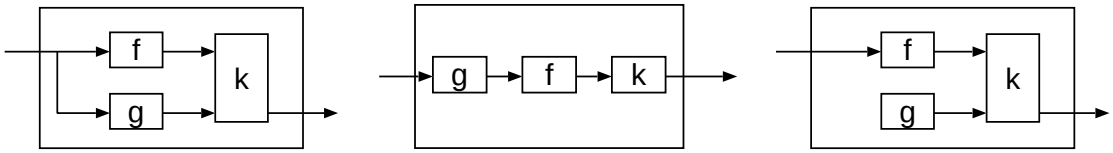


Abbildung 4.9: Vereinfachung eines Kombinator-Ausdrucks

Erweiterte Kombinatoren S', B' und C'

In Abbildung 4.10 sehen wir natürliche Darstellungen der erweiterten Kombinatoren **S'**, **B'** und **C'**. Hier ist erkennbar, dass jeweils der erste Parameter k nicht mit dem eingehenden Datenstrom versorgt wird, sondern nur durch die Ergebnisse der Funktionen von f bzw. g.



(a) **S'** k f g (b) **B'** k f g (c) **C'** k f g

Abbildung 4.10: Erweiterte Kombinatoren **S'**, **B'** und **C'**

Der Kombinator **B'** beschreibt dann z. B. nichts anderes als die Hinterschaltung dreier Funktionen, die jeweils das Funktionsergebnis der vorhergehenden Funktion verarbeiten.

Wir haben gezeigt, wie eine Datenflussdarstellung von Kombinatoren bestimmte Abhängigkeiten von Funktionen besonders anschaulich visualisieren, die auf den ersten Blick aus deren

Definition (z. B. aus Tabelle 4.7) nicht offensichtlich sind. Wir erhalten dadurch einen schnelleren Einblick in den Ablauf eines Programms und somit Hilfestellung sowohl bei der Programmerstellung wie auch der Programmänderung bzw. -korrektur.

4.4 Director Strings

Durch die Einführung der erweiterten Kombinatoren **S'**, **B'** und **C'** in Zusammenhang mit dem Abstraktionsalgorithmus konnten wir zwar die Größe des resultierenden Ausdrucks reduzieren. Doch der Algorithmus besitzt den unschönen Effekt, dass durch die Einführung der Kombinatoren der ursprüngliche Ausdruck nicht mehr ohne weiteres erkennbar bleibt. Dies wird umso deutlicher, wenn von mehreren Variablen abstrahiert wird. Hier überlagern sich die Kombinatoren, d. h. von einem Ausdruck, der bereits Kombinatoren enthält, wird wiederum abstrahiert.

Ein Software-Entwickler kann sich nun auf den Standpunkt stellen, dass Transformationsabläufe einen internen Automatismus darstellen, von denen er nicht behelligt werden möchte. Wir bieten jedoch dem Interessierten durch interaktive Transformationen Einblicke in Umwandlungen, in denen er stets die Ausgangsform seines Codes erkennen kann und trotzdem Einsichten in Optimierungen erhält. Dies ist ihm nicht möglich durch einen Compiler, der nach einem Blackbox-Prinzip arbeitet.

Director Strings sind komplementär zu den bisher eingeführten Kombinatoren **S**, **K** und **I** bzw. ihren Erweiterungen. Sie haben das Ziel, den Ausgangsausdruck zu erhalten, auch wenn von mehr als einer Variable abstrahiert wird. Der ursprüngliche Ausdruck bleibt weiterhin erkennbar.

4.4.1 Markierte Applikationen

Sieht man sich die Wirkungsweise der Kombinatoren **S**, **B** und **C** näher an, so stellt man fest, dass diese im Wesentlichen nur den Fluss der Parameterversorgung steuern. Kommen die Parameter in beiden Teilausdrücken vor, so werden die Parameter in beide Zweige distribuiert, ansonsten nur in einen der beiden weitergeleitet. Beim Kombinator **K** wird der Parameterfluss gänzlich unterbunden. Hier liegt die Idee nahe, die auf [Kennaway und Sleep 1988] zurückgeht und im Detail in [Stoye 1985] beschrieben ist, dass zu diesem Zweck nicht explizit Kombinatoren eingeführt werden, sondern dass bei den entsprechenden Applikationen lediglich der Parameterfluss (links, rechts, beidseitig, keiner) annotiert wird. Diese Aufgabe erledigen Markierungen (so genannte *directors*), welche je für eine zu abstrahierende Variable zuständig sind.

Tabelle 4.10 beschreibt den Abstraktionsalgorithmus, der zu dem Algorithmus für die Kombinatoren **S**, **K** und **I** aus Tabelle 4.6 komplementär ist. Hier wird jeweils zu einer bereits bestehenden Markierung $d_1 \ldots d_n$ des Operators @ eine zusätzliche Markierung eingeführt in Abhängigkeit davon, in welchem Teilausdruck sich die zu abstrahierende Variable x befindet.

$$[x] \mathbin{@} v \Rightarrow \mathbf{I} \qquad v \equiv x \text{ Variable} \tag{4.6}$$
$$[x] \mathbin{@}_{d_1 \ldots d_n} E \Rightarrow \mathbin{@}_{-d_1 \ldots d_n} E \qquad \text{wenn } x \notin FV(E) \tag{4.7}$$
$$[x] \mathbin{@}_{d_1 \ldots d_n} E_1\ E_2 \Rightarrow \mathbin{@}_{\backslash d_1 \ldots d_n} E_1\ ([x]E_2) \qquad \text{wenn } x \notin FV(E_1) \tag{4.8}$$
$$[x] \mathbin{@}_{d_1 \ldots d_n} E_1\ E_2 \Rightarrow \mathbin{@}_{/ d_1 \ldots d_n} ([x]E_1)\ E_2 \qquad \text{wenn } x \notin FV(E_2) \tag{4.9}$$
$$[x] \mathbin{@}_{d_1 \ldots d_n} E_1\ E_2 \Rightarrow \mathbin{@}_{\wedge d_1 \ldots d_n} ([x]E_1)\ ([x]E_2) \qquad \text{sonst} \tag{4.10}$$

Tabelle 4.10: Abstraktionsalgorithmus mittels Director Strings

Eine Markierung '$\wedge$' kann nun so interpretiert werden, dass der Datenfluss der zu abstrahierenden Variable auf beide Parameter verteilt wird. Eine Markierung '$\backslash$' versorgt nur den zweiten Parameter, eine Markierung '$/$' nur den ersten, und eine Markierung '$-$' unterbindet den Datenfluss[5].

Manche Director Strings können wir auf bereits bekannte Kombinatoren abbilden. So besitzen die @-Operatoren mit einer einstelligen Markierung folgende Äquivalente:

$\mathbin{@}_{\wedge} = \mathbf{S}$	beidseitiger Datenfluss
$\mathbin{@}_{/} = \mathbf{C}$	linker Datenfluss
$\mathbin{@}_{\backslash} = \mathbf{B}$	rechter Datenfluss
$\mathbin{@}_{-} = \mathbf{K}$	kein Datenfluss

In Termen, die entsprechend unseres neuen Abstraktionsalgorithmus transformiert wurden, ist die ursprüngliche Struktur weiterhin ersichtlich. Lediglich die Applikation (@) wird mit einer Markierung — einem so genannten *director string* — angereichert. Da die Applikationen nun markiert sind, müssen wir den Reduktionsalgorithmus entsprechend anpassen und gemäß den Markierungen erweitern:

$$\begin{aligned}
(\mathbin{@}_{\wedge d_1 \ldots d_n} E_1\ E_2)\ x &\Rightarrow \mathbin{@}_{d_1 \ldots d_n} (E_1\ x)(E_2\ x) \\
(\mathbin{@}_{/ d_1 \ldots d_n} E_1\ E_2)\ x &\Rightarrow \mathbin{@}_{d_1 \ldots d_n} (E_1\ x) E_2 \\
(\mathbin{@}_{\backslash d_1 \ldots d_n} E_1\ E_2)\ x &\Rightarrow \mathbin{@}_{d_1 \ldots d_n} E_1 (E_2\ x) \\
(\mathbin{@}_{-d_1 \ldots d_n} E_1\ E_2)\ x &\Rightarrow \mathbin{@}_{d_1 \ldots d_n} E_1 E_2
\end{aligned}$$

Tabelle 4.11: Reduktionsalgorithmus von Director Strings

Es ist sofort erkennbar, dass bei dem Abstraktionsalgorithmus Markierungen aufgebaut werden, bei dem Reduktionsalgorithmus Markierungen abgebaut werden. In dem Reduktionssystem

[5] Die Markierungen '$\wedge$', '$/$' und '$\backslash$' sollen die Pfeile '$\swarrow\!\!\searrow$', '$\swarrow$' und '$\searrow$' andeuten, die den Datenfluss in einem Termbaum von oben nach unten anzeigen.

sind die einzelnen Regeln mit Markierungen parametriert. Würden wir nur konstante Regeln ohne Parametrierung zulassen, so erhielten wir ein Reduktionssystem mit einer unbegrenzten Anzahl von Regeln, das durch die unbegrenzte Anzahl von verschieden markierten Applikationen ($@_{d_1 \ldots d_n}$) entsteht.

4.4.2 Datenfluss

Die Markierungen der Director Strings steuern die Parameterversorgung bei applizierten Teilausdrücken. Suchen wir nach einer geeigneten Visualisierung, so ist es nur natürlich, dafür eine Datenflusssicht zu wählen, die den Parameterfluss aufzeigt. Hier können wir sehen, wie sich die unterschiedlichen Markierungen auf die Verzweigung der Datenflusses auswirken.

(a) $@_{/\backslash}\ f\ g$ (b) $@_{/\wedge\backslash}\ f\ g$

Abbildung 4.11: Datenfluss markierter Applikationsknoten

Die kanonische Datenflussdarstellung für einstellige Markierungen haben wir bereits in den Gleichungen 4.7 und 4.8 kennengelernt, da diese ja den bereits bekannten Kombinatoren **S**, **C**, **B** und **K** entsprechen. Eine Darstellung für zwei ausgewählte, markierte Applikationsknoten mit mehreren Markierungen ($@_{/\backslash}$ und $@_{/\wedge\backslash}$) sehen wir in Abbildung 4.11. Die Anzahl der eingehenden Datenflussströme ergibt sich direkt aus der Anzahl der Markierungen.

4.4.3 Abbildung auf HOPS-Konstruktoren

Die markierten Applikationen aus Abschnitt 4.4.1 können wir auf Bausteine von HOPS abbilden. Es ergibt sich, wie bereits bei der Umsetzung der Kombinatoren, ein Satz an Basisbausteinen, dargestellt in Tabelle 4.12, die sich bei der Arbeit mit HOPS als grundlegend heraus kristallisiert haben.

$$
\begin{aligned}
(\textbf{FPair}\ f\ g)\ x &= (f\ x,\ g\ x)\\
(\textbf{Const}\ x)\ y &= x\\
\textbf{Curry}\ f\ x\ y &= f\ (x,\ y)\\
\textbf{Uncurry}\ f\ (x,\ y) &= f\ x\ y\\
\textbf{pi}\ (x,\ y) &= x\\
\textbf{rho}\ (x,\ y) &= y\\
\textbf{PUpd}\ f\ g\ (x,\ y) &= (f\ x,\ g\ y)
\end{aligned}
$$

Tabelle 4.12: Basisbausteine von HOPS

Wir sehen in der Tabelle eine textuelle Definition der Bausteine in Haskell-Notation [6]. Da es sich hier um jeweils eine einzeilige Implementierung handelt, fällt die Notation etwas platzsparender als in HOPS aus. Ebenso hätten wir natürlich eine äquivalente Definition in HOPS-Notation geben können. In Tabelle 4.13 sehen wir nun die Abbildung einer Auswahl von markierten Applikationsknoten auf grundlegende HOPS-Konstruktoren, geordnet nach der Anzahl der Markierungen[7].

Man kann bei allen Abbildungen mit n Markierungen ein einheitliches Muster

$$\textbf{Curry}^{n-1}\ (\textbf{FPair}\ (F[f])(G[g])\ \fatsemi\ @)$$

erkennen. Dies beruht auf der Tatsache, dass die verwendete Abbildung ein zweistelliges Tupel benutzt. Werden mehr als zwei Parameterflüsse benötigt, so werden zweistellige Tupel — jeweils auf der zweiten Tupelposition — geschachtelt. Diese Schachtelung, erzeugt durch den Operator **FPair**, wird durch die Anwendung des Muster $\textbf{Curry}^{n-1}$ aufgebrochen.

Der Term $F[f]$ wiederum wird zur Aufbereitung für die Applikation von f benutzt, deren Aufgabe im Herausprojizieren und Currysieren des entsprechenden Tupels liegt. Die gleiche Aufgabe kommt dem Term $G[g]$ für den zweiten Parameter g zu.

In Tabelle 4.13 haben wir es mit geschachtelten, zweistelligen Tupeln zu tun. Eine andere Möglichkeit besteht in der Verwendung von n-stelligen Tupeln, die das $n-1$-fache Currysieren und die Mehrfachprojektionen in $F[f]$ und $G[g]$ vereinfachen. Doch dies setzt wiederum beliebige n-stellige Tupel mit sämtlichen Projektionsvarianten [8] voraus. Um nicht eine unendliche Menge an Tupelkonstruktoren und -projektionen bereithalten zu müssen, haben wir uns auf zweistellige Tupel beschränkt.

[6] $(\_,\ \_)$ denotiert ein zweistelliges, nicht striktes Tupel.

[7] Hier ist zu beachten, dass der Baustein $\fatsemi$ geringer bindet als die Funktionsapplikation in Form der Juxtaposition (also: $f\ x\ \fatsemi\ g\ =\ (f\ x)\ \fatsemi\ g$).

[8] z. B. $Proj_3(a,b,c,d) = c$, d. h. projiziere die dritte Komponente aus einem vierstelligen Tupel.

@ mit 1 Markierung

$$@_{\wedge}\ f\ g = \textbf{FPair}\ f\ g \circ @ \quad (4.11a)$$
$$@_{\backslash}\ f\ g = \textbf{FPair}\ (\textbf{Const}\ f)\ g \circ @ \quad (4.11b)$$
$$= g \circ f \quad (4.11c)$$
$$@_{/}\ f\ g = \textbf{FPair}\ f\ (\textbf{Const}\ g) \circ @ \quad (4.11d)$$

@ mit 2 Markierungen

$$@_{\backslash\backslash}\ f\ g = \textbf{Curry}\ (\textbf{FPair}\ (\textbf{Const}\ f)\ (\textbf{Uncurry}\ g) \circ @) \quad (4.11e)$$
$$@_{//}\ f\ g = \textbf{Curry}\ (\textbf{FPair}\ (\textbf{Uncurry}\ f)\ (\textbf{Const}\ g) \circ @) \quad (4.11f)$$
$$@_{/\backslash}\ f\ g = \textbf{Curry}\ (\textbf{FPair}\ (\textbf{pi} \circ f)\ (\textbf{rho} \circ g) \circ @) \quad (4.11g)$$
$$= \textbf{Curry}\ (\textbf{PUpd}\ f\ g \circ @) \quad (4.11h)$$
$$@_{\wedge\backslash}\ f\ g = \textbf{Curry}\ (\textbf{FPair}\ (\textbf{Uncurry}\ f)\ (\textbf{rho} \circ g) \circ @) \quad (4.11i)$$

@ mit 3 Markierungen

$$@_{\backslash\backslash/}\ f\ g = \textbf{Curry}\ (\textbf{Curry}\ (\textbf{FPair}\ (\textbf{rho} \circ f)\ (\textbf{pi} \circ (\textbf{Uncurry}\ g)) \circ @)) \quad (4.11j)$$
$$@_{//\backslash}\ f\ g = \textbf{Curry}\ (\textbf{Curry}\ (\textbf{FPair}\ (\textbf{pi} \circ (\textbf{Uncurry}\ f))\ (\textbf{rho} \circ g) \circ @)) \quad (4.11k)$$
$$@_{/\backslash/}\ f\ g = \textbf{Curry}\ (\textbf{Curry}\ (\textbf{FPair}\ ((\textbf{PUpd}_{\mathbf{1}}\ \textbf{pi}) \circ (\textbf{Uncurry}\ f))\ (\textbf{pi} \circ \textbf{rho} \circ (\textbf{Uncurry}\ g)) \circ @)) \quad (4.11l)$$

@ mit n Markierungen

$$@_{d_1 \ldots d_n}\ f\ g = \textbf{Curry}^{n-1}\ (\textbf{FPair}\ (\ldots f)(\ldots g) \circ @) \quad (4.11m)$$

Tabelle 4.13: Abbildung der markierten Applikationen (@) auf HOPS-Konstruktoren

4.4.4 Verallgemeinerung auf beliebige Konstruktoren

Kennaway und Sleep haben Director Strings nur für den Konstruktor @ beschrieben. Wir können diesen Gedanken jedoch weiterführen und die Markierung und damit die Parameterversorgung auf andere, nicht nullstellige Konstruktoren (Knoten mit Nachfolgern) erweitern.

So sehen wir in Tabelle 4.14 den Abstraktions- und Reduktionsalgorithmus für die markierte Paarung (**Pair**$_{d_1 \ldots d_n}$).[9]

Abstraktionsalgorithmus:

$$
\begin{aligned}
&[x]\ \textbf{Pair}_{d_1 \ldots d_n}\ E_1\ E_2 = \textbf{Pair}_{\wedge d_1 \ldots d_n}\ ([x]E_1)\ ([x]E_2) \\
&[x]\ \textbf{Pair}_{d_1 \ldots d_n}\ E_1\ E_2 = \textbf{Pair}_{\backslash d_1 \ldots d_n}\ E_1\ ([x]E_2) \qquad \text{wenn } x \notin FV(E_1) \\
&[x]\ \textbf{Pair}_{d_1 \ldots d_n}\ E_1\ E_2 = \textbf{Pair}_{/ d_1 \ldots d_n}\ ([x]E_1)\ E_2 \qquad \text{wenn } x \notin FV(E_2)
\end{aligned}
$$

Reduktionsalgorithmus:

$$
\begin{aligned}
&(\textbf{Pair}_{\wedge d_1 \ldots d_n}\ E_1\ E_2)\ x \Rightarrow \textbf{Pair}_{d_1 \ldots d_n}\ (E_1\ x)\ (E_2\ x) \\
&(\textbf{Pair}_{\backslash d_1 \ldots d_n}\ E_1\ E_2)\ x \Rightarrow \textbf{Pair}_{d_1 \ldots d_n}\ E_1\ (E_2\ x) \\
&(\textbf{Pair}_{/ d_1 \ldots d_n}\ E_1\ E_2)\ x \Rightarrow \textbf{Pair}_{d_1 \ldots d_n}\ (E_1\ x) E_2
\end{aligned}
$$

Tabelle 4.14: Abstraktions-/Reduktionsalgorithmus der markierten Paarung (**Pair**)

Anstatt der Einführung neuer markierter Konstruktoren wie der markierten Paarung können wir diesen Konstruktor ebenso auf bekannte Konstruktoren aus HOPS abbilden. Die Abbildung einer Auswahl an markierten Paarungen sehen wir in Tabelle 4.15.

Ähnlich wie für den Paarungsoperator können wir die Markierungen auch auf andere Konstruktoren erweitern. Dabei ist neben der Einführung eines markierten Knotens stets dessen Aufbau und Abbau der Markierungen, d. h. die Abstraktion und Reduktion, zu berücksichtigen. Diesen Nachteil werden wir in Abschnitt 4.5 aufheben, in dem wir ganz auf markierte Knoten verzichten und einen modifizierten Abstraktionsalgorithmus einführen werden, der ausschließlich auf einer **konstanten Menge** von HOPS-Konstruktoren aufbaut.

[9] Der Konstruktor **Pair** $f\ g$ erzeugt aus den beiden Parametern f und g ein zweistelliges Tupel.

Pair mit 1 Markierung

$$\begin{aligned} \textbf{Pair}_{\backslash}\ f\ g &= \textbf{FPair}\ (\textbf{Const}\ f)\ g \\ \textbf{Pair}_{/}\ f\ g &= \textbf{FPair}\ f\ (\textbf{Const}\ g) \\ \textbf{Pair}_{\wedge}\ f\ g &= \textbf{FPair}\ f\ g \end{aligned}$$

Pair mit 2 Markierungen

$$\textbf{Pair}_{/\backslash}\ f\ g = \textbf{Curry}\ (\textbf{PUpd}\ f\ g)$$

Tabelle 4.15: Abbildung der markierten Paarung (**Pair**) auf HOPS-Konstruktoren

4.4.5 Beispiel

Wir wollen nun betrachten, wie sich die Einführung der erweiterten Bausteine auf ein kleines Beispiel-Programm auswirkt. Wir gehen von der Fakultätsfunktion

$$n! = \begin{cases} 1 & n = 0 \\ n * (n-1)! & n \neq 0 \end{cases}$$

aus. Eine Definition in Haskell-Schreibweise ist in Listing 4.1 angegeben; eine entsprechende Implementierung in Form eines HOPS-DAGs sehen wir in Abbildung 4.12.

Listing 4.1: Fakultät in Haskell

```
fac n = (either (const 1) (\x →mult (n, fac x ))) ( repr n)
```

Wenden wir darauf die Regeln in Gleichung (4.6) bis Gleichung (4.10) aus Tabelle 4.10 an, so kommen wir zu den transformierten Graphen aus Abbildung 4.13. Vergleicht man das Resultat mit seinem Urprungsgraphen, so sieht man, wie der allgemeine Aufbau beibehalten wurde und lediglich manche Knoten mit einer Markierung versehen wurden.

Zudem erkennen wir, dass uns diese Version noch nicht zufrieden stellt, da einige Abstraktionen noch vorhanden sind, unser Ziel aber darin bestand, Abstraktionen und somit auch mögliche, gebundene Variable zu eliminieren. Der Grund resultiert aus der Tatsache, dass wir durch die Beschränkung der Markierungen auf @-Knoten die Abstraktionen nicht über beliebige, mehrstellige Konstruktoren hinweg heben können und damit gebundene Variablen übrig bleiben.

Durch die Erweiterung des Mechanismus auf andere Konstruktoren kommen wir letztendlich zu den Graphen aus Abbildung 4.14.

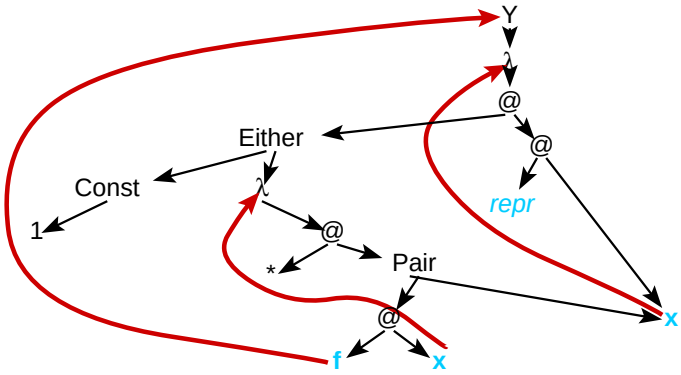
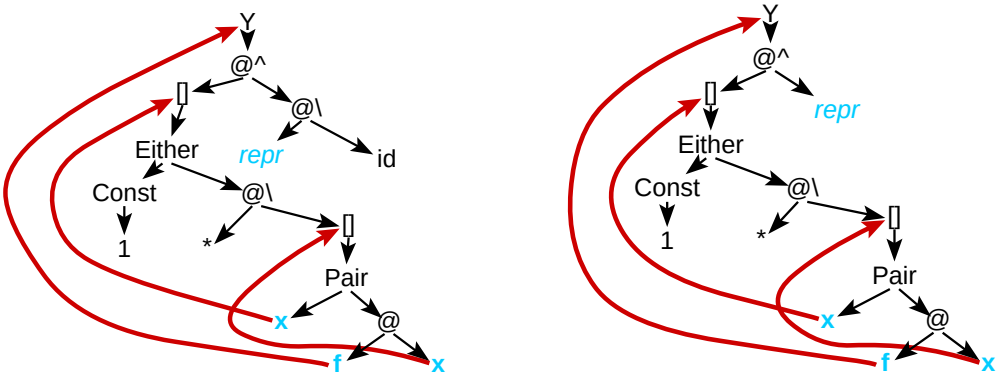


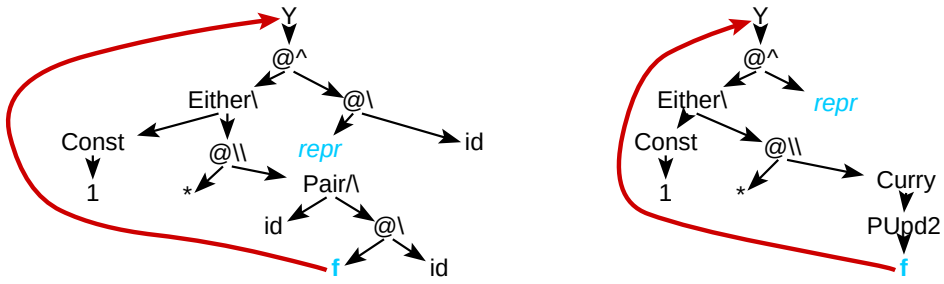
Abbildung 4.12: Definition der Fakultät



(a) mit Director Strings

(b) mit Director Strings (lokal optimiert)

Abbildung 4.13: Fakultät mit Director Strings für @



(a) mit Director Strings

(b) mit Director Strings (lokal optimiert)

Abbildung 4.14: Fakultät mit Director Strings für alle Konstruktoren

Neben den transformierten DAGs in Abbildung 4.13(a) und Abbildung 4.14(a) sehen wir in Abbildung 4.13(b) und Abbildung 4.14(b) jeweils noch optimierte Versionen, in denen einige Vereinfachungen, wie z. B.

$$\lambda x.x = \mathbf{id}$$
$$@_{\backslash}\ f\ \mathbf{id} = f$$

durchgeführt wurden. Sie sind semantisch identisch zu ihren nicht-optimierten Versionen, aber etwas kompakter in der Aufschreibung.

4.5 Kompositionaler Datenfluss

In Abschnitt 4.4 haben wir mit Hilfe von Director Strings eine Möglichkeit aufgezeigt, wie wir Abstraktionen eliminieren und dennoch die Struktur des Ausgangsterms erhalten können. Dazu mussten wir jedoch markierte Applikationen einführen. Zusätzlich haben wir Markierungen auf andere Konstruktoren erweitert und so eine flexible Anwendbarkeit für beliebige Termgraphen geschaffen. Anschließend gaben wir eine Abbildung der markierten Konstruktoren auf HOPS-Bausteine an.

In diesem Abschnitt wollen wir der Frage auf den Grund gehen, wie ein Abstraktionsalgorithmus aussehen kann, der direkt auf HOPS-Konstruktoren abbildet, also keine zusätzlichen Konstruktoren einführt, die dem HOPS-System fremd sind.

Außerdem möchten wir eine kompositionale, durch den Operator ‘$\fatsemi$’ erzeugte, Hintereinanderschaltung von Bausteinen erreichen. Durch den Operator ‘$\fatsemi$’ gewinnen wir eine besonders einfache Art einer Datenflussdarstellung, die einen Datenstrom kennzeichnet, der von einzelnen Funktionen in einer sequenziellen Weise bearbeitet wird.

Das Prinzip besteht wiederum darin, die Daten mittels geeigneter Kombinatoren zu denjenigen Stellen zu transportieren, an denen sie benötigt werden. Dies kommt — in der Datenflussdarstellung — einem Legen von Leitungen gleich, auf denen die Daten fließen.

4.5.1 Abbildung des λ-Kalküls

Zunächst betrachten wir einen Ausdruck des λ-Kalküls. Unsere Vorgehensweise ist, λ-Abstraktionen zu entfernen unter Einführung eines geeigneten Satzes von Kombinatoren. Bisher haben wir uns im Wesentlichen auf die Kombinatoren **S**, **K** und **I** gestützt. Im Folgenden führen wir nun einen Kombinatorsatz ein, der die kompositionale Verarbeitung von Funktionen unterstützt. Die Kombinatoren werden sein:

id, **Const**, $\circ\!\!\!\circ$, **FPair**, **Uncurry**.

Zuerst zeigen wir eine Version ohne Berücksichtigung vom tatsächlichen Vorkommen abstrahierter Variablen, die das Pendant zu Tabelle 4.3 ist:

$[x]\ v$	$\Rightarrow$ **id**	$v \equiv x$ Variable
$[x]\ E$	$\Rightarrow$ **Const** E	$x \notin FV(E)$
$[x]\ @\ E_1\ E_2$	$\Rightarrow$ (**FPair id** $[x]E_2$) $\circ\!\!\!\circ$ (**Uncurry** $([x]E_1)$)	

Tabelle 4.16: Einfacher Abstraktionsalgorithmus

Hierbei wird außer Acht gelassen, ob die zu abstrahierende Variable tatsächlich in einem Teilausdruck vorkommt. Dies hat zur Folge, dass evtl. Leitungen zu einer Funktion gelegt werden, die diesen Variablenstrom gar nicht ausnutzt. Das ist z. B. bei der Konstantenfunktion **Const** der Fall.

Aus diesem Grund werden wir den Kalkül etwas verfeinern und gelangen zu folgenden Regeln, die explizit das Vorkommen von freien Variablen in Ausdrücken berücksichtigen (vgl. dazu Tabelle 4.6):

$[x]\ v$	$\Rightarrow$ **id**	$v \equiv x$ Variable
$[x]\ E$	$\Rightarrow$ **Const** E	$x \notin FV(E)$
$[x]$ @ $E_1\ E_2$	$\Rightarrow [x]E_2 \fatsemi E_1$	$x \notin FV(E_1)$
$[x]$ @ $E_1\ E_2$	$\Rightarrow$ (**FPair id** (**Const** E_2)) $\fatsemi$ (**Uncurry**$([x]E_1)$)	$x \notin FV(E_2)$
$[x]$ @ $E_1\ E_2$	$\Rightarrow$ (**FPair id** $([x]E_2)$) $\fatsemi$ (**Uncurry**$([x]E_1)$)	sonst

Tabelle 4.17: Verfeinerter Abstraktionsalgorithmus

Wir sehen, dass wir bei der Variablenelimination zusätzliche Bausteine eingeführt haben. Unter den Bausteinen mit Nachfolgern,

Const, $\fatsemi$, **FPair**,

können nun wiederum abstrahierte Variablen auftreten[10], die wir in unserem Abstraktionsalgorithmus wiederum berücksichtigen werden. Das Verfahren wird nun wie folgt erweitert:

$[x]$ **Const** E	$\Rightarrow$ **Curry**(**pi** $\fatsemi$ $[x]E$)	
$[x]\ \fatsemi\ E_1\ E_2$	$\Rightarrow$ **Curry**(**FPair pi** (**Uncurry**$([x]E_1)$) $\fatsemi$ **Uncurry**$([x]E_2)$)	
$[x]\ \fatsemi\ E_1\ E_2$	$\Rightarrow$ **Curry**(**FPair pi** (**rho** $\fatsemi$ E_1) $\fatsemi$ **Uncurry**$([x]E_2)$)	$x \notin FV(E_1)$
$[x]\ \fatsemi\ E_1\ E_2$	$\Rightarrow$ **Curry**(**Uncurry**$([x]E_1)$ $\fatsemi$ E_2)	$x \notin FV(E_2)$
$[x]$ **FPair id** E_2	$\Rightarrow$ **Curry**(**FPair** (**rho** $\fatsemi$ **id**) **Uncurry**$([x]E_2)$)	
$[x]$ **FPair pi** E_2	$\Rightarrow$ **Curry**(**FPair** (**rho** $\fatsemi$ **pi**) **Uncurry**$([x]E_2)$)	

oder kürzer für $e_1 \in \{\textbf{id}, \textbf{pi}\}$:

$$[x]\ \textbf{FPair}\ e_1\ E_2[x] \Rightarrow \textbf{Curry}(\textbf{FPair}\ (\textbf{rho}\ \fatsemi\ e_1)\ \textbf{Uncurry}([x]E_2))$$

Nun bleiben noch die Bausteine **Curry** und **Uncurry** zur Bearbeitung. Auch unter diesen können Abstraktionen auftreten. Der Abstraktionsalgorithmus ist jedoch so konzipiert, dass in vielen Fällen die beiden Bausteine unmittelbar geschachtelt auftreten, so dass sie sich mit Hilfe der Vereinfachung

$$\textbf{Uncurry}(\textbf{Curry}(E)) = E$$
$$\textbf{Curry}(\textbf{Uncurry}(E)) = E$$

gegenseitig auflösen.

[10] Abstrahierte Variablen, die von außen gebunden sind

Treten dennoch Abstraktionen unter **Curry** bzw. **Uncurry** auf, die mit der Vereinfachung nicht aufzulösen sind, so können wir eine Erweiterung des Algorithmus um die beiden Bausteine dadurch vermeiden, dass wir jeweils eine Definition der beiden Bausteine angeben, die wiederum vom Abstraktionsalgorithmus behandelbar ist:

$$\textbf{Uncurry}\ E = @\ \textbf{uncurry}\ E$$
$$\textbf{Curry}\ E = @\ \textbf{curry}\ E$$

Dadurch ersetzen wir einen Baustein mit einem Nachfolger durch einen Term, der den entsprechenden Baustein ohne Nachfolger enthält.

Beispiel
Ein kleines Beispiel zeigt die Wirkung des Algorithmus. Dabei bezeichnet $\stackrel{*}{\Rightarrow}$ die mehrfache Anwendung des Abstraktionsalgorithmus, also die Elimination mehrerer gebundener Variablen.

$$\lambda y.(\ \lambda x.(+\ x)\ 3\ )\ y \stackrel{*}{\Rightarrow} \textbf{id} \circ (((\textbf{FPair id}\ (\textbf{Const}\ 3)) \circ \textbf{Uncurry}(\textbf{id} \circ +))$$

Abbildung des λ-Kalküls mit abkürzenden Kombinatoren

Betrachten wir die resultierenden Ausdrücke etwas genauer, so stellen wir sehr schnell fest, dass manche Teilausdrücke häufig ein einheitliches Schema aufweisen. Hier liegt die Idee nahe, einige wenige Kombinatoren einzuführen, die lediglich Abkürzungen größerer Ausdrücke sind und dieses Schema abdecken.

Ein Abstraktionsalgorithmus, der einige konstante Teilausdrücke in abkürzenden Kombinatoren belässt, die einer kompakteren Notation dienen, ist nachfolgend aufgeführt. Ähnlich wie bei den erweiterten Bausteinen mit Director Strings geschieht die Versorgung mit den Daten durch (geschachtelte) Produkte. Durch das Uncurrysieren des noch zu transformierenden Ausdrucks greifen wir auf den entsprechenden Wert zu.

Mit dem Wunsch nach einer kompositionalen Verkettung ('$\circ$'), also einer sequenziellen Hintereinanderschaltung, von Funktionen kommen wir zu folgenden Transformationsregeln:

$[x]\ v$	$\Rightarrow \textbf{id}$	$v \equiv x$ Variable
$[x]\ E$	$\Rightarrow \textbf{Const}\ E$	$x \notin FV(E)$
$[x]\ @\ E_1\ E_2$	$\Rightarrow [x]E_2 \circ E_1$	$x \notin FV(E_1)$
$[x]\ @\ E_1\ E_2$	$\Rightarrow \textbf{MkPair}_\textbf{2}\ E_2 \circ \textbf{Uncurry}([x]E_1)$	$x \notin FV(E_2)$
$[x]\ @\ E_1\ E_2$	$\Rightarrow \textbf{FPair}_\textbf{2}([x]\ E_2) \circ \textbf{Uncurry}([x]\ E_1)$	sonst

Tabelle 4.18: Verfeinerter Abstraktionsalgorithmus unter Einführung abkürzender Kombinatoren

Dabei ist zu berücksichtigen, dass unter den neu hinzugenommenen Kombinatoren **FPair 2** und **MkPair 2** wiederum abstrahierte Variablen auftreten können. Daher ist es erforderlich, für diese Kombinatoren ebenfalls eine Abstraktionsregel

$$[x]\ \textbf{FPair}_\textbf{2}\ E \Rightarrow \textbf{Curry}(\textbf{FPair rho Uncurry}([x]E))$$

bzw. eine Auflösung

$$\textbf{MkPair}_\textbf{2}\ E = \textbf{FPair}_\textbf{2}(\textbf{Const}\ E)$$

vorzusehen.

Wir möchten im Besonderen auf die induktiv definierten Regel aus Tabelle 4.18 hinweisen, bei denen auf der rechten Seite wieder der Abstraktionsoperator $[x]$ erscheint. Diese Regeln stützen sich jeweils auf den Kombinator '$\circ$' und führen dadurch eine kompositionale Hintereinanderschaltung ein.

Beispiel
Unser vorheriges Beispiel wird nun in eine kürzere Aufschreibung transformiert:

$$\lambda y.(\ \lambda x.(+\ x)\ 3\ )\ y \stackrel{*}{\Rightarrow} \textbf{id}\ \circ\ (\textbf{MkPair}_\textbf{2}\ 3\ \circ\ \textbf{Uncurry}(\textbf{id}\ \circ\ +))$$

Lokale Optimierungen

Aus dem vorangegangenen Beispiel lassen sich einige Optimierungen ablesen, die unnötige Berechnungen unterbinden. Dazu ist keine aufwändige Analyse des Terms notwendig, da die Optimierungen lokal zu erkennen sind. Einige lokale Optimierungen sind nachfolgend aufgeführt:

$\textbf{id}\ E$	$\equiv E$
$E\ \textbf{id}$	$\equiv E$
$\textbf{FPair}_\textbf{2}\ (\textbf{Const}\ E_1)\ \circ\ \textbf{Uncurry}(\textbf{Const}\ E_2)$	$\equiv \textbf{Const}(E_2\ E_1)$
$\textbf{FPair}_\textbf{2}\ (\textbf{Const}\ E)$	$\equiv \textbf{MkPair}_\textbf{2}\ E$
$\textbf{FPair}_\textbf{2}\ E_1\ \circ\ \textbf{Uncurry}(\textbf{Const}\ E_2)$	$\equiv E_1\ \circ\ E_2$

Tabelle 4.19: Lokale Optimierungen für Abstraktionsalgorithmus

Mit Hilfe dieser Transformationen lässt sich unser Beispiel weiter vereinfachen, so dass das Ergebnis sogar kompakter ausfällt als der Ausgangsausdruck.

Beispiel

$$\lambda y.(\ \lambda x.(+\ x)\ 3\ )\ y \stackrel{*}{\Rightarrow} \textbf{MkPair}_\textbf{2}\ 3\ \circ\ \textbf{Uncurry}\ +$$

4.5.2 Erweiterung auf andere Kombinatoren

Der bisherige Abstraktionsalgorithmus aus Tabelle 4.18 ist darauf ausgelegt — analog zu den konventionellen Director Strings (Tabelle 4.10 in Abschnitt 4.4) —, dass er für einen bestimmten Konstruktor gilt, nämlich die Applikation (**@**). Diese Einschränkung können wir ohne weiteres aufheben, indem wir für weitere Konstruktoren wie der Funktionssumme (**FSum**), Funktionspaarung (**FPair**) etc. entsprechende Regeln angeben, wie sie sich bzgl. der Variablenabstraktion verhalten sollen[11]:

$$
\begin{array}{lll}
[x]\ \mathbf{FSum}\ E_1\ E_2 \Rightarrow \mathbf{Curry}(\mathbf{RDistr} \circ \mathbf{FSum}\ \mathbf{Uncurry}([x]E_1)\ \mathbf{Uncurry}([x]E_2)) & & \\
[x]\ \mathbf{FSum}\ E_1\ E_2 \Rightarrow \mathbf{Curry}(\mathbf{RDistr_2} \circ \mathbf{FSum}\ E_1\ \mathbf{Uncurry}([x]E_2)) & & x \notin FV(E_1) \\
[x]\ \mathbf{FSum}\ E_1\ E_2 \Rightarrow \mathbf{Curry}(\mathbf{RDistr_1} \circ \mathbf{FSum}\ \mathbf{Uncurry}([x]E_1)\ E_2) & & x \notin FV(E_2) \\
[x]\ \mathbf{FPair}\ E_1\ E_2 \Rightarrow \mathbf{Curry}(\mathbf{FPair}\ \mathbf{Uncurry}([x]E_1)\ \mathbf{Uncurry}([x]E_2)) & & \\
[x]\ \mathbf{FPair}\ E_1\ E_2 \Rightarrow \mathbf{Curry}(\mathbf{FPair}\ (\mathbf{rho} \circ E_1)\ \mathbf{Uncurry}([x]E_2)) & & x \notin FV(E_1) \\
[x]\ \mathbf{FPair}\ E_1\ E_2 \Rightarrow \mathbf{Curry}(\mathbf{FPair}\ \mathbf{Uncurry}([x]E_1)\ (\mathbf{rho} \circ E_2)) & & x \notin FV(E_2) \\
[x]\ \mathbf{FPair_1}\ E \Rightarrow \mathbf{Curry}(\mathbf{FPair}\ \mathbf{Uncurry}([x]E)\ \mathbf{rho}) & & \\
[x]\ \mathbf{FPair_2}\ E \Rightarrow \mathbf{Curry}(\mathbf{FPair}\ \mathbf{rho}\ \mathbf{Uncurry}([x]E)) & & \\
[x]\ \mathbf{Pair}\ E_1\ E_2 \Rightarrow \mathbf{FPair}\ [x]E_1\ [x]E_2 & & \\
[x]\ \mathbf{Pair}\ E_1\ E_2 \Rightarrow \mathbf{FPair}\ (\mathbf{Const}\ E_1)\ [x]E_2 & & x \notin FV(E_1) \\
[x]\ \mathbf{Pair}\ E_1\ E_2 \Rightarrow \mathbf{FPair}\ [x]E_1\ (\mathbf{Const}\ E_2) & & x \notin FV(E_2) \\
[x] \circ E_1\ E_2 \Rightarrow \mathbf{Curry}(\mathbf{Keep_1}(\mathbf{Uncurry}([x]E_1)) \circ \mathbf{Uncurry}([x]E_2)) & & \\
[x] \circ E_1\ E_2 \Rightarrow \mathbf{Curry}(\mathbf{PUpd_2}(E_1) \circ \mathbf{Uncurry}([x]E_2)) & & x \notin FV(E_1) \\
[x] \circ E_1\ E_2 \Rightarrow \mathbf{Curry}(\mathbf{Uncurry}([x]E_1) \circ E_2) & & x \notin FV(E_2) \\
[x]\ \mathbf{Const}\ E \Rightarrow \mathbf{Curry}(\mathbf{pi} \circ [x]E) & &
\end{array}
$$

Tabelle 4.20: Erweiterter Abstraktionsalgorithmus für andere Kombinatoren

Lokale Optimierungen

Auch hier sind einige Vereinfachungen möglich, die im Zuge der obigen Transformationen häufig zur Anwendung kommen.

[11] Anzustreben ist eine Version mit äußerem **Curry** und innerem **Uncurry** (siehe **FSum**, **FPair**, $\circ$), die sich dann gegenseitig aufheben. Möglich ist die Platzierung des äußeren Currys jedoch nur bei Konstruktoren, deren Nachfolger bereits Funktionen sind. Nicht möglich ist dies z. B. bei (@ E_1 E_2), da E_2 nicht immer eine Funktion sein muss.

$$[x]\ \textbf{Const}\ x \quad \Rightarrow \textbf{const}$$
$$[x]\ \textbf{Uncurry const} \Rightarrow \textbf{pi}$$

Auch hier möchten wir wieder auf die Regeln aus Tabelle 4.20 hinweisen, bei denen insbesondere der Kompositionsoperator '$\circ$' auf der rechten Seite eingeführt wird, wo es möglich ist.

Wir sind nun in der Lage, von den folgenden Bausteinen mit Nachfolgern zu abstrahieren. Diese sind:

@, **FPair**, **FPair$_1$**, **FPair$_2$**, **Pair**, $\circ$, **Const**.

In den Regeln treten zwar auf der rechten Seite auch Bausteine wie

RDistr, **RDistr$_1$**, **RDistr$_2$** u. a.

auf, doch setzt der Transformationsalgorithmus unter diesen Bausteinen keine weiteren Abstraktionen ein, die in weiteren Schritten eliminiert werden müssten.

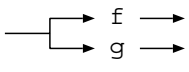
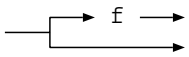
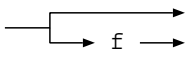
4.5.3 Datenfluss wichtiger Kombinatoren

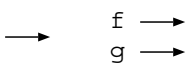
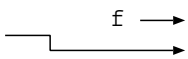
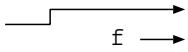
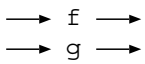
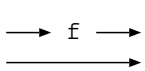
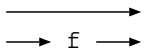
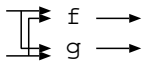
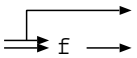
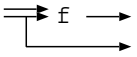
Die Semantik vieler Kombinatoren lässt sich oftmals durch die Angabe ihres Datenflusses leicht erkennen. Der Datenfluss wiederum ist bei grundlegenden Bausteinen oft schon an der Typisierung abzulesen. Nehmen wir z. B. den Baustein

$$\textbf{FPair} :: (\alpha \to \beta) \times (\alpha \to \gamma) \to (\alpha \to \beta \times \gamma)$$

Die gemeinsame Typvariable α für die Funktionen f und g ist bedingt durch den gemeinsamen Ursprung der Daten. Dementsprechend leiten wir den ausgehenden Datenfluss der beiden Funktionen zu einem Kreuzprodukt ($\times$) ab.

In der folgenden Tabelle geben wir eine Reihe von Bausteinen mit deren Datenflüssen an:

FPair f g	
FPair$_1$ f $\Rightarrow$ **FPair** f **id**	
FPair$_2$ f $\Rightarrow$ **FPair id** f	

MkPair $f\ g \Rightarrow$ **FPair** (**Const** f) (**Const** g)	
MkPair$_1$ $f \Rightarrow$ **FPair$_1$** (**Const** f)	
MkPair$_2$ $f \Rightarrow$ **FPair$_2$** (**Const** f)	
PUpd $f\ g \Rightarrow$ **FPair** (**pi** $\fatsemi$ f) (**rho** $\fatsemi$ g)	
PUpd$_1$ $f \Rightarrow$ **PUpd** f **id**	
PUpd$_2$ $f \Rightarrow$ **PUpd id** f	
Keep $f\ g \Rightarrow$ **FPair** $f\ g$	
Keep$_1$ $f \Rightarrow$ **FPair pi** f	
Keep$_2$ $f \Rightarrow$ **FPair** f **rho**	

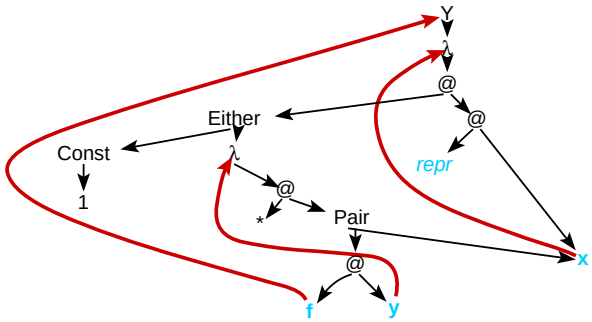
4.5.4 Beispiel

Die Transformationsvorschrift wollen wir nun an einem Beispiel demonstrieren. Wir wählen wieder die Fakultät, die in zwei verschiedenen Implementierungen vorgestellt wird. Zum einen geben wir eine Implementierung an, die die Definition kanonisch umsetzt. Anschließend geben wir eine Version an, die eine lineare Komplexität aufweist und einen Ergebnisparameter verwendet.

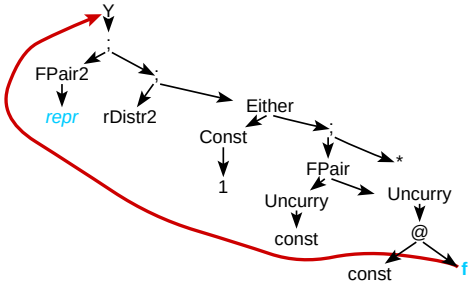
Fakultät laut Definition

Wir starten mit der Fakultätsfunktion aus Abbildung 4.15(a) und transformieren mit den angegebenen Regeln zu dem Graphen aus Abbildung 4.15(b).

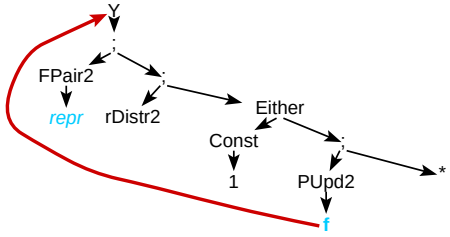
Wenden wir einige, lokale Optimierungen an, so kommen wir zu der Version aus Abbildung 4.15(c), die bemerkenswert wenige und trotzdem nur grundlegende Bausteine aufweist. Das gehäufte Auftreten des Bausteins $\fatsemi$ weist auf einen linearen Datenfluss hin.



(a) Fakultät



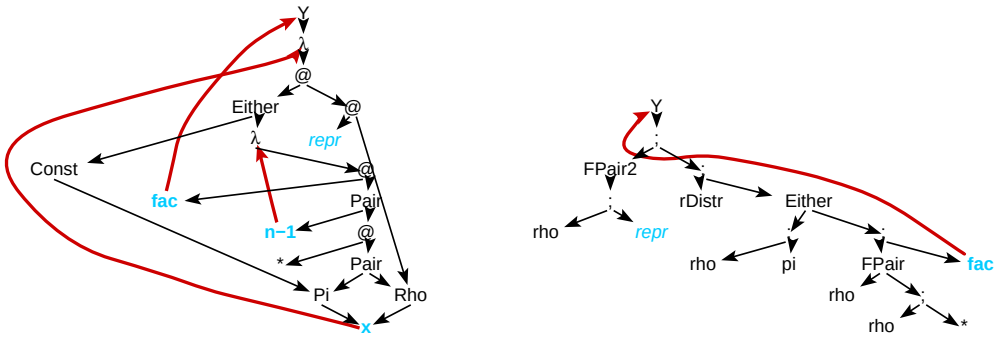
(b) Fakultät ohne Abstraktionen



(c) Fakultät ohne Abstraktionen (optimiert)

Abbildung 4.15: Einstellige Fakultät

Fakultät (linear, mit Ergebnisparameter)
Wir sind nicht nur an einer beliebigen Implementierung der Fakultät interessiert, sondern auch an deren Effizienz. Dazu führen wir — wie aus grundlegenden Informatikbüchern bekannt — einen weiteren (Ergebnis-)Parameter ein, der bisherige Berechnungen aufnimmt.



(a) Fakultät

(b) Fakultät ohne Abstraktionen

Abbildung 4.16: Zweistellige Fakultät

In Abbildung 4.16(a) sehen wir wieder den Ausgangs-DAG; Abbildung 4.16(b) beschreibt das transformierte Programm. Hier möchten wir auf das mehrfache Vorkommen des Kompositionsoperators ‘$\circ$’ hinweisen, dessen Einsatz im Ergebnisausdruck ja unser erklärtes Ziel war.

KAPITEL 5

DSP – Von der Differenzengleichung zu ausführbarem Code

Das Gebiet der Signalverarbeitung hat im letzten Jahrzehnt durch die anhaltende Digitalisierung enorm an Bedeutung gewonnen. So sind heutzutage kaum mehr Bereiche denkbar, in denen die digitale Signalverarbeitung (DSP, Digital Signal Processing) keine Anwendung gefunden hat, sei es in der Automobilindustrie, in der Haushaltstechnik oder in der Unterhaltungsindustrie.

Wir haben in den bisherigen Kapiteln Eigenschaften und Vorzüge von Datenflusssystemen aufgezeigt. Daneben haben wir einen Weg beschrieben, beliebige Terme von Variablenbindungen zu befreien bzw. in eine Datenflussversion zu transformieren. Unser Interesse für das überaus interessante Themengebiet der Signalverarbeitung liegt nun in der Tatsache begründet, dass es viele Gemeinsamkeiten mit einem Datenflusssystem aufweist. Daher liegt es nahe, bisher gewonnene Verfahren und Werkzeuge auf diesen Bereich anzuwenden. Wir werden dazu Algorithmen mit einer visuellen Datenflusssprache entwerfen und anschließend daraus ausführbaren Code mittels eines Transformationsverfahrens generieren. Diesen Ansatz werden wir an einer Teildisziplin der Signalverarbeitung, dem Gebiet der linearen zeitinvarianten Systeme, untersuchen.

Inhaltsverzeichnis

Digitale Signalverarbeitung befasst sich mit der Verarbeitung von Signalfolgen, die in digitalisierter Form vorliegen. Verwendung findet sie vor allem im Bereich der Kommunikationstechnik zur Bildung von Systemen zur Datenübertragung und -kompression, Geräuschunterdrückung aber auch in „eingebetteter Form“ wie etwa in der Fahrzeugtechnik, bei der z. B. Klangprozessoren genutzt werden, um in einem Fahrzeug synthetische, für das menschliche Ohr wohlklingende Motorgeräusche zu erzeugen.

Die Verarbeitung von Signalen beruht auf wenigen Grundschemata, die in verschiedenen Anwendungen immer wieder auftreten. Die Grundlagen der diskreten Signalverarbeitung werden wir in Abschnitt 5.1 genauer beleuchten. Hier führen wir auch eine mathematisch geprägte Definition von Signalfolgen und deren Verarbeitungsfunktionen ein, die sich an den Grundzügen des λ-Kalküls orientiert und manchmal etwas schärfer den Unterschied zwischen Signalfolge und -wert beschreibt.

Abschnitt 5.2 konzentriert sich auf ein Teilgebiet der Signalverarbeitung, das der linearen zeitinvarianten Systeme, die für den Rest des Kapitels die zentrale Rolle spielen werden. Mögliche Implementierungsformen für lineare zeitinvariante Systeme werden wir in Abschnitt 5.3 vorstellen.

Einen transformativen, rechnergestützten Ansatz beschreibt Abschnitt 5.4, der es erlaubt, bestimmte, sogenannte Systemfunktionen, die zur Beschreibung von Signalverarbeitungssystemen dienen, automatisch in eine Reihe von Implementierungsformen zu transformieren. Dazu wird eine spezielle Notation eingeführt, aus der wir nicht nur unterschiedliche Implementierungsformen ableiten können, sondern auch Programmtext für verschiedene Programmiersprachen generieren können.

5.1 Grundlagen der diskreten Signalverarbeitung

In der *digitalen Signalverarbeitung* werden Signale durch eine Folge von Zahlen endlicher Genauigkeit dargestellt, deren Verarbeitung meist durch Computer bewerkstelligt wird. Der Begriff *zeitdiskrete Signalverarbeitung* umfasst dabei digitale Signalverarbeitung und wird zumeist im Kontext digitaler Rechenanlagen als gleichbedeutend verstanden (siehe [Oppenheim und Schafer 1992]).

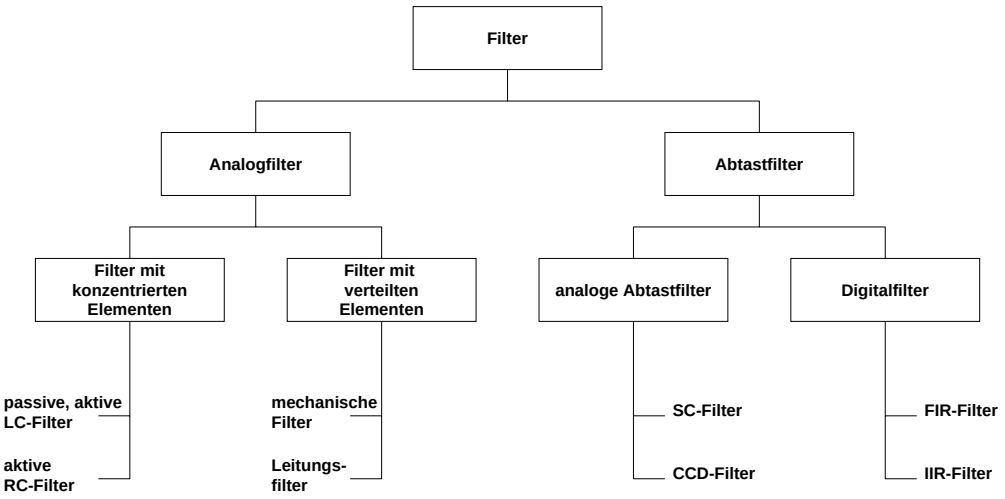


Abbildung 5.1: Übersicht der Filtertechniken

Die meisten Anwendungen in dieser Domäne betreffen zeitkontinuierliche Signale, die vor der digitalen Verarbeitung in eine Folge von Abtastwerten, also ein zeitdiskretes Signal, gewandelt werden, um nach der Verarbeitung wieder in ein zeitkontinuierliches Signal rücktransformiert zu werden. Bei der Verarbeitung kommen häufig Filter zum Einsatz, über deren Klassifizierung Abbildung 5.1 eine Übersicht gibt. In dieser Arbeit konzentrieren wir uns auf digitale Filter.

5.1.1 Zeitdiskrete Signalfolgen

Signale werden mathematisch als Funktionen einer oder mehrerer Variablen dargestellt. Beschränken wir uns jedoch auf *zeitdiskrete Folgen*, so ist es üblich, ein Signal als Folge von Zahlen darzustellen. Dabei erhalten wir über den Weg der periodischen Abtastung (Sampling) aus einem analogen ein diskretes Signal (Abbildung 5.2).

Im Folgenden wollen wir in Anlehnung an [Oppenheim und Schafer 1992] die Konvention einführen, dass x eine Folge von (komplexen) Zahlen bezeichnet, bei der man mit dem Infixoperator $[\,]$ durch $x[n]$ auf das n-te Element zugreift. Meinen wir ein analoges Signal $x_a(t)$, so greifen wir über den $()$-Operator, also die herkömmliche Funktionsapplikation, mit $x_a(t)$ auf den Wert von x_a zum Zeitpunkt t zu.

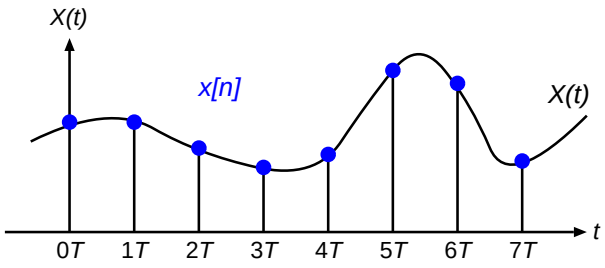


Abbildung 5.2: Abtastung eines zeitkontinuierlichen Signals

Zusätzlich wollen wir neben der in der Elektrotechnik üblichen Schreibweise an Stellen, wo Mehrdeutigkeiten herrschen könnten, auch eine mathematisch geprägte, an den λ-Kalkül angelehnte Notation angeben. Dabei soll eine Folge x als (mathematische) Funktion x kodiert werden. Um den Charakter der Folge beizubehalten, verwenden wir zur Applikation die Klammern $[\,]$.

In der Elektrotechnik ist die Notation

x	
$x[n]$	n-tes Element der diskreten Folge x
$x_a :: \mathbb{Z} \to \mathbb{C}$	
$x_a(t) :: \mathbb{C}$	Wert des analogen Signals x_a zum Zeitpunkt t

gebräuchlich, die wir mit der λ-Kalkül-artigen Schreibweise

$$\mathsf{x} :: \mathbb{Z} \to \mathbb{C} \qquad \text{mathematisch geprägt} \tag{5.1}$$

ergänzen wollen, um jederzeit ihre Typisierung ablesen zu können und somit z. B. an manchen Stellen den Unterschied zwischen Signalfolge und Signalwert herauszustellen.

Den Zusammenhang zwischen digitalem und analogem Signal veranschaulicht folgende Gleichung:

$$x[n] = x_a(nT) \qquad -\infty < n < \infty,$$

wenn wir annehmen, dass T die Abtastperiode bezeichnet.

Darüber hinaus wird im folgenden Text an manchen Stellen $x[n]$ verwendet, wo die gesamte Folge gemeint ist und nicht der n-te Abtastwert. Diese Schreibweise hat sich — obwohl mathematisch nicht sauber — in der Elektrotechnik eingebürgert, ähnlich einer Funktion $f(x)$, bei der die Funktion f und nicht der Funktionswert an der Stelle x gemeint ist.

Wir führen nun im weiteren Text zwei Notationen wo nötig parallel; zum einen die in der Elektrotechnik übliche, zum anderen die mathematisch geprägte in Klammern. Dies liest sich dann so: 'Eine Folge $x[n]$ (math: x) ...'

Verschiebung

Eine Folge $y[n]$ (math: y) bezeichnen wir gegenüber einer Folge $x[n]$ (math: x) um n_0 Abtastwerte verschoben, wenn gilt:

$$\begin{aligned} y[n] &= x[n-n_0] && \forall n \in \mathbb{Z} \\ \overrightarrow{y}^{n_0} &:: \mathbb{Z} \to \mathbb{C} \\ \overrightarrow{y}^{n_0}(n) &= x(n-n_0) && \forall n \in \mathbb{Z} \end{aligned}$$

Zusätzlich zu einer mathematischen Definition wollen wir für einige Funktionen eine Haskell-Version angeben[1]. So kann die Verschiebung folgendermaßen notiert werden:

Listing 5.1: Verschiebung

```
-- functionShift
-- verschiebt den Definitionsbereich einer Funktion f um n0 Einheiten nach rechts
functionShift :: (ℤ →ℚ) →ℤ→ℤ→ℚ
functionShift f n0 = \n →f (n−n0)
```

Einheitsimpuls

Bei der Betrachtung linearer Systeme spielt der *Einheitsimpuls* $\delta[n]$ (math: δ) eine wichtige Rolle, welcher folgendermaßen definiert ist:

$$\begin{aligned} \delta[n] &= \begin{cases} 0 & n \neq 0 \\ 1 & n = 0 \end{cases} \\ \delta &:: \mathbb{Z} \to \mathbb{C} \end{aligned}$$

[1] Dabei vereinfachen wir an manchen Stellen die Implementierung und schränken den Wertebereich von $\mathbb{C}$ auf $\mathbb{Z}$ bzw. $\mathbb{Q}$ ein aufgrund der Tatsache, dass Haskell keinen primitiven Datentyp für komplexe Zahlen zur Verfügung stellt. Die vollständigen Definitionen der Funktionen sind in Anhang B zu finden.

Listing 5.2: Einheitsimpuls

```
-- Einheitsimpuls
delta :: ℤ→ℚ
delta 0 = 1
delta _ = 0
```

Faltungssumme

Eine weitere wichtige Operation in der Elektrotechnik ist die *Faltungssumme* aus Gleichung (5.2), welche als Funktional über zwei Funktionen definiert ist:

$$f[n] = \sum_{k=-\infty}^{\infty} g[k]\, h[n-k] \tag{5.2}$$

$$f[n] = g[n] * h[n] \tag{5.3}$$

$$\mathsf{f} = \sum_{\mathsf{k}=-\infty}^{\infty} \mathsf{g}(\mathsf{k})\ \overset{\mathsf{k}}{\rightarrow}\ \mathsf{h} \tag{5.4}$$

$$\mathsf{f}(\mathsf{x}) = \sum_{\mathsf{k}=-\infty}^{\infty} \mathsf{g}(\mathsf{k})\ \overset{\mathsf{k}}{\rightarrow}\ \mathsf{h}\ (\mathsf{x}) \qquad \forall \mathsf{x} \in \mathbb{Z} \tag{5.5}$$

$$\circledast :: (\mathbb{Z} \to \mathbb{C}) \to (\mathbb{Z} \to \mathbb{C}) \to (\mathbb{Z} \to \mathbb{C})$$

$$\mathsf{f} = \mathsf{g} \circledast \mathsf{h} \tag{5.6}$$

Dabei bedeutet die Juxtaposition $g[k]\ h[n-k]$ in Gleichung (5.2) die punktweise Multiplikation einer Funktion $h[n]$ mit einem Skalar $g[k]$; $\sum_{k=-\infty}^{\infty}$ wiederum ist die punktweise Addition von Funktionen. Um Mehrdeutigkeiten mit der herkömmlichen Addition zu vermeiden, führen wir das Symbol $\circledast$ für die Faltungssumme zweier Funktionen ein.

In Gleichung (5.4) haben wir eine (von x) abstrahierte Definition der Faltungssumme angegeben, in der eine Funktionsmultiplikation zum Tragen kommt, während Gleichung (5.5) eine punktweise Definition beschreibt.

Listing 5.3: Faltungssumme

```
-- Faltungssumme (im endlichen Bereich von 0..n)
convolution :: (ℤ →ℚ) →(ℤ →ℚ) →ℤ→ℚ
convolution f g = \n →functionAddFold (map
                        (\k →(f k) ‘ functionMult ‘ ( functionShift g k)) [0 .. n]) n

-- functionShift
-- verschiebt den Definitionsbereich einer Funktion f um n0 Einheiten nach rechts
functionShift :: (ℤ →ℚ) →ℤ→ℤ→ℚ
functionShift f n0 = \n →f (n−n0)
```

```
-- Multiplikation einer Funktionen mit einer Konstanten
functionMult :: ℚ→(x →ℚ) →x →ℚ
functionMult a f = \n →a ∗ (f n)

-- punktweise Addition einer Liste von Funktionen
functionAddFold :: [ℤ →ℚ] →ℤ→ℚ
functionAddFold funcs n = foldr (\f x →(f n) + x) 0 funcs

-- Faltungssumme, punktweise definiert
-- berechne [f(0)..f(n)] und [g(0).. g(n)]
-- berechne [f(0)∗g(n)..f(n)∗g(0)]
-- addiere die resultierende Folge
convolution' :: (ℤ →ℚ) →(ℤ →ℚ) →ℤ→ℚ
convolution' f g = \n →foldr (+) 0 ( zipWith (∗) ( map f [0..n] ) ( reverse (map g [0..n] )))

-- Ist Faltungssumme kommutativ für Werte von 0..n?
convolutionCommTest f g n = and (map
                                  (\n0 →convolution' f g n0 == convolution' g f n0) [0..n])
```

Den Zusammenhang zwischen der punktweisen Definition aus Gleichung (5.5) und der in der Literatur vorzufindenden Gleichung (5.2) erläutert folgende Betrachtung, die sich auf den endlichen Fall beschränkt:

$$
\begin{aligned}
f(x_0) &= \sum_{k=0}^{x_0} g(k) \overset{k}{\rightarrow} h\,(x_0) && \forall x_0 \in \mathbb{Z} \\
&= g(0) \overset{0}{\rightarrow} h\,(x_0) + g(1) \overset{1}{\rightarrow} h\,(x_0) + \cdots + g(x_0-1) \overset{x_0-1}{\rightarrow} h\,(x_0) + g(x_0) \overset{x_0}{\rightarrow} h\,(x_0) \\
&= g(0)h(x_0) + g(1)h(x_0-1) + \cdots + g(x_0-1)h(1) + g(x_0)h(0) \\
&= f[x_0]
\end{aligned}
$$

Hier wird deutlich, wie die beiden Funktionen gegeneinander laufen, also die Funktionswerte der komplementären Indizes multipliziert und insgesamt akkumuliert werden.

Impulsfolge

Mit Hilfe des Einheitsimpulses und der Faltungssumme können wir nun eine beliebige Folge $x[n]$ (math: x) in Form einer *Impulsfolge* durch eine Summe von skalierten verzögerten Impulsen

darstellen:

$$x[n] = \sum_{k=-\infty}^{\infty} x[k]\ \delta[n-k]$$
$$\mathsf{x} :: \mathbb{Z} \to \mathbb{C}$$
$$\mathsf{x} = \sum_{\mathsf{k}=-\infty}^{\infty} \mathsf{x}(\mathsf{k})\ \overrightarrow{\delta}^{\mathsf{k}}$$
$$\mathsf{x} = \mathsf{x} \circledast \delta$$

Hier ist $\overrightarrow{\delta}^{k}$ der um k Einheiten verzögerte Einheitsimpuls. Folglich haben wir es mit einer unendlichen Summe über Funktionen zu tun, die jeweils nur einen Punkt $\neq 0$ zu der Gesamtfunktion beitragen. Im Kontext der Gruppentheorie spricht man bei der Funktion δ auch von einem Einselement.

Listing 5.4: Impulsfolge

```
-- Impulsfolge (über eine endliche Summe!)
-- impulseSequence func == func
impulseSequence xfun = convolution xfun delta

-- Test, ob die Impulsfolge mit der Originalfunktion übereinstimmt
impulseSequenceTest :: ℤ→ℤ→(ℤ →ℚ) →𝔹
impulseSequenceTest from to func =
    and (map (\n →impulseSequence func n == func n) [from..to])
```

Modifikation

Eine *Modifikation* einer Funktion f an der Stelle n ist definiert als:

$$(\mathbb{Z} \to \mathbb{C})[\mathbb{Z} \leftarrow \mathbb{C}] \to (\mathbb{Z} \to \mathbb{C})$$
$$f[n := v](n_0) = \begin{cases} v & n = n_0 \\ f(n) & \text{sonst} \end{cases}$$

Listing 5.5: Funktionsmodifikation

```
-- Abänderung einer Funktion func an Stelle n zum Wert v
functionModify func n v = \n0 →if n==n0 then v else (func n0)
```

Die Modifikation werden wir zur bequemen Definition von Eigenschaften zeitdiskreter Systeme in Abschnitt 5.1.2 benötigen.

5.1.2 Zeitdiskrete Systeme

Ein zeitdiskretes System wird im mathematischen Sinne als eine Transformation verstanden, die eine Eingangsfolge $x[n]$ (math: x) auf eine Ausgangsfolge $y[n]$ (math: y) abbildet:

$$\begin{array}{rcl} x[n] \longrightarrow \boxed{T\{\cdot\}} \longrightarrow y[n] & & \\ y[n] & = & T\{x[n]\} \\ \mathsf{y} & = & \mathsf{T}(\mathsf{x}) \end{array}$$

Notieren wir in unserer strengeren Notation, so ist $T\{\cdot\}$ (math: T) ein Funktional, welches eine Funktion auf eine Funktion abbildet:

$$\begin{array}{l} \mathsf{T} :: (\mathbb{Z} \to \mathbb{C}) \to (\mathbb{Z} \to \mathbb{C}) \\ \mathsf{T} = \lambda x.\lambda n.(T(x))[n] \end{array}$$

Systemklassen

In der Elektrotechnik haben sich wichtige Grundschemata herauskristallisiert, die sich in speziellen Systemklassen niederschlagen. Einige wichtige Klassen wollen wir nachfolgend vorstellen.

System ohne Speicher: Die Ausgangsfolge $y[n]$ ist bei jedem Wert n nur von dem dazugehörigen Wert $x[n]$ abhängig:

$$(\mathsf{T}(\mathsf{x}))(\mathsf{n}) = (\mathsf{T}(\mathsf{x}[\mathsf{m} := \mathsf{v}]))(\mathsf{n}) \qquad \forall \mathsf{n}, \mathsf{m}, \mathsf{v} : \mathsf{n} \neq \mathsf{m}$$

lineares System: Ein lineares System erfüllt die folgenden beiden Eigenschaften:

$$\begin{array}{rcll} T\{x_1[n] + x_2[n]\} & = & T\{x_1[n]\} + T\{x_2[n]\} & \textit{Überlagerung} \\ (\mathsf{T}(\mathsf{x}_1 + \mathsf{x}_2))(\mathsf{n}) & = & (\mathsf{T}(\mathsf{x}_1))(\mathsf{n}) + (\mathsf{T}(\mathsf{x}_2))(\mathsf{n}) \quad \forall \mathsf{n} \in \mathbb{Z} & \\ T\{ax[n]\} & = & aT\{x[n]\} & \textit{Skalierbarkeit} \\ \mathsf{T}(\mathsf{ax}) & = & \mathsf{a}(\mathsf{T}(\mathsf{x})) & \end{array}$$

zeitinvariantes System: Eine Zeitverschiebung der Eingangsfolge bewirkt eine entsprechende Zeitverschiebung der Ausgangsfolge:

$$\mathsf{T}(\overset{\mathsf{k}}{\rightarrow}(\mathsf{x})) = \overset{\mathsf{k}}{\rightarrow}(\mathsf{T}(\mathsf{x})) \qquad \forall \mathsf{k} \in \mathbb{Z}, \mathsf{x} \in \mathbb{C}$$

Kausalität: Ein Wert der Ausgangsfolge ist nur von vergangenen Werten der Eingangsfolge abhängig, also nichtvorhersagend:

$$(\mathsf{T}(\mathsf{x}))(\mathsf{n}) = (\mathsf{T}(\mathsf{x}[\mathsf{m} := \mathsf{v}]))(\mathsf{n}) \qquad \forall \mathsf{m}, \mathsf{n}, \mathsf{v} : \mathsf{m} \leq \mathsf{n}$$

5.2 Lineare zeitinvariante Systeme (LTI)

Benutzen wir die Eigenschaften der Linearität und der Zeitinvarianz, so können wir ein LTI-System (*linear time-invariant*) vollständig durch seine Impulsantwort $h_k[n]$ beschreiben, wenn $h_k[n]$ die Antwort auf $\delta[n-k]$ (math: $\overset{k}{\rightarrow}\delta$) ist:

$$y[n] = T\{\sum_{k=-\infty}^{\infty} x[k]\ \delta[n-k]\} \tag{5.7}$$

$$= \sum_{k=-\infty}^{\infty} x[k]\ T\{\delta[n-k]\} \tag{5.8}$$

$$= \sum_{k=-\infty}^{\infty} x[k]\ h_k[n] \tag{5.9}$$

$$= \sum_{k=-\infty}^{\infty} x[k]\ h[n-k] \tag{5.10}$$

$$\mathsf{y} = \sum_{\mathsf{k}=-\infty}^{\infty} \mathsf{x}(\mathsf{k})\ \overset{\mathsf{k},}{\rightarrow}\mathsf{h} \tag{5.11}$$

Diese besondere Gleichung können wir einfacher mittels der Faltungssumme ausdrücken:

$$y[n] = x[n] * h[n] \tag{5.12}$$

$$\mathsf{y} = \mathsf{x} \circledast \mathsf{h} \tag{5.13}$$

Zwei besondere Eigenschaften von linearen zeitinvarianten Systemen, die bei verschiedenen Implementierungsformen (Abschnitt 5.3) von besonderer Bedeutung sein werden, betrachten wir im Folgenden genauer.

Reihenschaltung

Durch die Kommutativität linearer zeitinvarianter Systeme können wir Teilsysteme, die in Reihe geschaltet sind, beliebig vertauschen, ohne das Verhalten des Gesamtsystems zu verändern, bzw. zu einem Gesamtsystem verschmelzen, indem wir die Impulsantworten miteinander multiplizieren.

Einen kurzen Beweis der Kommutativität liefert Folgendes:

$$
\begin{aligned}
(f \circledast g)(n) &= \sum_{k=-\infty}^{\infty} f(k) \overset{k}{\rightarrow} g(n) && \forall n \in \mathbb{Z} \\
&= \sum_{k=-\infty}^{\infty} f(k)\, g(n-k) \\
&= \sum_{m=-\infty}^{\infty} f(n-m)\, g(m) && \text{Var.Subst. } [m := n-k] \\
&= \sum_{m=-\infty}^{\infty} \overset{m}{\rightarrow} f(n)\, g(m) \\
&= \sum_{m=-\infty}^{\infty} g(m) \overset{m}{\rightarrow} f(n) && \text{Komm. Funktionsmult.} \\
&= (g \circledast f)(n)
\end{aligned}
$$

Graphisch lässt sich die Vertauschbarkeit der Reihenschaltung wie folgt ausdrücken:

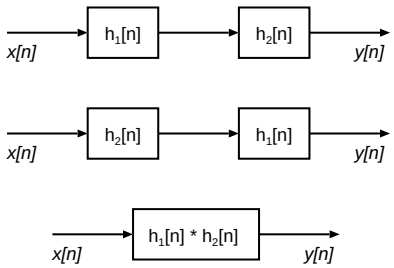


Abbildung 5.3: Reihenschaltung

Parallelschaltung

Die distributive Eigenschaft der Faltung besagt, dass zwei parallel geschaltete Teilsysteme äquivalent sind zu einem System, das die beiden Impulsantworten addiert.

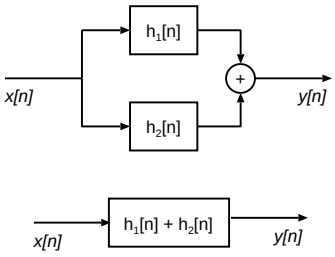


Abbildung 5.4: Parallelschaltung

Die Eigenschaften der Reihen- und Parallelschaltung werden bei den verschiedenen Implementierungsformen (Abschnitt 5.3) noch eine wichtige Rolle spielen.

5.2.1 Differenzengleichung

Eine interessante Unterklasse der linearen zeitinvarianten Systeme kann durch folgende lineare Differenzengleichung mit konstanten Koeffizienten ausgedrückt werden:

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k] \tag{5.14}$$

$$\sum_{k=0}^{N} a_k \overset{\overset{k}{\rightarrow}}{\mathsf{y}} = \sum_{k=0}^{M} b_k \overset{\overset{k}{\rightarrow}}{\mathsf{x}} \tag{5.15}$$

Durch diese Gleichung N-ter[2] Ordnung können alle zeitdiskreten, linearen, kausalen, zeitinvarianten Operatoren ausgedrückt werden, in der $x[n]$ (math: x) der Stimulus, also der Eingabestrom ist und $y[n]$ (math: y) den Ausgabestrom beschreibt ([Embree 1995], Seite 10).

Normieren wir in Gleichung (5.14) mit dem Wert a_0 und bringen $y[n-k]$ (math: $\overset{\overset{k}{\rightarrow}}{\mathsf{y}}$) auf die linke Seite, so erhalten wir

$$y[n] = \sum_{k=0}^{M} \frac{b_k}{a_0} x[n-k] \; - \; \sum_{k=1}^{N} \frac{a_k}{a_0} y[n-k] \tag{5.16}$$

$$\mathsf{y} :: \mathbb{Z} \to \mathbb{C}$$

$$\mathsf{y} = \sum_{k=0}^{M} \frac{b_k}{a_0} \overset{\overset{k}{\rightarrow}}{\mathsf{x}} \; - \; \sum_{k=1}^{N} \frac{a_k}{a_0} \overset{\overset{k}{\rightarrow}}{\mathsf{y}} \tag{5.17}$$

Um nun einen Operator mittels Gleichung (5.16) adäquat repräsentieren zu können, müssen wir nur ein geeignetes N finden. In der Tat ist es so, dass wir jeden Operator mit beliebiger Genauigkeit approximieren können, wenn wir ein genügend großes N wählen.

5.2.2 Digitale Filter

Lineare Differenzengleichungen können nach einem Schema klassifiziert werden, wenn ausgehend von der allgemeinen, normierten Form

$$y[n] = \sum_{k=0}^{M} b_k x[n-k] \; - \; \sum_{k=1}^{N} a_k y[n-k] \tag{5.18}$$

$$\mathsf{y} = \sum_{k=0}^{M} b_k \overset{\overset{k}{\rightarrow}}{\mathsf{x}} \; - \; \sum_{k=1}^{N} a_k \overset{\overset{k}{\rightarrow}}{\mathsf{y}} \tag{5.19}$$

[2] o. B. d. A.: $N \geq M$, ggf. sind einige $a_i, b_i = 0$

folgende digitale Filter eine vereinfachte charakteristische Differenzengleichung ([Embree 1995]) aufweisen.

IIR
Die Klasse der IIR-Filter (*infinite impulse response*) beschreibt einen Filter mit Rückkopplung und teilt sich auf in

- *AR*-Filter (*autoregressive*) und
- *ARMA*-Filter (*autoregressive moving average*).

ARMA-Filter weisen die allgemeinste Form auf und sind in Gleichung (5.18) beschrieben. Ein Wert $y[n]$ ist von beliebigen Eingangswerten $x[n]$, $x[n-1]\ldots$ und daraus berechneten Ausgangswerten $y[n-1]$, $y[n-2]\ldots$ abhängig. Ein AR-Filter hingegen besitzt die Form aus Gleichung (5.20), in der alle b_k mit $k = 1\,..\,M$ gleich 0 gesetzt werden und die übrigen Koeffizienten so normiert sind, dass $b_0 = 1$ gilt:

$$y[n] = \sum_{k=1}^{N} a_k y[n-k] \; + x[n] \tag{5.20}$$

$$\mathsf{y} = \mathsf{x} + \sum_{k=1}^{N} a_k \overrightarrow{\mathsf{y}}^{\mathsf{k}} \tag{5.21}$$

Blahut verweist in [Blahut 1985] auf die Tatsache, dass IIR-Filter eine Polynomdivision beschreiben.

FIR
Die Klasse der FIR-Filter (*finite impulse response*) hat die Eigenschaft, dass die Antwort auf einen Impuls nach einer endlichen Zeit ausläuft. Mit $a_k = 0$ ($k = 1\,..\,N$) kann sie nur eine (endliche) Anzahl von Eingabewerten berücksichtigen. Ihre Gleichung lautet:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k]$$

$$\mathsf{y} = \sum_{k=0}^{M} b_k \overrightarrow{\mathsf{x}}^{\mathsf{k}}$$

FIR-Filter sind dadurch inhärent stabil, d. h. sie erzeugen zu einer beschränkten Eingangsfolge eine beschränkte Ausgangsfolge. Sie haben aber den Nachteil, dass sie zumeist mehr Addierer und Speicherelemente benötigen als ein entsprechender IIR-Filter. Blahut zieht in [Blahut 1985] eine Parallele zu einem Polynomprodukt.

5.2.3 Systemfunktion

Bisher haben wir unser Augenmerk auf die Eigenschaften von zeitdiskreten Signalfolgen gelegt. Nun geben wir einen kurzen Einblick in deren Darstellung und Analyse. Dieser Hintergrund

wird eine bedeutende Rolle bei der Modellierung (Abschnitt 5.4) und Generierung in eine Programmiersprache (Abschnitt 5.5.1 und 5.5.2) spielen.

In Gleichung (5.7) haben wir gesehen, dass wir ein Ausgangssignal vollständig durch seine Impulsantwort beschreiben können. Gleichung (5.12) beschreibt den gleichen Sachverhalt in einer kompakteren Notation:

$$y[n] = x[n] * h[n]$$

Wenden wir auf die entsprechenden Signale eine aus den Grundlagen der Elektrotechnik bekannte z-Transformation[3] an, so kommen wir zu der Gleichung:

$$Y(z) = H(z)\,X(z) \tag{5.22}$$

Der Term $H(z)$ wird in diesem Zusammenhang auch *Systemfunktion* genannt.

Werfen wir noch einmal einen Blick auf die allgemeine Form der Differenzengleichung:

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

Darauf wenden wir auf beiden Seiten der Gleichung die z-Transformation an und kommen zu der Gestalt

$$\left(\sum_{k=0}^{N} a_k z^{-k}\right) Y(z) = \left(\sum_{k=0}^{M} b_k z^{-k}\right) X(z)$$

nachdem wir den konstanten Faktor $X(z)$ bzw. $Y(z)$ ausgeklammert haben.

Mit Hilfe von Gleichung (5.22) können wir nun die Systemfunktion $H(z)$, vorausgesetzt das System ist linear und zeitinvariant, in der Form

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{M} b_k z^{-k}}{\sum_{k=0}^{N} a_k z^{-k}} \tag{5.23}$$

beschreiben.

Die Systemfunktion aus Gleichung (5.23) ist ein Quotient aus Polynomen in z^{-1}. Sind wir an den Nullstellen bzw. Polen von $H(z)$ interessiert, können wir die Funktion auch in einer faktorisierten Notation angeben:

$$H(z) = \left(\frac{b_0}{a_0}\right) \frac{\prod_{k=1}^{M}(1 - c_k z^{-1})}{\prod_{k=1}^{N}(1 - d_k z^{-1})}$$

Hier repräsentieren die c_k die Nullstellen und die d_k die Pole.

[3] Die z-Transformation ist eine Verallgemeinerung der Fourier-Transformation, die eine Folge $x[n]$ in eine Funktion $X(z)$ (z komplexe Variable) konvertiert, d. h. eine diskrete Funktion in eine kontinuierliche Funktion umwandelt.

5.2.4 Blockdiagramm

In den vergangenen Abschnitten haben wir gesehen, dass uns mehrere Möglichkeiten zur Beschreibung von linearen zeitinvarianten Systemen offen stehen. So können wir einerseits eine lineare Differenzengleichung mit konstanten Koeffizienten heranziehen, zum anderen besteht die Möglichkeit, dass wir die Systemfunktion (Gleichung (5.23)) benutzen. Eine alternative Variante zur Beschreibung besteht in der Verwendung von *Blockdiagrammen*, die den Datenflussgraphen aus HOPS entsprechen.

Kommen wir zu dem Punkt, dass solche Systeme mit digitaler Schaltungstechnik realisiert werden sollen, so werden wir sehen, dass nur wenige elementare Bausteine zu deren Realisierung nötig sind. Diese Elementarstrukturen sind im Wesentlichen

- die *Addition*,
- die *Multiplikation mit einer Konstanten* und
- die *Verzögerung*.

Die Differenzengleichung aus Gleichung (5.18))

$$y[n] = \sum_{k=1}^{N} a_k y[n-k] \; + \; \sum_{k=0}^{M} b_k x[n-k] \tag{5.24}$$

gibt uns bereits Aufschluss, wie eine mögliche Implementierung auszusehen hat.

Wollen wir diese direkt in einer Programmiersprache kodieren, so erhalten wir durch den Term $\sum_{k=1}^{N} a_k y[n-k]$ eine Rückkopplung, d. h. einen Bezug auf ältere, ausgehende Signale. Bei einer direkten, naiven Implementierung bekämen wir eine N-stufige Kaskadenrekursion. Um das Laufzeitverhalten zu verbessern und dies überhaupt erst in Schaltungen realisieren zu können – Rekursion ist dynamisch und somit nicht auf Schaltungen abbildbar –, überführen wir dies in eine Schleife, die über ein begrenztes Gedächtnis (lokale $N+M$ Speicherplätze) verfügt, das eine endliche Zahl bereits berechneter, ausgehender Signale zwischenspeichert.

Rufen wir uns noch einmal die z-Transformierte des Ausgangssignals aus Gleichung (5.22) in Erinnerung, dann können wir die Systemfunktion $H(z)$ aufgrund ihrer LTI-Eigenschaften (Abschnitt 5.2) in zwei einzelne Teilsystemfunktionen $H_1(z)$ und $H_2(z)$ aufspalten.

$$\begin{aligned} Y(z) &= H(z)\, X(z) \\ Y(z) &= H_2(z)\, H_1(z)\, X(z) \\ Y(z) &= H_2(z)\, V(z) \qquad\qquad \text{mit} \\ V(z) &= H_1(z)\, X(z) \end{aligned}$$

Auf diese Weise können wir die Teilsysteme H_1 und H_2 unabhängig voneinander implementieren und anschließend kompositional (Hintereinanderschaltung) miteinander verbinden. Matchen wir nun die Systemfunktion auf die Form von Gleichung (5.23), so erhalten wir die beiden Teilsystemfunktionen[4]

$$H_2(z) = \frac{1}{1 - \sum_{k=1}^{N} a_k z^{-k}}$$
$$H_1(z) = \sum_{k=0}^{M} b_k z^{-k}$$

Direkte Form I

Die gewonnenen Teilsystemfunktionen, bzw. deren Pendant als Differenzengleichung, lassen sich in der *direkten Form I* in einem Blockdiagramm (Abbildung 5.5) darstellen ([Oppenheim und Schafer 1992], S. 333), indem wir die Systemfunktionen in einer Tabelle nachschauen. Die Abbildung zeigt, dass $M + N$ Verzögerungselemente, respektive Register, benötigt werden.

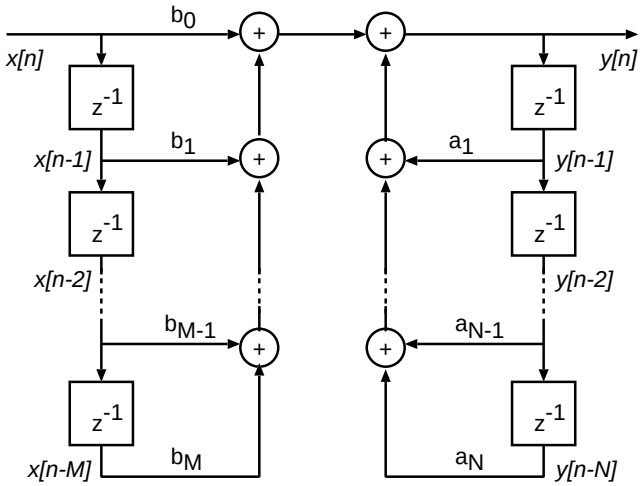


Abbildung 5.5: Blockdiagramm, direkte Form I

Modifizieren wir die Systemfunktion etwas, indem wir die Funktionen $H_1(z)$ und $H_2(z)$ kommutieren lassen

$$H(z) = H_1(z)\, H_2(z)$$

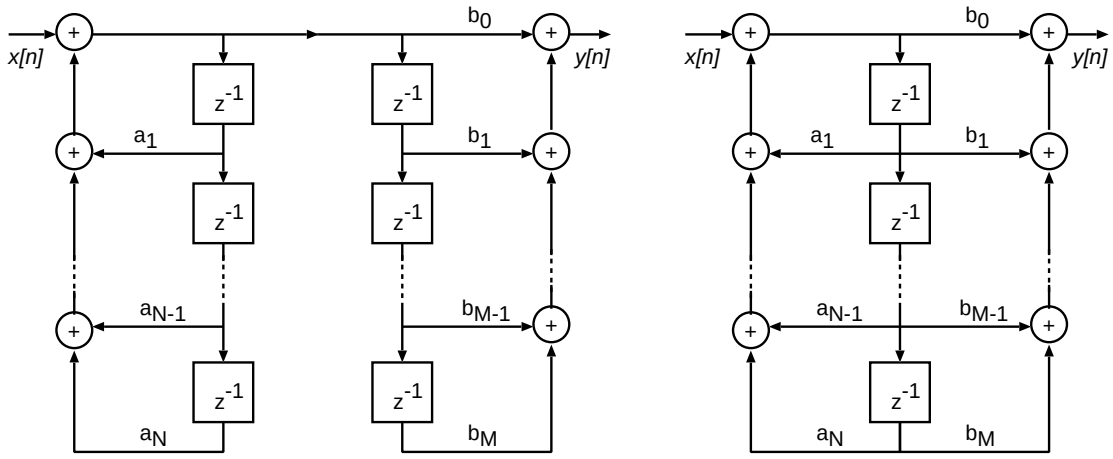
[4] Die Konstante 1 und die negativen Koeffizienten a_i sind durch die Normierung in Gleichung (5.18) bedingt.

so ergibt sich daraus ein geändertes Blockdiagramm mittels der Gleichungen

$$Y(z) = H_1(z)\, W(z) \text{ mit}$$
$$W(z) = H_2(z)\, X(z)$$

Direkte Form II

Aus dem Blockdiagramm können wir erkennen, dass die Register identische Werte aufnehmen und somit auf $min(N+M)$ Register verzichten. Hieraus können wir sofort die *direkte Form II* [5] (Abbildung 5.6(b)) ableiten.



(a) umgeformt

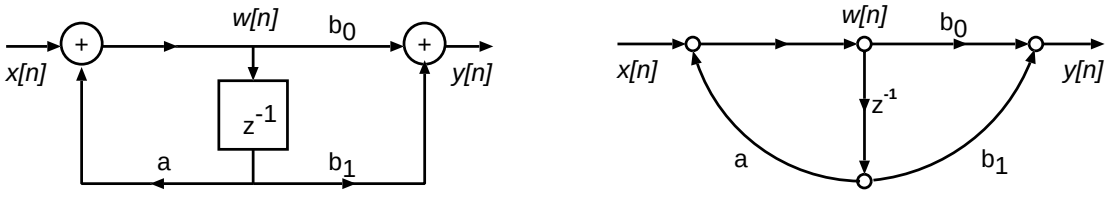
(b) vereinfacht

Abbildung 5.6: Blockdiagramm, direkte Form II

5.2.5 Signalflussgraph

Ein Signalflussgraph (Abbildung 5.7) ist einem Blockdiagramm sehr ähnlich, lediglich die Notation ist etwas anders. Tatsächlich ist der einzige wesentliche Unterschied der beiden Repräsentationen, dass ein Knoten im Signalflussgraph ein Astende und eine Additionsstelle symbolisiert, während es im Blockdiagramm für den Addierer ein besonderes Symbol gibt. Dabei besitzt ein Astende im Signalflussgraphen per Definition nur eine eingehende Kante, ein Addierer mehrere eingehende Kanten.

[5] o. B. d. A sei N=M bzw. einige a_i, $b_j = 0$



(a) Blockdiagramm (b) Signalflussgraph

Abbildung 5.7: Vergleich von Blockdiagramm und Signalflussgraph

5.3 Implementierungsformen

Wie wir gesehen haben, können wir aus einer gegebenen Systemfunktion $H(z)$ ein korrespondierendes Blockdiagramm bzw. einen äquivalenten Signalflussgraphen entwickeln. Dabei stellt sich die Frage, ob es nicht mehrere Möglichkeiten der Implementierung zu einer Systemfunktion gibt.

Dieser Abschnitt zeigt unterschiedliche Formen theoretisch äquivalenter Implementierungen auf, die verschiedene Aspekte der Komplexität berücksichtigen, wie z. B. minimale Anzahl der Speicherregister bzw. Multiplikationseinheiten. Auch die endliche Genauigkeit von elektronischen Recheneinheiten spielt bei einer Implementierung eine besondere Rolle, so dass wir u. U. durch geschickte Anordnung der Bausteine die signifikanten Stellen einer Ausgabe erhöhen können.

5.3.1 Direkte Formen

Direkte Form I

In Abschnitt 5.2 haben wir gesehen, wie wir die Differenzengleichung

$$y[n] - \sum_{k=1}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

mit der korrespondierenden Systemfunktion

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 - \sum_{k=1}^{N} a_k z^{-k}} \tag{5.25}$$

in der *direkten Form I* als Signalflussgraph (siehe Abbildung 5.8) implementieren.

Direkte Form II

Dem gegenüber steht die *direkte Form II* (siehe Abbildung 5.9), die eine etwas effizientere Implementierung im Hinblick auf die Anzahl der Speicherregister darstellt.

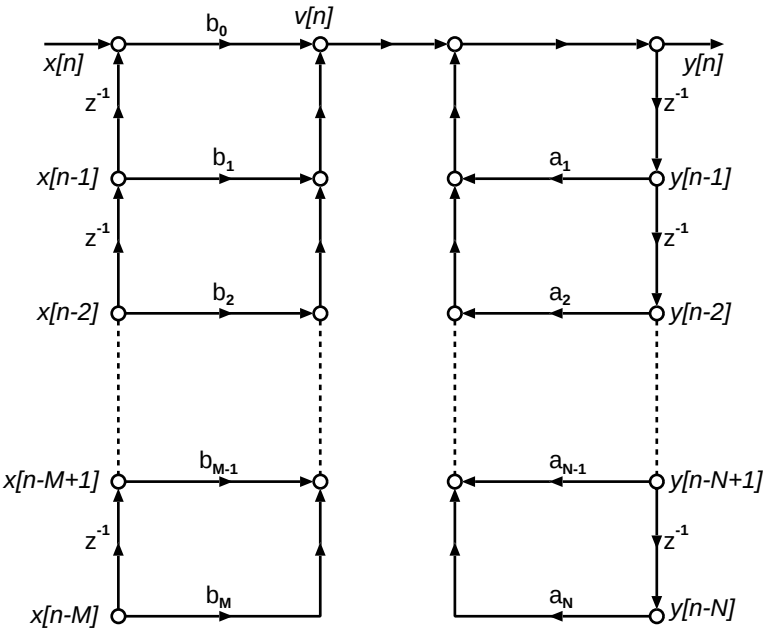


Abbildung 5.8: Signalflussgraph, direkte Form I

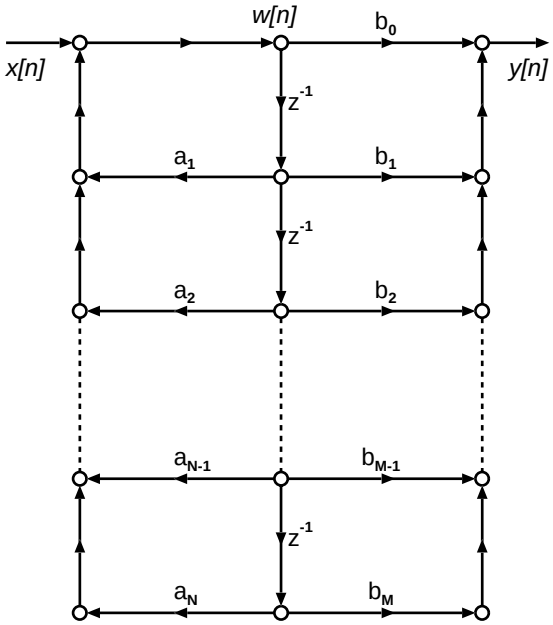


Abbildung 5.9: Signalflussgraph, direkte Form II

5.3.2 Kaskadierte Form

Kann man die direkte Form aus der Systemfunktion in Gleichung (5.25) ablesen, so kommt man zu der *kaskadierten Form*, indem wir das Zähler- und Nennerpolynom der Systemfunktion faktorisieren. Gleichung (5.26) zeigt das Ergebnis der genannten Umformung:

$$H(z) = A\frac{\prod_{k=1}^{M_1}(1-g_k z^{-1})\prod_{k=1}^{M_2}(1-h_k z^{-1})(1-h_k^* z^{-1})}{\prod_{k=1}^{N_1}(1-c_k z^{-1})\prod_{k=1}^{N_2}(1-d_k z^{-1})(1-d_k^* z^{-1})} \tag{5.26}$$

In diesem allgemeinsten Ausdruck stellen die Faktoren erster Ordnung reelle Nullstellen bei g_k und reelle Pole bei c_k dar, wo hingegen die Faktoren zweiter Ordnung durch die konjungiert komplexen Paare repräsentiert werden. Deren Nullstellen sind bei h_k bzw. h_k^* anzutreffen, deren Pole bei d_k bzw. d_k^*.

Aus Gleichung (5.26) können wir nun mehrere Beobachtungen ziehen. Zum einen haben wir es mit einer Reihe von Faktoren zu tun (z. B. $\prod_{k=1}^{M_1}$), was uns nahe legt, ein Produkt als Hintereinanderschaltung der Terme zu implementieren. Wir erinnern uns an Abschnitt 5.2, dass in linearen zeitinvarianten Systemen das Produkt von Impulsantworten einer Hintereinanderschaltung entspricht. Des Weiteren ist das Produkt kommutativ, so dass wir weitere Freiheitsgrade erhalten, nämlich in der beliebigen Anordnung der Faktoren bzgl. der Reihenfolge.

Wollen wir z. B. ein System mit maximal zweiter Ordnung entwerfen, so bringen wir Gleichung (5.26) in eine Form, die paarweise reelle Faktoren und konjugiert komplexe Paare zu Faktoren zweiter Ordnung kombiniert. Wir erhalten hierdurch eine Systemfunktion folgender Gestalt:

$$H(z) = \prod_{k=1}^{N_s}\frac{b_{0k}+b_{1k}z^{-1}+b_{2k}z^{-2}}{1-a_{1k}z^{-1}-a_{2k}z^{-2}}$$

Aus der Darstellung des Signalflussgraphen können wir erkennen, dass die Grundsysteme zweiter Ordnung sind, die durch eine Hintereinanderschaltung verknüpft sind, welche durch die Faktoren des Produkts induziert werden. Das Beispiel aus Abbildung 5.10 zeigt ein Gesamtnetz sechster Ordnung, das aus hintereinandergeschalteten Subsystemen zweiter Ordnung besteht, welche ihrerseits wiederum die direkte Form II aufweisen.

Die allgemeine Form der Differenzengleichungen für ein kaskadiertes System, das aus einer Reihe von Systemen zweiter Ordnung besteht, lautet folgendermaßen:

$$\begin{aligned}
y_0[n] &= x[n] \\
w_k[n] &= a_{1k}w_k[n-1]+a_{2k}w_k[n-2]+y_{k-1}[n], \quad k=1,2,\ldots,N_s \\
y_k[n] &= b_{0k}w_k[n]+b_{1k}w_k[n-1]+b_{2k}w_k[n-2], \quad k=1,2,\ldots,N_s \\
y[n] &= y_{N_s}[n]
\end{aligned}$$

Ebenso könnten wir durch Umformung der Systemfunktion auch Grundsysteme der dritten Ordnung herleiten, die wir wiederum in Reihe schalten. Hier ist ersichtlich, dass wir eine große

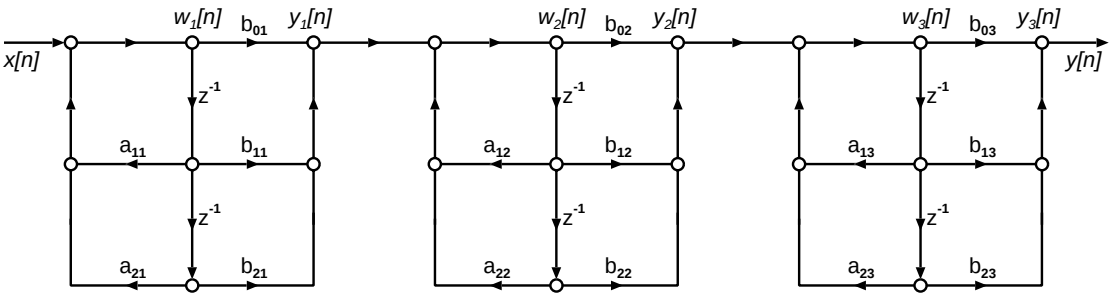


Abbildung 5.10: Kaskadierte Form

Anzahl an Freiheitsgraden haben, die es uns erlaubt, eine nach unterschiedlichen Gesichtspunkten optimierte Implementierung zu erhalten.

Hier stellt sich natürlich die Frage, ob die (mathematisch) äquivalenten Systeme, die ja alle die gleiche Systemfunktion implementieren, sich in der Praxis nicht doch unterschiedlich verhalten. In der Tat zeigen sich hier Unterschiede durch den Umstand, dass wir es bei einer digitalen Hardware mit einer Arithmetik von endlicher Genauigkeit zu tun haben. Auf die Thematik von Rundungsfehlern wollen wir aber hier nicht näher eingehen, da sie für die Beschreibung des allgemeinen Verfahrens nicht von besonderem Interesse ist.

5.3.3 Parallele Form

Nutzten wir bei der kaskadierten Form noch spezifische Eigenschaften der Reihenschaltung von linearen zeitinvarianten Systemen, so machen wir bei der *parallelen Form* von der Tatsache Gebrauch, dass der Zusammenfluss einer Parallelschaltung einer Addition gleichkommt.

So können wir die Systemfunktion durch Umformungen wie z. B. einer Partialbrucherweiterung auf folgende Gestalt bringen:

$$H(z) = \sum_{k=0}^{N_p} C_k z^{-k} + \sum_{k=1}^{N_s} \frac{e_{0k} + e_{1k} z^{-1}}{1 - a_{1k} z^{-1} - a_{2k} z^{-2}}$$

Hier haben wir es mit N_s parallel geschalteten Elementen zweiter Ordnung zu tun, die wir mittels der direkten Form modellieren können, und zusätzlich N_p mit einem Faktor gewichteten Verzögerungen. Dabei ist $N_s = \lfloor (N+1)/2 \rfloor$.

Daraus können wir eine beispielhafte parallele Form mit einer Gesamtstruktur sechster Ordnung ableiten, wie sie in Abbildung 5.11 zu sehen ist.

Aus dieser Abbildung lässt sich folgende Systemfunktion ablesen:

$$H(z) = C_0 + \sum_{k=1}^{3} \frac{e_{0k} + e_{1k} z^{-1}}{1 - a_{1k} z^{-1} - a_{2k} z^{-2}}$$

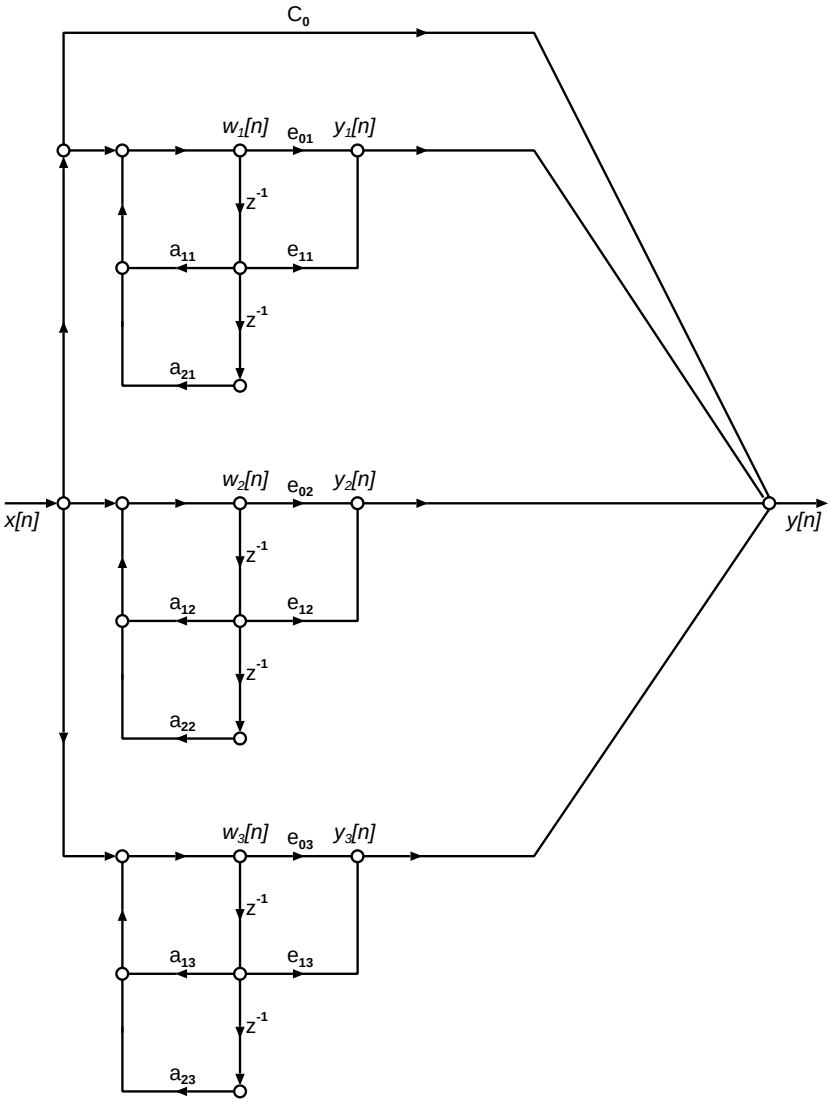


Abbildung 5.11: Parallele Form

5.3.4 FIR-Systeme

Haben allgemeine IIR-Systeme die Eigenschaft, dass sie sowohl Nullstellen als auch Pole (Rückwärtskanten im Signalflussgraphen) besitzen, so vereinfacht sich dies in FIR-Systemen dahingehend, dass wir es nur noch mit Nullstellen zu tun haben.

Somit sind FIR-Systeme spezielle Systeme ohne Rückkopplung, die folgende vereinfachte Differenzengleichung aufweisen:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k]$$

Aus dieser vereinfachten Faltung ergibt sich ein entsprechend einfacherer Signalflussgraph ohne Zyklen wie in Abbildung 5.12.

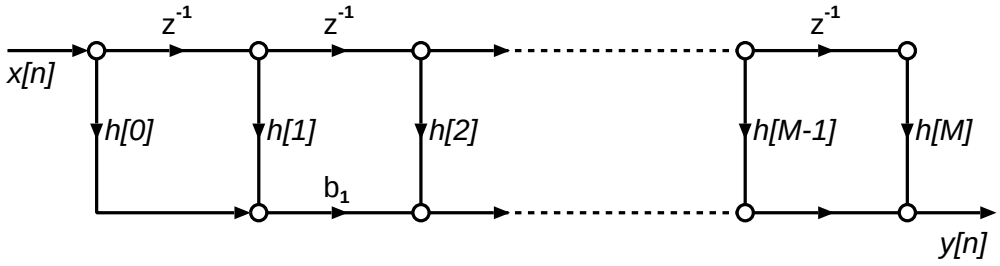


Abbildung 5.12: Signalflussgraph, FIR-System

Das FIR-System ist analog zu einem allgemeinen IIR-System in den Grundformen direkte Form, kaskadierte Form und parallele Form implementierbar.

FIR-Systeme mit linearer Phase

Ein kausales FIR-System besitzt eine verallgemeinerte lineare Phase, wenn es darüberhinaus die Eigenschaft

$$h[M-n] = h[n] \qquad \text{für } n = 1,\ldots,M \tag{5.27}$$

aufweist, aus der wir in seiner Implementierung weiteren Nutzen ziehen können. Dabei ist

$$h[n] = \begin{cases} b_n, & \text{für } n = 1,\ldots,M \\ 0 & \text{sonst} \end{cases} \tag{5.28}$$

die Impulsantwort, mit der ein Eingabestrom gefaltet wird.

Aus Gleichung (5.27) leiten wir ab, dass sich die Anzahl an Multiplikationen pro Taktzyklus im Wesentlichen halbiert, da sich die Koeffizienten b_i, die ja in der Impulsantwort stecken, doppelt vorkommen. So kommen wir von einer Faltungsgleichung unter der Voraussetzung, dass M

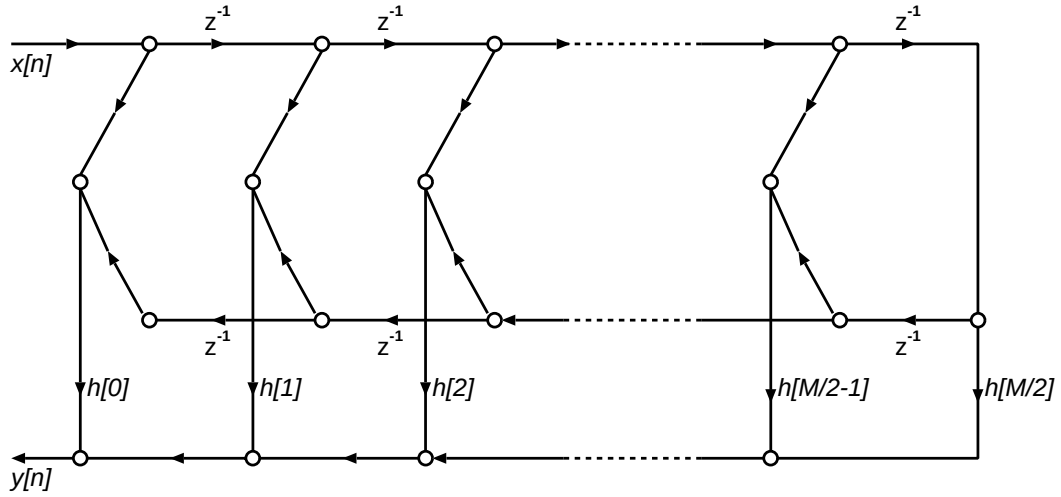


Abbildung 5.13: Signalflussgraph, FIR-System mit linearer Phase

gerade ist, mit einem Systemtyp I auf folgende Termumformung

$$y[n] = \sum_{k=0}^{M} h[k]x[n-k] \tag{5.29}$$

$$= \left(\sum_{k=0}^{M/2-1} h[k](x[n-k]+x[n-M+k]) \right) + h[M/2]x[n-M/2] \tag{5.30}$$

Hier sieht man, dass durch die vorherige Zusammenfassung von $x[n-k]+x[n-M+k]$ jetzt $M/2$ weniger Multiplikationen mit $h[k]$ anfallen. Abbildung 5.13 verdeutlicht diesen Aspekt.

5.4 Modellierung in HOPS

Wir haben in den vorangegangenen Abschnitten den theoretischen Hintergrund der digitalen Signalverarbeitung beleuchtet und gezeigt, wie lineare zeitinvariante Systeme in allgemeine Implementierungsformen transformiert werden können. Nun wollen wir diese Transformationsschritte in einem konkreten System (HOPS) beschreiben und einen semiautomatischen Weg aufzeigen, wie wir ausgehend von einer abstrakten Beschreibung eines linearen zeitinvarianten Systems eine konkrete Implementierung in Form eines ausführbaren Programms erzielen.

Das weitere Vorgehen beschreibt Abbildung 5.14. Ausgehend von einer allgemeinen Systemfunktion vereinfachen wir diese, indem wir sukzessive Bausteine wie Addition, Multiplikation und Verzögerung einführen. Hier sind wir daran interessiert, Bausteine mehrfach zu verwenden, um eine möglichst effiziente Implementierung zu erhalten.

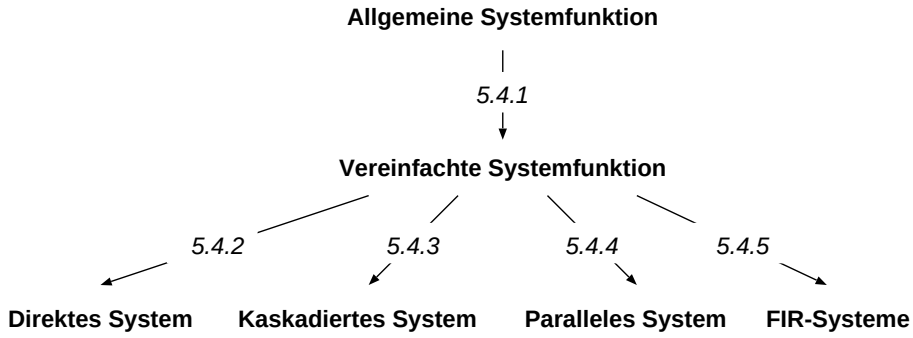


Abbildung 5.14: Transformation in unterschiedliche Systemformen

In weiteren Schritten instanzieren wir das System in die unterschiedlichen Ausprägungen direktes, kaskadiertes, paralleles und FIR-System.

5.4.1 Allgemeine Systemfunktion

Ausgehend von der Systemfunktion aus Gleichung (5.23) wollen wir nun eine Implementierung in HOPS entwickeln, die äquivalent zu einem Blockdiagramm bzw. Signalflussgraphen in der direkten Form I ist. Anschließend transformieren wir die Systemfunktion innerhalb des HOPS-Systems zu der direkten Form II, die einfacher im Hinblick auf die Anzahl der Register ist.

Eine allgemeine Systemfunktion für eine Differenzengleichung der Form

$$Y(z) = H_2(z) * H_1(z) * X(z)$$
$$Y(z) = \frac{1}{1 - \sum_{k=1}^{N} a_k z^{-k}} * \sum_{k=0}^{M} b_k z^{-k} * X(z)$$

lässt sich in HOPS in ein allgemeines Schema (siehe Abbildung 5.15) fassen:

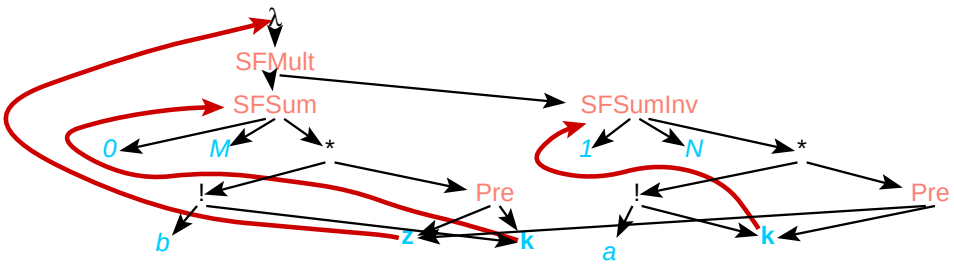
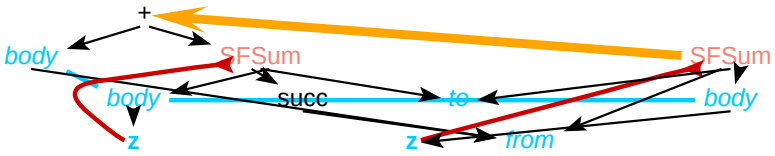


Abbildung 5.15: Allgemeines Schema einer Systemfunktion $H_2(z) * H_1(z) * X(z)$

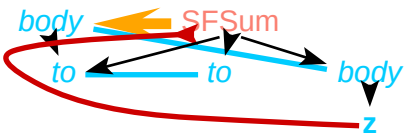
Hier ist zu beachten, dass die Multiplikation von Systemfunktionen in $H_2(z) * H_1(z) * X(z)$ von rechts her assoziiert (also $H_2(z) * (H_1(z) * X(z))$ entspricht), wo hingegen die ausgehenden Kanten im HOPS-Graphen von links nach rechts zu lesen sind. Eine Klammerung in HOPS benötigen wir aufgrund der eindeutigen Graphstruktur nicht.

Systemfunktion ohne Rückkopplung

Die Iteration der Systemfunktion SFSum von 0 bis M können wir mittels der Regel aus Abbildung 5.16 expandieren.



(a) Rekursion



(b) Rekursionsabbruch

Abbildung 5.16: Transformationsregel für SFSum

Um auf Werte aus vorhergehenden Iterationen zugreifen zu können, bedienen wir uns des Konstrukts Pre z k, das auf die Eingangsvariable z vor k Iterationen zurückgreift. Diese Zugriffe auf bereits erfolgte Berechnungen werden wir durch den Einsatz von Speicherelementen implementieren, die explizit einen Zustand einführen. Die Regeln aus Abbildung 5.19 bilden sukzessive den Baustein Pre auf ein Verzögerungselement (Z^{-1}) ab.

Später werden wir sehen, dass der Regelmechanismus in HOPS für unsere Zwecke zu scharfe Konsistenzbedingungen berücksichtigt, die die Anwendbarkeit dieser Regeln nicht ermöglichen. Wir werden daher einen allgemeiner gefassten Begriff der Anwendbarkeit und der interaktiven Substitution definieren.

Wir wollen nun ausgewählte Transformationsschritte zu einem Blockdiagramm exemplarisch

aufzeigen. Ausgehend von der Gleichung

$$H_1(z) = \sum_{k=0}^{2} b_k z^{-k} \tag{5.31}$$

die dem Graphen aus Abbildung 5.17(a)[6] entspricht, wenden wir die Regeln aus Abbildung 5.16 an, die gemeinsam die Summe über k abrollen und den Graphen aus Abbildung 5.17(b) liefern. Mit weiteren Regeln zur Einführung des Bausteins Z^{-1} kommen wir zu einer expandierten Version (Abbildung 5.17(c)) unseres Ursprungs-DAGs.

Gehen wir von dem Ergebnis zu einer Datenflussdarstellung über, so erzielen wir ein Blockdiagramm der direkten Form I aus Abbildung 5.18.

Beschränkung der Regeldefinition in HOPS

Beim Einführen des Baustein Z^{-1} sind einige Randbedingungen zu beachten, die Gegenstand dieses Abschnitts sind. Werfen wir dazu einen Blick auf die Regel von Pre, die ein Abstützen des Zugriffs auf die Variable z vor $k+1$ Iterationen auf den Zugriff vor k Iterationen einführt.

Ein direkte Herangehensweise, den gewünschten Baustein einzuführen, ist in den Regeln aus Abbildung 5.19 abgebildet.

Hier wird für jeden Zugriff auf den Wert von z von vor $n+1$ Inkarnationen ein Verzögerungsbaustein Z^{-1} eingeführt, der sich auf einen Zugriff vor n Inkarnationen abstützt. Der gravierende Nachteil bei diesem Verfahren besteht darin, dass neue Zugriffsbausteine (Pre) eingefügt werden und nicht auf bereits vorhandene abgestützt wird.

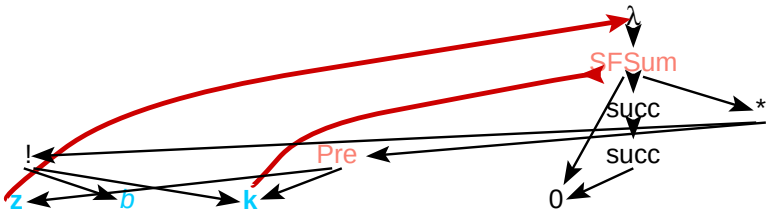
Also liegt die Idee nahe, mit mehrstelligen Metavariablen zu arbeiten, die eine vorhandene Graphenstruktur berücksichtigen und bestehende Teilstrukturen wiederverwenden. Die entsprechenden Regeln sind in Abbildung 5.20 zu sehen. Hier stoßen wir sofort auf das Problem, dass die Regel in Abbildung 5.20(a) nicht eindeutig ist, da die Metavariable MV zwei Nachfolger besitzt, die nicht eindeutig gekapselt sind. In dieser Regel weisen zwar die Nachfolger-DAGs der Metavariablen gebundene Variablen (z) auf, die von einem Binder gekapselt sind, doch sind diese nicht unterschiedliche Variable, so dass die Kapselung nicht eindeutig ist [7]. Zum anderen verhindert die Konsistenzbedingung der *Variablenkontrolle* [8], dass die Regel auf einen DAG anwendbar ist, der mehr als zwei Pre-Bausteine enthält.

Andererseits können wir eine eindeutige Regel mit nullstelligen Metavariablen (Abbildung 5.21) definieren, was zu Folge hat, dass an manchen Stellen das Sharing ungewollt aufgetrennt wird und Kopien von Teilstrukturen im Graphen entstehen. Außenverweise auf den Knoten ‘Pre z succ k’ der linken Regelseite werden nicht, wie man sich das wünschen würde, auf den Knoten Z^{-1} der rechten Regelseite überführt. Stattdessen bleibt der Knoten ‘Pre z succ k’ separat erhalten mit denjenigen Vorgängern, die nicht auf die Regel matchen. Betrachten wir

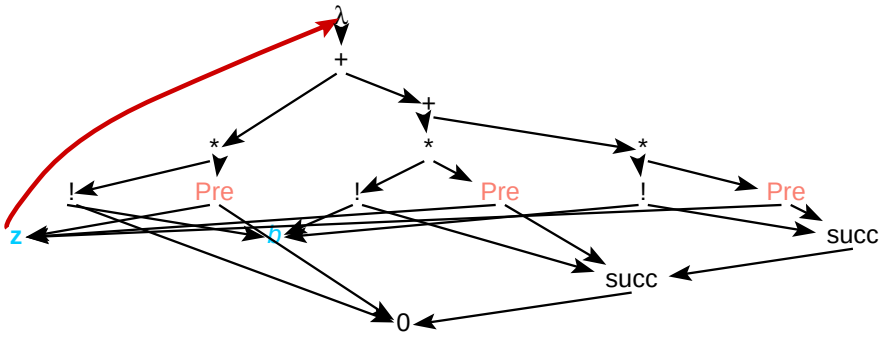
[6] Der Baustein Z^{-1} wird in den Graphen mit ‘Z-1’ bezeichnet.

[7] In einem DAG treten gewöhnlich mehrere Vorkommen von Pre auf.

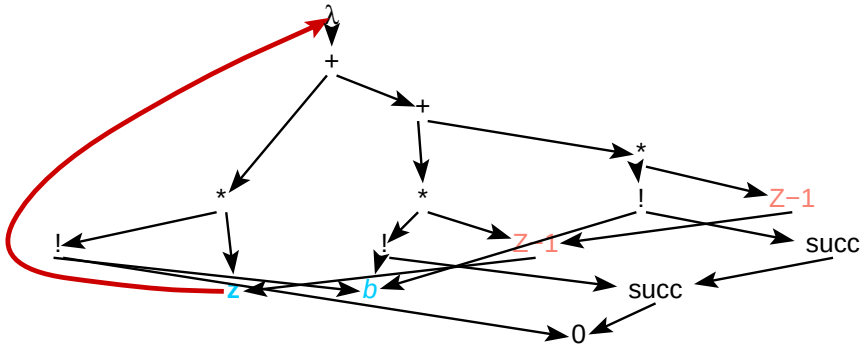
[8] Die Konsistenzbedingung der Variablenkontrolle besagt, dass das Bild einer gebundenen Variablen nicht frei vorkommen darf im Bild einer Metavariablen.



(a) SFSum



(b) SFSum abgerollt



(c) z^{-1} eingeführt

Abbildung 5.17: Beispieltransformation eines DAGs

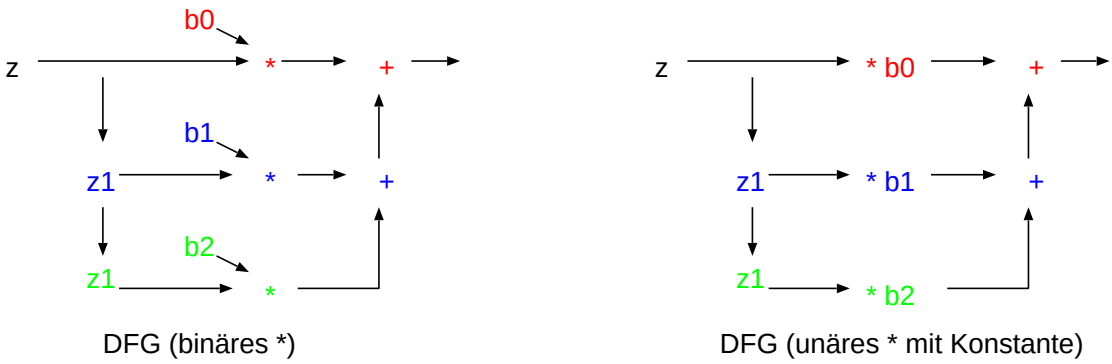
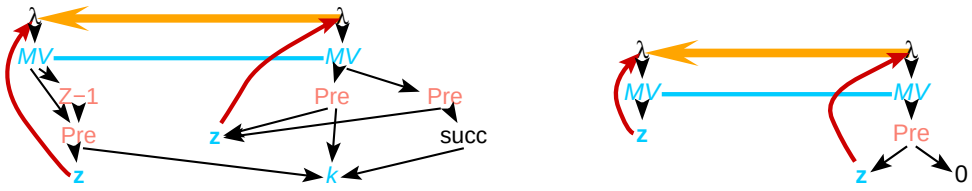


Abbildung 5.18: Beispieltransformation in Datenflussform

(a) Rekursion (b) Rekursionsabbruch

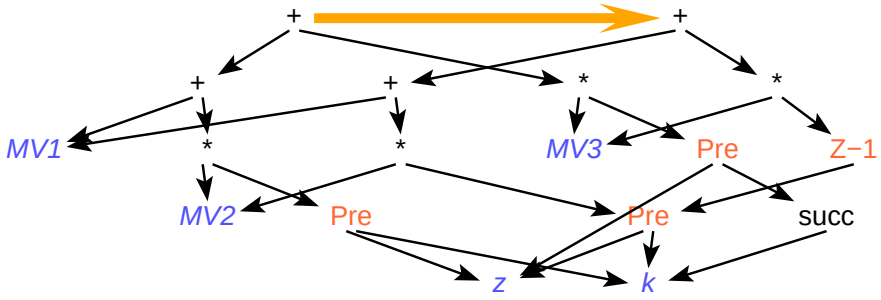
Abbildung 5.19: Direkte Transformationsregeln für Pre



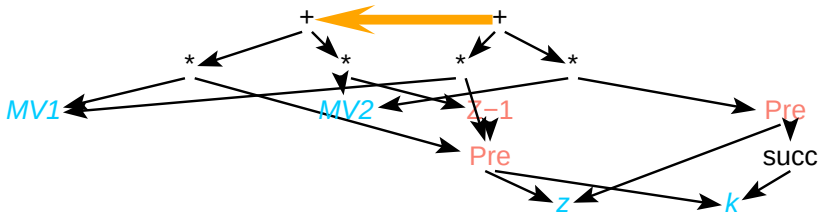
(a) Rekursion (b) Rekursionsabbruch

Abbildung 5.20: Mehrdeutige Transformationsregeln für Pre

dazu den Beispiel-DAG aus Abbildung 5.22, in dem wir nach Anwendung der Regel (Abbildung 5.21(b)) zwei unterschiedliche Knoten mit ‘Pre z …’ erhalten haben, anstatt dass sich ein Knoten Z^{-1} auf den anderen Knoten Z^{-1} abstützt, um Berechnungen wieder verwenden zu können.



(a) Rekursion

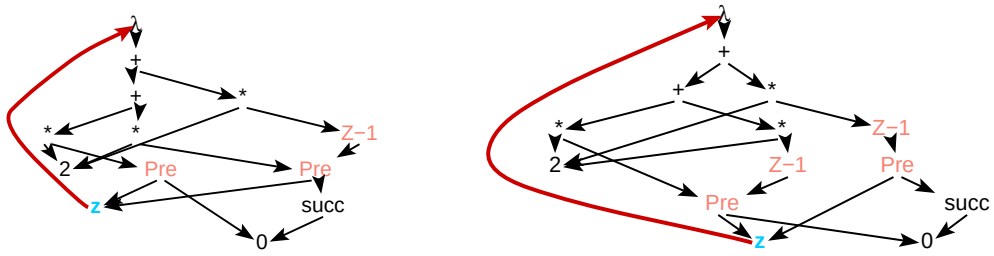


(b) Rekursionsabbruch

Abbildung 5.21: Eindeutige Transformationsregeln für Pre

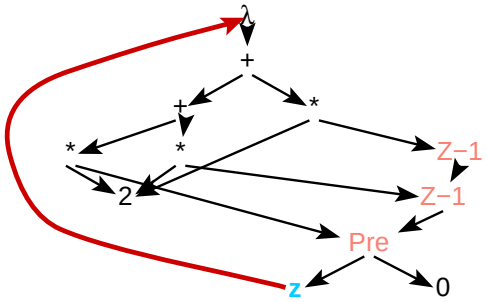
Eine Lösung zu der oben beschriebenen Problematik ist eine Flexibilisierung des Regelmechanismus. Lassen wir Regeln mit mehreren Spitzen zu, so erreichen wir, dass das Auftrennen des Sharings verhindert wird. Eine mehrspitzige Regel besagt, dass alle einlaufenden Kanten einer Regelseite auf die korrespondierende Regelseite übertragen werden. So erzielen wir unser gewünschtes Verhalten, wenn wir in Regel Abbildung 5.21(a) eine zweite Spitze einführen, die ‘Pre z $k+1$’ auf ‘Z^{-1} …’ überführt (siehe Abbildung 5.23).

Wenden wir nun den Lösungsvorschlag an, so gelangen wir zu dem bereits gezeigten DAG in Abbildung 5.17(c).



(a) vorher

(b) nachher



(c) gewünscht

Abbildung 5.22: Auftrennen des Sharings

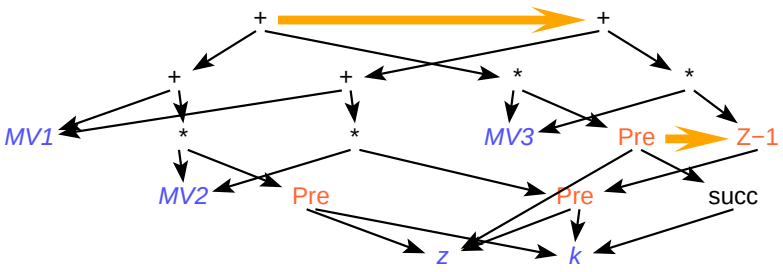


Abbildung 5.23: Eindeutige HOPS-Regel für Pre (mehrspitzig)

Hintereinderschaltung von Systemfunktionen

Teil-Systemfunktionen, die miteinander multipliziert werden, können wir als unabhängige Systemfunktionen behandeln, wenn wir sie hintereinanderschalten. Diese Idee ist in der Regel aus Abbildung 5.24[9] zu finden, die eine Multiplikation der Systemfunktion (SFMult) auf eine simple Hintereinanderschaltung von Funktionen (Komposition, $\fatsemi$) abbildet.

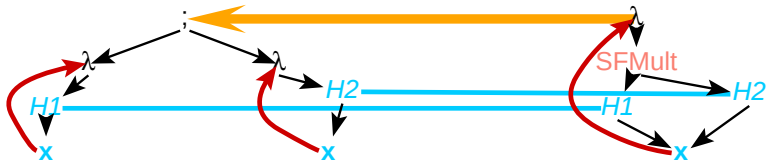


Abbildung 5.24: SFMult abgestützt auf Composition (';')

Systemfunktion mit Rückkopplung

Einen ähnlichen Weg können wir nun für die zweite Teilfunktion H_2

$$H_2(z) = \frac{1}{1 - \sum_{k=1}^{N} a_k z^{-k}} \tag{5.32}$$

gehen. Deren besonderes Merkmal ist ja, dass sie auf früher errechnete Ergebnisse zurückgreift, während sich die Teilfunktion H_1 lediglich auf frühere (unveränderte) Eingangswerte bezieht.

Wenn wir einen Blick auf das Blockdiagramm aus Abbildung 5.5 und 5.6 werfen, so sehen wir rückgerichtete Kanten, also Zyklen. DAGs in HOPS sind per definitionem azyklisch. Wollen wir uns dennoch dieses System nutzbar machen, müssen wir auf eine Technik zurückgreifen, die ein zyklisches Netzwerk als DAG modelliert[10]. Dazu ist ein Verfahren aus dem λ-Kalkül bekannt. Dort wird ein (bindender) Y-Kombinator eingeführt, der eine Rekursion über eine Funktion definiert.

Übertragen wir diesen Sachverhalt sinngemäß auf das Blockdiagramm, so können wir zwei Schemata (Abbildung 5.25) unterscheiden, die zur Auflösung der Zyklizität führen. Die beiden Formen der Implementierung unterscheiden sich durch die Art, an welchen Stellen wir Kanten auftrennen, um die Kreisfreiheit zu gewährleisten.

[9] Der Graph der linken Seite hat eine Typisierung von $\lambda :: \alpha \to \mathbb{Z}$, während der Graph der rechten Seite den Typ $\lambda :: \mathbb{Z} \to \mathbb{Z}$ aufweist. Folglich ist die Regel nur auf einen Graphen mit der Typisierung der rechten (schärferen) Regelseite anwendbar, da die Regel andernfalls die Typisierung verändern würde.

[10] Die Abbildung eines zyklischen Netzwerks auf einen azyklischen Graphen hat einen rein technischen Hintergrund, der durch eine Designentscheidung in HOPS bedingt ist, keine Zyklen zuzulassen. Ein flexiblerer Graphenbegriff würde das folgende Verfahren nicht notwendig machen.

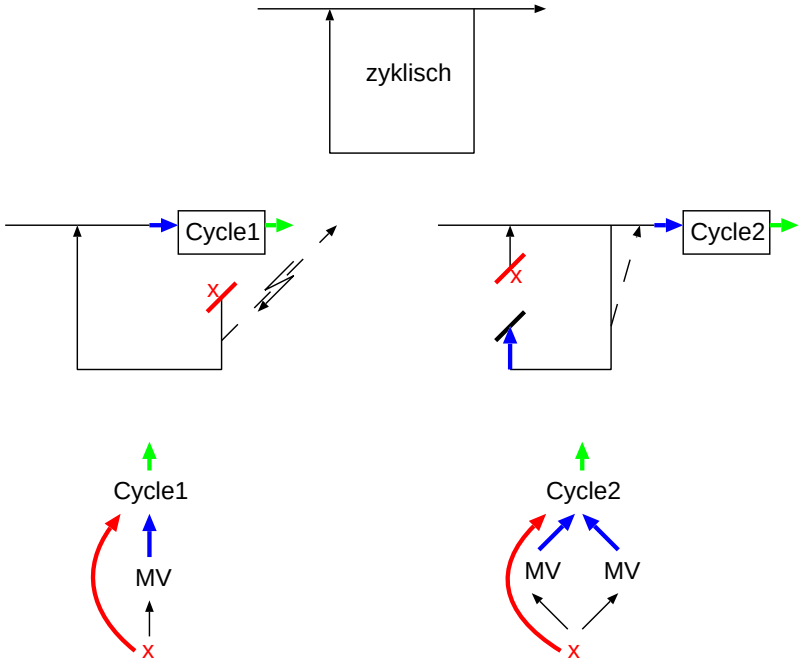


Abbildung 5.25: Zwei verschiedene DAG-Implementierungen eines Zyklus

Es ist offensichtlich, dass sich auf dem Zyklus, d. h. auf den rückführenden Kanten, verzögernde Elemente (Speicherelemente, Z^{-1}) befinden müssen, um als Ganzes eine definierte Funktion (ohne „Kurzschluss“) zu erlangen.

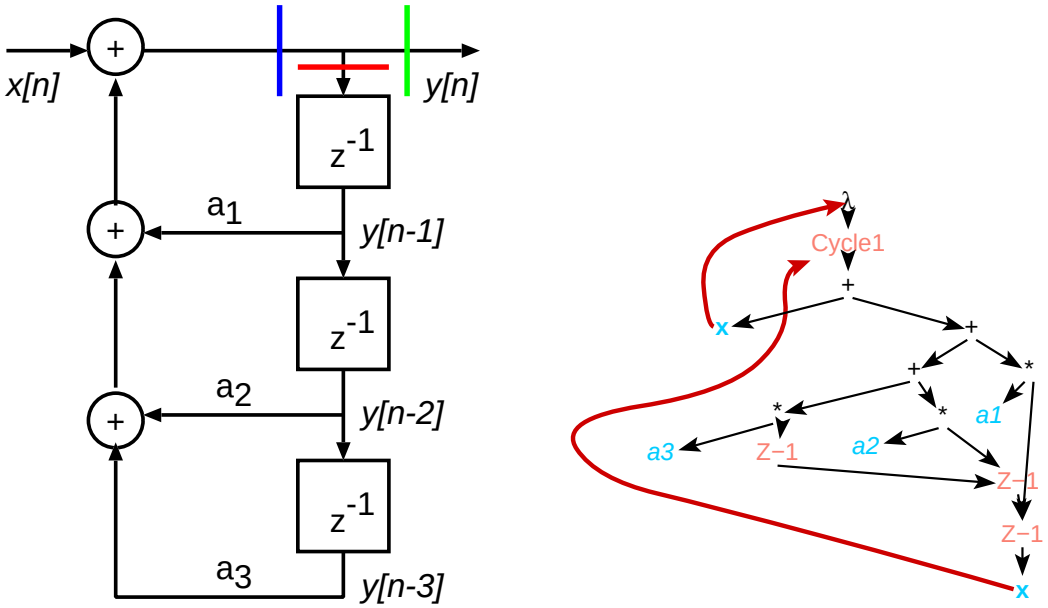
Baustein mit einem Nachfolger (Cycle1)

Eine Lösung des Problems besteht darin, die Kreisfreiheit mittels Durchschneiden des **Anfangs** der zyklenbildenden Kante herzustellen (Abbildung 5.26(a)).

Dazu abstrahieren wir das eine Ende mittels einer gebundenen Variable (x). Das gegenüberliegende Ende müssen wir nicht gesondert als Nachfolger des Cycle1-Bausteins aufhängen, da hier keine weiteren Bausteine enthalten sind. Als Ergebnis gelangen wir zu einem Baustein mit nur einem Nachfolger (Abbildung 5.26(b)).

Vorbedingung für die Anwendbarkeit dieses Verfahrens ist, dass die abstrahierte Variable von ihrem Binder dominiert wird. Dies ist jedoch nur gewährleistet, wenn auslaufende Kanten auf dem Zyklus nicht Vorwärtskanten darstellen, die hinter (hier: rechts) dem Cycle1-Baustein enden.

Erschwerend kommt hinzu, dass wir unseren Baustein nur an einer Verzweigung mit identischem Datenfluss, also ohne Operator, einfügen können. Da diese Einschränkung nicht von allen Datenflussgraphen erfüllt wird, gehen wir über zu einem etwas allgemeineren Baustein mit zwei Nachfolgern, der diese Restriktion nicht aufweist.



(a) Azyklisches Blockdiagramm (b) DAG

Abbildung 5.26: Kreisfreiheit mit Cycle1

Baustein mit zwei Nachfolgern (Cycle2)

Um beliebige Blockdiagramme zu implementieren, führen wir einen flexibleren Baustein (Cycle2) ein, mit dessen Hilfe beliebige, kreisbehaftete Netze abgebildet werden können. Hier wird der Zyklus dadurch unterbunden, dass wir die kreiserzeugende Kante an dessen **Ende** schneiden (Abbildung 5.27(a)). Dadurch gewinnen wir zwei Enden, mit denen wir gesondert verfahren. Das eine Ende abstrahieren wir mit einer gebundenen Variablen, das andere Ende 'hängen' wir an unseren neuen Baustein. Den Cycle2-Baustein selbst fügen wir in diejenige Kante ein, die den — jetzt entfernten — Kreis unmittelbar dominiert hat, wodurch wir einen weiteren Nachfolger unseres neu definierten zweistelligen Bausteins bekommen (Abbildung 5.27(b)).

Vorbedingung für die Anwendbarkeit dieses Verfahrens ist wiederum, dass die abstrahierte Variable von ihrem Binder dominiert wird. Dies erreichen wir dadurch, dass wir den Binder so weit (im DFG nach rechts) verschieben, dass Variablendominierung herrscht, was jederzeit möglich ist. Eine hinreichende, aber nicht notwendige Bedingung ist diejenige, dass keine Kante unter dem Cycle2-Baustein weiter nach vorne greift, der Baustein also **alle** Knoten unter ihm dominiert.

Nachdem wir in der Lage sind, Zyklen auf DAGs abzubilden, können wir die Teil-Systemfunktion H_2 auf einen Zyklenbaustein mit der Regel aus Abbildung 5.28[11] stützen. Durch das Abwälzen von SFSumInv auf SFSum können wir die Vorarbeit, die wir für letzteren Baustein bereits geleistet haben, auch hier anwenden.

Kommutativität der Systemfunktionen

Aus der Theorie der Signalverarbeitung wissen wir, dass die Multiplikation (SFMult) von Systemfunktionen kommutativ ist. Diese Tatsache können wir bei der Entwicklung einer Implementierung ausnutzen, um eine effiziente Version aus einem Schaltbild abzuleiten.

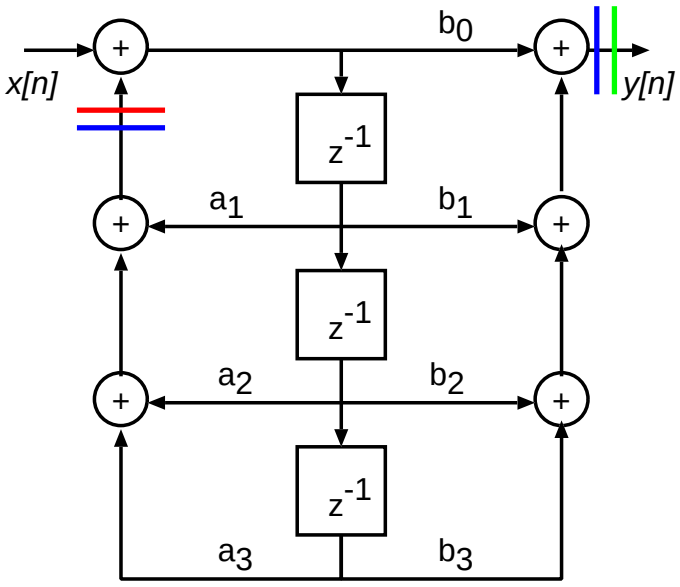
Daraus lässt sich mit Hilfe der bisherigen Transformationen eine direkte Form II (siehe Abbildung 5.6(a)) generieren, die aber noch redundante Speicherelemente aufweist. Offen bleibt vorerst die Frage, wie wir ein Sharing dieser Elemente erreichen (siehe Abbildung 5.6(b)).

5.4.2 Direktes System

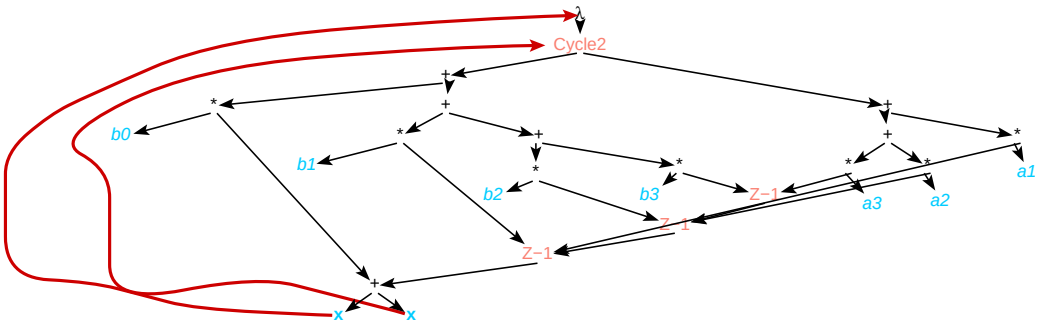
Nachdem wir in den vorangegangenen Abschnitten vorbereitende Maßnahmen getroffen haben, wollen wir nun beginnend mit der Spezifikation der allgemeinen linearen Differenzengleichung für bestimmte Koeffizienten N und M einen DAG derart transformieren, dass hieraus direkt Code in verschiedenen Programmiersprachen gewonnen werden kann.

Wir starten mit der Differenzengleichung in Graphnotation für $M = N = 3$ (Abbildung 5.29) und erreichen über mehrere Zwischenschritte den Ergebnisgraph aus Abbildung 5.30. Hier sehen wir die Programmgraph-Repräsentation über der Datenfluss-Notation, wie wir sie aus den Blockdiagrammen kennen.

[11] Beachte, dass der Index von SFSumInv nun bei 1 beginnt.



(a) Azyklisches Blockdiagramm



(b) DAG

Abbildung 5.27: Kreisfreiheit mit Cycle2

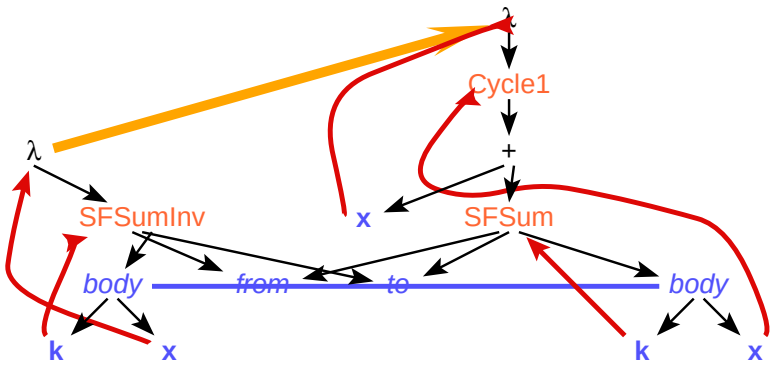


Abbildung 5.28: SFSumInv abgestützt auf SFSum

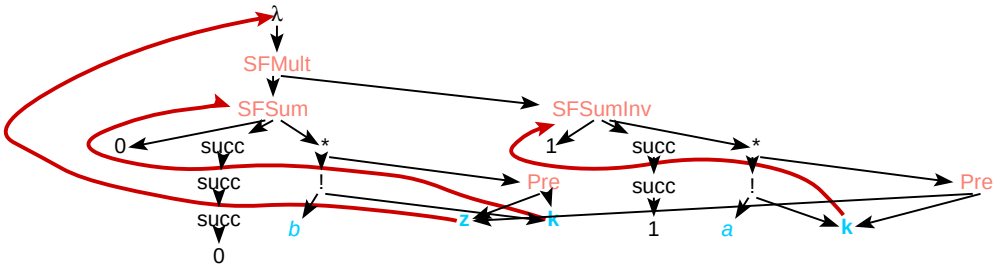
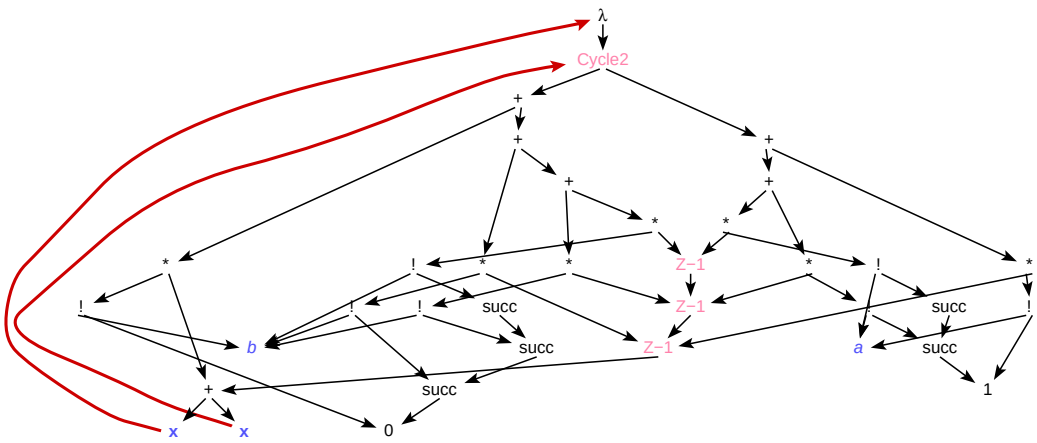
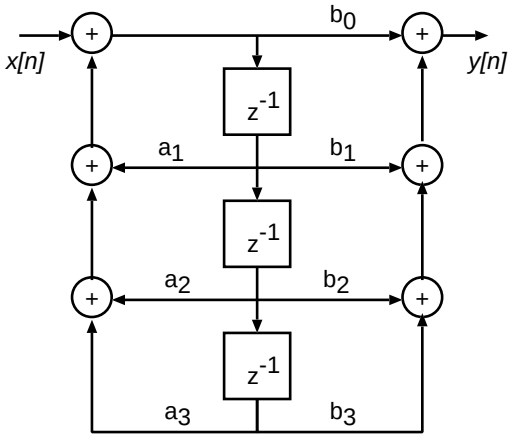


Abbildung 5.29: Ausgangsgraph der Transformationssequenz für die direkte Form



(a) DAG



(b) Blockdiagramm

Abbildung 5.30: Ergebnisgraph der Transformationssequenz für die direkte Form

Der interessierte Leser kann einige Zwischenschritte dieser Transformationssequenz im Anhang (Abschnitt A.1) nachschlagen. Dort wird auch erläutert, an welchen Anwendungsstellen die einzelnen Transformationsregeln greifen.

5.4.3 Kaskadiertes System

Die kaskadierte Form der Implementierung können wir modular auf die direkte Form (I oder II) abstützen, indem wir das Produkt abrollen. Die einzelnen Faktoren werden dann durch Terme der direkten Form zweiter Ordnung dargestellt.

$$H(z) = \frac{\sum_{k=0}^{6} b_k z^{-k}}{1 - \sum_{k=1}^{6} a_k z^{-k}} \qquad \text{für } M = N = 6 \tag{5.33}$$

$$\Rightarrow A \frac{\prod_{k=1}^{3}(1 - h_k z^{-1})(1 - h_k^* z^{-1})}{\prod_{k=1}^{3}(1 - d_k z^{-1})(1 - d_k^* z^{-1})} \tag{5.34}$$

$$\Rightarrow \prod_{k=1}^{3} \frac{\sum_{i=0}^{2} b_{ik} z^{-i}}{1 - \sum_{i=1}^{2} a_{ik} z^{-i}} \qquad \text{für } N_s = \lfloor (6+1)/2 \rfloor = 3 \tag{5.35}$$

$$\Rightarrow \prod_{k=1}^{3} \frac{b_{0k} + b_{1k} z^{-1} + b_{2k} z^{-2}}{1 - a_{1k} z^{-1} - a_{2k} z^{-2}} \qquad \text{für } N_s = \lfloor (6+1)/2 \rfloor = 3 \tag{5.36}$$

So sehen wir in Abbildung 5.31 das Ausgangssystem für Gleichung (5.35) in der kaskadierten Form. Dieses System können wir mit unseren bereits bekannten Regeln für die direkte Form I und II implementieren.

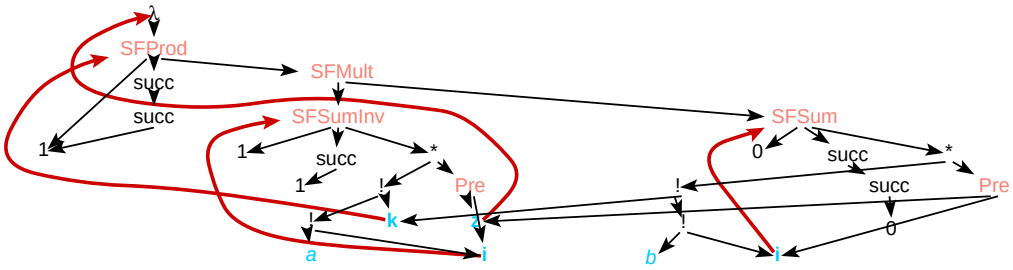


Abbildung 5.31: Ausgangsgraph der Transformationssequenz für die kaskadierte Form

Über eine Reihe von Transformationsschritten erreichen wir einen Graphen, aus dem wiederum direkt Code erzeugt werden kann. Der interessierte Leser kann die Zwischenschritte und den Ergebnisgraphen wiederum im Anhang (Abschnitt A.2) nachschauen.

5.4.4 Paralleles System

Das parallele System können wir mit bereits bekannten Mitteln entwickeln, indem wir die Summe analog zum kaskadierten System abrollen.

Als Beispiel werfen wir einen Blick auf die Systemfunktion

$$H(z) = \frac{1+2z^{-1}+z^{-2}}{1-0,75z^{-1}+0,125z^{-2}} \tag{5.37}$$

$$= 8 + \frac{-7+8z^{-1}}{1-0,75z^{-1}+0,125z^{-2}} \tag{5.38}$$

Dazu gehen wir von Gleichung (5.37) aus und entwickeln zunächst den zugehörigen HOPS-Graphen (Abbildung 5.32).

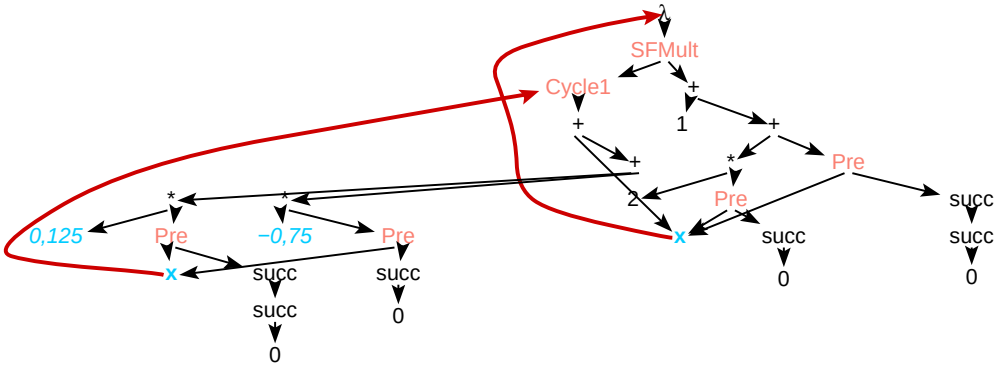


Abbildung 5.32: Ausgangsgraph der Transformationssequenz für die parallele Form

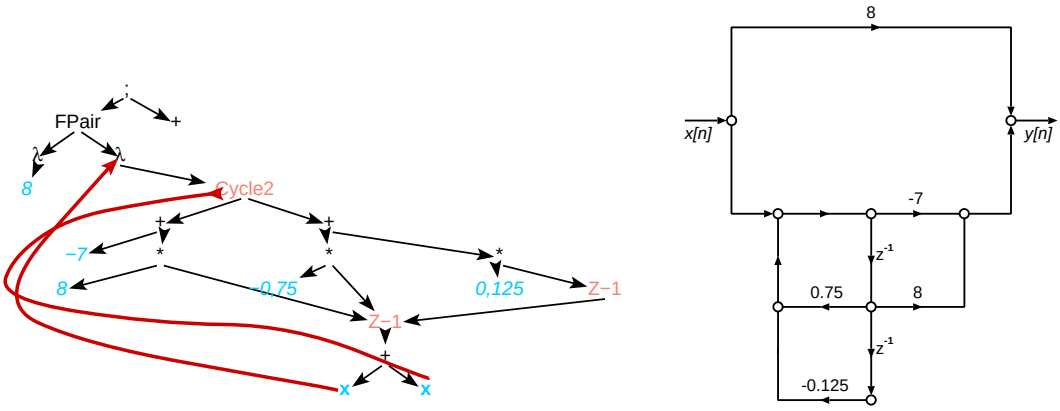
Nun können wir aus dem HOPS-Graphen aus Abbildung 5.32 den modifizierten Graphen aus Abbildung 5.33 ableiten, der durch eine entsprechende Partialbrucherweiterung zwei Summanden aufweist, die voneinander unabhängig zu implementierende Funktionen repräsentieren.

Sind wir indes an einer Form interessiert, die ausschließlich Grundelemente erster Ordnung enthält, so transformieren wir unsere Systemfunktion aus Gleichung (5.38) dementsprechend und gelangen zu der veränderten, jedoch semantisch äquivalenten Form

$$H(z) = 8 + \frac{18}{1-0,5z^{-1}} - \frac{25}{1-0,25z^{-1}} \tag{5.39}$$

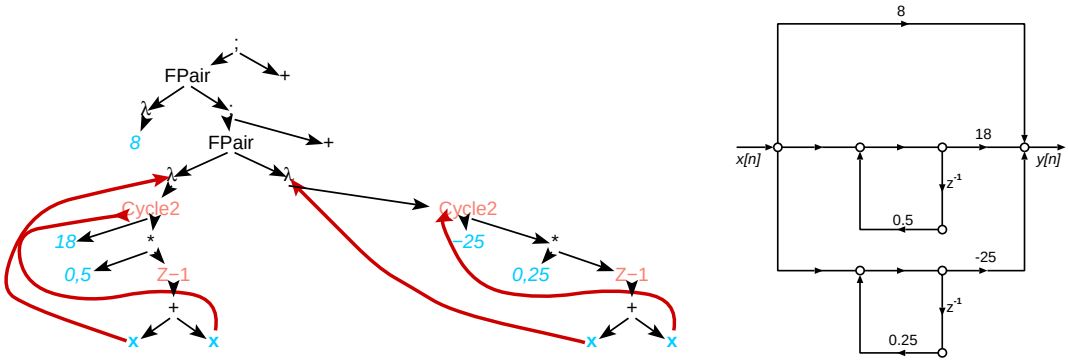
Der dazugehörige HOPS-Graph mit seinem Signalflussgraphen können wir in Abbildung 5.34 erkennen.

Dieser Abschnitt zeigt, wie wir in modularer Weise ausgehend von einer abstrakten Systemfunktion direkt zu einer Implementierung gelangen und uns immer wieder auf bereits bekannte Grundformen (direkte Form II) abstützen können.



(a) DAG (b) Signalflussgraph

Abbildung 5.33: Ergebnisgraph der Transformationssequenz für die parallele Form mittels eines Systems zweiter Ordnung



(a) DAG (b) Signalflussgraph

Abbildung 5.34: Ergebnisgraph der Transformationssequenz für die parallele Form mittels Systeme erster Ordnung

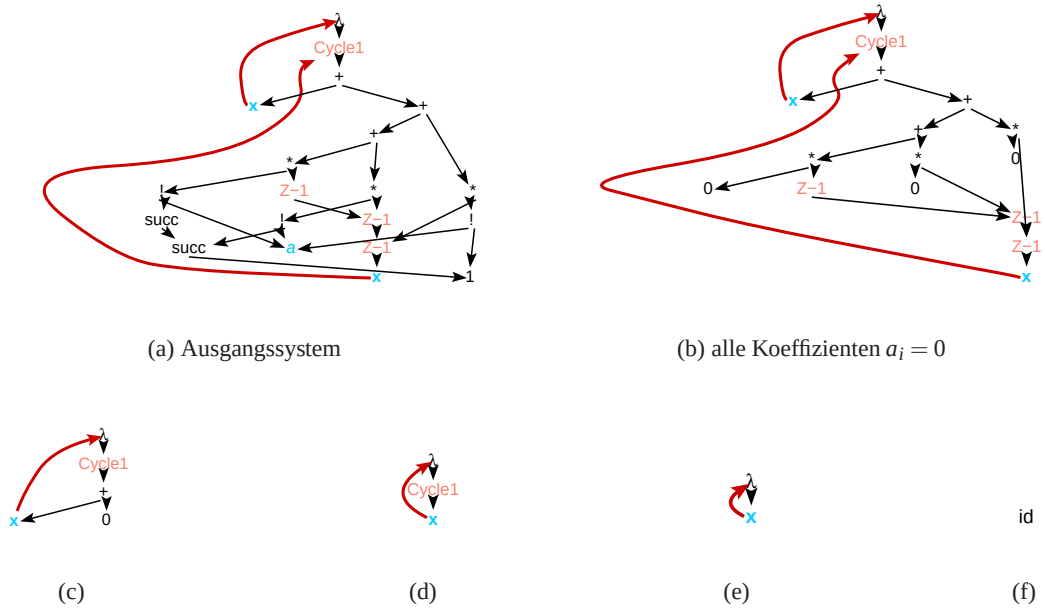
5.4.5 FIR-System

FIR-Systeme stellen einen Spezialfall der allgemeineren IIR-Systeme dar, da sie keine Zyklen besitzen.

Verfolgen wir den Ansatz, dass wir zunächst mit der allgemeinen Form der Systemfunktion starten, muss man erwarten können, dass sich die daraus entwickelte Implementierung an manchen Stellen bedeutend vereinfacht. Wenn unser Regelwerk wohldefiniert ist, so muss sich der Teilterm

$$H_2(z) = \frac{1}{1 - \sum_{k=1}^{N} a_k z^{-k}}$$

auflösen, in dem Termgraphen also zu der Identitätsfunktion vereinfachen. Wir wollen unser Verfahren nun in der Transformationsfolge (Abbildung 5.35) auf die Probe stellen.



(a) Ausgangssystem (b) alle Koeffizienten $a_i = 0$

(c) (d) (e) (f)

Abbildung 5.35: Transformationssequenz zur Vereinfachung eines FIR-Systems

Zu Beginn der Transformationsfolge (Abbildung 5.35(a)) sehen wir das Grundmuster, welches die Rückwärtskanten repräsentiert. Dadurch, dass sämtliche Koeffizienten $a_i = 0$ sind, wird der gesamte rechte Zweig der Summanden zu 0 (Abbildung 5.35(c)), wodurch die Addition ebenfalls wegfällt (Abbildung 5.35(d)). Da der Cycle1-Baustein keine gebundene Variable mehr enthält (Abbildung 5.35(e)), entfällt selbiger, weil er somit keinen Zyklus repräsentiert. Letztendlich kommen wir zu der Identitätsfunktion id (Abbildung 5.35(f)), welche sich neutral bzgl.

der Komposition verhält — wir erinnern uns, dass wir das System der Vorwärtskanten mit dem System der Rückwärtskanten komponiert, also hintereinander geschaltet haben — und hiermit ebenfalls herausfällt.

Galt das Vorstehende für ein System mit direkter Form, so können wir das FIR-System dementsprechend auch auf ein kaskadiertes bzw. paralleles System übertragen. Auch hier fallen die Rückwärtskanten heraus, und wir erhalten ein vereinfachtes kaskadiertes oder paralleles FIR-System.

Somit können wir innerhalb unseres Regelkalküls Vereinfachungen durchführen und dadurch spezielle, einfachere Strukturen ableiten, wie wir anhand eines FIR-Systems gezeigt haben.

FIR-Systeme mit linearer Phase

Wir haben im vorigen Abschnitt die Eigenschaften eines FIR-Systems mit linearer Phase kennengelernt. Die Besonderheit eines solchen Systems, dass sich Multiplikationsbausteine mehrfach verwenden lassen, wollen wir bei der Implementierung nutzen. Dazu stehen uns zwei Möglichkeiten offen.

Zum einen können wir die zusammengefasste Systemfunktion aus Gleichung (5.30) expandieren. Hier haben wir die Vorarbeit bereits außerhalb unseres Kalküls geleistet. Andererseits können wir von einer allgemeinen Systemfunktion ausgehen und unseren Transformationsmechanismus dazu nutzen. Dazu werden in der bereits expandierten Fassung Teile mittels des Distributivgesetzes wieder zusammen gefasst. Den zweiten Weg schildert die Transformationsfolge aus Abbildung 5.36 für den Fall, dass $M = 2$ ist.

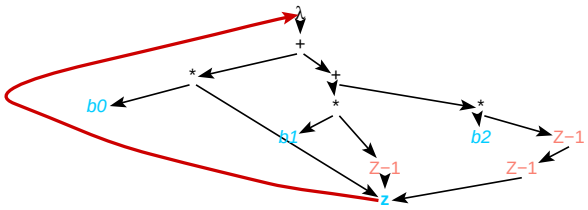
Dieses einfache Beispiel zeigt, dass selbst allgemeine Gesetze aus der Mathematik wie z. B. Kommutativität und Distributivität, die uns in HOPS innerhalb eines mathematischen Moduls zur Verfügung gestellt werden, zur Effizienzsteigerung (hier: weniger Multiplikationen) genutzt werden können.

5.5 Codierung in verschiedenen Programmiersprachen

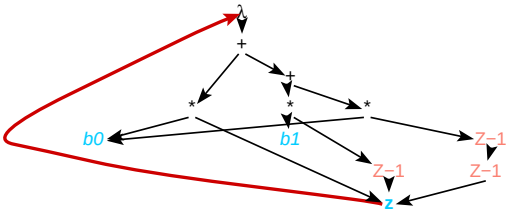
Wir haben nun die notwendigen Vorarbeiten geleistet und eine Differenzengleichung mit HOPS in eine Form gebracht, so dass der eingebaute Generierungsmechanismus funktionalen Code emittieren und ein Compiler eine lauffähige Anwendung erzeugen kann. Es sei darauf hingewiesen, dass die abgedruckten Programme mit den eingebauten Durchlaufmechanismen vollautomatisch aus HOPS generiert werden können.

Der generierten funktionalen Version stellen wir eine imperative Fassung in der Programmiersprache C, die aus [Oppenheim und Schafer 1992] übernommen wurde. Hier handelt es sich nicht um ein Programm, das mittels eines Kalküls aus einer Differenzengleichung erzielt wurde. Vielmehr ist es durch eine ad-hoc–Programmierung entstanden, die nicht auf einem formalen Modell basiert.

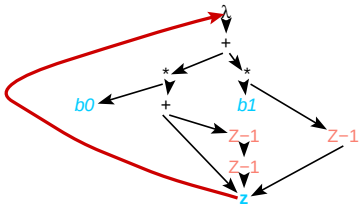
Natürlich kann man davon ausgehen, dass für bestimmte Anwendungsfälle geeignete Bibliotheken existieren. Die Besonderheit unseres Ansatzes eines formal abgeleiteten Programms liegt



(a) FIR-System



(b) FIR-System mit linearer Phase



(c) vereinfacht

Abbildung 5.36: Transformationssequenz zur Vereinfachung eines FIR-Systems mit linearer Phase

darin, in einer frühen Phase des Prototypings schnell korrekte, ausführbare Programme aus einer Differenzengleichung zu erstellen. Sollte sich der generierte Code immer noch als zu langsam erweisen, kann der Anwender darüber nachdenken, eine handoptimierte Version zu erstellen. Doch erscheint uns diese Angelegenheit ähnlich gelagert zu sein wie vor vielen Jahren die Diskussion Assembler versus C oder heutzutage C versus Java. Schnellere Prozessoren machen diese Diskussion über kurz oder lang obsolet.

5.5.1 Generierung in eine funktionale Programmiersprache

Funktionale Sprachen erleben dank schneller Rechner und nicht zuletzt Fortschritten bei der Optimierung in der Übersetzungstechnik einen neuen Aufschwung. Eine ihrer Domänen liegt in jenen Bereichen, in denen bereits eine mathematische Beschreibung einer Anforderung vorliegt. Von hier aus ist der Weg zu einer funktionalen Implementierung nicht mehr weit.

In unserem Kalkül transformieren wir eine abstrakte, mathematische Spezifikation sukzessive in eine implementierungsnähere Beschreibung, die stark an den λ-Kalkül angelehnt ist. Da liegt es nahe, ein ausführbares Programm durch ein Abstützen auf eine funktionale Sprache zu gewinnen. Im Folgenden untersuchen wir, welche Wege sich bei der Codeerzeugung anbieten.

Wir haben als einen Vertreter der funktionalen, nicht-strikten Programmiersprachengattung Haskell gewählt. Ebenso könnten wir aber auch eine andere nicht-strikte Programmiersprache wie Miranda oder LML[12] nehmen.

Wir stellen nun drei unterschiedliche Implementierungen vor:

- Direkte Implementierung
- Matrixmultiplikation
- Graphabbildung

Die direkte Implementierung basiert auf einer direkten Übersetzung der elektrotechnischen Funktionen ohne jegliche Optimierung. Die Matrixmultiplikation greift eine Technik auf, in der mathematische Matrizen multipliziert werden. Erst bei der Graphabbildung kommt das Transformationsverfahren zum Einsatz, das wir in Abschnitt 5.4 vorbereitet haben. Hier haben wir ja mittels eines mathematischen Kalküls eine Differenzengleichung in ein Blockdiagramm überführt, das in der HOPS-Terminologie einem Datenflussgraphen entspricht. Daran schließt sich eine vollautomatische Codeemission an, die zu den abgebildeten Programmen führt.

Es wird sich im Rahmen eines abschließenden Leistungsvergleichs herausstellen, dass sich unsere teils aufwändigen Transformationen gelohnt haben. So erreicht eine Prozessnetzimplementierung, die auf einem Blockdiagramm basiert, gegenüber einer direkten Implementierung ohne transformative Vorarbeiten einen Geschwindigkeitszuwachs mit einem Faktor von mehr als 10^4. Dies zeigt das Potenzial einer transformativen Programmierung gegenüber Compileroptimierungen, die meist nur lokal operieren.

[12] Lazy ML

Direkte Implementierung

Dieser Abschnitt befasst sich mit direkten Implementierungen in der Hinsicht, dass wir eine Gleichung, die wir aus der Theorie gewonnen haben, nahezu unverändert in Code gießen.

Faltungssumme
In Gleichung (5.12) haben wir die Faltungssumme

$$y[n] = x[n] * h[n]$$

kennengelernt. Die Faltungssumme besteht aus punktweisen Multiplikationen von Funktionen, die um entsprechende Phasen verschoben sind. In Listing 5.6 sehen wir eine direkte Implementierung, bei der Multiplikationen auf Funktionen angewandt werden.

Listing 5.6: Faltungssumme (convolution)

```
-- Faltungssumme (im endlichen Bereich von 0..n)
convolution :: (ℤ →ℚ) →(ℤ →ℚ) →ℤ→ℚ
convolution f g = \n →functionAddFold (map
                        (\k →(f k) ‘ functionMult ‘ ( functionShift g k)) [0 .. n]) n

-- functionShift
-- verschiebt den Definitionsbereich einer Funktion f um n0 Einheiten nach rechts
functionShift :: (ℤ →ℚ) →ℤ→ℤ→ℚ
functionShift f n0 = \n →f (n−n0)

-- Multiplikation einer Funktionen mit einer Konstanten
functionMult :: ℚ→(x →ℚ) →x →ℚ
functionMult a f = \n →a ∗ (f n)

-- punktweise Addition einer Liste von Funktionen
functionAddFold :: [ℤ →ℚ] →ℤ→ℚ
functionAddFold funcs n = foldr (\f x →(f n) + x) 0 funcs
```

Es ist offensichtlich, dass dieser direkte Ansatz ein vollkommen unbefriedigendes Laufzeitverhalten zur Folge hat, da viele Werte mehrfach berechnet werden. Etwas effizienter wird es, wenn wir nicht mehr explizit Funktionen miteinander multiplizieren, sondern direkt Folgen verarbeiten wie in der Funktion convolution' (Listing 5.7) ersichtlich.

Listing 5.7: Faltungssumme (convolution')

```
-- Faltungssumme, punktweise definiert
-- berechne [f(0)..f(n)] und [g(0) .. g(n)]
```

```
-- berechne [f(0)*g(n)..f(n)*g(0)]
-- addiere die resultierende Folge
convolution' :: (ℤ →ℚ) →(ℤ →ℚ) →ℤ→ℚ
convolution' f g = \n →foldr (+) 0 ( zipWith (*) ( map f [0..n] ) ( reverse (map g [0..n] )))

-- Ist Faltungssumme kommutativ für Werte von 0..n?
convolutionCommTest f g n = and (map
                                  (\n0 →convolution' f g n0 == convolution' g f n0) [0..n])
```

Differenzengleichung

Aufbauend auf der Multiplikation von Funktionen (functionShift) implementieren wir nun die Differenzengleichung aus Gleichung (5.24)

$$y[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$$

Listing 5.8: Differenzengleichung

```
-- FIR Differenzengleichung (ohne Rückkopplung)
yFIRFun = \n →(b!0)*(functionShift xFun 0 n) +
                (b!1)*( functionShift xFun 1 n) + (b!2)*( functionShift xFun 2 n)

-- IIR Differenzengleichung (mit Rückkopplung)
yIIRFun = \n →
          (a!1)*( functionShift yIIRFun 1 n) + (a!2)*( functionShift yIIRFun 2 n)
           + (b!0)*( functionShift xFun 0 n) + (b!1)*( functionShift xFun 1 n)
           + (b!2)*( functionShift xFun 2 n)
```

Wie die Faltungssumme ist auch diese Version ist in hohem Maße ineffizient. Das erkennt man allein daran, dass yIIRFun eine Kaskadenrekursion aufweist. So wird die Berechnung jedes weiteren Funktionswertes yIIRFun(n) unverhältnismäßig aufwändiger.

Einen Einblick in die Platzkomplexität geben die beiden folgenden Testläufe, bei denen eine FIR-Funktion auf 100 Werte angewandt wird und eine IIR-Funktion auf nur 20 Werte Anwendung findet, da bei einer größeren Anzahl an Werten ein Fehler auf Grund mangelnden Speichers auftritt.

Listing 5.9: Testläufe von yFIRFun und yIIRFun

```
testFIR x = map yFIRFun x

testIIR x = map yIIRFun x
```

```
-- Funktionsergebnisse von y(0) .. y(19)/y(99)

-- Performance: test1
-- (299531 reductions, 381308 cells, 4 garbage collections)
test1 = testFIR [0..99]

-- Performance: test2
-- (15259315 reductions, 20780493 cells, 243 garbage collections)
test2 = testIIR [0..19]
```

Um die schlechte Performance der direkten Implementierung zu verbessern, werden wir in der Graphabbildung die effiziente und flexible Listenverarbeitung ausnutzen, mit deren Hilfe wir unsere bisherigen Funktionswerte so weit wie nötig hinterlegen, um Doppelberechnungen zu vermeiden. Hierbei fließen auch die Resultate aus den Abschnitten 5.3 und 5.4 ein, so dass wir zu äußerst effizienten Implementierungen gelangen.

Matrixmultiplikation

Einen weiteren Weg zur Implementierung wollen wir in diesem Abschnitt studieren. Hier führen wir das Problem der Differenzengleichung auf eine Matrixmultiplikation zurück.

Eine Matrixdarstellung einer Differenzengleichung legt in der ersten Zeile die Koeffizienten a_i ($1 \leq i \leq N$) und b_i ($0 \leq i \leq M$) ab. Darunter ist eine schwach besetzte Matrix mit Einsen, durch die die alten Werte der Register gespeichert werden. Die Antwort auf eine Multiplikation in Form eines Vektors ist ein neuer Ausgabewert mit einem neuen Zustand der Registerinhalte. Das Weiterschalten um einen Takt erfolgt dadurch, dass der Ergebnisvektor ‘oben’ mit einem neuen Eingabelement x_n erweitert wird und damit implizit der Wert an der Stelle y_{n-1} durch y_n usw. belegt wird.

Für $M = N = 3$ können wir folgende Matrix der Dimension 7×6 definieren

$$\begin{pmatrix} b_0 & a_1 & a_2 & a_3 & b_1 & b_2 & b_3 \\ & 1 & & & & & \\ & & 1 & & & & \\ 1 & & & & & & \\ & & & & 1 & & \\ & & & & & 1 & \end{pmatrix} \times \begin{pmatrix} x_n \\ y_{n-1} \\ y_{n-2} \\ y_{n-3} \\ x_{n-1} \\ x_{n-2} \\ x_{n-3} \end{pmatrix} = \begin{pmatrix} y_n \\ y_{n-1} \\ y_{n-2} \\ x_n \\ x_{n-1} \\ x_{n-2} \end{pmatrix}$$

die einen Zustand inklusive Eingabe (x_n) nimmt und einen neuen Zustand (in Form eines Vektors) errechnet, der um die Größe der Eingabe kürzer ist. Dabei kann der Eingabestrom beliebig

groß sein.

Um von einem Eingabestrom der Breite k (hier x_n, $k = 1$), einem Zustand der Größe l (Tupel von Speicherelementen, hier $l = 7$) zu einem neuen Zustand der Größe l zu gelangen, (vorausgesetzt, der Zustand beinhaltet den Ausgabestrom, hier y_n), muss der Eingabe- und Zustandsvektor der Größe $k+l$ mit der (Zustandsübergangs-)Matrix multipliziert werden.

Durch n-maliges Multiplizieren wird n Takte weitergeschaltet, wobei vor dem Multiplizieren der Zustandsvektor um den Eingabevektor erweitert wird.

Eine entsprechende Implementierung in Haskell zeigt Listing 5.10.

Listing 5.10: Matrixmultiplikation

```
import DSP_Basic
import Konstanten

-- beispielhafte Übergangsmatrix
bspBlockMatrix = [[b0,a1,a2,a3,b1,b2,b3],
                  [  0, 1, 0, 0, 0, 0, 0  ],
                  [  0, 0, 1, 0, 0, 0, 0  ],
                  [  1, 0, 0, 0, 0, 0, 0  ],
                  [  0, 0, 0, 0, 1, 0, 0  ],
                  [  0, 0, 0, 0, 0, 1, 0  ]]
matrixMult :: Num a => Matrix a →Vector a →Vector a
matrixMult matrix@(m:ms) vector
  | not(and((map (\row →length m == length row) ms))) =
      error "matrixMult: Matrix nicht rechteckig"
  | length m /= length vector =
      error ("matrixMult: Matrix (" ++
        show (length (head matrix)) ++ "x" ++
        show (length matrix) ++
        ") und Vektor (" ++
        show (length vector) ++ ") nicht kompatibel!")
  | otherwise = map (addmult vector) matrix

-- Eingabestrom der Breite 1
blockInput :: Matrix ℚ→Vector ℚ→Vector ℚ
blockInput matrix vector = blockInputH matrix vector initState
                  --- Initialzustand mit 0en
          where initState = replicate (length matrix) 0

--blockInputH :: Matrix ℤ→Vector ℤ→Vector ℤ→Vector ℤ
blockInputH matrix (x:xs) state = y : blockInputH matrix xs newState
```

```
        where newState@(y:ss) = matrix 'matrixMult' (x:state)
blockInputH _ [] _ = []

testIIR  x = blockInput  bspBlockMatrix x
```

Graphabbildung

Auf der Suche nach effizientem Code untersuchen wir nun die Möglichkeit, Listen einzusetzen.

Hier kommen die Verfahren zum Einsatz, die wir in den Absätzen 5.3 und 5.4 erarbeitet haben. Im Zuge von Transformationssequenzen erzielten wir einen kompakten Graphen aus den Komponenten Addition, Multiplikation und Verzögerung. Abbildung 5.37 dienst uns als Vorlage, um aus diesem Graphen Code zu generieren.

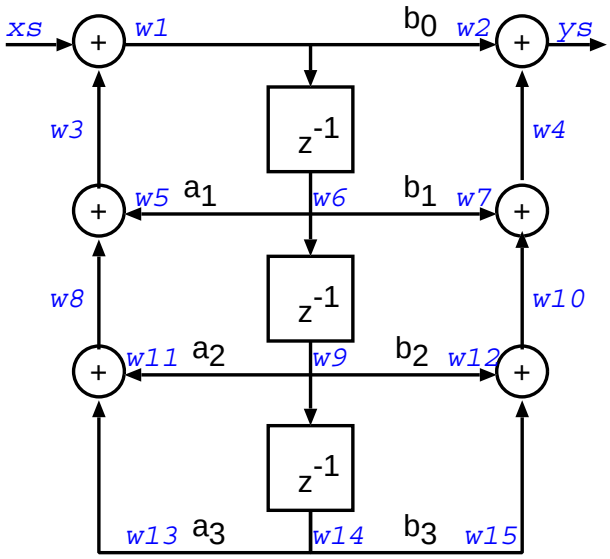


Abbildung 5.37: Blockdiagramm als Prozessnetz

Explizite Listen

Wir sehen in dem Blockdiagramm eine Funktion, die Listen verarbeitet. Nun kann man darauf direkt in Abhängigkeit des Eingangswertes einen Ausdruck generieren, der lc1 aus Listing 5.11 entspricht.

Listing 5.11: Differenzengleichung über explizite Liste definiert

```
> lc1 :: [ℚ] →[ℚ]
> lc1 xs = lc1 ' 0.0 0.0 0.0 xs
> lc1 ' _ _ _ [] = []
> lc1 ' z1 z2 z3 (x:xs) = r_1 : ( lc1 ' l_1 z1 z2 xs)
>     where r_1 =((+)
>                  ((∗) ( b!0) l_1)
>                 ((+)
>                   ((∗) ( b!1) z1)
>                   ((+)
>                     ((∗) ( b!2) z2)
>                     ((∗) ( b!3) z3)
>                   )
>                 )
>               )
>         l_1 = (+) x l_2
>         l_2 = ((+)
>                  ((+)
>                   ((∗) ( a!3) z3)
>                    ((∗) ( a!2) z2)
>                  )
>                   ((∗) ( a!1) z1)
>                 )

———— Performance: lc1 [1..100]
———— [1.0,4.75,11.4375,19.9844,29.5586,39.6709,50.0583,60.5849,71.1814 ...] :: [ℤ]
———— (26847 reductions, 36243 cells)
```

Implizite Listen (Prozessnetz)

Ein weitaus eleganterer Weg besteht darin, Abbildung 5.37 als *Prozessnetz* aufzufassen. Synchrone Prozessnetze (vgl. u. a. [Thiemann 1994]) können zur Beschreibung der Verarbeitung gegenseitig verschränkter, unendlicher Listen[13] herangezogen werden. Nachdem ein Signalflussgraph resp. Blockdiagramm ein bestimmtes Prozessnetz formiert, können wir diese Ergebnisse heranziehen und die Gleichungen, die ein Signalflussgraph induziert, direkt in Haskell implementieren.

Der folgende Ansatz, Verbindungen durch Lazy-Listen zu implementieren, erfordert eine Programmiersprache, die das Konzept der Lazy-Eigenschaft unterstützt, wie dies z. B. Haskell bereitstellt.

[13] Unendliche Listen im Sinne von beliebig langen Listen

So bilden wir Elemente des Signalflussgraphen auf Haskell folgendermaßen ab:

Graphelement	Kombinator	
$+$	wireAdd	Summieren zweier Datenströme
$\xrightarrow{a}$	wireMult	Skalieren eines Datenstroms mit konstantem Faktor
z^{-1}	wireReg	Verzögerungselement mit dem Ruhezustand 0

Dann können wir daraus den Code in Listing 5.12 generieren.

Listing 5.12: Differenzengleichung als Prozessnetz definiert

```
> lc2 :: [ℚ] →[ℚ]
> lc2 xs = ys
>  where ys = wireAdd w2 w4
>        w1 = wireAdd xs w3
>        w2 = wireMult (b!0) w1
>        w3 = wireAdd w8 w5
>        w4 = wireAdd w7 w10
>        w5 = wireMult (a!1) w6
>        w6 = wireReg w1
>        w7 = wireMult (b!1) w6
>        w8 = wireAdd w13 w11
>        w9 = wireReg w6
>        w10 = wireAdd w12 w15
>        w11 = wireMult (a!2) w9
>        w12 = wireMult (b!2) w9
>        w13 = wireMult (a!3) w14
>        w14 = wireReg w9
>        w15 = wireMult (b!3) w14

---- Performance: lc2 [1..100]
---- [1.0,4.75,11.4375,19.9844,29.5586,39.6709,50.0583,60.5849,71.1814 ...] :: [ℤ]
---- (6679 reductions, 12746 cells)
```

Die Kombinatoren wireAdd, wireMult und wireReg haben in Haskell folgende Gestalt:

Listing 5.13: Kombinatoren für das Prozessnetzdefinition

```
> wireAdd :: [ℚ] →[ℚ] →[ℚ]
> wireAdd = zipWith (+)

> wireMult :: ℚ→[ℚ] →[ℚ]
> wireMult n = map (∗n)
```

```
> wireReg :: [ℚ] →[ℚ]
wireReg = (0.0 :)
```

An dieser Implementierung sticht besonders hervor, dass die einzelnen Leitungen w_n wie in einem Gleichungssystem direkt aus dem Blockdiagramm abzulesen sind. Neben dieser Eleganz erhalten wir obendrein eine hochperformante Lösung.

5.5.2 Implementierung in einer imperativen Programmiersprache

Dieser Abschnitt befasst sich mit der Implementierung in einer imperativen Programmiersprache. Wir führen an dieser Stelle eine Beispielimplementierung an, um einen Vergleich der unterschiedlichen Programmierkonzepte (funktional versus imperativ) an Hand konkreter Programmzeilen ziehen zu können. Hier haben wir beispielhaft auf Grund deren weiten Verbreitung die Sprache C ausgewählt. Ebenso ist aber jede andere imperative Sprache wie Pascal oder Ada denkbar. Auch objektorientierte Sprachen wie C++ oder Smalltalk führen nicht zu einer grundlegenden Änderung des Programmaufbaus.

Wählen wir die Programmiersprache C, welche zu der großen Familie der strikten, imperativen Sprachen gehört, können wir uns nicht mehr auf die Lazy-Eigenschaft stützen, die uns z. B. Haskell bietet und uns von der expliziten Angabe der Abhängigkeiten zwischen den Leitungen entbindet[14]. Vielmehr müssen wir die Taktung explizit implementieren in der Weise, dass für jedes Fortschreiten der Zeit die Funktion iir_filter_lin aufgerufen wird.

Listing 5.14: Funktion iir_filter_lin

```
/*****************************************************************

  iir_filter_lin  – Perform linear (not cascaded)
                    iir  filtering  sample by sample on floats

                  Modified by Arne Bayer

Requires array of  filter  coefficients  and pointer to  history .
Returns one output sample for each input sample.
2∗n + 1  filter  coefficients  in  the  order a_1 .. a_n, b_0 .. b_n

float   iir_filter_lin  ()

    float  input          new float  input  sample
```

[14] Der abgedruckte Code basiert auf einer Implementierung aus [Embree 1995]. Er wurde nicht aus HOPS generiert. Dies ist zwar prinzipiell möglich, würde aber bedeutend mehr transformativen Aufwand bedeuten.

```
    float *coef_a      pointer to n (1..n) filter coefficients a
    float *coef_b      pointer to n+1 (0..n) filter coefficients b
    int n              number of coefficients in filter
    float *history     history array pointer

Returns float value giving the current output.

**********************************************************************/

double iir_filter_lin (double input,double *coef_a,double *coef_b,int n,double *history)
{
    int i;
    double *hist_ptr,*hist1_ptr,*coef_ptr_a,*coef_ptr_b;
    double l_output, r_output;

    hist_ptr = history;
    hist1_ptr = hist_ptr;              /* use for history update */
    coef_ptr_a = coef_a + n - 1;       /* point to last coef_a */
    coef_ptr_b = coef_b + n;           /* point to last coef_b */
    /* form output accumulation */
    printf ("\n%f ", *hist_ptr);
    l_output = *hist_ptr * (*coef_ptr_a--);
    r_output = *hist_ptr++ * (*coef_ptr_b--);
    for(i = 1; i < n; i++) {
       printf ("%f ", *hist_ptr);
        *hist1_ptr++ = *hist_ptr;          /* update history array */
        l_output += (*hist_ptr) * (*coef_ptr_a--);
        r_output += (*hist_ptr++) * (*coef_ptr_b--);
    }
    l_output += input;                     /* input tap */
    r_output += l_output * (*coef_ptr_b);  /* input tap */
    *hist1_ptr = l_output;                 /* last history */
    printf (": ");
    return(r_output);
}
```

Des weiteren sind die Datenelemente, die auf den logischen Leitungen anliegen, explizit durch einen Zustand zu implementieren. Dazu führen wir ein Array ein, das bei jedem Aufruf der Funktion iir_filter_lin als Parameter übergeben wird.

Listing 5.15: Funktion iir-test

```
/*****************************************************************

Test für   iir_filter

*****************************************************************/

const int INPUT_LEN = 5000;

void main()
{
  int i;
  double signal_out,
    *x,
    // x    [] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 },
    coef_a [] = { 0.75, -0.125, 0 },
    coef_b  [] = { 1, 2, 1, 0 };

  /* history for the filters */
  static double hist  [] = { 0, 0, 0 };

  x = malloc(INPUT_LEN * sizeof(double));
  for(i = 0; i < INPUT_LEN; i++)
    x[i] = i+1;

  for(i = 0; i < INPUT_LEN; i++) {
    signal_out =   iir_filter_lin  (x[i], coef_a, coef_b, 3, hist);
    sendout(signal_out);
  }
  putchar('\n');
}
```

Wir sehen, dass eine Codierung in einer imperativen Programmiersprache etwas mehr Aufwand erfordert als in einer funktionalen Sprache und das ursprüngliche Modell des Blockdiagramms nicht mehr so leicht erkennbar ist. So müssen wir sowohl die Abhängigkeit der Leitungen modellieren als auch das Fortschalten der Zeit. Nicht zuletzt ist die Funktion auf Differenzengleichungen zugeschnitten, die einen linearen Filter beschreiben. Wenn wir allgemeinere Filter implementieren wollen, haben wir zusätzliche, eigens darauf zugeschnittene Funktionen zu definieren.

5.5.3 Leistungsvergleich der unterschiedlichen Implementierungen

Wir vergleichen nun die verschiedenen Programme, die durch unterschiedliche Mechanismen generiert wurden, in Bezug auf deren Ausführungsgeschwindigkeiten.

Angefangen bei einer direkten Implementierung haben wir unseren Programmgraphen sukzessive in eine effiziente Implementierung transformiert, die in einem Prozessnetz resultiert. Sieht man sich die Ausführungsgeschwindigkeiten an, so steht die Implementierung durch ein Prozessnetz einer Implementierung in C, die Teil einer spezialisierten Bibliothek sein könnte, kaum nach, obwohl das Prozessnetz in der Sprache Haskell mit automatischer Speicherverwaltung inklusive Garbage Collection codiert ist.

Mechanismus	*Zielsprache*	*Name*	*Reduktionen*	*Zeit (in s)*
Direkt	Haskell	testIIR	3.109.005.837	11.916,3
Matrixmultiplikation	Haskell	testIIR	4.674.989	26,9
Graph, explizite Listen	Haskell	lc1	1.315.047	7,6
Graph, Prozessnetz	Haskell	lc2	295.274	1,7
Bibliothek	C	iir-test	*nicht ermittelbar*	0,9

Tabelle 5.1: Ausführung der Programme für 5000 Zahlen

Die einzelnen Durchläufe wurden jeweils mit der Gleichung

$$y[n] = \sum_{k=1}^{N} a_k y[n-k] \; + \; \sum_{k=0}^{M} b_k x[n-k] \tag{5.40}$$

und den Koeffizienten

Listing 5.16: Koeffizienten

```
a = listArray  (1,3)  [0.75, −0.125, 0]
b = listArray  (0,3)  [ 1, 2, 1, 0 ]
```

gestartet. Der Eingabestrom

Listing 5.17: Eingabestrom

```
x = [1.. 5000]
```

ist eine aufsteigende Sequenz von 5000 Zahlen[15].

[15] Der Mechanismus ‘Direkt´ wurde aus Zeitgründen **nach 32 Zahlen abgebrochen**!

Die vorstehenden Zahlen belegen, wie durch eine Reihe von Transformationen, die auf Effizienz ausgerichtet sind, aus einem Programm, das ein inakzeptables, exponentielles Laufzeitverhalten aufweist und kaum mehr als 32 Zahlen berechnet, äußerst effizienter Code entwickelt werden kann, der einer direkten Implementierung in der Programmiersprache C sehr nahe kommt. Da ist es umso bemerkenswerter, dass unser generiertes Haskell-Programm per se korrekt ist, da es auf einem mathematischen Kalkül basiert. Die Korrektheit unseres C-Programms kann nur mit unverhältnismäßig großem Aufwand geprüft werden. Die Semantiken von imperativen Programmiersprachen, die zudem Zeigermechanismen benutzen, sind äußerst komplex. Dementsprechend sind deren Korrektheitsbeweise.

Moderne leistungsfähige Computer lassen unterschiedliche Ausführungsgeschwindigkeiten im Bereich von 20–50 % ohnehin in den Hintergrund treten. Ein Kernpunkt in der Softwareindustrie ist das Aufzeigen eines reibungslosen Entwicklungsprozesses. Solch ein Verfahren, das zunehmend an Bedeutung gewinnen wird, haben wir in diesem Kapitel aufgezeigt.

KAPITEL 6

Resümee und Ausblick

Im letzten Kapitel wollen wir die Arbeit bewerten und mit anderen Systemen vergleichen. Daraus ergeben sich eine Reihe von Erweiterungsvorschlägen für das HOPS-System. Zudem werfen wir einen Blick in die Zukunft bzgl. der Bedeutung von transformationsgestützten Ansätzen in der Software-Entwicklung.

Inhaltsverzeichnis

6.1 Einordnung, Bewertung, Vergleich

Kapitel 4 zeigte eine Programmiermethodik mit Kombinatoren auf und präsentierte eine allgemeine Übersetzung eines Ausdrucks in einen semantisch äquivalenten Ausdruck ohne gebundene Variable. Anschließend beschrieb Kapitel 5 eine durchgängige Ableitung von einer mathematischen Beschreibung eines DSP-Operators in ein ausführbares Programm, insbesondere unter der Verwendung von Kombinatoren.

Wir haben HOPS als Werkzeug zur Entwicklung von Filtern im Kontext der digitalen Signalverarbeitung eingesetzt. HOPS ist vor allem als generisches Rahmenwerk zu verstehen und soll keineswegs mit spezialisierten Tools zur Entwicklung von Programmen aus diesem Bereich

konkurrieren. Vielmehr dient das Beispiel aus dem genannten Gebiet als running example, um allgemein den Einsatz und die Mächtigkeit der transformativen Programmentwicklung unter Beweis zu stellen. Das Beispiel stellt vor allem den methodischen Ansatz in den Vordergrund und bedient sich eines adäquaten Anwendungsbereiches, der ebenso ein anderer sein könnte.

Das Beispiel demonstriert einen Software-Entwicklungsprozess, der homogene Übergänge von der Spezifikation eines Problems bis hin zu einem fertigen Programm erlaubt. Voraussetzung dieses Verfahrens ist eine formale, mathematische Beschreibung des Problems. Der Autor ist sich im Klaren darüber, dass es für ein System oftmals schwammige und unpräzise Vorgaben gibt, die mit formalen Modellen nicht bzw. nur mit unverhältnismäßig großem Aufwand beschreibbar sind. Dies gilt umso mehr, wenn Personen zwar das nötige fachliche Verständnis für ihre Thematik aufbringen, jedoch mit einer formalen und eindeutigen Beschreibung schnell überfordert sind. Dies setzt versierte Spezialisten voraus, die ein Bindeglied zwischen den fachlichen Anforderungen und den technischen Spezifikationen bilden und aus informellen Beschreibungen formale Modelle ableiten. Daher muss es ein Streben der Informatik sein, einfache und doch mächtige Beschreibungsmittel zu finden, die eine präzise Semantik besitzen und auch ohne einen ausgeprägten mathematischen Formalismus anwendbar sind. Die zur Problemerfassung zur Verfügung gestellten Konzepte sollten daher einerseits einen Wiedererkennungswert für den Spezialisten besitzen, andererseits sollten sie ihn zur nötigen formalen Strenge in der Ausdrucksweise hinführen und unterstützend begleiten.

In dem Kontext eines solchen Bindeglieds ist auch HOPS zu verstehen. Ist das System einmal für einen bestimmten Anwendungsbereich instanziert und mit einem Satz an problemspezifischen Regeln ausgerüstet, muss der Benutzer keine tiefgreifenden Kenntnisse über Graphgrammatiken und Graphersetzungssysteme besitzen. Er kann sich vielmehr auf die Domäne seines Problembereichs konzentrieren. In diesem Sinne wollen wir HOPS vor allem als Richtung weisendes Rahmenwerk verstanden wissen und nicht als ein fertiges, vermarktbares Produkt.

Zur Idee der transformativen Programmentwicklung im Kontext des Software Engineerings gibt es natürlich Alternativen. Ein erfolgversprechender Ansatz besteht darin, für den jeweiligen Problembereich eine eigene Beschreibungssprache zu entwickeln oder eine bestehende zu erweitern. Den Weg einer Erweiterung der funktionalen Sprache Haskell verfolgen die Entwickler des Systems *Hawk*, das in [Matthews u. a. 1998] und [Launchbury u. a. 1999] beschrieben wird. Das System Hawk ist eine domain-spezifische Sprache zur Spezifikation und Simulation von Mikroprozessor-Architekturen. Die Autoren verwenden Haskell als Wirtssprache unter Ausnutzung von statischer Typisierung, parametrischem Polymorphismus, Higher-Order-Funktionen und Lazy-Auswertung. Ein bedeutender Unterschied zu der Herangehensweise in HOPS besteht darin, dass Haskell direkt als Codierungssprache verwendet wird, wir uns hingegen im (graphischen) HOPS-System bewegen und Haskell nur im Hintergrund als Zielsprache zur Code-Emission benutzen, die der Anwender nicht zu Gesicht bekommen muss.

Das System *PROGRES* (PROgrammed Graph REwriting System), das in [Schürr u. a. 1995] beschrieben ist, stellt ebenfalls den Gedanken einer ausführbaren, hochgradig abstrakten (engl.

high-level) Spezifikationssprache in den Vordergrund. Im Gegensatz zu Hawk verwendet PROGRES Graphgrammatiken und ein Graphersetzungssystem als Beschreibungsmittel. Im Vergleich zu HOPS verarbeitet PROGRES allgemeinere Graphen, die weder gerichtet noch kreisfrei sein müssen. Dafür besitzt HOPS einen etwas mächtigeren Transformationsmechanismus mit Regeln zweiter Stufe (Schemavariable mit Nachfolgern). Daran erkennt man auch die Ursprünge der beiden Systeme: HOPS verwendet Termgraphen als Verallgemeinerung von Syntaxbäumen, wie sie nach dem Parsen entstehen, und hat seine Wurzeln auf dem Gebiet der Programmiersprachen, PROGRES indes setzt von vorn herein auf allgemeine Graphen und Graphgrammatiken und wird im Bereich von Spezifikationstechniken eingesetzt, in denen ein ausführbares Programm nicht unmittelbar das Ziel ist.

6.2 Erweiterungen des HOPS-Systems

Die Entwicklungsumgebung HOPS ist ein sich ständig entwickelndes System, das in vielfacher Hinsicht erweitert werden kann.

Wir haben uns bisher mit der Datenflussansicht auf eine statische Visualisierung konzentriert. Wie aber können wir eine Abarbeitung allgemeiner Transformationen visualisieren? Eine Erweiterung, die sich sowohl im Kontext der interaktiven Transformation als auch der Ausführung von Modellen anbietet, ist eine Animation von Transformationssequenzen. Betrachtet man die Ausführung eines Programms als eine spezielle Transformationsstrategie, so gewinnt man durch eine animierte Ausführung einen detaillierten Einblick in das Ablaufverhalten eines Programms. Gerade im Zusammenhang mit datenflussorientierten Modellen ist es von Nutzen, dass der Entwickler einen Programmgraph entwirft und anschließend die Daten auf den Datenflusskanten ‘fließen’ sehen kann.

HOPS ist ein Rahmenwerk mit der Betonung auf Generizität. Wünscht der Entwickler ein System, das auf seinen Problemkontext zugeschnitten ist, so kann eine Spezialisierung auf den speziellen Anwendungskontext, z. B. digitale Signalverarbeitung, vorteilhaft sein. Eine mögliche Anpassung ist z. B. ein Parser, der eine textuelle Differenzengleichung in einen entsprechenden HOPS-Graphen umsetzt. Außerdem sind Kopplungen an spezielle Hardwaresysteme denkbar, bei denen z. B. ein DSP-Operator in einer Umgebung mit realer Hardware simuliert wird, wie dies auch Stand der Technik bei der Entwicklung eines Bordnetzes im Fahrzeugbau der Fall ist (z. B. im Rahmen einer Restbussimulation).

6.3 Ausblick

In der Informatik hat die transformationsgestützte Software-Entwicklung ihren verdienten Stellenwert erlangt und wird diesen in der Zukunft noch weiter ausbauen. An einigen einfachen Beispielen, bei denen weniger das Wort Transformation als vielmehr Generieren gebraucht wird, jedoch vergleichbare Techniken eingesetzt werden, kann dies verdeutlicht werden.

Viele Entwicklungsumgebungen unterstützen den Programmierer, indem sie einfache Codegerüste generieren, worin nur noch anwendungsspezifischer Code einzutragen ist. Dieses initiale Generieren hat natürlich Einmalcharakter und dient v. a. dem schnelleren Einstieg für Unerfahrene bzw. erspart lästige Tipparbeit.

Einen Schritt weiter gehen Generierungsmechanismen, bei denen im erzeugten Programm Bereiche ausgezeichnet sind, in die der Entwickler eigenen Code einfügen kann. Anschließende, weitere Generierungsläufe lassen den benutzerspezifischen Code unberührt (d. h. setzen diesen Code wieder an die entsprechende Stelle), so dass dieser Vorgang beliebig oft wiederholbar ist. Dieses gängige Verfahren setzen z. B. Systeme wie IBM Visual Age for Java in der Komponente GUI-Builder ein. Ein Nachteil besteht darin, dass man sich nach dem starren Schema der Programmerzeugung richten muss und manchmal auch nicht umhin kommt, den generierten Code zu verstehen, um hartnäckigen, selbst verschuldeten Programmierfehlern auf die Spur zu kommen.

Im Gegensatz dazu stehen Mechanismen, bei denen Code erzeugt wird, den der Programmierer gar nicht zu Gesicht bekommt. So ist die Komponentenarchitektur Enterprise Java Beans (EJB) Teil der Spezifikation der Java 2 Enterprise Edition (J2EE). Bei jeder Komponente in EJB handelt es sich um ein Dreigestirn von Klassen, wovon die beiden Anteile Stub und Skeleton von einem eigenen Programm aus einer XML-Spezifikation erzeugt und kompiliert werden. Der Entwickler ist nur noch für den dritten Teil verantwortlich und muss sich um Stub und Skeleton nicht kümmern, die ohnedies abhängig vom Hersteller des EJB-Servers erzeugt werden.

Ausgereifter sind Systeme wie XDE von Rational[1], die eine bidirektionale Verbindung zwischen Spezifikation (hier UML) und Programmcode (hier Java) herstellen. Werden Teile der Spezifikation geändert, so wird auf Wunsch automatisch der Code entsprechend abgeändert. Neu ist v. a. die umgekehrte Richtung, in welcher eine Änderung des Programmcodes eine sofortige Änderung der Spezifikation herbeiführt. Interessant ist die Technik von Transformationsregeln, die in diesem Zusammenhang eingesetzt werden und sogar durch den Benutzer geändert bzw. erweitert werden können. Damit stehen in einem weit verbreiteten System definierbare Transformationsregeln zur Verfügung.

Ein Software-System ist kein starres Gebilde, bei dem Klassen und Methoden unverändert bleiben, wenn sie einmal korrekt funktionieren. Vielmehr bedürfen Erweiterungen und Veränderungen des Systems oft neuer Abstraktionen, die homogen im bestehenden System Anwendung finden sollen. Diesen ständigen Prozess der Codeanpassung nennt man *Refactoring* ([Fowler 1999]). War der Programmierer bei dieser Aufgabe bisher auf sich allein gestellt, so findet diese Technik zunehmend Unterstützung in verbreiteten Entwicklungsumgebungen wie z. B. der Open Source-Initiative Eclipse[2]. Unterstützte Operationen sind das Verschieben von Objektattributen entlang einer Klassenhierarchie oder das Extrahieren von ausgewählten Code-

[1] Rational wurde im Jahr 2003 für 2,1 Mrd. USD von IBM gekauft. Daraus ist ersichtlich, welche Bedeutung der Markt von Spezifikationswerkzeugen in der Zukunft haben wird.

[2] IBM hat im Jahr 2001 der Weiterentwicklung von Eclipse 40 Mio. USD gestiftet. Auch Rational XDE basiert auf dieser Open Source-Umgebung.

zeilen zu Methoden. Viele Mechanismen erfordern eine entsprechende Syntaxanalyse, sind also keine gewöhnlichen Textfunktionen. Hervorzuheben ist, dass bei diesen Transformationen die Abstraktionsebene beibehalten wird, d. h. es wird Programmcode aus Programmcode erzeugt, im Unterschied zu Techniken, bei denen aus einer Spezifikation ein Codegerüst erzeugt wird.

Verfolgt man den Ansatz des Refactoring konsequent weiter, so möchte der Programmierer nicht nur eine transformationsgestützte Entwicklung auf diesem noch sehr bescheidenen Niveau, sondern wünscht sich weit mächtigere Operationen wie der Entrekursivierung von rekursiv definierten Funktionen. Solche Techniken sind nicht unbedingt Aufgabe eines hochoptimierenden Compilers, sondern bedürfen Entscheidungen des Menschen, sind also interaktive Transformationen. Eingabe und Ausgabe der Transformation bleiben im Blickfeld des Entwicklers und können bzw. sollen von ihm weiterverarbeitet werden.

Inzwischen beginnen sogar explizite Transformationssprachen sich aus dem Stadium der Forschung und Entwicklung in der Industrie zu etablieren. Jüngstes Beispiel ist XSLT (eXtended Style Sheet Language for Transformations), das eine Sprache zur Transformation von XML-Dokumenten in andere XML-Dokumente definiert. XSLT wurde als Teil von XSL (eXtended Style Sheet Language) entwickelt, mit dem Aussehen und Formatierung (engl. styling) von Dokumenten definiert wird.

Eine Transformation in der Sprache XSLT wird wiederum durch ein XML-Dokument ausgedrückt und beschreibt Regeln, wie ein Quell-Baum[3] in einen Ergebnis-Baum überführt wird. Dabei kommen Templates zum Tragen, die Muster beschreiben, auf die zu transformierende Teile matchen und entsprechend umgewandelt werden.

Obwohl XSLT ursprünglich im Kontext von XSL entstanden ist, kann es als eigenständige Sprache benutzt werden. Die Sprache ist mächtig genug, sich weitere Anwendungsfelder jenseits von Formatierungsbeschreibungen zu suchen.

Wenn wir unterschiedliche Ebenen der Abstraktion betrachten, z. B. die Ebene der Anforderung und der Implementierung, so spielt die iterative Verfeinerung einer Spezifikation, wie sie Gegenstand von Kapitel 5 ist, eine bedeutende Rolle. Ein schrittweises, interaktives Erzeugen von Code, ausgehend von einer abstrakten Ebene bis hin zu Details in der Implementierung, und mächtige Transformationssprachen werden die Zukunft des Software Engineering nachhaltig beeinflussen. Vorschläge wie MDA (Model Driven Architecture) der OMG (Object Management Group) untermauern diese Prognose.

Daher lautet das Fazit des Autors:

Wir befinden uns erst am Anfang eines Wandels hin zu einer
transformationsgestützten Entwicklung von Software-Modellen und Programm-Systemen!

[3] XML-Dokumente sind Bäume, die durch spezielle Markierungen ausgezeichnet sind.

ANHANG A

Transformationssequenzen zur Differenzengleichung

Kapitel 5 beschreibt einen durchgängigen Weg, wie wir aus einer abstrakten Differenzengleichung eine ausführbare Implementierung gewinnen können. Dabei kommt ein Transformationsverfahren zur Anwendung, das in einer automatisierten Weise eine Reihe von Transformationsregeln appliziert.

Nachfolgend zeigen wir einige Zwischenschritte dieser Transformationssequenz, um dem Leser den Ablauf näher zu bringen. Der daraus entstehende Programmgraph besteht am Ende des Verfahrens nur noch aus einfachen Operatoren wie Addition, Multiplikation und Verzögerung. Daraus erzeugen wir anschließend mittels eines Moduls zur Codeemission Programmcode für eine funktionale Programmiersprache.

Bei unserem Verfahren sind wir zuerst daran interessiert, den Graphen auf die direkte Form I zu transformieren. Anschließend zeigen wir, wie wir zu der effizienteren direkten Form II kommen.

A.1 Direktes System

Wir zeigen informell die einzelnen Regeln aus HOPS, die während des Transformationsablaufs zur Anwendung kommen. Dabei können die angegebenen Schritte innerhalb einer Transformationssequenz definiert werden, die vollautomatisch ausgeführt wird. Die Regeln sind jeweils unterstrichen (*Regelname*), ein $^+$ weist darauf hin, dass eine Regel möglicherweise mehr als einmal angewendet wird.

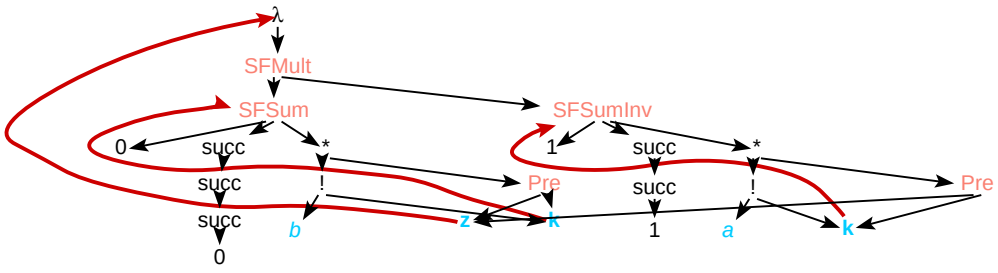


Abbildung A.1: Transformationssequenz (Start)

1. Einführung des Kompositionsoperators ';' mit der Regel *SFMult* $\longleftrightarrow$ *Comp*
2. Entfalten der Operators SFSum von oben nach unten, so dass rechtshängende Additionen nach dem Muster $x+(y+z)$ entstehen
   a) *SFSum-Unfold-Recursive-first*$^+$
   b) *SFSum-Unfold-Finish*
3. Abbilden des Operators Pre auf Z^{-1} von unten nach oben
   a) *Pre-init*
   b) *Pre-step (mr)*$^+$
   c) *Pre-Zero*
4. Entfalten des Operator SFSumInv (analog zu Schritt 2)
5. Abbilden des Operators Pre auf Z^{-1} mit *Pre-One* (siehe Schritt 3)
6. Auf die Form eines Blockdiagramms trimmen (*+ Commutativity*)
7. Cycle1 nach Cycle2 konvertieren (*Cycle1* $\longleftrightarrow$ *Cycle2*)
8. *Wahlweise Komposition auf Multiplikation zurückführen und kommutieren lassen, um auf direkte Form II zu kommen*

9. Zusammenfassen der Funktionen unter Komposition (*Composition Functions Merge*)
   a) nicht kommutiert: (siehe Abbildung A.4, Stopp der Transformation)
   b) kommutiert: (weiter in 10)
10. *Fortsetzung, wenn oben kommutiert:*
    Cycle2 erweitern (Abbildung A.5)
11. Zusammenfassen von gemeinsamen Teilausdrücken bei +, Z^{-1} (*Merge*) siehe Abbildung A.6
    Optimierung durch weniger Schaltelemente (Z^{-1})

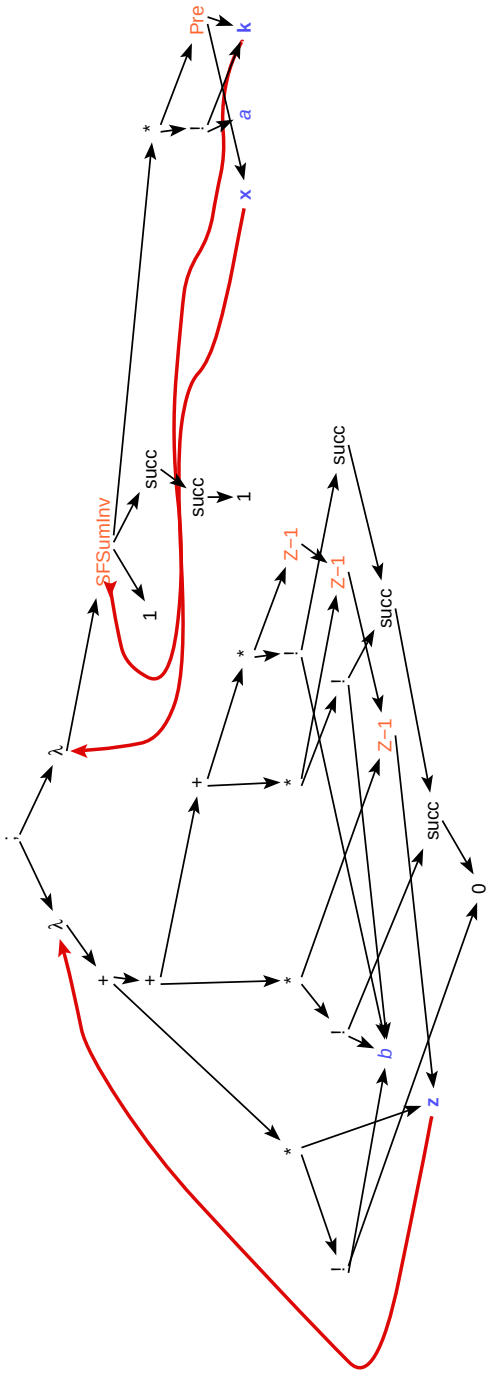


Abbildung A.2: Transformationssequenz (nach Schritt 3)

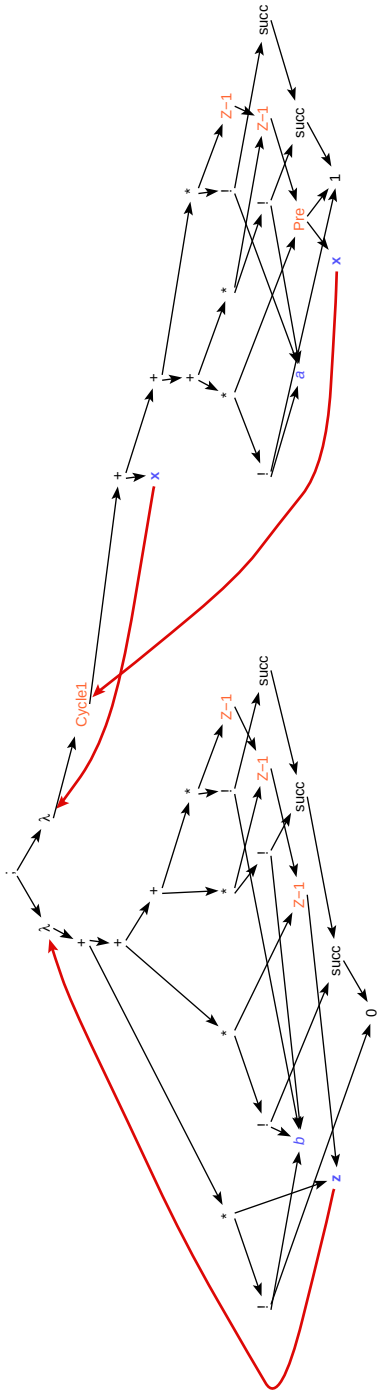


Abbildung A.3: Transformationssequenz (nach Schritt 6)

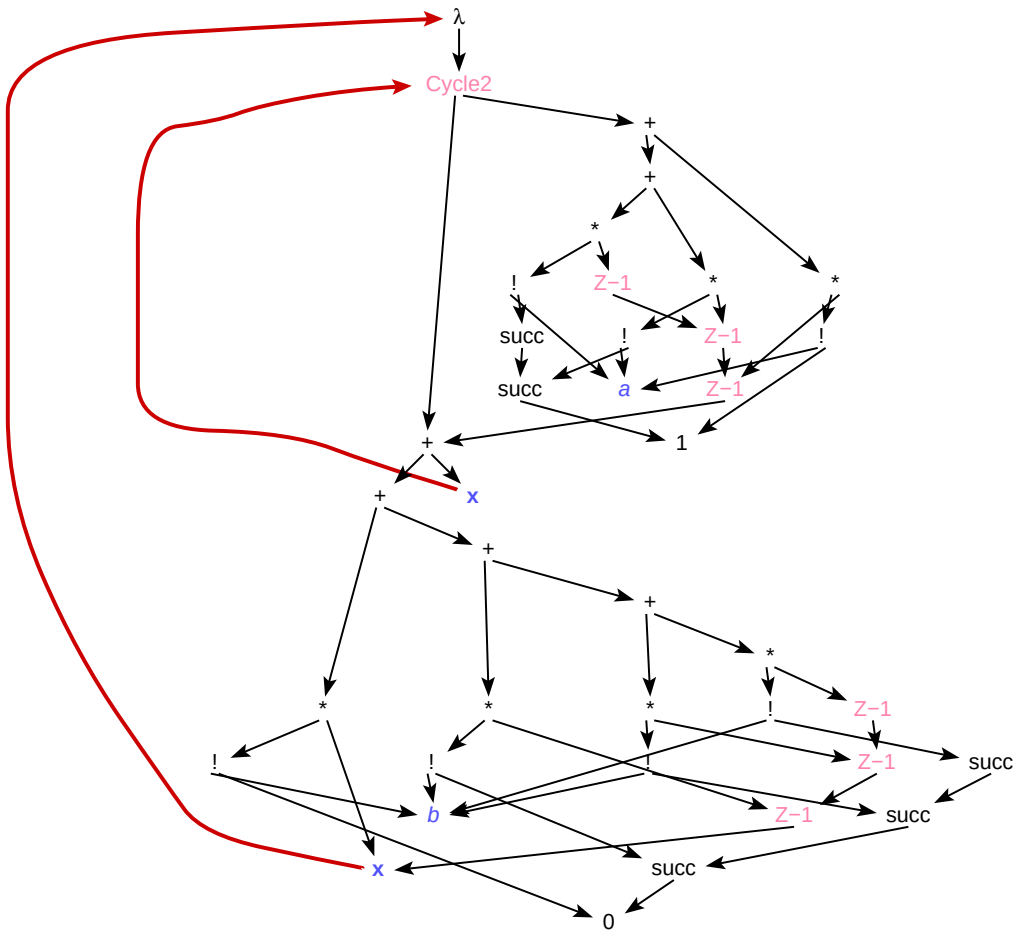


Abbildung A.4: Transformationssequenz (nach Schritt 9a)

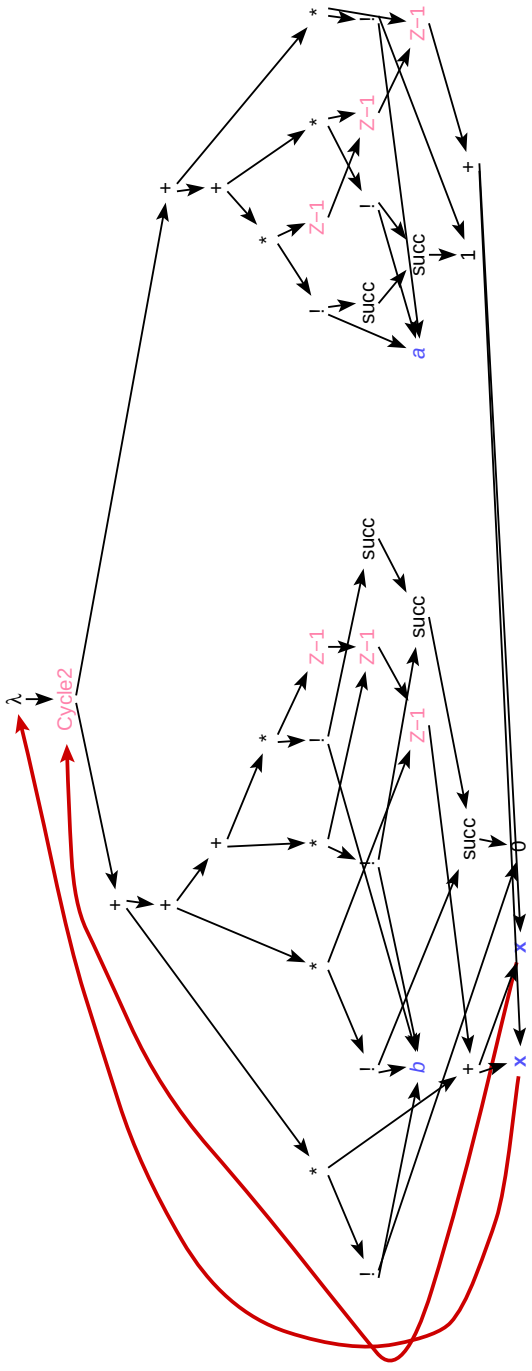
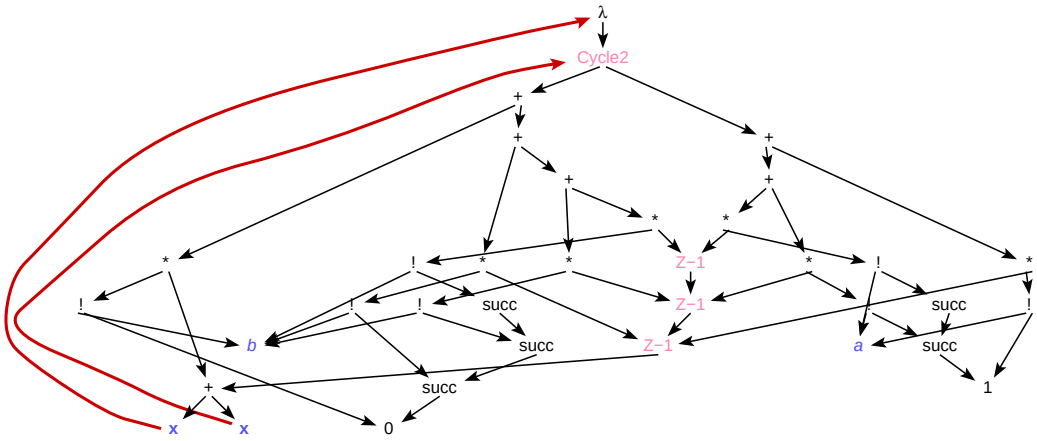
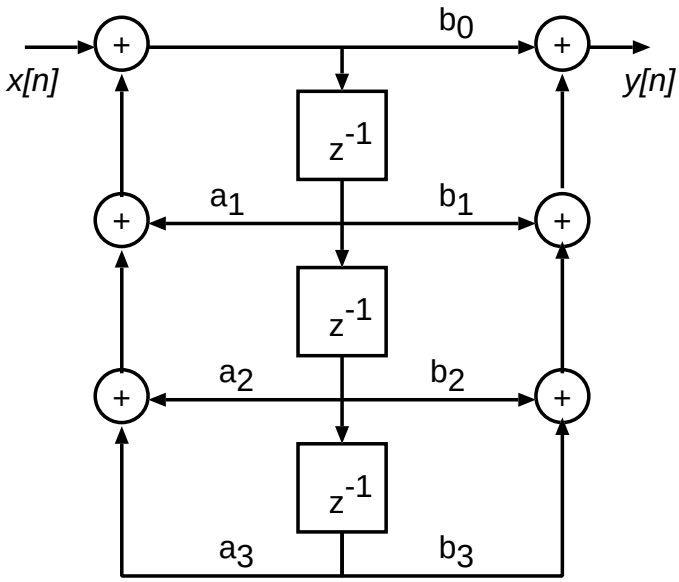


Abbildung A.5: Transformationssequenz (nach Schritt 10)



(a) DAG



(b) Blockdiagramm

Abbildung A.6: Transformationssequenz (nach Schritt 11)

A.2 Kaskadiertes System

Abbildung A.7 beschreibt das Ausgangssystem für ein kaskadiertes System, das Gleichung (5.35) entspricht. Dieses besteht aus einem Term (SFMult ...), den wir mit der bereits bekannten Regelsequenz aus Abschnitt A.1 für die direkte Form I und II auflösen können.

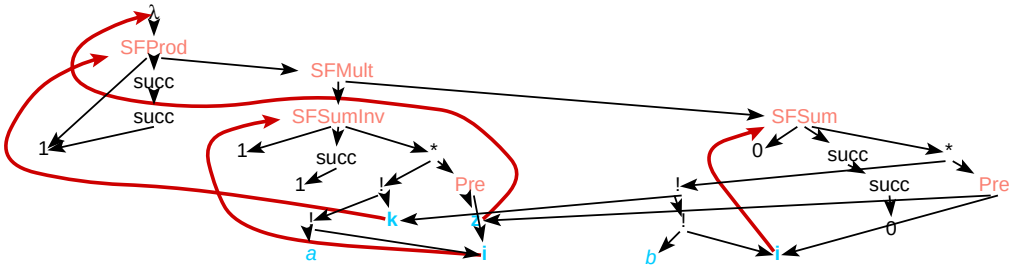


Abbildung A.7: Ausgangssystem für ein kaskadiertes System

Nun haben wir noch das Produkt SFProd zu behandeln. Dieses entfalten wir in einem nächsten Schritt und gelangen so zu Abbildung A.8.

Anschließend können wir die Multiplikation (SFMult) zu einer Komposition auflösen (;), so dass wir als nächsten Zwischenschritt Abbildung A.9 erreichen.

In weiteren Schritten können wir analog zu Abschnitt 5.4.2 vorgehen und die einzelnen, sequenziell hintereinander geschalteten Teilsysteme in einer direkten Form II implementieren. Sie verlaufen analog zu Abschnitt A.1.

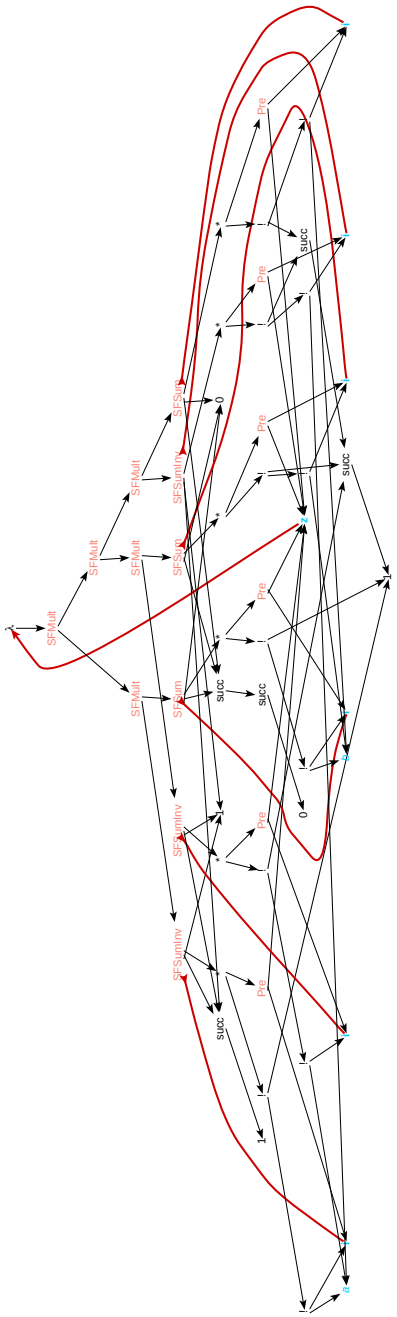


Abbildung A.8: SFProd aufgelöst

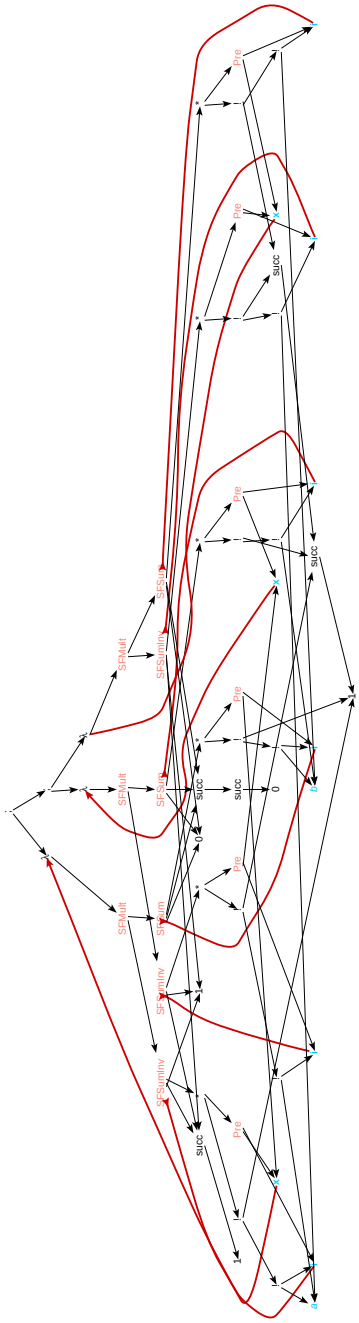


Abbildung A.9: SFMult aufgelöst

ANHANG B

Funktionen aus der digitalen Signalverarbeitung

In Abschnitt 5.5.1 haben wir zu einigen Funktionen aus der digitalen Signalverarbeitung eine Implementierung in der funktionalen Programmiersprache Haskell angegeben.

Hier geben wir nun die Module an, die zur Definition bzw. Generierung von Funktionen verwendet werden bzw. mittels einer Codeemissionstechnik generiert wurden.

Grundlegende Funktionen

Listing B.1: Grundlegende Funktionen

```
module Direkt where
import Array
import Konstanten

-- Faltungssumme (im endlichen Bereich von 0..n)
convolution :: (ℤ →ℚ) →(ℤ →ℚ) →ℤ→ℚ
convolution f g = \n →functionAddFold (map
                        (\k →(f k) ‘ functionMult ‘ ( functionShift g k)) [0 .. n]) n

-- functionShift
-- verschiebt den Definitionsbereich einer Funktion f um n0 Einheiten nach rechts
functionShift :: (ℤ →ℚ) →ℤ→ℤ→ℚ
functionShift f n0 = \n →f (n−n0)
```

```
-- Multiplikation einer Funktionen mit einer Konstanten
functionMult :: ℚ→(x →ℚ) →x →ℚ
functionMult a f = \n →a ∗ (f n)

-- punktweise Addition einer Liste von Funktionen
functionAddFold :: [ℤ →ℚ] →ℤ→ℚ
functionAddFold funcs n = foldr (\f x →(f n) + x) 0 funcs

-- Faltungssumme, punktweise definiert
-- berechne [f(0)..f(n)] und [g(0) .. g(n)]
-- berechne [f(0)∗g(n)..f(n)∗g(0)]
-- addiere die resultierende Folge
convolution' :: (ℤ →ℚ) →(ℤ →ℚ) →ℤ→ℚ
convolution' f g = \n →foldr (+) 0 ( zipWith (∗) ( map f [0..n] ) ( reverse (map g [0..n] )))

-- Ist Faltungssumme kommutativ für Werte von 0..n?
convolutionCommTest f g n = and (map
                                  (\n0 →convolution' f g n0 == convolution' g f n0) [0..n])

-- xFun als Funktion des Arrays x
xFun :: ℤ →ℚ
xFun = (\n →x !! n)

-- Einheitsimpuls
delta :: ℤ →ℚ
delta 0 = 1
delta _ = 0

-- verzögerter Einheitsimpuls
-- deltaShift == functionShift delta
deltaShift n0 = \n →if n==n0 then 1 else 0

-- Impulsfolge (über eine endliche Summe!)
-- impulseSequence func == func
impulseSequence xfun = convolution xfun delta

-- Test, ob die Impulsfolge mit der Originalfunktion übereinstimmt
impulseSequenceTest :: ℤ→ℤ→(ℤ →ℚ) →𝔹
impulseSequenceTest from to func =
    and (map (\n →impulseSequence func n == func n) [from..to])
```

```
-- Abänderung einer Funktion func an Stelle n zum Wert v
functionModify func n v = \n0 →if n==n0 then v else (func n0)

-- x-Funktion, Oppenheim/Schafer S. 24
xFoldFun :: ℤ→ℚ
xFoldFun = \n →case n of
                    (−2) →3.0
                    0 →4.0
                    3 →(−2.0)
                    _ →0.0

-- hFun: Antwortfolge auf delta
-- h-Funktion, Oppenheim/Schafer S. 24
hFun :: ℤ→ℚ
hFun = \n →case n of
                    0 →3.0
                    1 →2.0
                    2 →1.0
                    _ →0.0

-- y-Funktion, Oppenheim/Schafer S. 24
yFoldFun = convolution xFoldFun hFun


-- FIR Differenzengleichung (ohne Rückkopplung)
yFIRFun = \n →(b!0)∗(functionShift xFun 0 n) +
                (b!1)∗( functionShift xFun 1 n) + (b!2)∗( functionShift xFun 2 n)

-- IIR Differenzengleichung (mit Rückkopplung)
yIIRFun = \n →
          (a!1)∗( functionShift yIIRFun 1 n) + (a!2)∗( functionShift yIIRFun 2 n)
           + (b!0)∗( functionShift xFun 0 n) + (b!1)∗( functionShift xFun 1 n)
           + (b!2)∗( functionShift xFun 2 n)

testFIR x = map yFIRFun x

testIIR x = map yIIRFun x

-- Funktionsergebnisse von y(0) .. y(19)/y(99)
```

```
-- Performance: test1
-- (299531 reductions, 381308 cells, 4 garbage collections)
test1 = testFIR [0..99]

-- Performance: test2
-- (15259315 reductions, 20780493 cells, 243 garbage collections)
test2 = testIIR [0..19]
```

Matrixmultiplikation

Listing B.2: Matrixmultiplikation

```
import DSP_Basic
import Konstanten

-- beispielhafte Übergangsmatrix
bspBlockMatrix = [[b0,a1,a2,a3,b1,b2,b3],
                  [  0, 1, 0, 0, 0, 0, 0  ],
                  [  0, 0, 1, 0, 0, 0, 0  ],
                  [  1, 0, 0, 0, 0, 0, 0  ],
                  [  0, 0, 0, 0, 1, 0, 0  ],
                  [  0, 0, 0, 0, 0, 1, 0  ]]
matrixMult :: Num a => Matrix a →Vector a →Vector a
matrixMult matrix@(m:ms) vector
   | not(and((map (\row →length m == length row) ms))) =
        error "matrixMult: Matrix nicht rechteckig"
   | length m /= length vector =
       error ("matrixMult: Matrix (" ++
          show (length (head matrix)) ++ "x" ++
          show (length matrix) ++
           ") und Vektor (" ++
          show (length vector) ++ ") nicht kompatibel!")
   | otherwise = map (addmult vector) matrix

-- Eingabestrom der Breite 1
blockInput :: Matrix ℚ→Vector ℚ→Vector ℚ
blockInput matrix vector = blockInputH matrix vector initState
                  --- Initialzustand mit 0en
          where initState = replicate (length matrix) 0

--blockInputH :: Matrix ℤ→Vector ℤ→Vector ℤ→Vector ℤ
blockInputH matrix (x:xs) state = y : blockInputH matrix xs newState
          where newState@(y:ss) = matrix ‘matrixMult‘ (x:state)
blockInputH _ [] _ = []

testIIR x = blockInput bspBlockMatrix x
```

Graphabbildung

Listing B.3: Graphabbildung

```
#!/usr/bin/runhugs

> module Main where
> import Array
> import Konstanten
> main = putStr (show (lc1 [1..5000]) ++ "\n")

---- Generierter Code!!!
---- Shift mit verschobenen Parametern implementiert
---- l_1: links 1. shared Knoten; r_3: rechts 3. shared Knoten

> lc1 :: [ℚ] →[ℚ]
> lc1 xs = lc1 ' 0.0 0.0 0.0 xs
> lc1 ' _ _ _ [] = []
> lc1 ' z1 z2 z3 (x:xs) = r_1 : ( lc1 ' l_1 z1 z2 xs)
>       where r_1 =((+)
>                    ((∗) ( b!0) l_1)
>                    ((+)
>                      ((∗) ( b!1) z1)
>                      ((+)
>                        ((∗) ( b!2) z2)
>                        ((∗) ( b!3) z3)
>                      )
>                    )
>                  )
>             l_1 = (+) x l_2
>             l_2 = ((+)
>                      ((+)
>                       ((∗) ( a!3) z3)
>                        ((∗) ( a!2) z2)
>                      )
>                      ((∗) ( a!1) z1)
>                    )

---- Performance: lc1 [1..100]
---- [1.0,4.75,11.4375,19.9844,29.5586,39.6709,50.0583,60.5849,71.1814 ...] :: [ℤ]
```

```
----- (26847 reductions, 36243 cells)


> lc2 :: [ℚ] →[ℚ]
> lc2 xs = ys
>  where ys = wireAdd w2 w4
>        w1 = wireAdd xs w3
>        w2 = wireMult (b!0) w1
>        w3 = wireAdd w8 w5
>        w4 = wireAdd w7 w10
>        w5 = wireMult (a!1) w6
>        w6 = wireReg w1
>        w7 = wireMult (b!1) w6
>        w8 = wireAdd w13 w11
>        w9 = wireReg w6
>        w10 = wireAdd w12 w15
>        w11 = wireMult (a!2) w9
>        w12 = wireMult (b!2) w9
>        w13 = wireMult (a!3) w14
>        w14 = wireReg w9
>        w15 = wireMult (b!3) w14

----- Performance: lc2 [1..100]
----- [1.0,4.75,11.4375,19.9844,29.5586,39.6709,50.0583,60.5849,71.1814 ...] :: [ℤ]
----- (6679 reductions, 12746 cells)

---------- Nützliche Funktionen ----------

> wireAdd :: [ℚ] →[ℚ] →[ℚ]
> wireAdd = zipWith (+)

> wireMult :: ℚ→[ℚ] →[ℚ]
> wireMult n = map (∗n)

> wireReg :: [ℚ] →[ℚ]
> wireReg = (0.0 :)


-- Vorwärts-Komposition ";"
(.|) = flip (.)
(>.>) = flip (.)
```

```
-- Schieberegister
-- lösche linkes Element, hänge rechts eins dran
shiftL :: [a] →a →[a]
shiftL l x = ( tail l ) ++ [x]

-- Array aus Liste ab Index 1
listArray1 l = listArray (1, length l ) l
```

Globale Konstanten

Listing B.4: Konstanten

```
module Konstanten where
import Array

-- Definition globaler Konstanten

a1, a2, b0, b1, b2 :: ℚ
a1 = a!1
a2 = a!2
a3 = a!3
b0 = b!0
b1 = b!1
b2 = b!2
b3 = b!3


-- Beispielvektoren

bspBlockVektor :: [ℚ]
bspBlockVektor = [7,8,9,10,11]
x :: [ℚ]
x = [1..5000]


-- Konstantenarrays

a,b :: Array ℤℚ
--a = listArray (1,3) [1..3]
--b = listArray (0,3) [1..4]
a = listArray (1,3) [0.75, -0.125, 0]
b = listArray (0,3) [ 1, 2, 1, 0 ]
```

LITERATURVERZEICHNIS

[Ackerman und Dennis 1979] ACKERMAN, W. B. ; DENNIS, J. B.: VAL – A value-oriented algorithmic language / Computation Structures Group, Laboratory for Computer Science, MIT. Cambridge, MA, Juni 1979 (TR-218). – Forschungsbericht

[Aditya u. a. 1995] ADITYA, Shail ; ARVIND ; AUGUSTSSON, Lennart: Semantics of pH: A parallel dialect of Haskell / Massachusetts Institute of Technology, Laboratory for Computer Science. Cambridge, Massachusetts, Juni 1995 (Computer Structures Group Memo 377-1). – Forschungsbericht

[Arvind u. a. 1978] ARVIND ; GOSTELOW, K. P. ; PLOUFFE, W.: An Asynchronous Programming Language and Computing Machine / Department of Information and Computer Science, University of California. Irvine, Dezember 1978 (TR114a). – Forschungsbericht

[Backus 1978] BACKUS, John: Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. 21 (1978), August, Nr. 8, S. 613–641

[Bauer u. a. 1985] BAUER, F. L. ; BERGHAMMER, R. ; BROY, M. ; DOSCH, W. ; GEISELBRECHTINGER, F. ; GNATZ, R. ; HANGEL, E. ; HESSE, W. ; KRIEG-BRÜCKNER, B. ; LAUT, A. ; MATZNER, T. ; MÖLLER, B. ; NICKL, F. ; PARTSCH, H. ; PEPPER, P. ; SAMELSON, K. ; WIRSING, M. ; WÖSSNER, H.: *LNCS*. Bd. 183: *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L*. Berlin/Heidelberg/New York : Springer-Verlag, 1985

[Bayer 1995] BAYER, Arne: GUI-Programming with HOPS. In: MARGARIA, Tiziana (Hrsg.): *Kolloquium Programmiersprachen und Grundlagen der Programmierung, Adalbert Stifter Haus, Alt Reichenau, 11.–13. Oktober 1995*, Universität Passau, Fakultät für Mathematik und Informatik, Dezember 1995 (Bericht MIP-9519), S. 29–34

[Bayer und Derichsweiler 1997] BAYER, Arne ; DERICHSWEILER, Frank: HOPS as a link between functional and dataflow oriented programming. In: BERGHAMMER, Rudolph (Hrsg.) ; FRIEDEMANN, Simon (Hrsg.): *Programming Languages and Fundamentals of Programming*, Christian-Albrechts-Universität Kiel, November 1997, S. 209–217

[Bayer u. a. 1996] BAYER, Arne ; GROBAUER, Bernd ; KAHL, Wolfram ; KEMPF, Peter ; SCHMALHOFER, Franz ; SCHMIDT, Gunther ; WINTER, Michael: The **H**igher **O**bject **P**rogramming **S**ystem **HOPS** / Universität der Bundeswehr München, Fakultät für Informatik. 1996. – Forschungsbericht. Interner Bericht

[Bayer 1987] BAYER, Ludwig J.: *Entwicklung von komplexen Werkzeugen zur interaktiven Auswertung von DAGs im Programmiersystem HOPS*, TU München, Diplomarbeit, 1987

[Bayer u. a. 1990] BAYER, Ludwig J. ; BERGHAMMER, Rudolf ; KEMPF, Peter: Programmieren mit höheren Objekten: Bausteine und Regeln im HOPS-System / Universität der Bundeswehr München, Fakultät für Informatik. 1990 (9003). – Technischer Bericht

[Blahut 1985] BLAHUT, Richard E.: *Fast Algorithms for Digital Signal Processing*. Owego, New York : Addison-Wesley Publishing Company, 1985

[Böhm u. a. 1992] BÖHM, A. P. W. ; OLDEHOEFT, R. R. ; CANN, D. C. ; FEO, J. T.: *SISAL Reference Manual, Language Version 2.0*, 1992

[Cann 1992] CANN, David C.: SISAL 1.2: A Brief Introduction and Tutorial / Computing Research Group, L-306, Lawrence Livermore National Laboratory. Livermore, CA 94550, 1992. – Forschungsbericht

[Chang 1986] Kap. Introduction: Visual Languages and Iconic Languages In: CHANG, Shi-Kuo: *Visual Languages*. Plenum Press, New York, 1986 (Management and Information Systems)

[Christ 1996] CHRIST, Oliver: *Implementierung von Typisierungs- und Higher-Order-Matchingalgorithmen in HOPS*, Universität der Bundeswehr München, Fakultät für Informatik, Diplomarbeit, Dezember 1996

[Curry 1930] CURRY, H. B.: Grundlagen der kombinatorischen Logik. In: *American J. Math* 52 (1930), S. 509–536, 789–834

[Curry und Feys 1958] CURRY, H. B. ; FEYS, R.: *Combinatory Logic*. Bd. 1. North Holland, 1958

[Dennis 1984] DENNIS, Jack B.: Models of Data Flow Computation. In: BROY, Manfred (Hrsg.): *Control Flow and Data Flow, Concepts of Distributed Programming* Bd. 14, Springer-Verlag, 1984, S. 345 – 354

[Derichsweiler 1996] DERICHSWEILER, Frank: *Datenflußgraphen und Termgraphen im Higher Object Programming System HOPS*, Universität der Bundeswehr München, Fakultät für Informatik, Diplomarbeit, Dezember 1996

[Embree 1995] EMBREE, Paul M.: *C Algorithms for Real-Time DSP*. Prentice Hall PTR, 1995

[Endres und Müller 1991] ENDRES, Klaus ; MÜLLER, Frank: *Objektorientierte Implementierung eines Programmiersystems mit zweischichtig typisierter DAG-Sprache und Regelanwendung*, Universität der Bundeswehr München, Fakultät für Informatik, Diplomarbeit, Dezember 1991

[Field und Harrison 1988] FIELD, J. A. ; HARRISON, G. P.: *Functional Programming*. Addison-Wesley Publishing Company, 1988

[Fowler 1999] FOWLER, Martin: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Publishing Company, 1999 (Addison-Wesley Object Technology Series)

[Glaser und Hayes 1986] GLASER, H. ; HAYES, S.: Another implementation technique for applicative languages. In: *Proc. European Symposium on Programming*. Saarbrücken : Springer-Verlag, 1986 (LNCS 213), S. 70–81

[Held 1991] HELD, Christian: *Spezifikation und Implementierung eines Vergleichs- und Transformationssystems für eine typisierte DAG-Sprache als HOPS-Teilsystem*, Universität der Bundeswehr München, Fakultät für Informatik, Diplomarbeit, Dezember 1991. – ID 38/91

[Held und Zimmermann 1990] HELD, Christian ; ZIMMERMANN, Thomas. *Ausgabeschnittstelle von HOPS*. Trimesterarbeit IT 35/90, Universität der Bundeswehr München, Fakultät für Informatik. September 1990

[Hudak 1989] HUDAK, Paul: Conception, Evolution, and Application of Functional Programming Languages. In: *ACM Computing Surveys* 21 (1989), September, Nr. 3, S. 359–411

[Kahl 1991] KAHL, Wolfram: *HOPS — Higher Object Programming System, Functional Graphics-Based Interactive Programming, User Manual*. Fakultät für Informatik, Universität der Bundeswehr München, Januar 1991. – (internal manual)

[Kahl 1994] KAHL, Wolfram: Can Functional Programming Be Liberated from the Applicative Style? In: PEHRSON, Bjørn (Hrsg.) ; SIMON, Imre (Hrsg.): *Technology and Foundations, Information Processing '94, Proceedings of the IFIP 13th World Computer Congress, Hamburg, Germany, 28 August – 2 September 1994, Volume I* Bd. A-51 IFIP, North-Holland, 1994, S. 330–335

[Kahl 1996] KAHL, Wolfram: *Algebraische Termgraphersetzung mit gebundenen Variablen*. München : Herbert Utz Verlag Wissenschaft, 1996 (Reihe Informatik). – ISBN 3-931327-60-4; auch als Dissertation bei der Fakultät für Informatik, Universität der Bundeswehr München

[Kahl 1998] KAHL, Wolfram: *The **H**igher **O**bject **P**rogramming **S**ystem — User Manual for HOPS*. Fakultät für Informatik, Universität der Bundeswehr München, 1998. – electronically available via URL: `http://ist.unibw-muenchen.de/kahl/HOPS/hopsmanual.ps.gz`

[Kennaway und Sleep 1988] KENNAWAY, J.R. ; SLEEP, M.R.: Director Strings as Combinators. In: *ACM Transactions on Programming Languages and Systems* 10 (1988), Nr. 4, S. 602–626

[Kimura u. a. 1986] KIMURA, T. D. ; CHOI, J. W. ; MACK, J. M.: A Visual Language for keybordless Programming / Department of Computer Science, Washington University. St. Louis, Missouri 63130, 1986 (WUCS-86-6). – Forschungsbericht

[Kimura u. a. 1990] KIMURA, T. D. ; CHOI, J. W. ; MACK, J. M.: Show and Tell: A Visual Programming Language. In: GLINERT, E. P. (Hrsg.): *Visual Programming Environments*. Los Alamitos/CA : IEEE Computer Science Press, 1990, S. 397 – 404

[Landin 1966] LANDIN, P. J.: The Next 700 Programming Languages. In: *Communications of the ACM* 9 (1966), März, Nr. 3, S. 157–164. – Originally presented at the Proceedings of the ACM Programming Language and Pragmatics Conference, August 8–12, 1965.

[Launchbury u. a. 1999] LAUNCHBURY, John ; LEWIS, Jeff ; COOK, Byron: On embedding a microarchitectural design language within Haskell. In: *ACM Int. Conf. on Functional Programming*, 1999

[Libera 1987] LIBERA, Klaus: *DAG-Transformationen für benutzerdefinierte Typen im graphisch gestützten Dialog*, TU München, Diplomarbeit, 1987

[Louden 1994] LOUDEN, Kenneth C. ; MAHR, Bernd (Hrsg.) ; SCHILL, Alexander (Hrsg.) ; VOSSEN, Gottfried (Hrsg.): *Programmiersprachen — Grundlagen, Konzepte, Entwurf*. International Thomson Publishing GmbH, 1994

[Matthews u. a. 1998] MATTHEWS, John ; LAUNCHBURY, John ; COOK, Byron: Microprocessor Specification in Hawk. In: *International Conference on Computer Languages*. Chicago, 1998

[Milner 1978] MILNER, Robin: A Theory of Type Polymorphism in Programming. In: *Journal of Computer and System Sciences* 17 (1978), S. 348–375

[Najork und Golin 1990] NAJORK, Marc A. ; GOLIN, Eric: Enhancing Show-and-Tell with a polymorphic type system and higher-order functions. In: *Workshop on Visual Languages*. Skokie, IL : IEEE Computer Society Press, Oktober 1990, S. 215 – 220

[Nikhil u. a. 1995] NIKHIL, Rishiyur S. ; ARVIND ; HICKS, James ; ADITYA, Shail ; AUGUSTSSON, Lennart ; MAESSEN, Jan-Willem ; ZHOU, Yuli: *pH Language Reference*

Manual, Version 1.0 - preliminary. Digital Equipment Corp, Cambridge Research Lab, Januar 1995

[Oppenheim und Schafer 1992] OPPENHEIM, Alan V. ; SCHAFER, Ronald W.: *Zeitdiskrete Signalverarbeitung*. München : R. Oldenburg Verlag, 1992

[Peyton Jones 1998] PEYTON JONES, Simon et. a.: Haskell 98: A Non-strict, Purely Functional Language. 1998. – Forschungsbericht

[Peyton Jones 1999] PEYTON JONES, Simon et. a.: Standard Libraries for Haskell 98. 1999. – Forschungsbericht

[Révész 1988] RÉVÉSZ, György E.: *Lambda-calculus, Combinators and Functional Programming*. Cambridge : Press Syndicate of the University of Cambridge, 1988 (Cambridge tracts in theoretical computer science 4)

[Schmidt u. a. 1990] SCHMIDT, Gunther ; BAYER, Ludwig J. ; KEMPF, Peter: Hinweise zur interaktiven Konstruktion von Bereichen, Objekten und Programmtexten im HOPS-System / Universität der Bundeswehr München, Fakultät für Informatik. 1990 (9004). – Technischer Bericht

[Schönfinkel 1924] SCHÖNFINKEL, Moses: Über die Bausteine der mathematischen Logik. In: *Math. Annalen* 92 (1924), S. 305–316

[Schürr u. a. 1995] SCHÜRR, A. ; WINTER, A. ; ZÜNDORF, A.: Visual programming with graph rewriting systems. In: *IEEE Symposium on Visual Languages*. Darmstadt, Germany, 1995 (IEEE Computer Society Press), S. 326–333

[Shu 1988] SHU, Nan C.: *Visual Programming*. New York : Van Nostrand Reinhold Company Inc., 1988

[Stender 1992] STENDER, Jörg: *Erweiterung des DAG-orientierten Programmiersystems Smalltalk-HOPS um Modulverwaltung und Transformationssystem*, Universität der Bundeswehr München, Fakultät für Informatik, Diplomarbeit, 1992. – ID 42/92

[Stoye 1985] STOYE, W. R.: *The implementation of functional languages using custom hardware*, University of Cambridge, Diss., 1985

[Thiemann 1994] THIEMANN, Peter: *Grundlagen der funktionalen Programmierung*. B. G. Teubner, Stuttgart, 1994

[Turner 1979] TURNER, David A.: Another Algorithm for Bracket Abstraction. In: *J. Symbol. Logic* 44 (1979), S. 267–270

[Turner 1979] TURNER, David A.: A New Implementation Technique for Applicative Languages. In: *SOFTWARE — Practice and Experience* 9 (1979), S. 31–49

[Zierer u. a. 1986] ZIERER, Hans ; SCHMIDT, Gunther ; BERGHAMMER, Rudolf: An Interactive Graphical Manipulation System for Higher Objects Based on Relational Algebra. In: TINHOFER, Gottfried (Hrsg.) ; SCHMIDT, Gunther (Hrsg.): *Proc. 12th International Workshop on Graph-Theoretic Concepts in Computer Science*. Bernried, Starnberger See : Springer-Verlag, Juni 1986 (LNCS 246), S. 68–81

[Zimmermann 1991] ZIMMERMANN, Thomas: *Studie zum Modulkonzept in HOPS-3*, Universität der Bundeswehr München, Fakultät für Informatik, Diplomarbeit, Dezember 1991. – ID 17/91