# Multiparadigmen-Programmiersprachen

Martin Grabmüller

**Bericht Nr. 2003-15**

Oktober 2003

ISSN: 1436-9915

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Übersetzerbau und Programmiersprachen

Forschungsbericht

# Multiparadigmen-Programmiersprachen

Martin Grabmüller

Oktober 2003

# Inhaltsverzeichnis

# Kurzfassung

Dieser Bericht untersucht Programmiersprachen, die verschiedene Programmierparadigmen in sich vereinen. Dabei wird ein breiter Überblick über die existierenden Ansätze und Techniken bei Entwurf und Implementierung multiparadigmatischer Sprachen herausgearbeitet. Die Unterschiede der einzelnen etablierten Programmierparadigmen werden beschrieben und die Möglichkeiten zur Kombination dieser Paradigmen anhand der Untersuchung existierender Programmiersprachen erläutert. Nach einer genaueren Betrachtung syntaktischer und semantischer Eigenschaften unterschiedlicher Programmiersprachen werden Vorschläge zum Entwurf von Programmiersprachen, welche die Programmierung in unterschiedlichen Paradigmen erlauben, entwickelt.

# 1 Einführung

Verschiedene Menschen nehmen zur Lösung von gleichen Aufgaben unterschiedliche Sichtweisen ein, und ebenso kann es sein, dass ein Mensch für unterschiedliche Aufgaben verschiedene Lösungsansätze verfolgt. Für die wenigsten Probleme gibt es „die“ optimale Lösung, und so empfiehlt es sich, der- oder demjenigen, der das Problem löst, die Wahl zu überlassen, wie er oder sie am besten mit einer Aufgabe umgeht. Daher ist es für den praktischen Einsatz von Programmiersprachen notwendig, diese unterschiedlichen Sichtweisen, die zu verschiedenen Lösungsstrategien führen, in geeigneter Form ausdrücken zu können. Eine mathematische Aufgabe lässt sich am besten durch den Einsatz einer mathematischen Notation für den Lösungsalgorithmus lösen, da auf diese Weise keine unnötige Übersetzung in eine andere Denkweise notwendig ist. Ebenso wird ein nebenläufiges Programm besser in einer Sprache entwickelt, die geeignete Abstraktionen für Prozesse zur Verfügung stellt, als in einer sequenziellen Sprache, in der Prozesse durch einen anderen Mechanismus simuliert werden müssen.

Wenn man also davon ausgeht, dass der Einsatz unterschiedlicher Programmiersprachen notwendig ist, um geeignete Lösungen verschiedener Probleme zu formulieren, stellt sich die Frage, wie komplexere Problemstellungen behandelt werden sollen, die durch den Einsatz unterschiedlicher Sprachen zur Lösung von Teilproblemen besser gelöst werden können als durch die Verwendung einer einzelnen Sprache. Sollen unterschiedliche Sprachen eingesetzt werden, was zu Problemen bei der Integration der Teillösungen führt, oder soll eine einzige Sprache zur Lösung aller Teile benutzt werden, was zu inadäquaten (und damit fehlerhaften und schwer wartbaren) Programmen führt. Ein möglicher Ansatz, der in diesem Bericht untersucht werden soll, ist der Einsatz von Multiparadigmen-Programmiersprachen.

Multiparadigmen-Programmiersprachen vereinigen die Ausdrucksmöglichkeiten mehrerer Programmierparadigmen, wie zum Beispiel der logischen, funktionalen oder objektorientierten Programmierung, in einer integrierten Programmiersprache. Die Überlegung, dass jedes dieser Paradigmen spezifische Vorteile bei der Lösung bestimmter Probleme bietet, führte zur Forschung und zu Entwicklungen in dieser Richtung. Aufgabenstellungen mit Teilproblemen, die durch die Eigenschaften verschiedener Paradigmen gut einzeln bearbeitet werden, sollen sich so durch die Verwendung einer einzigen Sprache effektiv lösen lassen, ohne durch die Verwendung unterschiedlicher Sprachen und Implementierungen die Lösung „zusammenflicken“ zu müssen.

Als Paradigma bezeichne ich dabei die Sichtweise, die zur Lösung eines Problems mittels einer Programmiersprache eingenommen wird. Beispielsweise konstruiert der Benutzer einer funktionalen Programmiersprache ein Programm, indem er einzelne Funktionen definiert und diese geeignet kombiniert, während ein imperativ programmierender Benutzer die einzelnen Schritte formuliert, die – nacheinander ausgeführt – durch Änderungen des Programmzustandes zur Lösung des Problems führen. Diese unterschiedlichen Sichtweisen bewirken nicht nur, dass ein Algorithmus zur Problemlösung in unterschiedlichen Schreibweisen formuliert wird, es werden vielmehr unterschiedliche Algorithmen entwickelt: Budd

(1991) betont, dass der Unterschied zwischen einzelnen Paradigmen sich nicht nur auf die Syntax beschränkt, sondern dass es sich dabei um unterschiedliche „Weltbilder“ handelt.

Der Begriff „Paradigma“ lässt sich daher nur sehr schwer präzise definieren, und noch schwieriger ist es, „Multiparadigmen-Programmiersprachen“ zu definieren. In diesem Bericht soll daher auch keine präzise Einteilung von Programmiersprachen in „einzelparadigmatische“ und „multiparadigmatische“ Sprachen gegeben werden, genausowenig wie einzelne Sprachen genau einem Paradigma zugeordnet werden. Vielmehr soll der Schwerpunkt darauf liegen, in welchen Eigenschaften verschiedene Programmiersprachen sich ähneln und in welchen sie sich unterscheiden, sowie darauf, ob sich bei genauer Betrachtung scheinbar sehr unterschiedlicher Sprachen diese vielleicht doch unter einem allgemeineren Konzept zusammenfassen lassen.

Dieser Bericht soll eine ausführliche Übersicht über die Ansätze beinhalten, die zu Multiparadigmensprachen geführt haben. Weiterhin möchte ich die wesentlichen Eigenschaften dieser Entwürfe herausarbeiten und schließlich zu allgemeinen Entwurfsvorschlägen kommen, die in zukünftiger Arbeit zu einer Multiparadigmensprache führt. Diese Sprache soll syntaktisch und semantisch verschiedene Paradigmen so kombinieren, dass sie besser zur Implementierung großer Systeme mit unterschiedlichen Anforderungen geeignet ist als eine lose Kombination unterschiedlicher Sprachen.

## 1.1 Motivation

Dieser Abschnitt soll zur Einführung in das Thema einige Aspekte aus Forschung und praktischem Einsatz anreißen, die eine Beschäftigung mit dem Thema „Multiparadigmensprachen“ begründen und die Vorteile einer multiparadigmatischen Programmentwicklung verdeutlichen.

**Wartbarkeit und Erweiterbarkeit.** Praktisch alle größeren Softwaresysteme bestehen aus einer Vielzahl von Subsystemen, die unter Umständen sehr verschiedene Probleme lösen. Beispielsweise besteht ein modernes Datenbanksystem nicht nur aus der eigentlichen (meist relationalen) Datenbank, sondern auch aus Komponenten zur Interaktion mit den Benutzern, Verwaltungswerkzeugen und Programmierschnittstellen. Da diese unterschiedlichen Problemstellungen oft sehr verschiedene Lösungsansätze erforden, werden natürlich auch diverse Programmiersprachen zur Implementierung eingesetzt. Die Verwendung unterschiedlicher Notationen und Implementierungen erfordert immer einen erhöhten Aufwand zur Einarbeitung, Wartung und Integration. Der Einsatz einer Multiparadigmensprache verspricht Abhilfe, indem zwischen den einzelnen Einsatzgebieten lediglich die Denkweise, nicht aber die Notation bzw. Implementierung gewechselt werden muss.

**Programmdesign.** Krishnamurthi, Felleisen und Friedmann (1998) beschreiben, wie die Verwendung sowohl funktionaler als auch objekt-orientierter Techniken die Wiederverwendbarkeit von Programmen erhöht. Rekursiv definierte Datentypen (algebraische Datentypen und Klassen) eignen sich gut zum Modellieren vieler Problemstellungen. Wenn allerdings Erweiterungen dieser Datentypen zur Anpassung eines Programms notwendig werden, ergeben sich zwei Möglichkeiten: (1) dem Datentyp werden neue Varianten hinzugefügt, oder (2) neue Werkzeuge (Funktionen) zum Verarbeiten dieser Datentypen werden hinzugefügt.

Zur leichteren Durchführung solcher Erweiterung wird meistens eine der folgenden Designstrategien beim Entwurf des ursprünglichen Programms verwendet. Entweder der Aufbau des Programms ist vor allem „funktional“, basiert also auf Funktionen, die (z.B. mittels Pattern-Matching) die Datentypen verarbeiten, oder „objekt-orientiert“, d.h. dass die einzelnen Varianten eines Datentyps die dazugehörigen Operationen definieren. Der erste dieser Ansätze hat den Vorteil, dass leicht neue Werkzeuge hinzugefügt werden können, es ist aber schwierig, neue Varianten für die Datentypen zu definieren, weil dann alle Funktionen angepasst werden müssen. Beim objekt-orientierten Ansatz dagegen existiert das gegengesetzte Problem: Neue Varianten können (als Unterklassen) leicht hinzugefügt werden, ohne bestehende Klassen ändern zu müssen, es ist aber schwierig, neue Funktionen zu definieren. Dazu wären wiederum Änderungen an allen Klassen notwendig, die die einzelnen Varianten repräsentieren.

Eine Multiparadigmensprache, die sowohl funktionales und objekt-orientiertes Design als auch die Kombination dieser beiden Entwurfsvervahren unterstützt, ermöglicht daher die Entwicklung von Programmen, die leichter zu warten und zu erweitern sind als in einer konventionellen Sprache entwickelte Programme.

**Fortschritt in Wissenschaft und Entwicklung.** Auch die Wissenschaft und die Entwickler von Programmiersystemen versprechen sich vom Einsatz von Multiparadigmensprachen Vorteile. In der Literatur wird darauf hingewiesen, dass beispielsweise die Entwickler funktionaler und logischer Programmiersprachen oftmals die selben Probleme mehrfach lösen, weil es zwischen den einzelnen Forschungsgebieten zu wenig Austausch gibt (Lloyd, 1995). So gibt es sowohl auf theoretischer Seite (Typsysteme, Beschreibung der Semantik, Auswertungsstrategien) als auch auf praktischer (effiziente Implementierung) viele Überschneidungen, die bei gemeinsamer Forschung auf dem Gebiet der Multiparadigmensprachen vermieden werden könnten.

**Lehre.** Viele Autoren betonen die Eignung von Multiparadigmensprachen in der Lehre (Placer, 1993; Lloyd, 1995; Westbrook, 1999). Da von jedem Studierenden der Informatik erwartet wird, dass sie oder er sowohl die Prinzipien der imperativen als auch der deklarativen Programmierung erlernt, wird von einer Multiparadigmensprache erwartet, dass dies ohne die Notwendigkeit, verschiedene Programmiersysteme zu erlernen, schneller zum Erfolg führt. Die so gewonnene Zeit soll dazu genutzt werden, die den Paradigmen zugrundeliegenden Prinzipien zu verstehen und zu begreifen, dass ein Paradigma nicht von der verwendeten Programmiersprache abhängt, sondern von der Sichtweise auf eine Problemstellung und deren Lösung.

**Interesse der Forschung.** Das Interesse an Multiparadigmen-Programmierung beschränkt sich nicht nur auf einige, wenige Zweige der Informatik. Das zeigt sich in einer Vielzahl internationaler Konferenzen und Workshops, die sich unter anderem dieses Themas angenommen haben (International Conference on Functional Programming, ICFP; European Conference on Object-Oriented Programming, ECOOP; International Workshop on Multiparadigm Programming, MPOOL (Striegnitz u.a., 2002); Declarative Programming in the Context of Object-Oriented Languages, DP-COOL) oder sogar darauf spezialisiert sind (Workshop on Multiparadigm Logic Programming, MPLP (Chakravarty u.a., 1996); In-

ternational Workshop on Multiparadigm Constraint Programming Languages, MultiCPL (Hanus u.a., 2002)). Das Interesse der Forschung kann natürlich nur ein Indikator für ein allgemeines Interesse an diesem Thema sein und stellt für sich genommen keine Begründung dar, sich mit diesem Gebiet zu befassen. Es ist aber auch ein Anzeichen dafür, dass die Forschung auf den Einzelgebieten große Fortschritte gemacht hat, ohne allerdings alle programmiersprachlichen Probleme, die bei der Entwicklung großer Softwaresysteme auftreten, zufriedenstellend zu lösen.

Natürlich lassen sich diesen Begründungen auch Gegenargumente gegenüberstellen. So ist zu überlegen, ob es für angehende Informatiker nicht nützlich ist, wenn sie frühzeitig erlernen, sich innerhalb eines engen Zeitrahmens mit unterschiedlichen Systemen vertraut zu machen, da diese Fähigkeit in ihrem späteren Berufsleben sehr wichtig sein wird. Ebenso lässt sich die Ansicht vertreten, dass Forschung auf ähnlichen Gebieten, aber aus verschiedenen Blickwinkeln auch zu neuen Erkenntnissen führen kann, die bei einer einheitlichen Herangehensweise nicht entdeckt werden können.

## 1.2 Schwerpunkt

Natürlich kann in diesem Bericht nur ein Teilgebiet dessen, was sich unter Multiparadigmen-Programmierung zusammenfassen lässt, detailliert betrachtet werden. Der Schwerpunkt liegt auf den Multiparadigmen*sprachen*, also Programmiersprachen, die syntaktisch und semantisch die Multiparadigmen-Programmierung unterstützen. Verwandte Themengebiete – zum Beispiel Multiparadigmen-*Programmierumgebungen*, die die gemeinsame Verwendung verschiedener Programmiersprachen in einem Projekt durch Werkzeugintegration und definierte Schnittstellen zwischen den einzelnen Sprachen erlauben – sollen nur näher behandelt werden, wenn sie allgemeine Konzepte aufweisen, die auch auf Multiparadigmensprachen übertragbar sind.

Der wesentliche Beitrag dieses Berichts besteht einerseits in einer Bestandsaufnahme der Forschung und Entwicklung von Multiparadigmensprachen, andererseits in der Herausarbeitung wesentlicher Konzepte der beschriebenen Programmiersprachen, um so mögliche Ansätze einer sinnvollen Integration dieser Sprachen aufzuzeigen. Da bisher nur wenige Arbeiten zum spezifischen Gebiet der Multiparadigmen-Programmiersprachen und zur multiparadigmatischen Softwareentwicklung existieren – z.B. Budd u.a. (1995); Budd (1995); Vranić (2000); Hailpern (1987) – sollen die hier vorgestellten Untersuchungen thematisch eher in die Breite als in die Tiefe gehen und somit die Grundlage für weitergehende Arbeiten liefern. Dieser Bericht stellt auch das Ergebnis einer ausführlichen Literaturrecherche zum Thema Multiparadigmen-Programmierung und -Sprachen dar. Die verschiedenen Ansätze, die von den einzelnen Autoren verfolgt wurden, werden im Text gegenübergestellt und verglichen, und das Literaturverzeichnis soll zukünftiger Arbeit als Ausgangspunkt zu weiteren Untersuchungen dienen.

Ich verspreche mir von der Untersuchung multiparadigmatischer Sprachen ein besseres Verständnis von Programmiersprachen, Lösungsstrategien und -techniken, sowie der Implementierung und Integration verschiedener Paradigmen.

## 1.3 Gliederung

In Kapitel 2 werden zunächst die wichtigsten Begriffe definiert, die in diesem Bericht verwendet werden. Darauf folgt die Beschreibung einzelner Paradigmen und Paradigmenkombinationen sowie eine ausführliche Beschreibung existierender Multiparadigmen-Programmiersprachen. Dabei sollen die unterschiedlichen Ansätze auf ihre Vor- und Nachteile untersucht sowie die der Integration zugrundeliegenden Überlegungen betrachtet werden.

Kapitel 3 arbeitet die syntaktischen und semantischen Eigenschaften verschiedener Multiparadigmensprachen heraus und leitet daraus eine Vorschläge ab, die bei dem Entwurf solcher Sprachen befolgt werden sollten. Einige Aspekte (insbesondere die Integration deklarativer und imperativer Programmierung) werden auch etwas ausführlicher dargestellt, da sie in der existierenden Literatur weniger erschlossen sind.

Abgeschlossen wird dieser Bericht mit Kapitel 4, in dem zukünftige Arbeiten auf dem Gebiet der Multiparadigmen-Programmiersprachen vorgestellt werden und in dem der Inhalt des Berichts zusammengefasst wird.

# 2 Multiparadigmen-Programmiersprachen

Im Folgenden sollen verschiedene Programmiersprachen auf die Eigenschaften untersucht werden, durch die sie sich zur Multiparadigmen-Programmierung eignen. Einige der Sprachen wurden bewusst als Kombination verschiedener Paradigmen entwickelt, andere haben lediglich Aspekte anderer Sprachen übernommen, wenn dies sinnvoll erschien.

Nach der Definition der wesentlichen Begriffe dieser Arbeit wird zunächst ein Überblick über die verschiedenen untersuchten Einzelparadigmen gegeben. Anschließend werden in Abschnitt 2.3 grundlegende Überlegungen zum Entwurf multiparadigmatischer Sprachen dargestellt. Die darauf folgenden Abschnitte befassen sich mit verschiedenen Multiparadigmen-Ansätzen. Dabei werden zunächst die Kombinationen zweier Paradigmen betrachtet, dann werden in Abschnitt 2.11 Multiparadigmensprachen beschrieben, die Konzepte aus mehr als zwei Paradigmen entlehnt haben. Alle Paradigmen-Kombinationen werden durch Beispiele aus entsprechenden Programmiersprachen verdeutlicht. Abgeschlossen wird dieses Kapitel durch eine Zusammenfassung und kritische Diskussion der beschriebenen Ansätze.

## 2.1 Definitionen

Zunächst ist es sinnvoll, wesentliche Begriffe zu definieren. Die Begriffe *Paradigma* und *Multiparadigma* werden in diesem Bericht flexibel benutzt, um einen möglichst breiten Überblick über das Thema zu erhalten, dennoch soll zumindest eine weit gefasste Definition dieser Begriffe die Aussagen und Bemerkungen dieses Berichts verdeutlichen.

**Definition 1** *Ein Paradigma ist ein Musterbeispiel. (Berlin-Brandenburgische Akademie der Wissenschaften, 2003)*

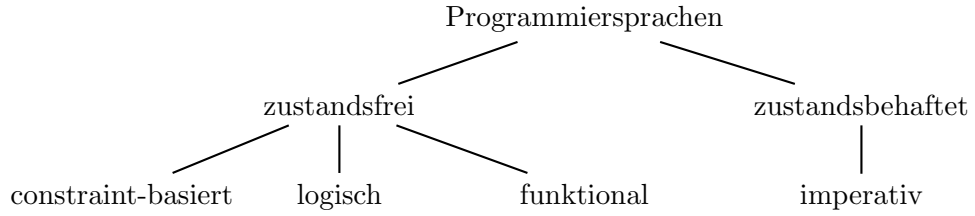
Damit repräsentiert ein Paradigma eine ganze *Klasse von Dingen*. In unserem Fall geht es um Problemlösungen, ein Paradigma repräsentiert also eine Sichtweise auf ein Problem. Da diese Definition zu allgemein ist, um im Zusammenhang mit Programmiersprachen damit zu arbeiten, beschränken wir uns im Folgenden auf Programmierparadigmen:

**Definition 2** *Ein Programmierparadigma ist eine Sichtweise, die zur Lösung eines Problems mittels einer Programmiersprache eingenommen wird.*

Eine Programmiersprache wird einem Paradigma zugeordnet, wenn sie Sprachmittel zur Verfügung stellt, um die der Sichtweise entsprechende Lösungsstrategie auszudrücken. Daraus ergibt sich die folgende Definition:

**Definition 3** *Eine Multiparadigmen-Programmiersprache ist eine Programmiersprache, die zur Problemlösung mit verschiedenen Programmierparadigmen die geeigneten Sprachmittel besitzt.*

**Abbildung 2.1** Programmiersprachen-Klassifizierung




## 2.2 Betrachtete Paradigmen

In diesem Abschnitt werden alle in diesem Bericht untersuchten Programmierparadigmen beschrieben, die im Sinne der obigen Definition die Bausteine für multiparadigmatische Sprachen darstellen. Abbildung 2.1 stellt die folgende Klassifizierung von Programmiersprachen grafisch dar.

Grob lassen sich Programmiersprachen zunächst in *zustandsbehaftete* und *zustandsfreie* Sprachen einteilen. Zustandsbehaftete Programmierung umfasst alle Sprachen, bei denen Daten neu erzeugt und im weiteren Programmverlauf explizit verändert werden, während bei der zustandsfreien Programmierung zwar Daten erzeugt werden, aber unverändert bleiben (Müller u.a., 1995). Beide Kategorien haben spezifische Vorteile, weshalb sie auch beide durch unterschiedlichste Programmiersprachen realisiert werden: die zustandsbehaftete Programmierung entspricht eher dem tatsächlichen Geschehen der realen Welt, während die zustandsfreie Programmierung vor allem leichter durch formale Methoden zu untermauern ist, beispielsweise durch die formale Definition der Semantik und die Programmverifikation durch Beweise.

Zustandsbehaftete Programmiersprachen sind prozedurale imperative Programmiersprachen wie Assemblersprachen, Fortran, Algol, Pascal, Modula, C, Ada, PL/1, Cobol, sowie objekt-orientierte imperative Sprachen wie Simula, Smalltalk, Eiffel, C++, Java und C#. Die zustandsfreien Sprachen (auch oft *deklarative Sprachen* genannt) werden im Wesentlichen in *funktionale*, *logische* und *constraint-basierte* Programmiersprachen[1] unterteilt. Dabei werden die logischen und constraint-basierten Sprachen mittlerweile als eine Untermenge der mächtigeren *constraint-logischen* Sprachen angesehen. Als Beispiele für funktionale Sprachen sollen (reines) Lisp, Standard ML, Miranda und Haskell dienen, für die logischen Sprachen klassisches Prolog sowie modernere Varianten wie Eclipse, Sicstus Prolog oder Mercury. Reine constraint-basierte Sprachen gibt es nur wenige, z.B. CONSTRAINTS (Sussman und Steele, 1980).

Während die obige Klassifikation vor allem darauf basiert, mit welcher Sichtweise Algorithmen zur Problemlösung formuliert werden, lassen sich Programmiersprachen aber auch danach unterteilen, wie einzelne Berechnungsschritte kombiniert werden. Dabei werden sequenzielle und nebenläufige Berechnungsverfahren unterschieden, deren Nutzung natürlich auch von Programmiersprachen unterstützt wird. Erweiterungen der Nebenläufigkeit sind die Parallelität und die Verteiltheit, bei denen einzelne Berechnungseinheiten nicht nur (konzeptionell) gleichzeitig, sondern tatsächlich physikalisch gleichzeitig bzw. auf verschiedenen

[1] Müller u.a. (1995) sprechen in diesem Zusammenhang auch von *gezielten* und *ungezielten* Berechnungen.

**Abbildung 2.2** Betrachtete Paradigmen

- funktional
- logisch
- constraint-basiert
- imperativ
- objekt-orientiert
- aspekt-orientiert
- access-orientiert
- intentional
- nebenläufig
- verteilt
- generisch
- generativ
- literate

Prozessoren oder Rechnern ausgeführt werden.

Da das Konzept der zustandsbehafteten bzw. -freien Programmierung prinzipiell unabhängig von eventueller Nebenläufigkeit oder Verteiltheit ist, existieren praktisch von allen Sprachparadigmen sowohl sequenzielle als auch nebenläufige Vertreter, oft auch verteilte. Es sei aber angemerkt, dass einige Paradigmen sich besonders für bestimmte Erweiterungen eignen. So lassen sich zustandsfreie Sprachen prinzipbedingt besser parallel implementieren, da die deklarative Sicht von vornherein auf eine zeitliche Abfolge einzelner Berechnungseinheiten verzichtet. Ähnliches gilt für die objekt-orientierte Programmierung: sowohl für imperative als auch für deklarative Programmiersprachen existieren entsprechende Erweiterungen.

Bei der weiteren Betrachtung sollen die Paradigmen in Abbildung 2.2 unterschieden werden.

Natürlich lassen sich wenige Programmiersprachen eindeutig einem bestimmten Paradigma zuordnen, ebensowenig wie alle Programmiersprachen eines Paradigmas identische Eigenschaften haben (beispielsweise werden funktionale Sprachen in *strict* und *lazy* Programmiersprachen unterteilt, objekt-orientierte Sprachen können entweder klassen- oder delegationsbasiert sein oder gar keine Vererbung zur Verfügung stellen). Die Zuordnung einzelner Sprachen zu Programmierparadigmen soll daher auf ihren wesentlichen Eigenschaften basieren, sowie darauf, wie sie von ihren Entwicklern eingeordnet wurden.

Im Folgenden sollen die oben aufgeführten Programmierparadigmen genauer betrachtet werden. Neben einer Beschreibung ihrer spezifischen Eigenschaften sollen auch ihre Vor- und Nachteile sowie einige Programmiersprachen, die sich den einzelnen Paradigmen zuordnen lassen, aufgeführt werden.

### Funktionale Programmierung

Unter dem Begriff der *funktionalen Programmiersprachen* sollen alle Sprachen zusammengefasst werden, bei denen die Definition, Applikation und Komposition von Funktionen sowie Funktionen höherer Ordnung zur Verfügung stehen und die wichtigsten Mechanismen zur Programmierung sind.

Als Unterteilungsmerkmale für funktionale Sprachen werden einerseits der Grad der „Reinheit“ (Abwesenheit von Seiteneffekten) sowie der Auswertungsmechanismus von Aus-

drücken (*strict* bzw. *call-by-value*, *lazy* bzw. *call-by-need* oder *call-by-name*) betrachtet. Vertreter der „reinen“ funktionalen Sprachen betrachten *referenzielle Transparenz* oft als ein wichtiges Kriterium von funktionalen Sprachen, da es Korrektheitsbeweise und Progammtransformationen erheblich vereinfacht. In den funktionalen Sprachen, die Seiteneffekte erlauben, gilt dieses Prinzip nicht. Sabry definiert ein präzises Kriterium für rein funktionale Sprachen:

> *A language is purely functional if (i) it includes every simply typed λ-calculus term, and (ii) its call-by-name, call-by-need, and call-by-value implementations are equivalent (modulo divergence and errors).* (Sabry, 1998, S. 2)

Funktionale Programme werden im Allgemeinen durch Termreduktion ausgeführt, d.h. dass das Programm aus einer Menge von Termersetzungsregeln sowie einem Ausdruck besteht, welcher zum Ergebnis umgeformt werden soll. Dann werden, solange möglich, Termersetzungen anhand der gegebenen Regeln ausgeführt, bis der Ausdruck eine Form erreicht hat, die nicht weiter reduziert werden kann. Eine solche Form wird *Normalform* genannt.

Ein weiteres wichtiges Merkmal funktionaler Sprachen ist die Verwendung *algebraischer Datentypen*, das sind Datentypen, die durch ihre Konstruktoren definiert werden und rekursive Summen- oder Produkttypen sein können. Durch *Pattern-Matching* können Funktionen anhand der Struktur der zu bearbeitenden Daten definiert werden.

Die Vorteile der funktionalen Programmierung basieren im Wesentlichen auf ihrer sauberen mathematischen Fundierung, die zur Programmtransformation und für Korrektheitsbeweise genutzt werden kann. Außerdem lassen sich funktionale Programme aufgrund der deterministischen Termersetzung effizient ausführen (Hanus, 1997).

Die Anbindung funktionaler Sprachen an externe Systeme ist aufgrund ihrer deklarativen Natur schwierig, da die zeitlichen Abhängigkeiten der Realität irgendwie auf das Programm abgebildet werden muss. In Abschnitt 3.4 wird beschrieben, wie dieses Problem gelöst wird. Gegenüber der logischen Programmierung hat die funktionale Programmierung den Nachteil, dass sie nur deterministische Reduktionen erlaubt und Nichtdeterminismus explizit nachgebildet werden muss, was bei vielen Problemen, die mehrere Lösungen haben können (z.B. Such- oder Planungsprobleme), zu umständlicheren Programmen führt als bei logischen Sprachen.

Beispiele funktionaler Programmiersprachen sind Backus' FP (Backus, 1978), Haskell (Peyton Jones, 2003), Standard ML (Milner u.a., 1997), Common Lisp (Steele, 1990), Scheme (Kelsey u.a., 1998) und Opal (Pepper, 2003).

#### Logische Programmierung

Die logische Programmierung baut auf die Prädikatenlogik auf. Programme bestehen dabei aus einer Menge von Fakten und Regeln, anhand derer das System die Erfüllbarkeit bzw. Nichterfüllbarkeit von Anfragen berechnen kann. Bei dieser Berechnung werden Variable an Werte gebunden, die dann das Ergebnis der Berechnung darstellen. Im Zuge der Berechnung kann es nötig sein, mehrere alternative Berechnungsstränge abzuarbeiten, um nach einer Lösung zu suchen bzw. um alle Lösungen zu ermitteln. In den meisten Implementierungen logischer Sprache wird dies über eine Tiefensuche mittels *Backtracking* realisiert. Dabei wird an jedem *Wahlpunkt* (engl. *choice point*) der Programmzustand gesichert und beim Fehlschlagen einer Berechnung wieder restauriert.

Das Verfahren, das zur Auswertung logischer Sprachen verwendet wird, heißt Resolution und wird durch Unifikation und Substitution implementiert.

Wie Hanus (1997) ausführt, bietet die logische Programmierung die Vorteile, dass durch die Verwendung von logischen Variablen mit partiellen Datenstrukturen gearbeitet werden kann, sowie dass mittels der eingebauten Suche mehrere Lösungen eines Problems berechnet werden können.

Gegenüber der funktionalen Programmierung sind logische Programmiersprachen aufgrund des inhärenten Nichtdeterminismus weniger effizient. In einigen Implementierung werden aufwändige Analysen der Programme durchgeführt, um deterministische Programmteile herauszufinden und diese dann effizienter zu übersetzen. Bei manchen logischen Sprachen hat der Programmierer die Möglichkeit, durch Anmerkungen im Programmtext den Übersetzungsvorgang einzelner Prädikate zu steuern (z.B. bei Mercury, siehe Abschnitt 2.6.4). Weiterhin bestehen natürlich die gleichen Probleme in Bezug auf die Interaktion mit externen Systemen wie bei funktionalen Sprachen.

Prolog war die erste verbreitete logische Programmiersprachen, weitere Vertreter dieses Paradigmas sind die Prolog-Nachfolger Prolog II und Prolog III (Colmerauer, 1990) sowie Mercury (Somogyi u.a., 1995).

#### Constraint-basierte Programmierung

Bei der constraint-basierten Programmierung bestehen Programme im Wesentlichen aus der Spezifikation von Einschränkungen bzw. Randbedingungen (engl. *constraints*), die das zu lösende Problem deklarativ beschreiben. Idealerweise ist es die Aufgabe des Programmiersystems, aus dieser Spezifikation die Lösung(en) zu berechnen. Da constraint-basierte Programmiersprachen oft eingesetzt werden, um extrem aufwändige Probleme zu lösen (z.B. Such- und Planungsprobleme), geben alle Systeme den Programmierern die Möglichkeit, Such- und Berechnungsstrategien zu beeinflussen.

Mittlerweile werden die logischen Programmiersprachen als eine Untermenge der constraint-logischen Sprachen angesehen. Dabei basieren die logischen Programmiersprachen auf dem speziellen Constraint-System der unendlichen Bäume.

Die meisten modernen Prolog-Dialekte (Mercury, Eclipse etc.) wurden mittlerweile um die Fähigkeit zum Umgang mit Constraints erweitert. Reine Constraintsprachen sind selten, ein Beispiel ist CONSTRAINTS (Sussman und Steele, 1980).

#### Imperative Programmierung

Die imperative Programmierung basiert darauf, dass jedes Programm bei der Ausführung einen Zustand besitzt, der während des Programmablaufs durch Anweisungen modifiziert wird. Der Programmzustand besteht zum einen aus dem Arbeitsspeicher, auf dem das Programm arbeitet, zum anderen schließt er aber auch die gesamte Umgebung (Peripherie, Dateisystem etc.) mit ein. Diese Modifikation des Zustandes wird durch Seiteneffekte (Verändern von Speicherzellen, Ein-/Ausgabe) bewirkt.

Die *prozedurale Programmierung* ermöglicht das Zusammenfassen von Anweisungen zu Unterprogrammen bzw. Prozeduren. Diese Prozeduren können dann von verschiedenen Stellen des Programms aufgerufen werden, was zur Ausführung der Anweisungen im Prozedurrumpf führt. Da Unterprogramme beinahe so alt sind wie die Programmierung selbst, sind

praktisch alle imperativen Sprachen auch prozedural, weshalb die beiden Begriffe in diesem Bericht synonym verwendet werden.

Imperative Sprachen haben den Vorteil, dass es aufgrund ihrer weiten Verbreitung gute Werkzeuge und Compiler gibt. Weiterhin gibt es sehr viele erprobte Bibliotheken und effiziente Algorithmen. Das Hauptargument zur Benutzung imperativer Sprachen ist heutzutage deren höhere Geschwindigkeit, die von deklarativen Sprachen im Allgemeinen nicht erreicht werden kann (vgl. Okasaki (1999)).[2]

Im Gegensatz zu deklarativen Sprachen ist vor allem das Beweisen der Korrektheit eines Programms wesentlich schwieriger, weil einerseits die anweisungsorientierte Abarbeitung imperativer Programme wesentlich kleinschrittiger ist und andererseits durch veränderbare Variablen Invarianten schwieriger zu definieren und zu prüfen sind. Dazu kommt, dass es für die Semantik der meisten verwendeten imperativen Sprachen keinerlei formale Definition gibt, die das Verhalten eines Programms in dieser Sprache festlegt.

Algol, Pascal, Modula, C, Ada, PL/1, Cobol und Fortran zählen zu den imperativen Programmiersprachen, es gibt aber für fast alle dieser Sprachen auch objekt-orientierte Erweiterungen.

### Objekt-orientierte Programmierung

Objekte in der *objekt-orientierten Programmierung* sind konzeptionell Einheiten, die einen Zustand und ein Verhalten besitzen. Der Zustand besteht im Allgemeinen aus einer Menge von veränderbaren Variablen (Attributen), während das Verhalten über Operationen definiert wird, welche diesen Zustand manipulieren (Methoden). Ein objekt-orientiertes Programm besteht aus einer Menge von Objekten, die durch die Anwendung von Methoden mit anderen Objekten interagieren. In der Literatur wird diese Methodenanwendung oft als *Nachrichtenaustausch* bezeichnet. Da Objekte die Attribute anderer Objekte nur durch Methodenanwendungen beeinflussen und auslesen können, bilden Objekte eine Modularisierungsmöglichkeit, die nur kontrollierten Zugriff auf den internen Zustand eines Objekts erlaubt.

Ein weiteres wichtiges Merkmal vieler objekt-orientierter Sprachen ist die Möglichkeit, gleiches Verhalten mehrerer Objekte durch die Definition von *Klassen* zusammenzufassen. Eine Klasse beschreibt eine Menge von Attributen und Methoden und kann als „Schablone“ für die Erzeugung neuer Objekte benutzt werden. Objekt-orientierte Sprachen, die solche Klassendefinitionen zulassen, werden *klassenbasiert* genannt. Andere Sprachen, oft als *objektbasiert* bezeichnet, kennen das Konzept „Klasse“ nicht; bei diesen Sprachen wird ähnliches Verhalten mehrerer Objekte entweder dadurch ermöglicht, dass neue Objekte aus bestehenden durch das Kopieren von Attributen und Methoden erzeugt werden (*Cloning*), oder dass Objekte bestimmte Methodenaufrufe zur Bearbeitung an andere Objekte weiterreichen (*Delegation*).

Sowohl bei klassen- als auch objektbasierten Sprachen ist es möglich, von bestehenden Klassen/Objekten neue Klassen/Objekte als Spezialisierungen abzuleiten, indem diese neuen Klassen das Verhalten der bestehenden „erben“, aber einige der geerbten Methoden umdefinieren (überschreiben) bzw. neue Methoden hinzufügen.

[2] Für viele Zwecke ist allerdings die Geschwindigkeitsdifferenz vernachlässigbar und als Argument gegen deklarative Sprachen nicht haltbar.

Objekte werden weiterhin in *passive* und *aktive* Objekte unterschieden. Passive Objekte verändern nur dann ihren Zustand und führen Berechnungen aus, wenn sie durch Methodenaufrufe dazu aufgefordert werden, während aktive Objekte unabhängig von anderen Objekten Berechnungen ausführen und mit anderen Objekten kommunizieren. Oft werden aktive Objekte auch als *Agenten* bezeichnet. Agenten werden in vielen Bereichen eingesetzt, wo durch die Verwendung vieler autonom rechnender Einheiten, die mit wenig Kommunikation und ihrer lokalen Sicht auf ein Problem arbeiten, komplexe Probleme effizient gelöst werden können. Agentensysteme eignen sich wegen der autonom arbeitenden Teilsysteme, die möglichst wenig miteinander kommunizieren, auch sehr für verteilte Implementierungen (siehe unten).

Die Aufteilung von Programmen in Objekte, die jeweils lokal mit ihren Attributen arbeiten und über wohldefinierte Schnittstellen mit anderen Objekten kommunizieren, führen in vielen Fällen zu einer guten Modularisierung und gut verständlichen und wartbaren Programmen. Außerdem lassen sich Objekte der realen Welt oft hervorragend auf Objekte in einer objekt-orientierten Sprache abbilden.

Andererseits werden Klassen in vielen Sprachen nicht nur zur Beschreibung von Objekten, sondern auch zur Modularisierung von Programmen verwendet. Dies führt nicht immer zu einer optimalen Klassen- und Modulstruktur. Ein weiterer Nachteil besteht darin, dass sich nicht alle Probleme angemessen durch Objekte modellieren lassen. Beispielsweise ist die Darstellung von Relationen zwischen mehreren Objekten als Methoden einer Klasse nicht die natürliche Darstellung, da eine Relation keiner ihrer Argumente mehr oder weniger zugeordnet werden kann.

Als erste objekt-orientierte Sprache wird in der Literatur Simula (Dahl und Nygaard, 1966) beschrieben, doch die erste auch heute noch populäre Sprache dieses Paradigmas ist Smalltalk (Goldberg und Robson, 1983). Mittlerweile existieren von fast allen verbreiteten Programmiersprachen objekt-orientierte Erweiterungen, so z.B. für Common Lisp, Cobol und Fortran. C++ und Objective C sind aus der imperativen Sprache C hervorgegangen, und neuere Varianten aus dieser Sprachfamilie sind Java und C#.

### Aspekt-orientierte Programmierung

Zu lösende Probleme werden in den meisten Paradigmen in kleinere, funktional zusammenhängende Teilprobleme aufgeteilt, getrennt gelöst und dann diese Teillösungen zu einer Lösung des Gesamtproblems zusammengefasst. Bei der aspekt-orientierten Programmierung (Kiczales u.a., 1997) betrachtet man nicht nur die funktionalen Zusammenhänge, sondern auch andere Abhängigkeiten, die bei einer Aufteilung nach funktionalen Kriterien „quer“ zur Programmstruktur stehen. So lassen sich beispielsweise viele Programme gut in einzelne Funktionen aufteilen, bei der immer eine Funktion das Ergebnis einer anderen verarbeitet. Sobald man aber eine Fehlerbehandlung hinzufügen will, müssen alle Funktionen so angepasst werden, dass sie diese Fehlerbedingungen berücksichtigen können. Auf diese Weise wird die zunächst klare funktionale Dekomposition des Problems wirr und unübersichtlich.

In der aspekt-orientierten Programmierung werden deshalb Komponenten und Aspekte unterschieden. Komponenten bestehen aus funktional abhängigen Programmteilen, die in einer prozeduralen Sprache programmiert werden. Aspekte werden dagegen in einer separaten Aspektsprache beschrieben und es gibt ein Werkzeug, das aus den Komponenten und den Aspekten das vollständige Programm generiert.

Die Implementierungen der aspekt-orientierten Programmierung unterstützen verschiedene Aspekte, z.B. Synchronisation nebenläufiger Prozesse und Methodenaufrufe zwschen verteilten Objekten in AspectJ, einer aspekt-orientierten Erweiterung von Java. Leider zeichnen sich die Umsetzungen verschiedener Aspekte durch unterschiedliche Notationen aus. Die Kombination mehrerer Aspekte in einem Programm ist aufgrund der unterschiedlichen eingesetzten Werkzeuge ebenfalls schwierig.

### Access-orientierte Programmierung

Die access-orientierte Programmierung (Stefik u.a., 1986) erweitert bestehende Programmierparadigmen um Anmerkungen (engl. *annotations*). Diese sind entweder Eigenschaftslisten (engl. *property lists*) oder aktive Werte (engl. *active values*) und können Variablen zugeordnet werden. Eigenschaftslisten können beliebige Informationen enthalten, die den Variablen zugeordnet sein sollen. Aktive Werte sind Funktionen bzw. Prozeduren, die immer aufgerufen werden, wenn der Wert der zugehörigen Variablen gelesen oder geschrieben wird.

Wichtig sind bei der access-orientierten Programmierung die folgenden Eigenschaften. Zunächst sind Anmerkungen für Programme, die nicht mit ihnen arbeiten, unsichtbar, stören also nicht den normalen Programmablauf. Weiterhin haben Anmerkungen nur geringen negativen Einfluss auf die Effizienz. Ein wesentlicher Punkt ist die Tatsache, dass Anmerkungen rekursiv sein können, dass sie also selbst auch wieder Anmerkungen besitzen können. Um die Programmierung mit Anmerkungen zu vereinfachen, können sie einfach und effizient direkt von den betreffenden Variablen, auf die sie sich beziehen, verwaltet werden. Es sind also keine umständlichen Datenstrukturen zu bearbeiten, die Variablen und Anmerkungen miteinander verbinden. Schließlich sind Anmerkungen normale Objekte, die spezialisiert werden können und die durch Attribute einen eigenen Zustand verwalten können.

Access-orientierte Programmierung wird beispielsweise durch das Loops-Programmiersystem (Stefik u.a., 1986) unterstützt. Dieses basiert auf Lisp und damit auf der funktionalen Programmierung und wurde um regelbasierte und objekt-orientierte Programmierung erweitert. Ein Beispiel für die Anwendung access-orientierter Sprachen ist eine Simulation, bei der das simulierte (interne) Modell durch Anmerkungen mit einer grafischen Darstellung verbunden wird, die bei jeder Änderung des Modells aktualisiert werden kann.

Aus heutiger Sicht könnte man die access-orientierte Programmierung der aspekt-orientierten Programmierung als Teilgebiet unterordnen, wenn man die Verbindungen zwischen Variablen und von den Variablenwerten abhängigen Objekte als Aspekt betrachtet.

### Intentionale Programmierung

Die *intentionale Programmierung* (IP) (Simonyi, 1995) stellt die Quellen eines Programmsystems in das Zentrum der Entwicklung. Sogenannte „aktive“ Quellen, die ihr eigenes Verhalten in Bezug auf Bearbeitung, Ansicht, Kompilierung und Debugging definieren, sollen dem Entwicklungsprozess zu mehr Flexibilität, Erweiterbarkeit und Produktivität verhelfen. IP umfasst daher nicht nur eine Programmiersprache, sondern ein komplettes Entwicklungssystem, das sich durch Erweiterungsbibliotheken an sich ändernde Bedingungen anpassen lassen soll. Die Bibliotheken können Spracherweiterungen, Bearbeitungs- und Debugging-Funktionen oder neue Compiler enthalten. Die Sprachabstraktionen, die IP zur

Verfügung stellt, werden *Intentionen* genannt. Intentionen können durch Hinzufügen von Methoden erweitert werden, allerdings werden diese Methoden im Gegensatz zu Klassenmethoden objekt-orientierter Sprachen zur Kompilierzeit ausgeführt, um auf den Quellen des Programms (repräsentiert als Graph) zu operieren.

Ein IP-Programmiersystem stellt zunächst keine verschiedenen Paradigmen zur Programmierung zur Verfügung. Es beinhaltet aber die notwendige Infrastruktur, um das System an verschiedene Anforderungen anpassen zu können, beispielsweise durch die Integration von Interpretern oder Compilern für verschiedene Paradigmen. Interessant im Zusammenhang mit diesem Bericht ist der Ansatz, sich bei Entwicklung nicht auf Sprachkonstrukte wie „Prozeduren“ oder „Datentypen“, sondern auf die „Intentionen“ des Programmierers zu konzentrieren. Da IP aber alle Paradigmen zulässt, aber nicht deren Interaktion definiert, wird dieser Ansatz im folgenden nicht weiter betrachtet.

### Nebenläufige Programmierung

In der nebenläufigen Programmierung laufen mehrere Stränge eines Programm konzeptionell unabhängig voneinander und evtl. sogar gleichzeitig ab. In der Implementierung kann diese Nebenläufigkeit entweder auf einer sequenziellen Maschine simuliert, oder aber tatsächlich parallel auf mehreren Prozessoren realisiert werden. Bei der Parallelverarbeitung unterscheidet man noch Datenparallelität, bei der ein Programm gleichzeitig auf mehreren Daten arbeitet, und Taskparallelität, bei der mehrere unterschiedliche Programme gleichzeitig abgearbeitet werden. Bei der Taskparallelität müssen die einzelnen Programme miteinander kommunizieren, was zu ähnlichen Problemen führt wie bei der verteilten Programmierung (siehe unten).

Die Synchronisation nebenläufiger Programmteile erfordert einen nicht zu vernachlässigenden Aufwand. Außerdem ist der Ablauf eines Programms schwer vorauszusehen bzw. nachzuvollziehen, was vor allem bei der Fehlersuche Schwierigkeiten verursacht. Besonders problematisch ist dabei, dass minimale Änderungen am Programm (z.B. eine Anweisung zur Ausgabe des Programmzustands) bereits das Verhalten nebenläufiger Programmstränge so verändern kann, dass ein Fehler nicht mehr oder in anderer Form auftritt.

### Verteilte Programmierung

Verteilte Programme werden nicht nur auf einem einzelnen Prozessor, sondern auf mehreren Prozessoren oder auch verteilt auf mehreren Rechnern ausgeführt. Die einzelnen verteilten Programmteile können dabei gleiche oder auch unterschiedliche Programme ausführen und auf den gleichen oder unterschiedlichen Daten operieren. Wichtig ist bei der verteilten Programmierung die Kommunikation zwischen den verteilten Komponenten, die den Programm- und Datenfluss steuert bzw. durchführt. Diese Kommunikation kann entweder explizit durch den Austausch von Nachrichten erfolgen, oder durch die Verwendung von Speicher, der von den einzelnen Komponenten gemeinsam ausgelesen und beschrieben werden kann. Gemeinsamer Speicher bietet sich an, wenn die einzelnen Prozessoren physikalisch eng miteinander verbunden sind, bei der Verwendung eines Netzwerks zur Verbindung verschiedener Rechner werden meistens Nachrichten verschickt. Ähnlich wie bei der nebenläufigen Programmierung muss ein verteiltes Programmiersystem Mechanismen zur Synchronisation einzelner Programmstränge zur Verfügung stellen.

Der Vorteil der verteilten Programmierung besteht in der Geschwindigkeitssteigerung, da mehrere Prozessoren natürlich schneller ein Problem lösen können als ein einzelner. Voraussetzung ist, dass sich das Problem in geeigneter Form aufteilen lässt, und dass bei der Lösung wenig Kommunikation zwischen den beteiligten Prozessoren notwendig ist, um den Geschwindigkeitszuwachs durch zusätzliche Hardware nicht durch den Kommunkationsaufwand zunichte zu machen.

Die Aufteilung eines Programms in mehrere verteilte Teilsysteme ist sehr aufwändig und erfordert unter Umständen völlig neue Algorithmen. Ähnlich wie bei der oben genannten nebenläufigen Programmierung ist auch bei der verteilten Programmierung der Programmablauf von vielen Details (bei der verteilten Programmierung z.B. zusätzlich die Zeit zur Nachrichtenübertragung und Latenz im Netzwerk) abhängig und schwer nachzuvollziehen.

### Generische Programmierung

Generische Programmierung (Backhouse u.a., 1999; Jansson, 2000) soll Programme auf einem sehr hohen Abstraktionsniveau ermöglichen, so dass beispielsweise Funktionen automatisch auf beliebigen induktiv definierten Datentypen arbeiten können. Der Vorteil dieses Vorgehens besteht in der hohen Wiederverwendbarkeit und der Modularität generischer Algorithmen.

PolyP ist eine Programmiersprache zur generischen (bzw. polytypischen) Programmierung (Jansson, 2000). Die generische Programmierung dient der Definition generischer Funktionen, die beispielsweise auf Werte verschiedener Datentypen angewendet werden können, ohne deren Struktur kennen zu müssen. Realisiert wird dies, indem die Funktionen nicht durch Pattern-Matching über der Struktur der Datentypen definiert werden, sondern über ein Matching der den Datentypen zugrundeliegenden Strukturen (Summentypen, Produkttypen etc.). Ein Beispiel für die Nutzung generischer Programmiersprachen ist ein generischer Unifikationsalgorithmus, der von Backhouse u.a. (1999) beschrieben wird. Dieser Unifikationsalgorithmus erhält die Struktur der Terme, die er unifizieren soll, als Parameter, und kann so Terme beliebiger Darstellung verarbeiten.

Da dieses Programmierparadigma sich ähnlich wie Verteiltheit oder Nebenläufigkeit orthogonal zu den klassischen Paradigmen verhält, wollen wie nicht näher darauf eingehen.

### Generative Programmierung

Die generative Programmierung (Czarnecki und Eisenecker, 2000) ist ein abstrakteres Konzept, das z.B. der aspekt-orientierten Programmierung übergeordnet wird. Die generative Programmierung umfasst alle Programmiersprachen, -techniken und -systeme, die die automatische Herstellung von Software aus geeigneten Komponenten ermöglichen. Ein wichtiger Aspekt ist dabei die Wiederverwendbarkeit, Konfiguration und Kombinationsfähigkeit der Komponenten.

Dieser Ansatz soll in diesem Bericht nicht weiter verfolgt werden, da der Schwerpunkt auf Multiparadigmensprachen – und damit der Formulierung von Algorithmen – und nicht auf den softwaretechnischen Anforderungen an ein Programmiersystem liegt.

### Literate Programming

*Literate Programming* wurde von Knuth (1983) entwickelt, ursprünglich, um ihm die Ar-

beit an seinem TEX-System zu erleichtern. Bei diesem Verfahren steht die Dokumentation eines Programms bei der Programmierung im Mittelpunkt, das Programm wird also so strukturiert, dass es bestmöglich zu lesen und zu verstehen ist. Um die Lesbarkeit zu unterstützen, wird die Dokumentation und der Programmtext gemeinsam geschrieben und es ist die Aufgabe zweier Übersetzungsprogramme, aus diesem integrierten Text einerseits die Dokumentation, andererseits den Programmtext zur Weiterverarbeitung durch einen Compiler zu extrahieren. Das erste Programm erzeugt seine Ausgabe in der Textsatzsprache TEX, so dass sie zu einer gut lesbaren Druckvariante weiterverarbeitet werden kann. Das zweite Programm entfernt die Dokumentation aus seiner Eingabe und setzt das Ausgabeprogramm aus den Programmfragmenten zusammen, in die das Programm zur besseren Bearbeitung und Lesbarkeit in der Eingabedatei aufgeteilt werden kann.

Die Beschreibung von Haskell (Peyton Jones, 2003) sieht eine ähnliche, aber stark vereinfachte Form des *Literate Programming* vor. Neben normalem Haskell-Code können Haskell-Compiler auch sogenannte *literate Haskell scripts* verarbeiten. Diese bestehen aus LATEX-Code, in dem Haskell-Fragmente durch Markierungen hervorgehoben werden (entweder durch das Einschließen in

```
\begin{code} ... \end{code}
```

oder das Voranstellen eines >-Zeichens vor jeder Zeile).

Auch das *Literate Programming* lässt sich problemlos mit anderen Paradigmen kombinieren, da es lediglich die Darstellung von Programmen betrifft, nicht aber deren Ausführung.

## 2.3 Design-Überlegungen

Bevor wir in den nächsten Abschnitten zu Kombinationen der beschriebenen Paradigmen kommen, sollen zunächst einige Überlegungen zum Entwurf von Multiparadigmen-Programmiersprachen besprochen werden. Auf einige dieser Prinzipien und Konzepte wird bei der Beschreibung der Paradigmen-Kombinationen verwiesen. In Kapitel 3 werden sie erneut augegriffen und vertieft, wenn die Syntax und Semantik von Multiparadigmensprachen beschrieben werden.

Hailpern (1987) beschreibt grundlegende Überlegungen zum Design von Multiparadigmensprachen und benennt sowohl ein objekt-orientiertes Berechnungsmodell, auf dem eine solche Sprache aufgebaut werden kann, als auch Mechanismen zur Erweiterung auf andere Paradigmen, z.B. funktionale und logische Programmierung.

Das Buch von Budd (1995) befasst sich ausführlich mit den Eigenschaften verschiedener Multiparadigmensprachen und den Überlegungen, die zum Design der Programmiersprache Leda (Budd, 1991) geführt haben.

Eine wichtige Frage, die sich beim Entwurf einer Multiparadigmensprache zunächst stellt, ist, ob es wirklich sinnvoll ist, eine neue Programmiersprache zu entwickeln, oder ob es nicht evtl. effektiver ist, eine bestehende Programmiersprache durch bestehende Erweiterungsmechanismen um ein Paradigma zu erweitern. Natürlich setzt letzteres Vorgehen die Existenz eines geeigneten Erweiterungsmechanismus voraus. Beispielsweise wurde C++ unter Verwendung von Operator-Überladung durch die Constraint-Bibliothek ILOG (Puget, 1994) so erweitert, dass mit relativ geringem syntaktischen Aufwand Constraint-Programmierung (wenn auch in eingeschränktem Umfang) möglich ist. Common Lisp eignet sich aufgrund

der einfachen Syntax von Lisp und dem mächtigen Makro-System ebenfalls für solche Erweiterungen. Screamer (Siskind und McAllester, 1993) ist ein Makro-Paket für Common Lisp, das die Sprache um nichtdeterministische und Constraint-Programmierung erweitert.

Spinellis u.a. (1995) befassen sich zwar mit Multiparadigmen-Umgebungen und nicht speziell mit Sprachen, diskutieren aber einige wichtige grundlegende Überlegungen. Eine erfolgreiche Umsetzung der Multiparadigmen-Programmierung in dem von Spinellis u.a. vorgestellten System erfordert die Betrachtung und Lösung verschiedener Probleme:

**Unterschiedliche Notation.** Für jedes Paradigma sollte die bestmöglichste Notation verfügbar sein, um das Paradigma zu betonen und die Portierung von Programmen in diesem Paradigma zu erleichtern.

**Unterschiedliche Ausführung.** Prinzipiell lassen alle Paradigmen sich als Turing-Maschine modellieren, daher ist eine Integration theoretisch kein Problem. Praktisch relevant sind aber Effizienzprobleme.

**Implementierungsstrategien.** Manche Paradigmen eines Systems werden durch Interpretation, andere durch Kompilierung, wieder andere durch eine Zwischenform realisiert. Ein Multiparadigmensystem muss all diese Strategien unterstützen.

**Beliebige Paradigmen-Kombinationen.** Die Integration sollte sich nicht nur auf bestimmte Paradigmen beschränken, die evtl. sowieso gut zueinander passen. Weiterhin muss die Integration transparent sein, so dass es für den Benutzer einer bestimmten Bibliothek unwichtig ist, in welchem Paradigma diese programmiert wurde.

Weiterhin weisen Spinellis u.a. auf ein Defizit vieler bestehender Multiparadigmensprachen hin: Diese lösen immer nur ein Teilproblem und sind unflexibel, beispielsweise nicht erweiterbar auf weitere Paradigmen.

Ein wichtiges Kriterium für den Programmiersprachenentwurf, das die Ausdrucksfähigkeit einer Sprache stark beeinflussen kann, ist die Menge der Sprachkonstrukte, die innerhalb der Sprache selbst verarbeitet werden kann. Backhouse u.a. (1999) geben diesen Konstrukten den Namen „First-class Citizen“:

Ein „First-class Citizen“ ist etwas, das

- benannt ist und mit diesem Namen verwendet werden kann,
- als Parameter angegeben werden kann, und
- „anonym“, also ohne, dass es mit einem Namen versehen werden muss, verwendet werden kann.

Dies gilt z.B. für Funktionen höherer Ordnung, higher-order Unifikation, higher-order Constraint-Programmierung, Referenzen in Standard ML, first-class continuations in Scheme etc. Programmiersprachen werden mächtiger, wenn first-order Konzepte auf higher-order erweitert werden, die Semantik und Implementierung aber unter Umständen komplizierter, weniger effizient und evtl. sogar unentscheidbar (z.B. Unifikation höherer Ordnung).

Im Folgenden sollen einige verbreitete Paradigmen-Kombinationen untersucht werden, die in Abbildung 2.3 zusammengefasst sind.

**Abbildung 2.3** Paradigmen-Kombinationen

| | |
|---|---|
| – funktional und imperativ | – constraint-funktional |
| – logisch und imperativ | – constraint-imperativ |
| – funktional und logisch | – nebenläufig und verteilt |
| – constraint-logisch | – multiparadigmatisch |

## 2.4 Funktionale und imperative Programmierung

Funktionale Programmiersprachen sind zustandsfrei, lassen sich also nicht unmittelbar in eine zustandsbehaftete Umgebung, wie sie z.B. durch Betriebssysteme dargestellt wird, einbetten. Notwendig ist ein Mechanismus, diese beiden Paradigmen miteinander zu verbinden.

Einige funktionale Programmiersprachen integrieren veränderbaren Zustand in der Form imperativer Variablen, welche oft als Datenobjekte eingegliedert werden, die modifizierbare Speicherzellen repräsentieren. In Standard ML werden diese sog. *reference cells* explizit erzeugt, verändert und ausgelesen, in Lisp und Scheme können alle Variablen modifiziert werden.

Funktionale Sprachen dieser Art werden von reinen funktionalen Sprachen, wie z.B. Haskell, unterschieden, da sie ein wichtiges Prinzip der funktionalen Programmierung verletzen: referenzielle Integrität. Dieses Prinzip besagt, dass ein Ausdruck immer den selben Wert haben muss, also gleiches durch gleiches ersetzt werden kann. Zustandsbehaftete Berechnungen müssen also in rein funktionalen Sprachen auf eine andere Art und Weise eingebettet werden. In Haskell wird das Konzept eines Programmzustandes durch Monaden eingeführt, welche den Kontrollfluss so serialisieren, dass die referenzielle Integrität erhalten bleibt.

### 2.4.1 Standard ML

Standard ML (Milner u.a., 1997) ist eine strikt auswertende funktionale Programmiersprache. Ähnlich wie Scheme (Kelsey u.a., 1998) oder Common Lisp (Steele, 1990) erlaubt Standard ML aber auch Seiteneffekte, wie z.B. nicht-deklarative Ein- und Ausgabe sowie veränderbare Variablen. Damit ist Standard ML keine rein funktionale Sprache, wie z.B. Haskell (Peyton Jones, 2003) oder Opal (Pepper, 2003).

In Standard ML werden veränderbare Variablen als Datenobjekte betrachtet, die eine einzelne Speicherzelle (*reference cell*) repräsentieren. Diese werden explizit mit einem initialen Wert erzeugt. Zum Zugriff muss der Wert aus der Zelle extrahiert werden und er kann mit einer Zuweisung verändert werden. Das folgende Beispiel zeigt die Erzeugung einer änderbaren Speicherzelle, eine Zuweisung und die Extraktion des zuvor zugewiesenen Wertes. Das Ergebnis des gesamten Ausdrucks ist 1.

```
let r = ref 0 in
  r := 1; !r
```

Abgesehen davon, dass solche veränderbaren Variablen die referenzielle Transparenz zerstören, kann es auch zu Problemen bei der Typisierung dieser Variablen kommen. Leroy

und Weis (1991) geben ein Beispiel, bei dem der normale für Standard ML verwendete Typinferenz-Algorithmus (erweitert um Referenztypen) fehlerhafte Typen zuweisen würde:

```
let r = ref [] in
  r := [1];
  if head(!r) then
    ...
  else
    ...
```

Das Problem hierbei ist, dass für $r$ der Typ $\alpha$ *list ref* (Referenz auf Liste vom Typ $\alpha$) inferiert wird, da `[]` polymorph ist. Dieser Typ ist sowohl für die Anweisung `r := [1]` als auch für `if head(!r) ...` gültig, wurde also ungültigerweise auf *int list ref* bzw. *bool list ref* spezialisiert. Die physische Veränderung von Datenstrukturen zerstört die Typsicherheit, da statische Typannahmen dynamisch verletzt werden. Zur Lösung können entweder erweiterte Typinferenz-Algorithmen verwendet oder die Struktur der Argumente des *ref* Konstruktors eingeschränkt werden.

### 2.4.2 Haskell

Ein ähnliches Programm wie das obige Beispiel für Standard ML kann in Haskell folgendermaßen geschrieben werden. Änderbare Referenzen müssen auch in Haskell explizit erzeugt werden. In diesem Beispiel wird die Referenz in die IO-Monade eingebettet, die für die Verwaltung eines Haskell-Programms verantwortlich ist.

```
import IORef
main = do ref <- newIORef 0
          writeIORef ref 1
          c <- readIORef ref
          putStr (show c ++ "\n")
```

In Kapitel 3 werde ich noch ausführlicher auf die Problematik der Kombination deklarativer und imperativer Sprachen eingehen.

## 2.5 Logische und imperative Programmierung

Bei der Integration logischer und imperativer Sprachen steht vor allem das relationale Programmiermodell und der damit verbundenen Nichtdeterminismus im Vordergrund. Ein eingebauter Mechanismus für Backtracking und Referenzparameter ermöglichen vor allem elegante Algorithmen zur Lösung von Suchproblemen. Die Multiparadigmensprache Leda (siehe Abschnitt 2.11.2) integriert auf ähnliche Weise das logische Paradigma in die imperative Basissprache.

### 2.5.1 Paslog

Paslog (Radensky, 1990) ist eine erweiterte Pascal-Variante, die Prolog-ähnliche logische Programmierung erlaubt. Die Erweiterung besteht aus folgenden Sprachergänzungen bzw. -modifikationen:

**Split statements.** Ein *split statement* enthält eine oder mehrere Anweisungen (genannt Alternativen), die in separaten Prozessen quasi-parallel ausgeführt werden. Nach seiner Abarbeitung setzt jeder dieser Prozesse die Programmausführung mit der auf das *split statement* folgenden Anweisung fort. Prozesse, die im weiteren Verlauf terminieren, beeinflussen nicht die übrigen Prozesse, die durch ein split statement erzeugt wurden.

**For split statements.** Diese Anweisungen kombinieren die *for*-Schleifen von Pascal mit den oben genannten *split statements*. Bei der Ausführung dieser Schleifen erhält eine Variablen nacheinander die Werte in einem gegebenen Intervall, und für jeden dieser Werte wird ein Prozess gestartet, der den Rumpf der Schleife ausführt.

**Erweiterungen vordefinierter Funktionen, Prozeduren und Operatoren.** Für die nichtdeterministische Programmierung wurden die vordefinierten Funktionen *first* und *last* hinzugefügt. Sie liefern zu jedem Aufzählungstyp das kleinste bzw. größte Element und können beispielsweise für Schleifen, die über alle Werte eines solchen Typs iterieren sollen, genutzt werden. Weiterhin wurden die Ein- und Ausgabeprozeduren so erweitert, dass sie mit Aufzählungstypen umgehen können. Die logischen Verknüpfungsoperatoren *and* und *or* sind im Gegensatz zu Standard-Pascal nicht-strikt.

**Programmausführung.** Anders als bei normalen Pascal-Systemen wird Paslog nicht vollständig kompiliert. Nach der Kompilierung aller Prozeduren und Funktionen sowie der Ausführung des Hauptprogramms beginnt ein Dialog, so dass der Benutzer ähnlich wie in interaktiven Prolog-Systemen Anfragen an das System stellen kann.

Paslog ähnelt mit dem Ansatz, eine imperative Sprache durch Hinzufügen bzw. Ändern weniger Spracheigenschaften mit logischen Programmierkonzepten zu erweitern, sehr der später entwickelten Sprache Alma-0. Allerdings geht bei Alma-0 die Integration weiter, da diese Sprache Backtracking unterstützt und auch mit Constraints erweitert werden soll (siehe unten).

### 2.5.2 Alma-0

Alma-0 ist die erste Sprache, die von Apt u.a. im Rahmen des Alma-Projekts entwickelt wurde (Apt und Schaerf, 1999). Das Ziel dieses Projekts ist die Kombination der imperativen und deklarativen Programmierung. Der Entwurf der Sprache basiert auf einer imperativen Programmiersprache (Modula-2), die schrittweise um deklarative Elemente erweitert werden soll. Für die erste Version von Alma (Alma-0) wurden zunächst nur Sprachkonstrukte für nichtdeterministische Auswahl, erfolgreiche und fehlschlagende Anweisungen und eingebautes Backtracking hinzugefügt.

Erfolgreiche und fehlschlagende Anweisungen sind boolsche Ausdrücke, die in Alma-0 an die Stelle von normalen Anweisungen geschrieben werden können.[3] Wenn das Ergebnis der Auswertung eines solchen Ausdrucks der Wert *falsch* ist, schlägt die Anweisung fehl und das Programm wird am letzten aktiven Wahlpunkt (engl. *choice point*) weiter ausgeführt. Durch einige weitere Anweisungen können solche Wahlpunkte erzeugt werden, um beispielsweise zu

[3] Die Sprache Modula-2, auf der Alma-0 basiert, unterscheidet zwischen Ausdrücken, die Werte berechnen, und Anweisungen, die nur ihrer Seiteneffekte wegen ausgeführt werden.

überprüfen, ob es eine Lösung für ein Problem gibt oder um in einer Schleife alle Lösungen zu berechnen.

Mit den eingebauten Suchfähigkeiten eignet sich Alma-0 gut zur Lösung vieler kombinatorischer Probleme (Beispiele finden sich in der Literatur (Apt und Schaerf, 1997; Apt u.a., 1998)). Die Sprachimplementierung basiert auf einer traditionellen imperativen Programmiersprache unter Hinzufügung eines *Trail*, der Variablenbindungen speichert, die für das Backtracking rückgängig gemacht werden können.

Für spätere Sprachversionen ist die Integration von Constraint-Programmierung geplant (Apt und Schaerf, 1999). Eine detaillierte Beschreibung dieser Pläne und eine Implementierung existieren allerdings noch nicht. Alma-0 ist eine logisch-imperative Sprache, während geplante zukünftige Versionen als constraint-imperativ bezeichnet werden könnten.

## 2.6 Funktionale und logische Programmierung

Funktionale und logische Programmiersprachen lassen sich besonders gut kombinieren, da es sich bei beiden um deklarative Sprachen handelt, die sich in ein gemeinsames Berechnungsmodell einbetten lassen, wie es von Hanus (1997) beschrieben wird.

Die funktional-logische Programmierung entsteht im Wesentlichen dadurch, dass freie Variablen in Termen erlaubt werden. Der Auswertungsmechanismus funktionaler Sprachen (*Reduktion*) muss dafür durch den Auswertungsmechanismus logischer Sprachen (*Resolution*) erweitert werden. Dabei wird das bei funktionalen Sprachen übliche Pattern-Matching durch die aus der logische Programmierung bekannte Unifikation der formalen und aktuellen Parameter eines Funktionsaufrufs ersetzt. Der resultierende Mechanismus wird *Narrowing* genannt.

Ein anderer Auswertungsmechanismus wird als *Residuation* bezeichnet. Bei der Residuation werden Funktionsaufrufe, bei denen uninstanziierte Variablen als Parameter vorkommen, solange verzögert, bis alle Variablen von parallel laufenden Prozessen gebunden wurden, bzw. bis die Variablen soweit instanziiert wurden, dass eine eindeutige Regelauswahl möglich ist. Dies erlaubt auch eine eingeschränkte Nebenläufigkeit. Der Nachteil dieser Auswertungsstrategie besteht darin, dass sie unvollständig ist und für manche Terme keine Lösung berechnen kann, selbst wenn die darin enthaltenen ungebundenen Variablen nicht zur Lösung beitragen. Narrowing dagegen ist vollständig, da es die Reduktion funktionaler Sprachen mit Unifikation bei der Parameterübergabe verbindet. Es können also auch Funktionsaufrufe reduziert werden, die ungebundene Variablen enthalten.

Einige funktional-logische Sprachen trennen Funktionen, die deterministisch ausgewertet werden, und Prädikate, die nichtdeterministisch berechnet werden, wie z.B. Escher (siehe Abschnitt 2.6.3). In anderen Sprachen, wie z.B. Curry (Abschnitt 2.6.1), existiert keine solche Trennung, vielmehr entscheidet die Form der jeweiligen Gleichung über deren Auswertungsstrategie.

Neben den im Folgenden ausführlicher vorgestellten Sprachen sind noch die Sprachen ALF (ein Vorgänger von Curry) und Babel zu erwähnen. Beide sind funktional-logische Sprachen, wobei die Auswertung von ALF auf Resolution, Rewriting und Narrowing basiert (Hanus, 1990), ähnlich wie Curry. Syntaktisch gesehen kombiniert Babel (Moreno-Navarro und Rodriguez-Artalejo, 1992) reines Prolog mit einer Funktionsnotation erster Ordnung. Babel nutzt Narrowing als Grundlage der verzögerten Reduktions-Semantik und umfasst

damit sowohl Termersetzung als auch Resolution.

### 2.6.1 Curry

Curry (Hanus u.a., 1995) ist eine funktional-logische Erweiterung der funktionalen Programmiersprache Haskell (Peyton Jones, 2003). Sie wurde entwickelt, um einen Standard im Bereich der funktional-logischen Sprachen zu etablieren.

Das Berechnungsmodell, das Curry zu Grunde liegt, vereinigt sowohl das funktionale als auch das logische Modell (Hanus, 1997). Ein Curry-Programm besteht aus einer Menge von Datentypdeklarationen, mit denen algebraische Datentypen eingeführt werden können, und Funktionsdefinitionen der Form

```
function f: T1 -> T2 -> ... -> Tn
f p1 ... pn = t <= C
```

Die Terme $p_i$ sind pattern, über die beim Funktionsaufruf eine anwendbare Regel ausgewählt wird. Dabei wird eine Funktionsregel nur dann zur Reduktion angewendet, wenn die Bedingung $C$ wahr ist. Da $C$ freie Variablen enthalten darf, wird an dieser Stelle Nichtdeterminismus eingeführt, der in Curry standardmäßig mit Tiefensuche bearbeitet wird, die freien Variablen werden also nacheinander mit allen möglichen Werten instanziiert, und wenn die Reduktion fehlschlägt, wird ein Backtracking ausgelöst. Es ist aber auch möglich, eigene Suchstrategien zu programmieren.

Wenn eine Funktion angewendet (aufgerufen) wird, werden die Argumente und die formalen Parameter unifiziert. Dazu muss das Programm die Argumente evtl. erst zur Kopfnormalform auswerten, um entscheiden zu können, welche der linken Regelseiten einer Funktionsdefinition zu den gegebenen Parametern passen. Funktionsaufrufe werden mit Narrowing ausgeführt, es ist aber auch möglich, durch die Angabe von Auswertungseinschränkungen diesen Mechanismus für bestimmte Funktionen zu beinflussen. Eine Einschränkung wie die Folgende hat beispielsweise zur Folge, dass beim Aufruf der Funktion `append` immer das erste Argument ausgewertet wird, bevor eine Regel für diese Funktion gesucht wird:

```
eval append 1:rigid
```

Ein formaler Parameter, der als `rigid` deklariert wird, verhält sich wie bei funktional-logischen Sprachen, die durch Residuation ausgewertet werden: Aktuelle Parameter werden immer soweit ausgewertet, dass anhand des Kopfkonstruktors des Paramters entschieden werden kann, welche Regel auszuwählen ist.

Die logischen Programmiermöglichkeiten von Curry erlauben es sogar, nach Funktionen zu suchen. Dazu unterstützt Curry eine eingeschränkte Variante der *Unifikation höherer Ordnung*, wobei durch eine Suche über die definierten Funktionen eines Programms auch Funktionen berechnet werden können. Außerdem kann eine Funktion auch dadurch gesucht werden, dass einfach alle definierten Funktionen des gesuchten Typs aufgezählt werden.

Ein- und Ausgaben werden in Curry durch ein deklaratives Modell integriert. Ähnlich wie in Haskell wird dazu monadische Ein-/Ausgabe benutzt. Der Compiler benutzt die Auswertungseinschränkungen, um zu prüfen, dass keine Ein- oder Ausgaben ausgeführt werden, solange ein Programm eine Suche durchführt, die evtl. ausgeführte Ein-/Ausgabeoperationen ungültig machen könnte. Die seiteneffektbehafteten Operationen von Prolog, wie der Cut oder `assert/retract`, stehen nicht zur Verfügung. Stattdessen existieren Implikationen und

Quantoren, mit deren Hilfe Regeln lokal für die Auswertung eines Ziels definiert werden können, die nach Beendigung dieser Auswertung nicht mehr existieren.

### 2.6.2 Gödel

Die logische Programmiersprache Gödel (Hill und Lloyd, 1994) wurde als deklarativer Nachfolger von Prolog konzipiert. Gödel verfügt sowohl über ein Typ- als auch ein Modulsystem und ersetzt alle nicht-deklarativen Sprachelemente (nicht-logische Prädikate wie z.B. für Ein- und Ausgabe, der Cut-Operator) durch deklarative Alternativen. Das Typsystem basiert auf viel-sortiger Logik mit parametrischem Polymorphismus. Weiterhin werden in Gödel Prädikate und (deterministische) Funktionen unterschieden. In beschränktem Umfang ist auch Constraint-Programmierung über endlichen Wertebereichen und lineare rationale Constraints möglich. Constraints werden so lange verzögert, bis darin enthaltene Variablen soweit instanziiert sind, dass die Menge der Lösungen endlich ist.

Weiterere Schwerpunkte des Sprachentwurfs sind Erweiterungen zum Schreiben von Meta-Programmen für die Analyse, Transformation, Kompilierung, Verifizierung und das Debugging von Gödel-Programmen.

### 2.6.3 Escher

Auch Escher (Lloyd, 1995) ist eine Verbindung funktionaler und logischer Programmierung. Allerdings gibt es in Escher keine nichtdeterministischen Funktionen, sondern alle Funktionen liefern ein Ergebnis, das allerdings eine Disjunktion verschiedener Ergebnisse sein kann. Auf diese Weise wird in Escher Nichtdeterminismus modelliert.

Funktionen, die einen booleschen Wert als Resultat liefern, werden in Escher auch als Prädikate bezeichnet. Die Auswertungsstrategie ist Residuation, wobei durch die Angabe eines sog. Modus für jede Funktion spezifiziert werden kann, auf welche Art und Weise Funktionsaufrufe verzögert werden. Wenn für einen Parameter der Modus `NONVAR` festgelegt wird, werden alle Aufrufe dieser Funktion verzögert, bis der entsprechende Parameter instanziiert ist. Auf diese Weise lässt sich der Nichtdeterminismus beim Aufruf von Prädikaten einschränken.

Eine Besonderheit von Escher sind die syntaktischen Elemente `SOME` und `ALL`, welche als Existenz- und Allquantor freie Variablen deklarieren. `SOME` als rechte Seite einer Funktion führt dazu, dass es für die entsprechenden Variablen im Ergebnis eine gültige Belegung gibt, `ALL` fordert, dass alle möglichen Belegungen auch gültig sind.

Escher bietet auch einige syntaktische und semantische Erweiterungen, die aus funktionalen Programmiersprachen bekannt sind. Ähnlich zu Haskell unterstützt Escher *list comprehensions* zur prägnanten Konstruktion von Listen sowie Funktionen höherer Ordnung und anonyme Funktionen ($\lambda$-Ausdrücke).

### 2.6.4 Mercury

Mercury (Somogyi u.a., 1995) ist eine rein deklarative logische Programmiersprache. Sie wurde zwar nicht als multiparadigmatische Sprache entwickelt, besitzt aber einige Eigenschaften, die effiziente funktionale Programmierung unterstützen. So ist es beispielsweise möglich, neben Prolog-artigen Prädikaten deterministische Prädikate in einer Notation wie in funktionalen Sprachen zu definieren:

```
fibonacci(N) = F :-
  ( if N =< 2 then F = 1
              else F = fibonacci(N - 1) + fibonacci(N - 2) ).
```

Mercury unterstützt auch Funktionen höherer Ordnung. Funktionswerte können durch $\lambda$-Ausdrücke erzeugt werden und auf Argumente angewendet werden. Dies geschieht entweder durch Verwendung des eingebauten Prädikats `apply` oder durch die Anwendung einer Variablen auf Parameter, wobei die Variable an einem $\lambda$-Ausdruck gebunden sein muss.

Durch ein ausgefeiltes Typ- und Modus-System können weiterhin alle Typen und Parameter-Modi (unbestimmt, *nonvar*, *in*, *out*) deklariert werden. Der Compiler prüft diese Deklarationen und nutzt die Informationen zur Optimierung. Neben den Parameter-Modi können Prädikate als deterministisch (immer genau einmal erfolgreich), semi-deterministisch (erfolglos oder genau einmal erfolgreich) sowie nicht-deterministisch deklariert werden. Der Compiler kann anhand dieser Informationen (semi-)deterministische Funktionsaufrufe effizienter übersetzen als nicht-deterministische.

Die Integration von Ein- und Ausgabe in die deklarative Sprache wird über ein Datenobjekt bewerkstelligt, das den Zustand des Programms und seiner Umgebung repräsentiert. Jedes Prädikat, das Ein- oder Ausgaben durchführt, erhält dieses Objekt als Parameter, wobei spezielle Modusdeklarationen für diese Parameter dafür sorgen, dass der Zustand immer auf einem Pfad durch den Programmablauf geschleust wird und kein Backtracking über Ein-/Ausgabeoperationen hinaus stattfindet (siehe auch Abschnitt 3.4.3).

### 2.6.5 $\lambda$-Prolog

$\lambda$-Prolog (Nadathur und Miller, 1988) ist eine logische Programmiersprache, die Funktionen höherer Ordnung, polymorphe Typisierung, abstrakte Datentypen sowie $\lambda$-Abstraktionen in Datenstrukturen erlaubt. Die Erweiterung um $\lambda$-Terme gegenüber Prolog erfordert eine erweiterte Auswertungsstrategie, und zwar Unifikation höherer Ordnung zur Berechnung von Funktionen.

### 2.6.6 $\mathrm{L}_\lambda$

Im Gegensatz zu $\lambda$-Prolog erfordert die funktional-logische Sprache $\mathrm{L}_\lambda$(Miller, 1991) keine Unifikation höherer Ordnung, da sie das Auftreten von Funktionsvariablen einschränkt. Der Vorteil dieser Restriktionen ist, dass die Unifikation in diesem Fall entscheidbar ist und immer ein allgemeinster Unifikator gefunden wird, falls dieser existiert.

Die Motivation zur Entwicklung von $\mathrm{L}_\lambda$ ist die Meta-Programmierung, also die Verarbeitung von Programmen durch Programme. Dazu ist die Manipulation syntaktischer Strukturen (Programme, Formeln, Typen und Beweise) notwendig. Die Unifikation erleichtert den Umgang mit diesen Termen, z.B. die Berechnung der Termgleichheit modulo $\alpha$, $\beta$ und $\eta$-Konvertierungen.

## 2.7 Constraint-logische Programmierung

Die constraint-logische Programmierung (CLP) (Jaffar und Lassez, 1987; Frühwirth u.a., 1993; Jaffar und Maher, 1994) kombiniert constraint-basierte und logische Programmierung.

CLP-Sprachen werden als Obermenge der klassischen logischen Programmiersprachen (wie z.B. Prolog) angesehen, da eine CLP-Sprache, die auf Herbrand-Termen arbeitet, einer logischen Sprache entspricht. Der Begriff CLP steht für eine ganze Familie $CLP(X)$ von Sprachen, wobei $X$ für die einzelnen Constraint-Systeme steht: $CLP(FD)$ bezeichnet die constraint-logische Programmierung mit endlichen Wertebereichen (engl. *finite domain*), $CLP(R)$ entsprechend für reell-wertige Constraints.

Die meisten modernen Prolog-Dialekte, wie z.B. Eclipse, Ciao-Prolog (Bueno u.a., 2002), GNU Prolog (Diaz, 2002), beherrschen den Umgang mit Constraints und lassen sich daher den constraint-logischen Sprachen unterordnen. Abgesehen von teilweise unterschiedlichen Constraint-Domains ähneln sich diese Sprachen sehr und werden deshalb nicht im Detail besprochen.

### 2.7.1 AKL – AGENTS Kernel Language

AKL (Janson, 1994) wurde für den Einsatz in Wissens- und Informationsverarbeitungssystemen entwickelt, mit Hinblick auf die Möglichkeiten moderner Multiprozessor-Computer. Daher wurde darauf geachtet, dass die Sprache Möglichkeiten zur parallelen Ausführung bietet. Diese Voraussetzungen führten zum Entwurf einer logischen Programmiersprache, welche sowohl Nichtdeterminismus als auch die Beschreibung von Prozessen ermöglichen soll.

Syntax und Ausführungsmodell von AKL basieren auf der *Concurrent Constraint* Programmierung (Saraswat, 1993). In diesem Modell besteht eine Berechnung aus einer Gruppe von Agenten und einem gemeinsamen Speicher. Agenten können diesem Speicher Informationen hinzufügen bzw. auf die Verfügbarkeit von Informationen im Speicher warten. Auf diese Weise werden die einzelnen, parallel laufenden, Agenten synchronisiert. Die Informationen werden durch Constraints repräsentiert, es ist also möglich, mit unvollständigen Informationen zu rechnen.

AKL unterstützt sowohl *don't care*- als auch *don't know*-Nichtdeterminismus durch die explizite Verwendung entsprechender Sprachkonstrukte.

In einem früheren Papier (Janson und Haridi, 1991) sprechen Janson und Haridi von der Verwendung mehrerer Paradigmen. Es handelt sich dabei allerdings nicht um eine umfassende Integration sehr unterschiedlicher Paradigmen, sondern um sich ohnehin ähnelnde, nämlich der logischen Programmierung in Prolog und in *Guarded Horn Clauses*, sowie der Constraint-Programmierung.

## 2.8 Constraint-funktionale Programmierung

Die Erweiterung von Constraint-Sprachen um deterministische Funktionen wird constraint-funktionale Programmierung genannt. Dies wird durch die Erweiterung des Constraint-Systems einer Constraint-Programmiersprache um Funktionen bewerkstelligt, wobei diese Integration von Funktionen unterschiedlich eng sein kann. Teilweise werden Funktionen soweit integriert, dass der Constraint-Lösungsmechanismus in der Lage ist, Funktionen aus gegebenen Constraints zu berechnen, teilweise werden Funktionen ausschließlich als deterministische Berechnungen eingebaut.

Als Beispiele dieser Paradigmen-Kombination sollen die Sprachen Goffin und Falcon beschrieben werden.

### 2.8.1 Goffin

Goffin (Chakravarty u.a., 1997) ist eine constraint-funktionale Erweiterung der Programmiersprache Haskell (Peyton Jones, 2003), bei der Funktionen in eine nebenläufige Constraint-Programmiersprache eingebettet sind (ähnlich zu AKL, siehe Abschnitt 2.7.1). Die grundliegende Idee dieser Integration basiert auf der Überlegung, eine Programmiersprache aus einer Berechnungssprache und einer Koordinationssprache aufzubauen. Die Berechnungssprache führt sequenziell die eigentlichen Berechnungen aus, während die Koordinationssprache Berechnungen erzeugt und für die Kommunikation der einzelnen Berechnungen untereinander sorgt. Der Constraint-Teil von Goffin der Sprache ist explizit getrennt vom funktionalen Teil und dient diesem als Koordinationssprache, stellt also Mittel zur Organisation funktionaler Reduktionen zur Verfügung.

Im Constraint-Teil können logische Variablen erzeugt werden, während die Aufrufe von Funktionen verzögert werden, bis ein Parameter instanziiert ist. Funktionen können Constraints erzeugen und erlauben damit die Definition parametrisierter Koordinations-Konstrukte.

Constraints werden in geschweifte Klammern ({ und }) eingeschlossen, wobei ungebundene (logische) Variablen explizit deklariert werden:

$$\{x, y \textbf{ in } e\}$$

Logische Variablen werden durch Gleichheitsconstraints (geschrieben als $\langle exp \rangle_1 \leftarrow \langle exp \rangle_2$) instanziiert. Mehrere dieser einfachen Constraints können durch Kommata getrennt zu Konjunktionen zusammengefasst werden:

$$\{x, y, r \textbf{ in } x \leftarrow foo\ a, y \leftarrow bar\ a, r \leftarrow x + y\}$$

In Goffin können logische Variablen nur an Datenstrukturen gebunden sein, die keine Funktionen enthalten, dadurch wird Unifikation höherer Ordnung vermieden. Natürlich ist es damit in Goffin nicht möglich, Funktionen zu berechnen, wie dies z.B. Curry (siehe Abschnitt 2.6.1) erlaubt.

Die Möglichkeit, durch Funktionen Constraints zu berechnen, wird genutzt, um z.B. Berechnungen zu parallelisieren, da die einzelnen Constraints einer Konjunktion nebenläufig abgearbeitet werden können. Weiterhin können Funktionen aus Constraints andere Constraints bilden, also als Kombinatoren für Koordinationsstrukturen dienen.

### 2.8.2 Falcon

Falcon (Guo und Pull, 1993) kombiniert funktionale und logische Programmierung mit Constraints. Syntaktisch bestehen Falcon-Programme aus bewachten (*guarded*) funktionalen Termersetzungsregeln sowie aus Relationen über funktionalen Ausdrücken. Semantisch ist Falcon eine Constraint-logische Programmiersprache, bei der das unterliegende Constraint-System mit funktionalen Programmen erweitert werden kann. Guo beschreibt diese Kombination mit der Gleichung $Falcon = CLP(CFP)$, die constraint-funktionale Programmierung stellt also das Constraint-System für ein allgemeines constraint-logisches System (siehe Abschnitt 2.7) dar.

Die funktionale Programmierung eignet sich gut zur Definition eines Constraint-Systems, da sie benutzerdefinierte Objekte und die Fähigkeit, Constraints über allgemeinen Ausdrücken zu lösen, zur Verfügung stellt.

Die Integration der funktionalen Programmierung als Constraint-System in einer constraint-logischen Sprache stellt den besonders interessanten Ansatz von Falcon dar.

## 2.9 Constraint-imperative Programmierung

Die constraint-imperative Programmierung umfasst alle Programmiersprachen, die sowohl constraint- als auch imperative Charakteristiken besitzen. Im wesentlichen handelt sich es dabei um imperative bzw. objekt-orientierte Sprachen, bei denen es möglich ist, zwischen den Variablen des Programms und/oder den Attributen von Objekten Constraints zu definieren. Ein in das Programmiersystem integrierter Constraint-Löser ist dafür zuständig, für die Variablen Belegungen zu berechnen, welche die definierten Constraints erfüllen.

Die Programmiersprache Kaleidoscope (Lopez u.a., 1994) ist eine objekt-orientierte Sprache, die die Definition von Constraints auf Objekt-Attributen erlaubt. Die oben bereits genannte Sprache Alma-0 (Apt u.a., 1998) soll um Constraints erweitert werden (Apt und Schaerf, 1999), wodurch auch sie sich zu einer constraint-imperativen Sprache entwickeln wird. Turtle (Grabmüller, 2003) ist eine prozedurale Sprache, bei der Constraints zwischen speziell deklarierten Variablen festgelegt werden können, um Werte für diese zu bestimmen.

Der wesentliche Nachteil constraint-imperativer Sprachen besteht in der Zusammenführung zustandsbehafteter und zustandsfreier Semantiken. Dieses Problem wird unterschiedlich gelöst. Bei der Sprache Kaleidoscope werden alle zustansbehafteten Operationen (z.B. Zuweisungen) auf Constraints abgebildet, die dann zustandsfrei von dem integrierten Constraint-Löser verarbeitet werden. In Turtle dagegen wurde der Ansatz gewählt, Constraints nur zur Berechnung und Prüfung von Constraints zu verwenden, während außerhalb dieser Berechnungsphasen normal imperativ gearbeitet wird. Der Constraint-Solver wird also jeweils explizit zur Berechnung aufgerufen und der imperative Programmfluss für die Zeit dieser Berechnung angehalten.

### 2.9.1 Kaleidoscope

Kaleidoscope (Lopez u.a., 1994) ist eine objekt-orientierte Sprache mit Constraints. Constraints können zwischen den Attributen verschiedener Objekte spezifiziert werden, und das System sorgt durch die Anpassung der Attributwerte dafür, dass die Constraints eingehalten werden.

In Kaleidoscope können Constraint-Konstruktoren definiert werden, diese ähneln Methoden, werden aber benutzt, um neue Constraints zu beschreiben. Beispielsweise kann ein Gleichheitsconstraint zwischen zwei Objekten durch die Gleichheit aller seiner Attribute definiert werden. Das System ermittelt anhand der Klassen der beteiligten Parameter eines solchen Constraints den zugehörigen Constraint-Konstruktor und führt diesen aus.

Die Semantik von Kaleidoscope geht von einer unterliegenden Maschine aus, die Constraints verarbeiten kann. Alle imperativen Konstrukte der Sprache werden zur Ausführung auf dieser Maschine in Constraints umgewandelt, Zuweisungen beispielsweise in Gleichheitsconstraints.

### 2.9.2 Turtle

Turtle (Grabmüller, 2003; Grabmüller und Hofstedt, 2003) erweitert eine imperative Basissprache mit einigen funktionalen Sprachkonstrukten durch vier neue Konzepte zu einer constraint-imperativen Sprache: Constrainable Variable, Constraint-Anweisungen, benutzerdefinierte Constraints und Constraint-Löser. Constrainable Variable unterscheiden sich von normalen Variablen dadurch, dass sie ihre Werte nicht durch Zuweisungen bekommen, sondern dadurch, dass Constraints für diese Variablen definiert werden. Dies geschieht mittels Constraint-Anweisungen, welche die Variablen mit anderen Variablen oder Konstanten in Verbindung setzen. Constraint-Anweisungen bestehen aus einer Constraint-Konjunktion und einem Anweisungsblock. Die Constraints der Konjunktion werden vom System solange durch geeignete Variablenbelegungen erfüllt, bis der Anweisungsblock komplett abgearbeitet ist. Am Anfang des Blocks werden die Constraints den Constraint-Lösern übergeben, die dafür verantwortlich sind, die Erfüllbarkeit der Constraints zu prüfen und Variablenbelegungen zu berechnen. Benutzerdefinierte Constraints dienen als Abstraktionsmittel für Constraints: ähnlich wie Funktionen Anweisungen zusammenfassen, können in benutzerdefinierten Constraints mehrere Constraint-Anweisungen und andere Berechnungen zusammengefasst werden, um dann in verschiedenen Constraint-Anweisungen in den Constraint-Konjunktionen aufgerufen zu werden.

Constraints werden in Turtle also vor allem dazu benutzt, um Werte für Variablen zu berechnen. Da bei Nichterfüllbarkeit der Constraints Ausnahmesituationen ausgelöst werden, die vom Programm abgefangen und behandelt werden können, lassen sie sich aber auch benutzen, um den Programmablauf zu steuern.

## 2.10 Nebenläufige und verteilte Sprachen

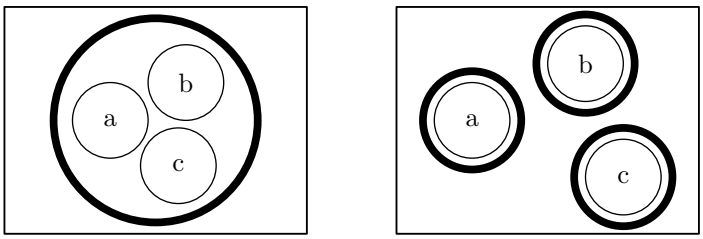
Die in diesem Abschnitt vorgestellten Sprachen besitzen als besonderes Merkmal Konstrukte zur nebenläufigen und verteilten Programmierung. Wie bereits in Abschnitt 2.2 bemerkt, sind nebenläufige oder verteilte Sprachen immer nebenläufige oder verteilte Versionen anderer Paradigmen. In diesem Abschnitt sollen Sprachen beschrieben werden, deren wichtigste Eigenschaft die Nebenläufigkeit bzw. Verteiltheit ist.

### 2.10.1 Erlang

Erlang ist eine nebenläufige und verteilte funktionale Sprache (Sahlin, 1996). Die Syntax lehnt sich an die von Prolog an (Variablennamen werden groß geschrieben, Funktionsdefinitionen ähneln Klauseldefinitionen in Prolog), es handelt sich aber um keine logische Programmiersprache. Erlang besitzt eine dynamische Typisierung, ähnlich wie Prolog oder Lisp, d.h. dass nicht die Variablen typisiert werden, sondern die Werte, die zur Laufzeit an diese Variablen gebunden werden.

Die Nebenläufigkeit wird durch drei Sprachkonstrukte unterstützt: Die primitive Funktion `spawn` erzeugt einen neuen Prozess, der Operator `!` (genannt *send*) verschickt einen Konstruktorterm als Nachricht und der nichtdeterministische Empfang wird durch `receive`-Anweisungen durchgeführt. Bei der Erzeugung eines Prozesses kann optional auch noch der Rechner (*Knoten* in der Erlang-Terminologie) angegeben werden, auf dem der Prozess ablaufen soll. Durch diese Erweiterung wird Verteiltheit ermöglicht; die Kommunikation läuft

**Abbildung 2.4** Nebenläufigkeit in deklarativen Sprachen




(a) Deklarative Prozesse (b) Monadische Prozesse

bei verteilten Systemen genauso durch Prozesskommunikation ab wie bei nicht verteilten.

Die Auswertungsstrategie in Erlang ist strikt, und Variablen können nur einmal an einen Wert gebunden und danach nicht mehr geändert werden. Daher ist der Charakter der Sprache funktional. Ein- und Ausgabe sind aber nicht deklarativ, und es gibt über die Standard-Bibliothek die Möglichkeit, globale und veränderbare Variablen zu definieren.

### 2.10.2 Eden

Eden ist eine nebenläufige deklarative Sprache zur Programmierung reaktiver Systeme und paralleler Algorithmen auf Systemen mit verteiltem Speicher (Breitinger u.a., 1996). Dabei handelt es sich um eine Erweiterung von Haskell. Die funktionale Sprache dient als Berechnungssprache für sequenzielle Programme, welche um eine Koordinationssprache zur Spezifizierung von Nebenläufigkeit erweitert wurde. Die Aufteilung in zwei Teilsprachen zur Berechnung und Koordination (ähnlich wie bei Goffin, siehe Abschnitt 2.8.1) erlaubt die Beibehaltung der rein funktionalen Semantik der Kernsprache und deren einfache Erweiterung.

Nebenläufigkeit wird in Eden durch Prozesse ausgedrückt. In der Sprache können Prozessabstraktionen definiert werden, welches statische Konstrukte – ähnlich Lambda-Ausdrücken – sind. Durch die Instanziierung (entspricht der Applikation eines Lambda-Ausdrucks) werden dann Prozesse erzeugt, die nebenläufig ausgeführt werden. Untereinander können Prozesse über Kanäle kommunizieren, die als verzögerte Listen modelliert werden. Lesende Prozesse greifen einfach auf Listenelemente zu, während schreibende Prozesse Listenelemente generieren. Die Kommunikation und Synchronisierung ist für die Prozesse transparent. Es ist aber für einzelne Prozesse möglich, Antwort-Kanäle explizit zu erzeugen und zu nutzen, um die Kommunikation zwischen mehreren Prozessen gleichzeitig durchzuführen.

### 2.10.3 Nebenläufigkeit in anderen deklarativen Sprachen

Die nebenläufige Constraint-Programmierung (engl. *concurrent constraint programming, cc*) (Saraswat, 1993) erweitert die Constraint-Programmierung, indem nichtdeterministische Auswertungen nebenläufig abgearbeitet werden. Innerhalb eines nebenläufigen Constraint-Programms laufen also gleichzeitig verschiedene Auswertungen. Beispiele dieser Programmierung sind AKL und Goffin (siehe oben). In Abbildung 2.4 (a) ist dieses Konzept dargestellt. Die dünn umrandeten Kreise repräsentieren drei Prozesse, die in einem gemeinsamen Umfeld ablaufen (dick umrandeter Kreis) und miteinander durch den gemeinsamen

Constraint-Speicher kommunizieren. Die Rechtecke in der Abbildung 2.4 umschließen das gesamte Programm.

Abbildung 2.4 (b) stellt einen anderen Ansatz dar, nämlich eine nebenläufige Erweiterung von Haskell namens Concurrent Haskell (Peyton Jones, 2001). Hierbei werden nebenläufige Prozesse jeweils als monadische Berechnungen (siehe Kapitel 3) modelliert, wobei jede einzelne dieser Berechnungen nicht explizit nebenläufig ist. Der wesentliche Unterschied besteht darin, dass die einzelnen Berechnungen rein funktional sind und keine Seiteneffekte haben dürfen. Lediglich die Aktionen, welche die Prozesse darstellen, dürfen dies und können somit damit mit anderen Prozessen kommunizieren.

## 2.11 Multiparadigmensprachen

In den vorhergehenden Abschnitten wurden Sprachen untersucht, die zwei unterschiedliche Paradigmen integrieren. Dieser Abschnitt soll Sprachen beschreiben, bei denen mehr als zwei Paradigmen zusammengefasst wurden.

### 2.11.1 CFLP($\mathcal{R}$)

Dies ist eine Sprache, die verzögerte funktionale Programmierung, logische Programmierung und Constraint-Lösen mit reellen Zahlen kombiniert (Kobayashi u.a., 2002). Damit ist die Sprache sowohl Obermenge von higher-order polymorphen verzögerten funktionalen Sprachen als auch der Constraint-logischen Programmierung. Die Auswertungsstrategie basiert auf *Lazy Narrowing* und dem Lösen von Constraints. Der Name lehnt sich an die allgemeine constraint-logische Programmierung $CLP(X)$ an, nur dass in diesem Fall die funktionale Programmierung ebenfalls eine wichtige Rolle spielt.

### 2.11.2 Leda

Leda ist eine streng getypte Programmiersprache, die sowohl objekt-orientierte (imperative) als auch funktionale und logische (bzw. relationale) Aspekte in einer Sprache vereint. Der logische Anteil wird durch ein neues Sprachkonstrukt, die *Relation*, in die Sprache integriert.

Relationen ähneln Prozeduren und Funktionen, werden aber als eine Art boolscher Wert behandelt, der in Konjunktionen und Disjunktionen allerdings durch Backtracking ausgewertet wird, um so Lösungen zu generieren. Syntaktisch sind Relationen Funktionen, die Werte des speziellen Typs `relation` als Ergebnis haben. Einfache Relationen werden gebildet, indem mit dem relationalen Zuweisungsoperator `<-` Werte an eine Variable zugewiesen werden. Dieser Operator unterscheidet sich von der normalen Zuweisung dadurch, dass die Variable auf den vorigen Wert zurückgesetzt wird, sollte das Backtracking diese Operation rückgängig machen. Boolsche Ausdrücke können überall verwendet werden, wo Relationen syntaktisch erwartet werden und komplexere Relationen können dann durch Dis- und Konjunktionen von Relationen gebildet werden.

Wenn eine Relation als Bedingung in einer Fallunterscheidung oder als Bereich in einer `for`-Schleife angewendet wird, beginnt der Berechnungsprozess. In einer Fallunterscheidung gilt die Bedingung als wahr, falls die Relation erfüllt ist, in einer Schleife können alle Lösungen einer Relation bearbeitet werden.

Die Auswertungsstrategie ist call-by-value, es ist aber auch möglich, durch Parameterdeklarationen call-by-reference und call-by-name für Funktionen zu deklarieren. Referenzparameter werden benutzt, um beim Aufruf von Relationen Variablen zu binden. Durch die Verwendung von call-by-name Parametern ist es sogar möglich, nicht-strikte Kontrollstrukturen in Leda selbst zu programmieren.

Die Unterstützung funktionaler Programmierung basiert auf Funktionen höherer Ordnung und der Möglichkeit, Datentypen (Klassen) und Funktionen mit Datentypen zu parametrisieren. Objekt-orientierte Programmierung wird durch Klassendefinitionen, Vererbung und dem Überschreiben von Methoden durch dynamische Bindung ermöglicht.

Relationen eignen sich sehr gut, um Iteration auszudrücken. Wenn eine Relation z.B. alle Elemente einer Datenstruktur aufzählt, kann eine spezielle Schleifenstruktur (die oben genannte for-Schleife) für alle diese Elemente ausgeführt werden, ohne dass der interne Aufbau der Datenstruktur bekannt sein muss. Die Klasse `List` implementiert eine einfach verkettete Liste, die Ganzzahl-Werte aufnehmen kann. (Der Konstruktor sowie Zugriffsfunktionen wurden aus Platzgründen weggelassen.)

```
class List {
  var
    value: int;
    next: List;
  ...
  function items(byRef val: int)->Relation
  begin
    return unify(val, value)
         | defined(next) & next.items(val);
  end;
end;
```

Da der Parameter `val` call-by-reference deklariert ist, wird er beim Aufruf der `items`-Funktion mit dem aktuellen Listenelement unifiziert. Da Relationen mit Backtracking ausgeführt werden, wird erst das erste Element der Liste mit `val` unifiziert, dann das zweite, u.s.w. Die folgende for-Schleife iteriert daher über alle Elemente der Liste `aList`:

```
var aList: List;
    val: int;
...
for aList.items(val) do
  ...
```

Diese Anwendung von Relationen ist eine mächtige und konzeptionell saubere Verallgemeinerung der Iteratoren, wie es sie in anderen Sprachen zur Aufzählung von Elementen einer Datenstruktur gibt, wie z.B. in der Standard Template Library (STL) von C++ oder der Java Standard-Bibliothek.

Generell schlägt Budd als möglichen Ansatz zum Programmentwurf mit einer multiparadigmatischen Sprache folgendes Vorgehen vor:

- Die Grobstruktur ist objekt-orientiert. Objekte besitzen einen Zustand und Methoden, die diesen Zustand verwenden und/oder verändern.

– Operationen auf den Objekten können auch Funktionen sein, die den Zustand nicht verändern und so die Vorteile der funktionalen Programmierung nutzen.

– Die logische Programmierung eignet sich nicht nur für offensichtliche Suchprobleme, sondern auch als Iterationsmechanismus (s.o.). Durch die Verwendung definierter und undefinierter Werte als Parameter kann die Suche bzw. Iteration eingeschränkt werden.

Natürlich ist auch ein anderes Vorgehen beim Entwurf möglich. Beispielsweise kann der grundlegende Entwurf zunächst funktional sein, und im Wesentlichen in der Anwendung von Funktionen auf einfache Objekte bestehen.

### 2.11.3 J/mp

Die Konzepte, die Budd in Leda umgesetzt hat, wurden in neueren Arbeiten von ihm auf Java übertragen (Budd, 2002). Die resultierende Sprache J/mp stellt eine Erweiterung von Java zur Multiparadigmen-Programmierung dar. Folgende fünf Erweiterungen wurden Java hinzugefügt:

– Funktionen „erster Klasse“, also Funktionen, die als Parameter an Funktionen übergeben, von Funktionen zurückgegeben werden und in Datenstrukturen gespeichert werden können. Diese Funktionen werden außerhalb von Klassen definiert; normale Klassenmethoden können nicht in der erweiterten Form benutzt werden.

– Operator-Überladung, also die Möglichkeit, die eingebauten Operatoren von Java für eigene Klassen umzudefinieren. Dies ist hauptsächlich eine syntaktische Erweiterung.

– Pattern-Matching wie in funktionalen Sprachen. Der **instanceof**-Operator wurde so erweitert, dass damit nicht nur die Zugehörigkeit zu einer Klasse getestet, sondern gleichzeitig die Bestandteile eines Objekts extrahiert werden können.

– Relationales Programmieren. Relationen wurden als Datentyp hinzugefügt und in Kombination mit o.g. Spracherweiterungen und einer Bibliothek für Backtracking und Unifikation ist Prolog-artige logische Programmierung möglich.

Die Übertragung der Multiparadigmenkonzepte von Leda nach Java wurde so durchgeführt, dass die sonstigen Eingenschaften von Java erhalten wurden.

Die Implementierung besteht aus einem Präprozessor, der alle Spracherweiterungen von J/mp in Java-Sprachkonstrukte übersetzt. Auf diese Weise sind J/mp-Programme in hohem Maße kompatibel zu Java und überall dort lauffähig, wo eine Java-Implementierung existiert.

### 2.11.4 G, G-2 und GED

G war der ursprüngliche Sprachentwurf von Placer für eine von Grund auf neuentwickelte Multiparadigmensprache. Die Nachfolgersprache wurde G-2 (Placer, 1992) genannt und die neueste Version der Sprachfamilie heißt GED.

Allen diesen Sprachentwürfen ist gemein, dass sie auf einer ausdrucksbasierten Programmierung mit verzögerter Auswertung basieren, ähnlich wie nicht-strikte funktionale Programmiersprachen.

*Stream*-Ausdrücke werden zu einer verzögerten Liste ausgewertet. Das bedeutet, dass die einzelnen Elemente der Liste erst berechnet werden, wenn ihre Auswertung für den weiteren Programmablauf unvermeidlich ist. Folgende Beispiele für die Auswertung zweier Stream-Ausdrücke ist Placer (1992) entnommen:

```
-->[1,"hi",['a',12.3]].
[ 1 "hi" ['a' 12.3] ]
-->[local[x:0], x, x+1, x]
[ 0 1 0]
```

Der zweite ausgewertete Ausdruck lässt erkennen, dass es möglich ist, innerhalb eines Stream-Ausdrucks lokale Variablen zu definieren.

Die wesentliche Neuerung im Design der Sprache G-2 im Vergleich zu G ist das Sprachkonstrukt *Block*. Block-Ausdrücke ähneln den oben eingeführten Stream-Ausdrücken, für sie gelten aber andere Auswertungsregeln. Innerhalb eines Blocks werden alle Berechnungen strikt ausgeführt und die Anweisungen im Rumpf des Blocks werden sequenziell von links nach rechts ausgeführt. Weiterhin sind in Blöcken im Gegensatz zu Stream-Ausdrücken imperative Zuweisungen an Variablen erlaubt und die Auswertung eines Block resultiert in einem einzigen Wert, nicht in einem Strom von Werten. Das folgende Beispiel soll dies illustrieren.

```
-->{local[x:0]; x:=x+1;x}.
1
```

Für die Interaktion von Stream-Ausdrücken und Blöcken definiert Placer vier Regeln, die den folgenden Möglichkeiten zur Schachtelung dieser beiden Sprachkonstrukte entsprechen:

1. Stream geschachtelt in Stream,
2. Stream geschachtelt in Block,
3. Block geschachtelt in Stream und
4. Block geschachtelt in Block.

Die ersten drei Fälle werden ähnlich behandelt: Das eingeschachtelte Konstrukt erbt die Variablen und Bindungen des äußeren, darf diese Bindungen aber nicht durch Zuweisungen modifizieren. Im letzten Fall darf der innere Block die Variablen des äußeren Blocks verändern. Dies ermöglicht auch die objekt-orientierte Programmierung in G-2, indem Objekte als Blöcke repräsentiert werden. Neben funktionaler, imperativer und objekt-orientierter Programmierung unterstützt G-2 auch die logische Programmierung, indem Streams als Relationen behandelt werden, die durch Filterausdrücke verknüpft werden können. Ein Filterausdruck ist ein Streamausdruck, dem ein Prädikat in eckigen Klammern nachgestellt wird. Alle Werte, die das Prädikat nicht erfüllen, sind in dem resultierenden Stream nicht enthalten.

Aus der Literatur wird nicht klar, wie innerhalb von Blöcken mit Streams gearbeitet werden soll, da alle Verzögerungen aufgehoben werden. Dies scheint besonders ein Problem zu sein, wenn ein Programm auf höherer Ebene ein Block ist (beispielsweise für die Interaktion mit dem Benutzer), aber darunter liegend verzögerte Streams verwendet werden sollen.

Westbrook (1999) gibt einige Beispiele zur Programmierung in unterschiedlichen Paradigmen mit GED, der dritten Variante der Sprachfamilie. Alle Beispiele implementieren ein Unterprogramm zum Löschen eines Elementes *x* aus einer Liste *list*. In GED werden Listen in eckige Klammern eingeschlossen, „`[]`“ steht für die leere Liste und „`|`“ ist der Verkettungsoperator für Listen.

Die erste, imperative Lösung nutzt eine Schleife und eine Zuweisung um die Ergebnisliste aufzubauen:

```
func delete(x, list) {
  local [a:[]];
  foreach i in list do
    if (i != x) then
      a := a | [i];
    end
  end;
  a;
}.
```

Die funktionale Variante ist rekursiv definiert:

```
func delete(x, list)
  if list then
    if (x = head(list)) then
      delete(x, tail(list))
    else
      [head(list)] | delete(x, tail(list))
    end
  end.
```

Für die logische Implementierung wird eine logische Regel `delete` definiert. Die Regel instanziiert zunächst die lokale Variable `temp` mit jedem Element der Liste, indem `temp` als Filterausdruck für die Liste `list` verwendet wird. Dann wird geprüft, ob `temp` mit dem zu löschenden Element identisch ist. Wenn nicht, ist `temp` ein Ergebnis und wird der Ergebnisliste hinzugefügt. Die Listenelemente, die durch den Filterausdruck ausgewählt wurden und ungleich `x` sind, werden in einer Liste gesammelt und als Ergebnis zurückgeliefert.

```
func delete(x, list) {
  local [temp];
  list[temp], temp != x -> temp
}.
```

Zur Illustration der objekt-orientierten Programierung in GED wird eine Klasse definiert, die eine Liste repräsentiert und eine Methode zum Löschen eines Elements besitzt. Zunächst wird eine Unterklasse von `List` definiert, dann wird ein Konstruktor sowie eine Methode zum Hinzufügen eines Elements und die Methode zum Löschen hinzugefügt.

Interessant an diesem Beispiel ist neben der Möglichkeit zur objekt-orientierten Programmierung vor allem die Mischung unterschiedlicher Paradigmen. Die Grundstruktur des Programms ist objekt-orientiert, es werden aber auch imperative Zuweisungen, Funktionen (wie

z.B. der *append*-Operator) und logische Konstrukte zum Filtern von Listen (z.B. `list[<x]`) verwendet.

```
addtype (MyList, List, local[list], #, #).

func {MyList} init(me, x:[])
  list := x.

func {MyList} add(me, x)
  list := list | [x].

func {MyList} delete(me, x)
  list := list[<x] | list[>x].
```

### 2.11.5 Oz

Oz (Smolka, 1995) vereint logische, funktionale, sowie objekt-orientierte Programmierung. Die Sprache basiert auf AKL (siehe Abschnitt 2.7.1). Die Erweiterungen bestehen aus Funktionen höherer Ordnung sowie einem Zustand, der objekt-orientierte Programmierung ermöglicht. Die Programmierung mit Oz wird von Müller u.a. (1995) folgendermaßen beschrieben: Eine Berechnung ist als ein Netzwerk nebenläufiger Objekte organisiert, die miteinander und mit der umgebenden Welt (Ein-/Ausgabe) interagieren. Innerhalb der Objekte werden Funktionen und Prädikate verwendet, um zustandsloses Wissen in algorithmischer oder deklarativer Form zu verarbeiten.

Nichtdeterministische Berechnungen werden in Oz explizit durch Sprachkonstrukte für logische Disjunktionen sowie für die Suche beschrieben. Durch die Nebenläufigkeit und die Möglichkeit zur Suche ist es möglich, in vielen Fällen, in denen äquivalente Prolog-Programme keine Lösung finden, mit Oz alle Lösungen zu berechnen.

Objekte in Oz sind Prozeduren, die eine Zelle referenzieren, die den aktuellen Zustand des Objektes repräsentiert. Zellen können in einer ununterbrechbaren Operation modifiziert werden. Da Objekte Prozeduren sind, entspricht das Versenden einer Nachricht einem Prozeduraufruf, bei dem die entsprechende Methode mit dem aktuellen Zustand und einer logischen Variable, die den Zustand zum Methodenende aufnimmt, aufgerufen wird. Der Rumpf einer Methode wird so ausgeführt, dass der Zustand von einer Anweisung zur nächsten „durchgefädelt“ wird, so dass er alle Attributzugriffe und -zuweisungen widerspiegelt.

Haridi, Van Roy und Smolka haben Oz zu einer verteilten Variante namens *Distributed Oz* erweitert (Haridi u.a., 1996). Durch die Einführung von *Mobilität* (Objekte können verteilt sein) und *asynchrone, geordnete Kommunikation* (zur Kommunikation zwischen verteilten Objekten) können verteilte Applikationen erstellt werden, ohne dass die Verteilung der Komponenten explizit formuliert werden muss.

### 2.11.6 LIFE

LIFE (Aït-Kaci und Podelski, 1993) ist eine Weiterentwicklung von Prolog. Im Gegensatz zu Prolog basiert LIFE nicht auf Termen (erster Ordnung), sondern auf sog. $\psi$-Termen.

$\psi$-Terme bezeichnen Mengen von Werten und jeder Term gehört zu einer Sorte, wobei Sorten in einer partiellen Ordnung stehen, die durch eine Untermengenrelation definiert ist. Die größte Sorte heißt $\top$ („Top“), die kleinste Sorte, die die leere Menge bezeichnet, $\bot$ („Bottom“). Regelanwendungen werden wie in Prolog durch Resolution ausgewertet, wobei die Unifikation auf $\psi$-Terme erweitert wurde. Während Terme in Prolog durch den Namen des Operatorsymbols (Funktor) und die Anzahl der Argumente (Arität) bezeichnet werden, haben $\psi$-Terme keine festgelegte Arität. $\psi$-Terme können beliebig um Attribute ergänzt werden, die einen (numerischen oder alphanumerischen) Schlüssel mit einem Wert assoziieren.

In dem folgenden Beispiel wird ein $\psi$-Term definiert, der drei Attribute besitzt. Der Wert des Attributs `number_of_wheels` (4) ist eine Spezialisierung der Sorte *int*, und durch Spezialisierungen der anderen Sorten *string* und *real* ließen sich aus dem $\psi$-Term `car` Terme ableiten, die speziellere Autos beschreiben. Auf diese Weise stellt LIFE einen Mechanismus zur Vererbung zur Verfügung, der für objekt-orientierte Programmierung benutzt werden kann.

```
car(number_of_wheels => 4,
    manufacturer => string,
    maximum_speed => real)
```

Neben der nichtdeterministischen Programmierung durch Prädikate wie in Prolog unterstützt LIFE auch die Definition von Funktionen. Funktionen werden immer deterministisch ausgewertet, wobei die passende Funktionsdefinition für einen Funktionsaufruf durch Pattern-Matching ausgewählt wird. Wenn ein Parameter nicht ausreichend instanziiert ist, wird der Funktionsaufruf wie in Oz (siehe Abschnitt 2.11.5) verzögert (residuiert). Eine interessante Eigenschaft der Funktionen in LIFE ist, dass diese mit weniger als den erforderlichen Parametern aufgerufen werden können (*currying*), wie in funktionalen Sprachen üblich. Darüberhinaus ist es aber auch möglich, nicht nur ein oder mehrere Parameter am Ende der Parameterliste wegzulassen, sondern auch am Anfang oder in der Mitte. Da Funktionsdefinitionen und -aufrufe $\psi$-Terme sind, können Parameter über ihren Attributnamen oder die Attributposition angegeben werden. Der $\psi$-Term:

```
+(2 => 1)
```

benennt eine Funktion, die zu ihrem Argument 1 addiert, indem sie die Funktion `+` in ihrem zweiten Parameter spezialisiert.

Für die eingebauten (binären) arithmetischen Operatoren und Funktionen stellt LIFE eine weitere Eigenschaft zur Verfügung. Diese Funktionen sind zwar ebenfalls deterministisch, sind aber darüberhinaus in der Lage, einen der Parameter zu berechnen, wenn der andere Parameter und das Ergebnis feststeht. So führt die Ausführung des Ziels `0 = B - C` dazu, dass `B` und `C` unifiziert werden. Diese Eigenschaft ermöglicht eine eingeschränkte Form der Constraint-Programmierung.

LIFE verbindet die logische und funktionale Programmierung und stellt einen Mechanismus zur Verfügung, der objekt-orientierte Programmierung unterstützt. Darüberhinaus ist (in bescheidenem Maße) Constraint-Programmierung möglich. LIFE kann also als Multiparadigmensprache bezeichnet werden.

## 2.12 Diskussion

In diesem Kapitel wurden Programmiersprachen betrachtet, die entweder explizit als Multiparadigmensprachen entwickelt wurden, oder die aufgrund der Integration zweier oder mehrerer Paradigmen als solche bezeichnet werden können.

Drei Gruppen von Programmiersprachen lassen sich aus den untersuchten Sprachen bilden: zum ersten deklarative Sprachen, die logische, funktionale und teilweise auch constraintbasierte Sprachen vereinigen, zum zweiten Sprachen, welche deklarative und imperative Paradigmen auf die eine oder andere Weise verbinden und drittens die Gruppe der „echten“, da als solche entwickelten, Multiparadigmen-Programmiersprachen.

Für die deklarativen Sprachen lässt sich sagen, dass die Integration sowohl auf syntaktischer als auch semantischer Ebene gelungen ist, was allerdings aufgrund der Ähnlichkeit der einzelnen deklarativen Sprachen nicht so problematisch ist, wie bei den anderen. Bei der Kombination deklarativer und imperativer Sprachen lassen sich zwei Strategien beobachten: entweder wird die deklarative von der imperativen Ausführungsweise möglichst entkoppelt, um so die rein deklarative Semantik beizubehalten (z.B. durch die Verwendung von Monaden), oder es wird versucht, eine Semantik zu definieren, die den Anforderungen beider Berechnungsmodelle entspricht (siehe z.B. Boehm (1982)).

Die Integration der deklarativen Sprachen ist theoretisch zufriedenstellend, da für die einzelnen Paradigmenkombinationen die Semantik der Sprachen formal definiert wurde und die Fähigkeiten und Einschränkungen (Vollständigkeit, Entscheidbarkeit, etc.) bekannt sind. Für die imperativen und deklarativen Kombinationen gibt es teilweise ebenfalls formale Spezifikationen (z.B. Seiteneffekte in Standard ML, monadische Integration von Ein-/Ausgabe), oftmals wurden imperative Sprachkonstrukte aber auch ad hoc integriert, ohne die semantischen Auswirkungen zu benennen. Die Multiparadigmensprachen Leda, J/mp und GED besitzen keine formale Definition; die Effekte, die diese Sprachkombinationen nach sich ziehen, sind, wenn überhaupt, nur informell aufgeführt. Die Problematik der formalen Semantik von Multiparadigmensprachen soll in Kapitel 3 näher betrachtet werden.

In jedem Fall ist zu erwähnen, dass die eingeschränkten Möglichkeiten zur Ausführung von Programmen (endlicher Speicher, Geschwindigkeit der Computer) in keiner der betrachteten Programmiersprachen berücksichtigt werden. Zwar definieren einige Sprachen einen Mechanismus zur Behandlung von Ausnahmesituationen (Exceptions), dessen Semantik ist aber selten eindeutig definiert. Ein Beispiel für diese Problematik ist ein möglicher Programmfehler in einer verzögerten funktionalen Sprache: Da der Fehler bei der Auswertung eines verzögerten Ausdrucks auftreten kann, ist es schwierig, zu einem geeigneten Zeitpunkt darauf zu reagieren.

Diese Problematik macht deutlich, dass es für eine sichere realistische Programmiersprache notwendig ist, auch für solche Situationen Lösungen zu bieten, die einfach anzuwenden, effizient und sicher sind.

# 3 Syntax und Semantik

Nachdem im vorangegangenen Kapitel konkrete Sprachen untersucht wurden, sollen in diesem Kapitel allgemeine syntaktische und semantische Kriterien von Multiparadigmen-Programmiersprachen näher beschrieben und verglichen werden.

## 3.1 Syntax

Generell lässt sich für alle Programmiersprachen sagen, dass sie eine

- ausdrucksstarke,
- kurze bzw. prägnante,
- lesbare und
- möglichst an bestehenden Sprachen orientierte

Syntax zur Verfügung stellen sollten. Dies gilt natürlich im gleichen Maße für Multiparadigmensprachen, die allerdings eine Problematik aufweisen, die für einzelparadigmatische Sprachen weniger ins Gewicht fällt: Multiparadigmensprachen vereinigen nicht nur verschiedene Paradigmen auf der semantischen Ebene, diese Paradigmen müssen sich auch geeignet ausdrücken lassen. Daraus resultieren im Allgemeinen mehr Konzepte und Notationen als in Einzelparadigmensprachen, die sich unter Umständen widersprechen. Die notwendige Integration ist zumindest für die vierte oben genannte Anforderung problematisch und erfordert bei der zweiten Anforderung Kompromisse.

Die in Kapitel 2 beschriebenen Programmiersprachen verfolgen bei der Sprachintegration unterschiedliche Philosophien. Zum einen existieren minimalistische Syntaxkonzepte, die möglichst wenige, dafür aber mächtige Konstrukte mit den dazugehörigen Kombinationsmöglichkeiten zur Verfügung stellen, um so ein möglichst klares und überschaubares Gesamtbild der Sprache zu erreichen. Die funktional-logische Sprache Curry (siehe Abschnitt 2.6.1) sieht z.B. keine syntaktisch offensichtliche Unterscheidung zwischen nichtdeterministischen und deterministischen Regeln vor und sieht daher aus wie eine funktionale Sprache. Ob eine Berechnung deterministisch oder nichtdeterministisch abläuft, hängt davon ab, ob der auszuwertende Term freie Variablen enthält oder nicht.

Auf der anderen Seite gibt es Sprachen, welche die Sprachelemente der unterschiedlichen Paradigmen auch auf syntaktischer Ebene unterschiedlich darstellen, um die Semantikunterschiede deutlich hervorzuheben. Ein Beispiel dafür ist die Sprache Gödel (siehe Abschnitt 2.6.2), welche zwischen nichtdeterministischen Prädikaten und deterministischen Funktionen unterscheidet. Ein Vorteil dieser Herangehensweise ist ein erhöhter Dokumentationswert, da durch eine syntaktische Unterscheidung das Verhalten einzelner Programmteile verdeutlicht wird. Weiterhin hilft das Wissen, das der Programmierer durch die Verwendung unterschiedlicher Konstrukte explizit in das Programm schreibt, einerseits dem

Compiler beim Übersetzungs- und Optimierungsvorgang, andererseits menschlichen Lesern beim Verständnis und bei der Wartung des Programmcodes..

### 3.1.1 Minimale Anforderungen

Entsprechend den oben genannten Überlegungen können ein paar Grundsätze aufgestellt werden, die beim Programmiersprachenentwurf im Allgemeinen, aber auch im Speziellen bei Multiparadigmensprachen berücksichtigt werden sollten. Folgende minimale Anforderungen sollte die Syntax einer multiparadigmatischen Programmiersprache erfüllen:

- Die Syntax sollte deutlich machen, welche Effekte ein gegebenes syntaktisches Konstrukt hat: handelt es sich um eine Deklaration (relevant zur Kompilierzeit) oder eine Anweisung (ausgeführt zur Laufzeit); welche Seiteneffekte hat eine Anweisung etc.
- Im gleichen Maße sollte anhand der Syntax erkennbar sein, welches Paradigma bei der Formulierung eines einzelnen Programmteils im Vordergrund stand, um die Wartbarkeit zu erleichtern.
- Der Kompromiss zwischen Knappheit und Ausführlichkeit muss so geschlossen werden, dass die Sprache sich sowohl gut schreiben als auch gut lesen lässt.
- Alle Sprachelemente, die mit gleicher Bedeutung in etablierten Sprachen zur Verfügung stehen, sollten auch in neuen Sprachen gleich aussehen, sofern dadurch keine Konflikte mit anderen Elementen entstehen. Dies erleichtert das Erlernen der Sprache und beugt Missverständnissen vor.
- Die Syntax sollte auch in einer Multiparadigmensprache die Programmierung in einem einzelnen Paradigma erlauben, und zwar mit der syntaktischen Unterstützung, die von etablierten Einzelparadigmensprachen geboten wird.

Ob die bestmögliche Programmiersprache eher ausführlich ist und viele Schlüsselwörter besitzt, oder ob sie eher knappe Symbole zur Formulierung von Programmen anbietet, lässt sich unter Berücksichtigung der Geschichte der Entwicklung von Programmiersprachen nicht abschließend beantworten. Dazu ist eine solche Entscheidung zu sehr abhängig von den Gewohnheiten und dem Geschmack des Sprachdesigners. Die genannten Anforderungen können also nur grundsätzlich die Richtung des Sprachentwurfs beeinflussen und keine präzisen Entwurfsvorschriften darstellen.

### 3.1.2 Ansätze

In der Literatur finden sich wenige Ansätze zum multiparadigmen-spezifischen Syntax-Entwurf. Neben den syntaktisch minimalistischen Sprachen wie z.B. Curry wurden im Wesentlichen Sprachkonstrukte aus bestehenden Sprachen unverändert übernommen, teilweise aber auch leicht abgewandelt.

Budd (1998) beschreibt interessante syntaktische Konventionen seiner Multiparadigmensprache Leda (siehe Abschnitt 2.11.2), die die Integration funktionaler und objekt-orientierter Programmierung vereinfachen. In Leda ist es möglich, normale Funktionen und Methoden beim Aufruf zu mischen, so dass es letztlich egal ist, welche der Konventionen man benutzt. Wenn ein Methodenaufruf der Form

```
receiver.method(param)
```

benutzt wird, prüft der Compiler, ob in der Klasse von `receiver` eine Methode `method` definiert ist. In diesem Fall wird ein normaler Methodenaufruf generiert. Sollte andererseits eine Funktion existieren, welche als ersten Parameter einen Wert der Klasse `receiver` erwartet, wird ein Funktionsaufruf generiert und `receiver` als erster Parameter hinzugefügt (vorausgesetzt natürlich, die anderen Parameter passen zum Typ der Funktion). Der tatsächlich erzeugte Aufruf ist also zu Folgendem äquivalent:

```
method(receiver, param)
```

Ähnlich verfährt der Compiler bei einem Funktionsaufruf

```
func(param1, param2)
```

Sollte keine passende Funktion definiert sein, die Klasse von `param1` aber eine Methode `func` besitzen, so wird für diese Methode ein Aufruf generiert:

```
param1.func(param2)
```

Diese Syntaxerweiterung erlaubt sowohl eine objekt-orientierte als auch eine funktionale Notation und damit eine nahtlose Integration der Interaktionen zwischen Objekten und Funktionen.

Die Sprachen G, G-2 und GED sind Beispiele dafür, dass Sprachkonstrukte anderer Sprachen mit abgewandelter und auf den ersten Blick mehrdeutig erscheinender Bedeutung in eine Multiparadigmensprache übernommen wurden. Die eckigen Klammern `[` und `]` werden sowohl als Listenkonstruktoren als auch als syntaktische Begrenzungselemente und zur Kennzeichnung von Filterausdrücken benutzt. Geschweifte Klammern `{` und `}` dienen ebenfalls als Begrenzungselemente, sowie zur Kennzeichnung imperativer Befehlssequenzen. Diese Vielzahl an unterschiedlichen Bedeutungen führt zur schlechten Lesbarkeit von Programmen. Ein besserer Syntaxentwurf sollte unterschiedliche semantische Konzepte, also z.B. Deklarationen und Ausdrücke, auch in der Schreibweise widerspiegeln.

## 3.2 Semantik

Die Semantiken existierender einzelparadigmatischer Programmiersprachen müssen in einer Multiparadigmensprache gleichberechtigt und sowohl praktisch als auch theoretisch angemessen integriert sein. Dabei sollte kein Paradigma in einem der anderen „ersticken“, also beispielsweise imperative Programmteile nur indirekt durch komplexe Operationen programmiert werden können, wie dies z.Z. in vielen funktionalen Sprachen der Fall ist. Genau so wenig dürfen einzelne Paradigmen nur zweitrangig, z.B. in Form einer Programmbibliothek, in der Gesamtsprache repräsentiert sein.

Generell lässt sich bemerken, dass bestimmte Paradigmen sich sehr gut zur Kombination mit anderen Paradigmen eignen. Dies trifft vor allem dann zu, wenn die zu integrierenden Paradigmen auf verschiedene Aspekte der Problemlösung spezialisiert sind. So ist beispielsweise die Erweiterung verschiedener Paradigmen um objekt-orientierte Konzepte wie Objekte, Klassen und Vererbung oder die Integration von Nebenläufigkeit durch das Hinzufügen

weniger sprachlicher Mittel aus heutiger Sicht gut untersucht. Für fast alle funktionalen, (constraint-)logischen oder imperativen Sprachen existieren entsprechende Erweiterungen.

Deklarative Sprachen lassen sich aufgrund ihrer ähnlichen Eigenschaften gut miteinander verbinden, was sich in zahlreichen Arbeiten zur constraint-logischen und funktional-logischen Programmierung niederschlägt. Problematisch, da noch nicht so tiefgehend untersucht, ist dagegen die Kombination deklarativer und imperativer Programmiersprachen. Beide Kombinationsmöglichkeiten sollen in den folgenden Abschnitten genauer untersucht werden.

## 3.3 Semantik deklarativer Sprachen

Die Integration funktionaler und logischer Sprachen ist weit fortgeschritten, es gibt theoretisch fundierte und praktikable Implementierungen (Albert u.a., 2002; Lloyd, 1995; Müller u.a., 1995).

Funktional-logische Sprachen verwenden prinzipiell die Reduktionssemantik funktionaler Sprachen und unterscheiden sich vor allem bei der Behandling nichtdeterministischer Regelauswertungen. Zwei Auswertungsmechanismen haben sich hierbei etabliert:

**Residuation** wird u.a. in Escher, Le Fun, Life, NUE-Prolog und Oz verwendet. Funktionsaufrufe werden durch deterministische Funktionsaufrufe ausgewertet; das erfordert, dass die Argumente soweit instanziiert sind, dass eine eindeutige Regelauswahl möglich ist. Wenn ein Argument nicht ausreichend an einen Wert gebunden ist, wird die Regelanwendung verzögert. Sobald ein separater Berechnungsvorgang das Argument instanziiert, kann die Berechnung fortgesetzt werden. Diese Verzögerung bedeutet, dass nichtdeterministische Suche durch Prädikate oder explizite Disjunktionen dargestellt werden muss, wenn sie erwünscht ist.

**Narrowing** wird u.a. in Curry und CFLP($\mathcal{R}$) verwendet. Beim Narrowing wird das Pattern-Matching funktionaler Sprachen durch Unifikation ersetzt. Dadurch können auch unvollständig instanziierte Regelanwendungen ausgeführt werden, die formalen Parameter werden in diesem Fall einfach an die Variablen gebunden, welche die nicht instanziierten Parameter darstellen. Da durch die Unifikation Informationen nicht nur in eine Regel hinein-, sondern auch herausfließen können, sind größere Modifikationen der Reduktionssemantik funktionaler Sprachen notwendig als bei der Residuation. Der Vorteil des Narrowing-Verfahrens besteht darin, dass die Effizienz deterministischer Funktionen mit den Suchmöglichkeiten und dem Umgang mit unvollständigen Informationen logischer Sprachen kombiniert wird.

In der Sprache Curry ist es durch optionale Deklarationen möglich, für einzelne Regeln anzugeben, mit welcher Auswertungsstrategie diese auszuwerten sind. Dies ermöglicht bei der Übersetzung eine Auswahl zwischen Residuation und Narrowing zur Anwendung bestimmter Regeln.

Hanus hat ein Berechnungsmodell für deklarative Sprachen entwickelt (Hanus, 1997), das funktionale und logische Berechnungsmodelle formal vereinigt. Sowohl Reduktion als auch nicht-deterministische Suche sowie Residuation und Narrowing werden kombiniert. Der Vorteil dieses Ansatzes ist, dass sowohl die effiziente funktionale Auswertung als auch die Suchmöglichkeiten logischer Sprachen genutzt werden können.

Die Integration von constraint-basierter Programmierung und der logischen sowie der funktionalen Programmierung wurde in mehreren Schritten vollzogen. Der Übergang von der logischen zur constraint-logischen Programmierung besteht im Wesentlichen aus der Erweiterung uninterpretierter Terme (Herbrand-Terme) um interpretierte Terme des Problembereichs (engl. *domain of discourse*) (Chakravarty u.a., 1997, S. 4). Darauf aufbauend lässt sich die Einbindung deterministischer Funktionen, also der Übergang von der constraint-logischen zur funktional-constraint-logischen Programmierung ähnlich erfassen: interpretierte Terme werden von eingebauten Funktionen auf generelle Ausdrücke (auch definierte Funktionen) erweitert. (Chakravarty u.a., 1997, S. 4). Ein ähnlicher Ansatz wird beim *Definitional Constraint Programming* verfolgt (Guo und Pull, 1993). Dieser Ansatz wird in der Sprache Falcon verwendet (siehe Abschnitt 2.8.2).

## 3.4 Semantik deklarativer und imperativer Sprachen

Die Kombination deklarativer und imperativer Berechnungen ist aufgrund ihrer unterschiedlichen Eigenschaften bezüglich des Zeitverhaltens eines Programms problematisch. Deklarative Programme sind unabhängig von der Zeit, wodurch die Auswertung dieser Programme weniger Einschränkungen unterworfen ist. Dies führt zu einer Reihe von Vorteilen, wie z.B. der Möglichkeit zur verzögerten Auswertung, die gegenüber imperativen Sprachen neue Programmiertechniken ermöglichen, beispielsweise den Umgang mit unendlichen Datenstrukturen. Durch die Integration imperativer Berechnungen, die durch Seiteneffekte charakterisiert sind, gehen diese Möglichkeiten bei einer direkten Kombination verloren, da der Einsatz von Seiteneffekten eine vorhersehbare zeitliche Abarbeitung der Programmschritte erfordert. Wenn man die Nachteile betrachtet, die mit der Einbeziehung von seiteneffektbehafteten Operationen einhergehen, stellt sich die Frage, ob eine solche Paradigmen-Kombination überhaupt sinnvoll und wünschenswert ist. Viele Gründen sprechen aber dafür.

Zunächst ist die reale Welt zustandsbehaftet, und eine Programmiersprache, welche keine Interaktion mit ihrer Umgebung erlaubt, kann nicht für realistische Programme eingesetzt werden. Imperative Seiteneffekte sind also zur Realisierung praktischer Programme zwingend erforderlich.

Weiterhin bieten Seiteneffekte aber auch in einem deklarativen Umfeld Vorteile, wie Swarup u.a. (1991) ausführen: Die imperative Programmierung erlaubt eine effiziente Darstellung dynamischer Daten, z.B. die Abbildung sich ändernder realer physikalischer Messwerte. Gemeinsamer, geteilter Speicher (engl. *shared data*) erlaubt weiterhin verschiedene Zugriffspfade auf ein und dasselbe Datenobjekt. Änderungen werden sofort an allen Programmpunkten sichtbar, die Zugriff auf die Daten haben. Auf diese Weise lassen sich viele physikalische Prozesse viel effizienter modellieren als rein deklarativ.

Oft sind Seiteneffekte auch die einzige Möglichkeit, bestimmte Probleme in deklarativen Sprachen vergleichbar effizient zu lösen, wie dies in imperativen Sprachen möglich ist. Launchbury (1993) nennt als Beispiel Graphtraversierungsalgorithmen wie Tiefensuche oder topologische Sortierung.

Swarup u.a. (1991) betonen, dass die Integration von Seiteneffekten in deklarative Sprachen symmetrisch sein muss: die deklarative und die imperative Untersprache sollen in beide Richtungen eingebettet werden können.

Zustandsbasierte Berechnungen lassen sich in zwei Arten aufteilen (Launchbury, 1993):

– Externe zustandsbasierte Berechnungen sind außerhalb des Programms sichtbar und müssen daher auf jeden Fall und in der richtigen Reihenfolge abgearbeitet werden. Beispiele für diese Berechnungen sind Operationen im Dateisystem oder die Interaktion mit Benutzern oder anderen Programmen.

– Interne zustandsbasierte Berechnungen – z.B. die Generierung von eindeutigen Namen bei der Kompilierung, Pseudo-Zufallsgeneratoren oder effiziente Hash-Tabellen – haben nur innerhalb eines eingeschränkten Bereichs des Programmablaufs Auswirkungen. Der Zustand, auf dem die Veränderungen ausgeführt werden, wird nach Beendigung der Berechnung verworfen. Da die Änderungen von außerhalb nicht sichtbar sind, müssen diese Berechnungen auch nur dann ausgeführt werden, wenn sie zum sichtbaren Ergebnis beitragen.

Beide Arten von zustandsbasierten Berechnungen können gleich behandelt werden, interne Berechnungen lassen sich aber aufgrund ihres eingeschränkten Wirkungsbereichs optimieren, indem Teilberechnungen nicht ausgeführt werden.

Zur Integration deklarativer und imperativer Sprachen werden verschiedene Ansätze verfolgt, welche im Folgenden einzeln beschrieben werden:

– Interaktionen eines Programms werden durch synchronisierte Ströme (verzögerte Listen) dargestellt, ein Strom wird für die Eingabe des Programms und ein Strom für die Ausgabe genutzt.

– Die zeitliche Abfolge der einzelnen Berechnungen wird durch die Aufrufreihenfolge erzwungen, indem seiteneffektbehaftete Operationen den Rest des Programms als sogenannte *Continuation* als Parameter übergeben bekommen.

– Die reale Welt wird durch *eindeutige Typen* modelliert, die durch ihre Eigenschaften eine korrekte Ausführungsreihenfolge von Seiteneffekten garantieren.

– Die deklarative Programmausführung und die Ausführung seiteneffektbehafteter Operationen werden getrennt, indem zwischen *Berechnungen* (engl. *calculations*) und *Aktionen* (engl. *actions*) unterschieden wird. Dieser Ansatz basiert auf *Monaden*.

– Deklarative Kalküle werden um *seiteneffektbehaftete primitive Operationen* erweitert.

– Ein anderer Ansatz besteht darin, eine Programmiersprache anhand der Seiteneffekte, die bestimmte Sprachkonstrukte haben können, in mehrere Untersprachen zu unterteilen: *Berechnungs-* und *Koordinationssprachen*.

Zwischen diesen Konzepten gibt es auch Überschneidungen, und man kann auch mehrere dieser Ansätze zusammenfassen, z.B. stellt die Trennung in seiteneffektfreie und seiteneffektbehaftete Operationen auch eine Aufteilung in Teilsprachen dar. Im Folgenden sollen aber die Unterschiede der Ansätze, die sich auch in der Anwendbarkeit in der Praxis auswirken, beschrieben werden.

### 3.4.1 Synchronisierte Ströme

Wadler (1997) beschreibt die Integration von Seiteneffekten durch synchronisierte Ströme.

Bei der Modellierung von Interaktionen durch synchronisierte Ströme ist ein Programm eine Funktion, die Eingaben auf Ausgaben abbildet. Diese Funktion wird auch *Dialog* genannt. Sowohl Ein- als auch Ausgabe werden als verzögerte Listen (Ströme) dargestellt. In Haskell-Syntax ist ein Programm also eine Funktion des Typs:

```
type Dialogue = [Response] -> [Request]
```

wobei `Request` Anfragen zur Interaktion darstellt und `Response` die Antworten der Umgebung. Ein Programm interagiert mit seiner Umgebung, indem es Anfragen an diese stellt und die Antworten darauf als Eingabe verarbeitet.

Das wesentliche Problem dieser Modellierung ist, dass die Datentypen `Request` und `Response` von vornherein festgelegt sind und alle Interaktionsmöglichen abdecken müssen. Weiterhin ist es schwierig, beim Programmieren die Synchronisation der beiden Ströme – schließlich muss jede Antwort zu der zugehörigen Anfrage passen – aufrecht zu erhalten.

Synchronisierte Ströme wurden in den ersten Versionen der Programmiersprache Haskell sowie in einigen weiteren rein funktionalen Sprachen eingesetzt, sind aber aufgrund ihrer mangelnden Flexibilität in neueren Sprachen nicht mehr verwendet worden.

### 3.4.2 Continuations

Im Continuation-Modell (Wadler, 1997) erhält jede Operation ein zusätzliches Argument, genannt die Continuation (Fortsetzung), die den gesamten Rest der Berechnung darstellt. Dieses Argument kann entweder ein abstrakter Datentyp, oder auch eine Funktion sein, die aufgerufen wird, um den Rest der Berechnung auszuführen. In Wadlers Darstellung werden Continuations durch den abstrakten Datentyp `Answer` dargestellt, der das Ergebnis einer Berechnung verkörpert. Die primitiven seiteneffektbehafteten Operationen zum Lesen bzw. Schreiben eines Zeichens haben folgende Typen:

```
putc :: Char -> Answer -> Answer
getc :: (Char -> Answer) -> Answer
```

Das Schreiben eines Zeichens führt die Operation aus und verhält sich wie die übergebene Continuation, während das Lesen eines Zeichens die übergebene Funktion mit dem gelesenen Zeichen aufruft, welche wiederum den Rest der Berechnung darstellt.

Continuations wurden für die funktionale Sprache Hope (Burstall u.a., 1980) verwendet. Das Konzept ist auch als Zwischensprache für Compiler verwendet worden, da der Kontrollfluss explizit dargestellt wird (Appel, 1992). Ähnlich wie bei den synchronisierten Strömen ist die Programmierung mit Continuations aber etwas unübersichtlich, weshalb sie in neueren Sprachen nicht verwendet wurde.

### 3.4.3 Eindeutige Typen

Eindeutige Typen (auch *lineare Typen*) sind eine Erweiterung gängiger Typsysteme, welche die einmalige Verwendung eines jeden Datenobjektes (engl. *single-threadedness*) garantieren. Auf diese Weise werden Berechnungen, die mit Objekten dieser Typen operieren, so serialisiert, dass sie sich zum Ausdrücken imperativer Berechnungen eignen.

Dieser Ansatz wurde bei der Entwicklung einiger deklarativer Sprachen verfolgt, von denen wir uns zwei genauer ansehen wollen: die logische Sprache Mercury und die funktionale Sprache Clean.

### Fallbeispiel Mercury

Die Ein- und Ausgabe bei Mercury wird über ein Datenobjekt serialisiert, das den Zustand der Welt repräsentiert. Jedes Prädikat, welches Ein- oder Ausgabe durchführt, besitzt dieses Objekt als zusätzlichen Ein- und Ausgabeparameter. Beispiel:

```
main(IO_In, IO_Out) :-
  io__write_string("Hello, World!\n", IO_In, IO_Out).
```

Da das „Durchreichen“ dieses Zustandsobjekts schreibaufwändig ist und die Definition des Prädikats unnötig verkompliziert (besonders wenn mehrere Ein-/Ausgabeoperationen in einem Prädikat ausgeführt werden), bietet Mercury eine alternative Notation für Prädikate an: sogenannte *Definite Clause Grammars* (DCG). In dieser Notation werden die Zustandsparameter automatisch an alle enthaltenen Prädikate angehängt, so dass obiges Prädikat sich wie folgt darstellen lässt:

```
main -->
  io__write_string("Hello, World!\n").
```

Für den Fall, dass ein Prädikat innerhalb eines in DCG notierten Prädikats nicht das Zustandsobjekt übergeben bekommen soll, ist es in geschweifte Klammern einzufassen, wie im folgenden Beispiel dargestellt.

```
main -->
  io__write_string("factorial(7) = "),
  { factorial(7, X) },
  io__write_int(X),
  io__nl.
```

### Fallbeispiel Clean

Die rein funktionale Programmiersprache Clean (Brus u.a., 1987) verfügt über ein Typsystem, das eindeutige Typen (eng. *unique types*) unterstützt (Smetsers u.a., 1994). Werte dieser Typen haben die Eigenschaft, *single-threaded* zu sein, also nicht mehrfach verwendet werden zu können. Ähnlich wie die *in*- und *out*-Typen bei Mercury können diese genutzt werden, um den Zustand der Welt zu repräsentieren.

Plasmeijer und van Eekelen (2001) beschreiben eindeutige Typen folgendermaßen: wenn ein Argument einer Funktion als eindeutig deklariert wird, ist garantiert, dass zur Laufzeit das entsprechende Datenobjekt lokal ist, also keine weiteren Referenzen auf dieses Objekt existieren. Da keine Referenzen von außerhalb einer Funktion auf das Objekt existieren, kann eine destruktive Änderung keinen Einfluss auf andere Berechnungen haben und kann durchgeführt werden, ohne die Sicherheit des Programms zu zerstören.

Eindeutige Typen in Clean werden mit einem Stern (`*`) markiert. Als Beispiel für die Anwendung eindeutiger Typen soll eine Funktion zum Schreiben eines Zeichens in eine Datei betrachtet werden (Plasmeijer und van Eekelen, 2001):

```
fwritec:: Char *File -> *File
```

Das zweite Argument ist als eindeutig markiert, kann also destruktiv verändert werden. Das Gleiche gilt für das Resultat der Operation, das verwendet werden kann, um weitere Operationen auf der Datei durchzuführen:

```
WriteABC:: *File -> *File
WriteABC file = fwritec 'c' (fwritec 'b' (fwritec 'a'))
```

Wenn eine Operation keine Seiteneffekte auf ihrem Argument ausführt, sind keine Eindeutigkeitsannotationen notwendig. Die Funktion `freadc`, welche ein Zeichen aus einer Datei liest, ist ein Beispiel dafür:

```
freadc:: File -> (Char, File)
```

Eine Datei, die an `freadc` übergeben wird, kann sowohl eindeutig als auch nicht eindeutig sein.

### 3.4.4 Monaden

Bei der Modellierung von Seiteneffekten durch Monaden konstruiert das deklarative Programm eine Datenstruktur, welche die imperativen Anweisungen enthält und in einer weiteren Ausführungsphase werden diese Anweisungen unter Ausführung der Seiteneffekte abgearbeitet. Da diese Datenstrukturen prinzipiell unendlich groß sein können, existiert diese Ausführungstechnik nur konzeptionell, in der Realität werden beide Phasen nebeneinander ausgeführt. Die Serialisierung, die durch die Unterscheidung der Ausführungsphasen realisiert wird, kann vom Compiler ausgenutzt werden, um Seiteneffekte kontrolliert in die Ausführung der deklarativen Sprachen einzuführen, ohne dass beispielsweise die referenzielle Transparenz verloren geht. Diese Strategie zur Integration findet sich sowohl bei der Implementierung von Ein-/Ausgabe durch verzögerte Streams als auch durch Monaden (Sabry, 1998).

Andere Versuche, imperative Sprachkonstrukte, oder allgemeiner, zustandsbehaftete Berechnungen mit deklarativen Sprachen zu verbinden, sind seltener. Die Verwendung von Monaden für deklarative Ein- und Ausgabe (Wadler, 1997) sowie zustandsbehaftete Berechnungen scheint sich bei (rein) deklarativen Sprachen mittlerweile durchgesetzt zu haben, beispielsweise in der funktionalen Sprache Haskell (Peyton Jones, 2003).

#### Fallbeispiel Haskell

Auch in Haskell werden Seiteneffekte durch ein Datenobjekt, welches den Zustand der „Welt“ darstellt, in die deklarative Sprache eingebunden. Allerdings wird hier nicht explizit dieses Objekt als Parameter bzw. Rückgabewert behandelt, sondern durch sogenannte *monadische Ein- und Ausgabe* versteckt.

Ein- und Ausgaben werden in Haskell durch sogenannte Ein-/Ausgabe-Transformierer ausgeführt, das sind Funktionen, die den aktuellen Zustand der Welt als Eingabe erhalten und ein Ergebnis sowie den veränderten Zustand der Welt zurückliefern. In Haskell-Notation haben diese Transformierer folgenden Typ:

```
type IO a = World -> (a, World)
```

Da Haskell Funktionen höherer Ordnung unterstützt, werden Ein- und Ausgabefunktionen aus den primitiven E/A-Funktionen sowie geeigneten Kombinatoren gebildet. Diese Kombinatoren heißen `>>=` (genannt *bind*) und `return`[1].

```
(>>=) :: IO a -> (a -> IO b) -> IO b
return :: a -> IO a
```

`>>=` kombiniert einen Transformierer sowie eine Funktion, die einen Transformierer konstruiert und erzeugt daraus einen neuen Transformierer, während `return` einen Wert in einen Transformierer des gleichen Typs umwandelt.

Wie man an den Typen von `>>=` und `return` sehen kann, taucht der Typ der „Welt“ bei diesen Transformierern gar nicht mehr auf, und tatsächlich kann ein Haskell-Programm auf diesen Zustand überhaupt nicht zugreifen.

Ein Programm in Haskell ist eine Funktion vom Typ `IO a` und wird dadurch ausgeführt, dass diese Funktion auf den Zustand der Welt angewendet wird. Durch die Kombinatoren wird gewährleistet, dass der Zustand genau einmal durch das gesamte Programm geschleust wird, ohne jemals in einer Datenstruktur gespeichert oder dupliziert zu werden. Dadurch bleibt er eine Repräsentation der Welt und serialisiert die Ein-/Ausgabe-Operationen.

In Haskell sieht das Beispielprogramm, dass in Abschnitt 3.4.3 für Mercury formuliert wurde, folgendermaßen aus:

```
main :: IO a
main = putStr "Hello World\n";
```

### Semantik monadischer Seiteneffekte

Als momentan wichtigster Vertreter der deklarativen Ein- und Ausgabe stellt der monadische Ansatz ein wichtiges Konzept der Programmiersprachen-Integration dar. Aus diesem Grund soll im Folgenden die Semantik monadischer Ein- und Ausgaben formal beschrieben werden. Diese Darstellung dient dem Verständnis der Abläufe monadischer Berechnungen und soll in die Formalismen zu deren Darstellung einführen.

Die Darstellung ist Peyton Jones (2001) entnommen. Da sich Monaden nicht nur zur Ein- und Ausgabe eignen, sondern auch zur Modellierung anderer zustandsbehafteter Operationen, finden sich dort auch Erweiterungen dieser Semantik auf Nebenläufigkeit und Fehlerbehandlung mit Ausnahmebedingungen (engl. *exceptions*). Die Semantik ist keine vollständige Beschreibung der Semantik funktionaler Sprachen mit Seiteneffekten, sondern konzentriert sich auf die Behandlung monadischer Operationen. Vorausgesetzt wird eine Auswertungsfunktion $\mathcal{E}$, welche die Semantik der rein funktionalen Aspekte der Sprache definiert.

Um die Semantikregeln zu beschreiben, muss zunächst der Begriff des *Auswertungskontexts* (engl. *evaluation context*) eingeführt werden. Ein Auswertungskontext E ist ein Term mit einer Lücke (genannt Loch, engl. *hole*), geschrieben E[·]. Der Ausdruck E[$M$] bezeichnet das Ergebnis, wenn das Loch im Term mit $M$ aufgefüllt wird. Das Loch ist immer so in einem Auswertungskontext platziert, dass es angibt, welcher Teil des Terms als nächster

[1] In anderen Programmiersprachen tragen diese Kombinatoren auch andere Namen, wie z.B. *yield* bzw. *lift* für *return*.

**Abbildung 3.1** Syntax von Werten und Termen

| | | | |
|---|---|---|---|
| | $x, y$ | $\in$ | *Variable* |
| | $k$ | $\in$ | *Constant* |
| | $c$ | $\in$ | *Constructor* |
| | $ch$ | $\in$ | *Char* |
| | $d$ | $\in$ | *Integer* |
| Values | $V$ | $::=$ | $\lambda x \to M \mid k \mid cM_1 \ldots M_n \mid ch \mid d \mid$ `return` $M \mid M >\!\!>\!\!= N \mid$ `putChar` $V \mid$ `getChar` |
| Terms | $M, N, H$ | $::=$ | $x \mid V \mid MN \mid \cdots$ |
| Evaluation contexts | $E$ | $::=$ | $[\cdot] \mid E >\!\!>\!\!= M$ |

ausgewertet werden muss. Dies ist notwendig, um die richtige Reihenfolge der Ausführung der verschiedenen Seiteneffekte in einem Term zu garantieren.

Abbildung 3.1 stellt die Syntax einer funktionalen Sprache mit monadischen Seiteneffekten dar. Die seiteneffektbehafteten Aktionen werden als Werte dargestellt und das funktionale Programm generiert diese Werte, um die Ausführung von Seiteneffekten zu veranlassen.

Die Auswertungsregeln sind in Abbildung 3.2 dargestellt. Der Zustand des Programms wird durch einen noch auszuführenden Term $M$ dargestellt, eingeschlossen in geschweifte Klammern: $\{M\}$. Die Auswertung des Programms besteht in der Überführung eines Anfangszustandes anhand der angegebenen Regeln in einen Endzustand, in dem keine weiteren Regeln angewendet werden können.

Anhand eines Beispiels soll die Ausführung eines Programms veranschaulicht werden. Gegeben sei das folgende Programm:

$$\texttt{main} \;=\; \texttt{getChar} \;>\!\!>\!\!=\; \lambda\texttt{c} \to \texttt{putChar c}$$

Zu diesem Term passt der Auswertungskontext $[\cdot] >\!\!>\!\!= \lambda c \to \texttt{putChar}\ c$. Durch Anwendung der Auswertungsregeln ergibt sich folgende Zustandssequenz:

$$\{\texttt{getChar} \;>\!\!>\!\!= \lambda c \to \texttt{putChar}\ c\} \xrightarrow{?ch} \{\texttt{return}\ ch >\!\!>\!\!= \lambda c \to \texttt{putChar}\ c\}$$

$$\longrightarrow \{(\lambda c \to \texttt{putChar}\ c)\, ch\} \longrightarrow \{\texttt{putChar}\ ch\} \xrightarrow{!ch} \{\texttt{return}\ ()\}$$

Die Anmerkungen an den Zustandsübergängen bezeichnen die Seiteneffekte, die durch die Ausführung des Programms verursacht wurden. Dabei steht $!\,ch$ für die Ausgabe eines Zeichens $ch$ und $?\,ch$ für die Eingabe.

Ariola und Sabry (1998) kombinieren imperative und funktionale Programmiersprachen, indem sie beide Teilsprachen durch monadischen Zustand voneinander isolieren. Durch die Verwendung neuer Term- und Typ-Konstruktoren wird die Reihenfolge von Zuweisungen explizit gemacht. Semantisch ist ein Programm mit monadischem Zustand äquivalent zu einem funktionalen Programm, das den änderbaren Zustand explizit herumreicht und teilweise kopiert, um Änderungen zu simulieren. Wichtig für die effiziente Implementierung

**Abbildung 3.2** Auswertungsregeln

$$\{\mathrm{E}[\texttt{putChar}\ ch]\} \xrightarrow{!ch} \{\mathrm{E}[\texttt{return}\ ()]\}$$

$$\{\mathrm{E}[\texttt{getChar}]\} \xrightarrow{?ch} \{\mathrm{E}[\texttt{return}\ ch]\}$$

$$\{\mathrm{E}[\texttt{return}\ N >\!\!>\!\!= M]\} \longrightarrow \{\mathrm{E}[M\ N]\}$$

$$\frac{\mathcal{E}[\![M]\!] = V \quad M \not\equiv V}{\{\mathrm{E}[M]\} \longrightarrow \{\mathrm{E}[V]\}}$$

ist, dass kein Code für die Sequenzialisierung von Zuweisungen generiert werden muss, und ebenfalls kein Code für das Herumreichen des Speichers. Im Wesentlichen umfasst die Arbeit die Erweiterung des *call-by-need* $\lambda$-Kalküls um imperative änderbare Variable.

#### Monaden in anderen Programmiersprachen

Monaden werden nicht nur zur Beschreibung rein funktionaler Sprachen wie Haskell eingesetzt. Die Sprache Standard ML integriert imperative Berechnungen durch änderbare Speicherzellen. Daher lassen ML-Programme sich nicht so gut algebraisch manipulieren wie beispielsweise Haskell-Programme. Basierend auf monadischem Zustand wurde eine ML Sprache mit Zuweisungen und einem Operator zur Kapselung von Seiteneffekten entwickelt (Semmelroth und Sabry, 1999). Dieser Ansatz benutzt ein Typsystem mit vereinfachter Effektinferenz und Effektmaskierung, ähnlich dem von Lucassen und Gifford (1988) (siehe Abschnitt 3.4.6).

### 3.4.5 Imperative Erweiterungen des $\lambda$-Kalküls

Einige funktionale Sprachen, z.B. Standard ML oder Lisp, besitzen keine Vorkehrungen, um die Eigenschaften rein deklarativer Sprachen zu erhalten. Seiteneffekte werden einfach durch primitive Operationen, die neben der Berechnung eines Ergebnisses die Umgebung beeinflussen, eingeführt.

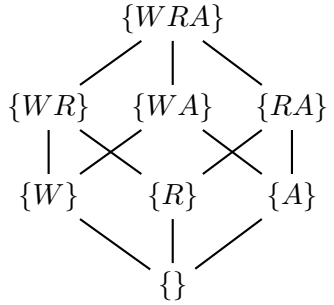
Es gab aber auch Bestrebungen, Seiteneffekte in kontrollierter Art und Weise in das funktionalen Sprachen zu Grunde liegende Kalkül einzubetten und so ein solides theoretisches Fundament zu schaffen. Die folgenden Arbeiten unterscheiden sich von bereits erwähnten Erweiterungen des $\lambda$-Kalküls dadurch, dass sie nicht durch Monaden modelliert werden.

Felleisen (1988) entwickelte den $\lambda$-v-CS-Kalkül, der eine Erweiterung des $\lambda$-Kalküls um Referenzen und Zuweisungen darstellt. Ähnlich wie der $\lambda_{\mathrm{var}}$-Kalkül von Odersky u.a. (1993) handelt es sich bei diesem Kalkül nicht um die Erweiterung des $\lambda$-Kalküls durch einen expliziten änderbaren Speicher, sondern vielmehr um eine Erweiterung der Substitutionssemantik, indem bei jeder Zuweisung alle Terme, die durch das Überschreiben einer Variablen beeinflusst werden, simultan mit dem neuen Wert substituiert werden.

### 3.4.6 Effektsysteme

Ein anderer Ansatz, Seiteneffekte in kontrollierter Form zuzulassen, besteht darin, sie in die Typisierung von Programmen einfließen zu lassen und so die zusätzlichen Informatio-

**Abbildung 3.3** Kombinationsmöglichkeiten von $W, R$ und $A$




nen über das Verhalten des Programms zu nutzen. Effekte sind statische Beschreibungen des dynamischen Verhaltens von Ausdrücken. Typen beschreiben, *welche* Werte berechnet werden und Effekte beschreiben, *wie* sie berechnet werden. Im Gegensatz zu linearen Typen enthält die Typisierung dabei nicht nur Informatione zur Eindeutigkeit eines Datenobjektes, sondern es werden explizit die Seiteneffekte berücksichtigt, die durch die Auswertung eines bestimmten Ausdrucks verursacht werden.

Gifford und Lucassen (1986) klassifizieren Ausdrücke nach drei Kriterien:

1. Allocate ($A$): eine veränderbare Speicherzelle wird erzeugt
2. Read ($R$): eine veränderbare Speicherzelle wird ausgelesen
3. Write ($W$): eine veränderbare Speicherzelle wird modifiziert

Mit diesen drei Kriterien lässt sich ein Verband aufstellen, wie er in Abb. 3.3 dargestellt ist. Die entstehenden acht Möglichkeiten wurden von Gifford und Lucassen in vier sog. *Effektklassen* zusammengefasst:

- $\{W, WR, WA, WRA\}$: Procedure
- $\{R, RA\}$: Observer
- $\{A\}$: Function
- $\{\}$: Pure (reine Funktion)

Die einzelnen Effektklassen bilden eine Ordnung, wobei *Procedure* das größte Element und *Pure* das kleinste Element darstellt.

Ein Ausdruck der Klasse *Procedure* darf beliebige Seiteneffekte verursachen, ein *Observer* darf diese Seiteneffekte nur beobachten, eine *Function* darf veränderbare Speicherzellen erzeugen, aber weder lesen noch schreiben. Ein Ausdruck der Klasse *Pure* darf keinerlei Seiteneffekte auslösen, dafür ist er aber referenziell transparent und muss für gleiche Parameter nur ein einziges Mal ausgewertet werden, danach kann dieser Wert wiederverwendet werden (*Memoization*).

Jedem Ausdruck der Sprache wird nun eine dieser Effektklassen zugeordnet, und ähnlich wie ein Programm auf Typverträglichkeit geprüft wird, wird nun geprüft, dass Ausdrücke höherer Effektklassen nur Unterausdrücke enthalten, die zur gleichen oder einer untergeordneten Effektklasse gehören.

**Abbildung 3.4** Effekt-Sprache

| | | | |
|---|---|---|---|
| $E$ | ::= | $x$ | $Variable$ |
| | | $\lambda x{:}\,T.\,E$ | Abstraction |
| | | $E_1\,E_2$ | Application |
| | | $\Lambda d{:}\,K.E$ | Polymorphic abstraction |
| | | PROJ $E\,D$ | Polymorphic application |
| | | NEW $R\,T\,E$ | Allocation |
| | | $!\,E$ | Read |
| | | $E := E$ | Write |
| $T$ | ::= | $d$ | Descriptor |
| | | $T_1 \xrightarrow{\epsilon} T_2$ | Function type |
| | | $\forall d{:}\,K \xrightarrow{\epsilon} T$ | Type abstraction |
| | | Ref $R\,T$ | Type of locations |

### Polymorphe Effektsysteme

Eine Erweiterung des oben dargestellten Modells besteht darin, Terme nicht nur mit Effekten zu annotieren, sondern auch über Effekte zu abstrahieren. Dies geschieht mit den polymorphen Effektsystemen von Lucassen und Gifford (1988), die zur Parallelisierung von Programmen benutzt wurden. Sie verwenden dazu ein Typsystem mit drei „Arten“ (engl. *kinds*): Typen, Effekte und Regionen. Typen beschreiben die Werte, die eine Operation als Ergebnis haben kann, Effekte beschreiben die Seiteneffekte einer Operation und Regionen stehen für den Teil des Speichers, der von den entsprechenden Seiteneffekten betroffen ist. Effektmaskierungen dienen zum Verstecken (operational) unsichtbarer Seiteneffekte. Ein weiteres Konzept ist die Effekt-Polymorphie: Effekte sind vollwertige Beschreibungen, die auch als Parameter an Funktionen übergeben werden können.

Die Sprache, die Lucassen und Gifford als Grundlage für ihr Typ- und Effektsystem benutzen, ist in Abbildung 3.4 zusammengefasst. Im Folgenden soll nicht das gesamte Inferenzsystem dargestellt, sondern nur der Grundgedanke erläutert werden. Für die präzisen Regeln sei auf die Originalliteratur (Lucassen und Gifford, 1988) verwiesen.

Zunächst werden Axiome definiert, welche die Typen und Effekte von Variablen $x$ und Konstanten $v$ definieren.

$$\begin{array}{rcl} A, B & \vdash & x : A(x) \\ x & ! & \text{PURE} \\ v & ! & \text{PURE} \end{array}$$

$A$ weist jeder Wertvariable einen Typ zu, während $B$ jeder Artvariablen eine Art (Typ, Effekt, Region) zuweist. Weiterhin werden für alle Sprachkonstrukte Inferenzregeln definiert, die ähnlich zu gebräuchlichen Typinferenzregeln auch auf Effekte angewendet werden können.

Bei $\lambda$-Abstraktionen werden der Effekt des Rumpfes sowie Argument- und Resultattyp zu einem Funktionstyp zusammengefasst. Der Effekt des Rumpfes $\epsilon$ ist ein sogenannter *latenter*

Effekt, da er erst bei der Applikation der Funktion zum Tragen kommt (siehe unten).

$$\frac{A[x \leftarrow \tau], B \vdash e : \tau' \quad A[x \leftarrow \tau], B \vdash e!\epsilon}{A, B \vdash \lambda x{:}\,\tau.\, e : \tau \xrightarrow{\epsilon} \tau'} \quad \text{(Abstraction)}$$

Der Effekt einer Funktionsapplikation ergibt sich aus den Effekten des Funktions- sowie des Argumentausdrucks und aus dem latenten Effekt des Funktionsausdrucks. Der Effekt wird dabei von der Funktion MAXEFF berechnet, die die kleinste obere Schranke der Effekte berechnet. Bei der Regel für die Applikation wird deutlich, weshalb Typen und Effekte gemeinsam berechnet werden: Funktionstypen beinhalten den latenten Effekt der Funktion, der erst bei der Applikation in die Berechnung einfließt.

$$\frac{A, B \vdash e_1 : \tau_1 \xrightarrow{\epsilon} \tau_2 \quad A, B \vdash e_2 : \tau \wedge \tau \sqsubseteq \tau_1 \quad A, B \vdash e_1!\epsilon_1 \quad A, B \vdash e_2!\epsilon_2}{A, B \vdash e_1\, e_2 : \tau_2 \quad A, B \vdash e_1\, e_2!(\text{MAXEFF } \epsilon_1\, \epsilon_2\, \epsilon)} \quad \text{(Application)}$$

Die übrigen Regeln des Inferenzsystems ähneln den hier vorgestellten.

Eine Erweiterung des beschriebenen Effektsystems stellen *Effektmaskierungen* dar. Hierzu wird die Syntax der Ausdrücke in Abbildung 3.4 um PRIVATE-Ausdrücke erweitert:

$$E \quad ::= \text{PRIVATE } D\, E$$

Dieser Ausdruck deklariert eine private Region $D$, die nur innerhalb des Ausdrucks $E$ verändert werden darf. Da diese Änderungen außerhalb des Ausdrucks $E$ nicht sichtbar sind, müssen sie nicht in den Effekt des gesamten PRIVATE-Ausdrucks aufgenommen werden.

#### Algebraische Rekonstruktion von Typen and Effekten

Die beschriebenen Effektsysteme sind explizit getypt und erfordern daher sowohl Typ- als auch Effektannotationen. Dies ist ein Nachteil gegenüber funktionalen Sprachen, für die der Compiler Typen inferieren kann. Jouvelot und Gifford (1991) beschreiben einen Algorithmus zur Rekonstruktion der Typen und Effekte von Ausdrücken in einer polymorph getypten Sprache, der diesen Nachteil aufwiegen kann. Im Gegensatz zu dem Ansatz von Lucassen und Gifford sind bei diesem Ansatz keine expliziten Typ- oder Effektdeklarationen notwendig, sie werden automatisch vom System berechnet.

Ähnlich wie bei Lucassen und Gifford besitzen Prozeduren (bzw. Funktionen) latente Effekte, welche die Effekte des Rumpfes einer Prozedur von der Definition zu den Punkten des Aufrufs tragen.

### 3.4.7 Berechnungs- und Koordinationssprachen

Die in den vorangegangenen Abschnitten beschriebenen Integrationskonzepte basieren alle auf einer engen Kombination unterschiedlicher Paradigmen. Ein anderer Ansatz besteht in einer stärkeren Trennung der Untersprachen.

Gelernter und Carriero (1992) betonen die Wichtigkeit von Koordinationssprachen. Sie unterscheiden in einem Programmiermodell das *Berechnungsmodell* und das *Koordinationsmodell*. Das Berechnungsmodell erlaubt die sequenzielle und schrittweise Beschreibung einer einzelnen Berechnung. Das Koordinationsmodell dient der Verbindung mehrerer getrennter

solcher Berechnungen. Dabei verkörpern gängige Programmiersprachen das Berechnungsmodell, während das Koordinationsmodell, das Operationen zum Erzeugen von Berechnungen und zur Kommunikation zwischen diesen anbieten muss, durch Koordinationssprachen verkörpert wird.

Diese Trennung zwischen mehreren Sprachen in einem System findet sich in vielen Programmiersprachen und -systemen in mehr oder weniger ausgeprägter Weise wieder. Einige constraint-funktionale Sprachen (Goffin, Abschnitt 2.8.1; Falcon, Abschnitt 2.8.2) besitzen eine funktionale Sprache als Berechnungssprache und nutzen Constraints zur Koordination der in dieser Sprache formulierten Berechnungen.

Ähnlichkeiten bestehen auch zur aspekt-orientierten Programmierung, die zwischen Komponenten (programmiert in einer prozeduralen Sprache) und Aspekten (beschrieben in einer Aspektsprache) unterscheidet. Aspektsprachen dienen dabei unter anderem zur Koordination von Komponenten.

Aber auch die Integration von deklarativen und imperativen Sprachen spiegelt diese Sicht wieder. Die monadische Ein- und Ausgabe trennt den Programmablauf (zumindest konzeptionell) in zwei Phasen: erst werden in einer funktionalen Sprache Ein-/Ausgabekommandos erzeugt, dann werden diese Kommandos (seiteneffektbehaftet) ausgeführt. Die deklarative Sprache dient in diesem Fall als Koordinationssprache für monadische Operationen. Umgekehrt lässt sich die Integration deklarativer und imperativer Programmierung aber auch so betrachten, dass einzelne imperative Programmschritte jeweils seiteneffektfreie, deklarative Berechnungen koordinieren, um diese einzelnen Berechnungen dann durch seiteneffektbehaftete Operationen zu verknüpfen. Die Kombination dieser beiden Sichten auf die deklarativ-imperative Programmierung führt dann zu der Ansicht, dass sich beide beliebig „aufeinander stapeln“ lassen, und so die freie Kombination dieser Paradigmen ermöglicht wird.

Gelernter und Carriero (1992) untersuchen auch spezifische Koordinationssprachen. Als Beispiel betrachten sie Linda, eine Koordinationssprache, die zu (fast) jeder Programmiersprache hinzugefügt werden kann.

Grundlage ihrer Argumentation ist der Begriff des *Asynchronen Ensembles*, das eine Ansammlung asynchroner kommunizierender Aktivitäten ist. Eine Aktivität kann dabei ein Prozess, Thread, Agent, aber auch ein Mensch sein, und sogar (rekursiv) ein Ensemble. Asynchrone Ensembles existieren überall, besonders aber in heterogenen Umgebungen. Koordinierung ist orthogonal zur Berechnung, daher sollte dafür eine eigene Notation verwendet werden. Weiterhin sollte eine Koordinierungssprache so allgemein wie möglich sein, also nicht an eine bestimmte Sprache gebunden. Selbst fundamentale Berechnungen bedürfen der Koordination: Ein- und Ausgaben zur Umgebung (Benutzer, Dateisystem, Netzwerk) müssen koordiniert werden, ebenso wie die Parameterübergabe zwischen Funktionen und Threads. In aktuellen Programmiersprachen werden diese Koordinierungen meist ad hoc in die Sprache eingebaut. Letztendlich versprechen Koordinationssprachen eine Erleichterung bei der Portierung von Berechnungen auf neue Architekturen und Systeme (z.B. massiv parallele Prozessoren).

#### Paradigmen-Komposition

Ein ähnliches Vorgehen schlägt Zave (1989) vor: das Zusammenfügen verschiedener Paradigmen durch Komposition mit geeigneten Mitteln. Zave untersucht drei verschiedene Ansätze

der Paradigmen-Kombination, die sie beobachtet bzw. selbst entwickelt hat:

1. Zunächst werden semantische und implementierbare Grundlagen geschaffen, auf denen jedes Paradigma realisiert werden kann. Der Nachteil dieses Vorgehens ist, dass man wissen muss, wie die einzelnen Paradigmen auf der gemeinsamen Grundlage implementiert sind, um zu verstehen, wie sie miteinander interagieren.

2. Der Begriff „Multiparadigmensprache“ wird oft benutzt, um (anders als in diesem Bericht) die Integration ungewöhnlicher, aber kompatibler Sprachelemente zu beschreiben, z.B. die Kombination funktionaler Sprachen und Arrays. Nach Zaves Ansicht handelt es sich dabei nicht um die Integration unterschiedlicher Paradigmen, sondern um Spracherweiterungen. Unterschiedliche Paradigmen unterscheiden sich normalerweise um radikal verschiedene Kontrollstrukturen.

3. Zaves eigener Ansatz besteht in der Paradigmen-Komposition durch die Verbindung der Einzelparadigmen als nebenläufige, kooperierende Elemente. Dabei basieren die Interaktionen der Paradigmen auf den konzeptionellen Grundlagen der teilnehmenden Paradigmen.

Das einzige allgemein bekannte und verwendete Mittel zur Komposition in gängigen Programmiersprachen ist nach Zaves Meinung der Funktionsaufruf. Sie beschreibt in ihrem Ansatz aber noch zwei weitere: Ströme (*streams*) und Ereignisse (*events*). Ströme dienen der kontinuierlichen, verzögerten Übertragung von Daten zwischen den einzelnen Paradigmen, während Events Ereignisse einmalig signalisieren.

### 3.4.8 Diskussion

Die in diesem Abschnitt beschrieben Integrationkonzepte von Seiteneffekten in deklarative Sprachen besitzen viele Ähnlichkeiten bzw. bauen aufeinander auf.

Effektsysteme wurden für strikt auswertende Sprachen entwickelt, auf Sprachen mit verzögernder Auswertungsstrategie lassen sie sich nicht ohne weiteres übertragen. Allerdings sind Ähnlichkeiten zwischen den Maskierungskonstrukten der Effektsysteme, die lokal begrenzte Seiteneffekte verstecken sollen, und der Behandlung interner Seiteneffekte mittels monadischer Zustandstransformierer (Launchbury, 1993) zu erkennen. Beide dienen dazu, interne Seiteneffekte so zu kapseln, dass seiteneffektbehaftete Operationen in seiteneffektfreie Berechnungen eingebettet werden können.

Monadische Seiteneffekte kombinieren die Serialisierung von seiteneffektbehafteten Operationen, wie sie durch synchronisierte Ströme und Continuations geleistet wird, mit dem Prinzip des „Information Hiding“, indem die notwendigen Verknüpfungsoperationen in den Kombinatoren `return` und `>>=` versteckt werden. Weiterhin wird bei Monaden der Zustand der Welt als abstrakter Datentyp behandelt, auf den nicht zugegriffen werden kann. Dies sichert die single-threadedness des Zustands, wodurch die zustandsverändernden Operationen als destruktive Seiteneffekte effizient implementiert werden können.

Monaden können durch synchronisierte Ströme modelliert werden, genauso wie umgekehrt, wenn auch ineffizient. Continuations und Monaden können ebenfalls ineinander überführt werden. Die Modellierung von Monaden durch eindeutige Typen ist möglich, umgekehrt jedoch nicht, Seiteneffekte und monadische Operationen können zumindest in Sprachen mit strikter Auswertung miteinander kombiniert werden. Diese theoretischen Erkennt-

nisse (Wadler, 1997) legen nahe, dass alle beschriebenen Ansätze in ihrer Ausdrucksstärke ähnlich sind, und die praktischen Vorteile über ihren Einsatz entscheiden sollten. Zur Zeit erscheint der monadische Ansatz aufgrund der oben beschriebenen Kapselung der Serialisierung und des Zustands am vielversprechendsten.

Auch in neueren Arbeiten werden die Zusammenhänge verschiedener Integrationskonzepte näher untersucht. Benton u.a. (2002) (in Barthe u.a. (2002)) beschreiben ebenfalls sehr ausführlich den monadischen Ansatz, sowohl von theoretischer Seite als auch die Umsetzung in der Sprache Haskell. Weiterhin erläutern sie Effektsysteme sehr detailliert und stellen die Zusammenhänge dieser beiden Techniken dar.

## 3.5 Praktische Überlegungen

In den vorangehenden Abschnitten dieses Kapitels wurden vor allem theoretische Aspekte der Paradigmenintegration betrachtet. Im Folgenden sollen kurz einige praktische Probleme und Lösungsmöglichkeiten angesprochen werden, die eine solche Integration mit sich bringt.

### 3.5.1 Implementierung

Die Implementierung monadischer Seiteneffekte kann durch verschiedene Verfahren erfolgen (Peyton Jones, 2001). Entweder werden die einzelnen Aktionen (Kommandos), die die Interaktion mit der Außenwelt repräsentieren, als primitive Datenobjekte betrachtet, die der Compiler gesondert behandelt oder Monaden werden als Funktionen implementiert, die vom Compiler genauso übersetzt werden, wie alle anderen Funktionen. Bei der zweiten Methode ist es allerdings notwendig, dass der Compiler bei einer Sprache mit verzögernder Auswertung keine Redexe (reduzierbare Ausdrücke) dupliziert, da dies die Semantik (nämlich die einmalige Ausführung monadischer Seiteneffekte) zerstört.

### 3.5.2 Reflection

Unter *Reflection* (Kiczales u.a., 1991; Sun Microsystems, 2002) versteht man die Fähigkeit eines Systems, die eigenen Eigenschaften abzufragen und untersuchen zu können. So ist es beispielsweise für ein Programm in einem System, das Reflection unterstützt, möglich, herauszufinden, welchen Typ eine Funktion oder ein Datenobjekt besitzt. Reflection kann zur Kooperation verschiedener Paradigmen genutzt werden, z.B. wenn zur Durchführung eines Funktionsaufrufs oder zur Interpretation einer Datenstruktur in einem anderen Paradigma Informationen über diese notwendig sind. Beispiele sind Aufrufkonventionen, Datenrepräsentation oder andere Meta-Informationen, beispielsweise, ob eine Funktion deterministisch ist oder nicht.

### 3.5.3 Datenrepräsentation

Unterschiedliche Paradigmen und Programmiersprachen legen auch verschiedene Schwerpunkte auf die Datenstrukturen, mit denen vornehmlich gearbeitet wird. So arbeiten imperative Programme oft mit änderbaren Arrays, während deklarative Sprachen eher mit dynamischen Datenstrukturen, wie z.B. Listen umgehen. Dies führt dazu, dass diese Datenstrukturen unterschiedlich realisiert werden, um die für das jeweilige Paradigma effizientesten

Implementierungen zu ermöglichen. Bei einer Neuimplementierung einer Multiparadigmensprache kann man bereits im Entwurf auf die erweiterten Anforderungen eingehen, bei der Erweiterung einer existierenden Sprache sind unter Umständen Kompromisse notwendig.

Eine häufig verwendete Methode, um unterschiedliche Sprachen und Systeme miteinander zu verbinden, ist die Definition einer gemeinsamen Datenrepräsentation zum Datenaustausch. Alle Programme, die an diesem Austausch teilnehmen, müssen dann natürlich beim Verschicken ihrer Daten die eigene Repräsentation in die externe übersetzen und beim Empfang wiederum eine Übersetzung durchführen. Wesentlich effizienter ist natürlich, wenn von vorneherein alle beteiligten Teilsysteme mit derselben Datenrepräsentation arbeiten. Beispielsweise wurden für *Microsoft .NET* präzise die Datenformate definiert, die alle Sprachimplementierungen für diese Plattform einhalten müssen.

## 3.6 Andere Arbeiten

Andere Arbeiten auf dem Gebiet der Multiparadigmen-Programmierung befassen sich meistens mit kleinen Ausschnitten dieses Themenkomplexes, wie z.B. der funktional-logischen Programmierung oder der Integration von Seiteneffekten in eine deklarative Sprache. Diese Arbeiten wurden bereits in den entsprechenden Abschnitten dieses Berichts zitiert. Der vorliegende Abschnitt soll dagegen die Arbeiten zusammenfassen, die sich mit der Multiparadigmen-Programmierung „an sich“, sowie mit dem Einsatz von multiparadigmatischen Entwurfstechniken befassen.

Budd (1995) beschäftigt sich hauptsächlich mit der Multiparadigmen-Programmierung mit der von ihm entwickelten Programmiersprache Leda, enthält aber umfangreiche allgemeine Überlegungen zu diesem Thema, die auf den Entwurf von Leda Einfluss genommen haben. Aber auch Programmiertechniken, zum Beispiel der Einsatz funktionaler Programmieridiome in imperativen Sprachen oder die Verwendung von logischen Relationen als Iteratoren werden erläutert. Vranić (2000) schreibt zu multiparadigmatischer Software-Entwicklung, und geht dabei vor allem auf Entwicklungstechniken ein, die sich mehrerer Paradigmen bedienen, wie z.B. auf die aspekt-orientierte Programmierung. Westbrook (1999) und Placer (1993) befassen sich mit dem Einsatz von Multiparadigmensprachen in der Lehre. Sie betonen, dass solche Sprachen den Vorteil haben, keine umfangreiche Einarbeitung in neue Programmiersystem zu erfordern, um die Prinzipien verschiedener Paradigmen zu vermitteln.

## 3.7 Diskussion

Bezüglich der Syntax von Multiparadigmensprachen lässt sich im Rahmen dieses Berichts keine verbindliche Empfehlung geben, abgesehen von den genannten, eher allgemeinen, Anforderungen.

Zur Semantik integrierter Programmiersprachen ist zu sagen, dass die Kombination deklarativer Sprachen bereits sehr erfolgreich durchgeführt wurde und zukünftige Arbeit sich auf die Verbindung deklarativer und imperativer Sprachen konzentrieren sollte. Die existierenden Ansätze sind entweder informell oder beschränken sich auf theoretische Kalküle, die in der Praxis kaum umgesetzt wurden. Welches der vorgestellten Konzepte sich langfristig bei der Integration deklarativer und imperativer Programmiersprachen durchsetzen

wird, ist noch nicht abzusehen, allerdings ist aus heutiger Sicht der monadische Ansatz am vielversprechendsten, da er die Vorteile der anderen Ansätze (Serialisierung von seiteneffekbehafteten Operationen, einmalige Ausführung) mit dem Verstecken der internen Abläufe verbindet.

Beim Entwurf multiparadigmatischer Sprachen ist wichtig, dass es sich dabei nicht um ein Aufeinanderstapeln verschiedenster Sprachkonstrukte geht, sondern um die Integration mehrerer Paradigmen in einem möglichst einfachen Modell (Müller u.a., 1995). Ein solches einfaches Modell emöglicht dem Programmierer, die Semantik der Programmiersprache vollständig zu erfassen, um so (möglichst) korrekte und robuste Programme zu schreiben, ohne in einer Unmenge verschiedenster (Un-)Möglichkeiten der Sprache die Orientierung zu verlieren.

# 4 Zukünftige Arbeiten und Zusammenfassung

Nachdem in den Kapiteln 2 und 3 die existierende Literatur zu Multiparadigmensprachen sowie deren Implementierungen vorgestellt wurden, soll ein Ausblick auf zukünftige Arbeiten gegeben werden, die sich an den vorliegenden Bericht anschließen sollten.

## 4.1 Zukünftige Arbeiten

Die Forschung auf dem Gebiet der Multiparadigmen-Programmierung scheint sehr aufgesplittert zu sein. Zum Beispiel existiert umfangreiche Literatur zur Integration deklarativer (logischer und funktionaler) Programmiersprachen einerseits, und zur Integration von Seiteneffekten in funktionale Sprachen andererseits, aber wenig Literatur zum grundlegenden Problem der Kombination all dieser Berechnungskonzepte und -techniken. Aus diesem Grund halte ich weitere Forschung in diesem speziellen Bereich zunächst für die wichtigste. Dabei sollte der Schwerpunkt zunächst auf der Identifikation der wesentlichen und grundlegenden Unterschiede und Gemeinsamkeiten zustandsbehafteter und -freier Paradigmen liegen, darauf aufbauend kann dann die Integration in eine gemeinsame Semantik erfolgen.

Diese Entwicklung eines allgemeinen Modells zur Multiparadigmen-Programmierung erfordert die Definition eines Kalküls, mit dessen Hilfe formale Aussagen über die darauf aufbauenden Sprachen getroffen werden können. Diese Aussagen sind unbedingt erforderlich, um mit formalen Mitteln Programme analysieren und verfizieren zu können. Weiterhin sollte eine operationale Semantik dieser Sprache entwickelt werden, die als Grundlage einer Implementierung dienen kann.

Wichtig scheint mir, dass diese Forschungen zu einer formalen Definition einer repräsentativen Multiparadigmensprache führt, auf der weitere Untersuchungen und Erweiterungen aufbauen können, ohne jeweils neue Formalismen einführen zu müssen.

Für den praktischen Einsatz von Multiparadigmensprachen sollten geeignete Mechanismen zur syntaktischen Erweiterung von Programmiersprachen (z.B. für generative Programmierung oder Meta-Programmierung) entwickelt werden. Dieser Aspekt wurde beim Entwurf der meisten in diesem Bericht genannten Sprachen nicht berücksichtigt, scheint aber für den praktischen Einsatz sehr wichtig zu sein (siehe dazu auch Hill und Lloyd (1994); Lloyd (1995)).

## 4.2 Zusammenfassung

Dieser Bericht gibt einen breiten Überblick über verschiedene Programmierparadigmen, deren Unterschiede und Gemeinsamkeiten, sowie über unterschiedliche Ansätze zur Integration mehrerer Paradigmen in kombinierten Programmiersprachen.

Zunächst werden verschiedene Argumente zur Unterstützung der Forschung auf dem gebiet der Multiparadigmen-Programmierung vorgetragen. Im zweiten Kapitel wird erläutert, was unter der Bezeichnung Multiparadigmensprachen verstanden wird und wie sie zu klassifizieren sind. Weiterhin werden die wichtigsten Einzelparadigmen (funktional, logisch, imperativ, objekt-orientiert, verteilt, usw.) beschrieben und verbreitete Programmiersprachen diesen zugeordnet. Kombinationen zweier oder mehrerer Paradigmen in der Literatur werden genannt und beschrieben und schließlich werden solche Sprachen genannt, die als Multiparadigmensprachen entworfen wurden. Das dritte Kapitel untersucht – unabhängig von konkreten Sprachen – wichtige Kriterien zur Klassifizierung und zum Entwurf von Multiparadigmensprachen. In diesem Kapitel werden außerdem Ansatzpunkte für zukünftige Forschung zur Multiparadigmen-Programmierung genannt.

Die Multiparadigmen-Programmierung stellt eine attraktive Lösung für vielschichtige und komplexe Probleme dar, wie sie in großen Softwaresystemen vorkommen. Leider sind weder die Ansätze zur Werkzeugintegration (Multiparadigmen-Programmierumgebungen) noch die integrierten Programmiersprachen (Multiparadigmen-Sprachen) genug entwickelt, um sowohl effektive Softwareentwicklung als auch die notwendige Sicherheit durch Korrektheitsprüfungen zu erreichen. Diese Ziele erfordern eine multiparadigmatische Programmiersprache, die sowohl eine praktikable Syntax zur effektiven Programmierung als auch eine theoretisch fundierte Semantik aufweist, um eine formale Verifikation durchführen zu können. Dieser Bericht legt den Grundstein, um auf der Grundlage der existierenden Arbeiten eine solche Sprache zu entwickeln.

# Literaturverzeichnis


Aït-Kaci, H. und Podelski, A. (1993): Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3 & 4):195–234.

Albert, E., Hanus, M., Huch, F., Oliver, J. und Vidal, G. (2002): An operational semantics for declarative multi-paradigm languages. In *Proc. of the 11th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2002)*, 7–20, Grado (Italy). Research Report UDMI/18/2002/RR, Università degli Studi di Udine.

Appel, A. W. (1992): *Compiling with Continuations*. Cambridge University Press.

Apt, K. R., Brunekreef, J., Partington, V. und Schaerf, A. (1998): Alma-0: An imperative language that supports declarative programming. *ACM Toplas*, 20(5):1014–1066.

Apt, K. R. und Schaerf, A. (1997): Search and imperative programming. In *Conference Record of POPL 97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France*, 67–79.

Apt, K. R. und Schaerf, A. (1999): The Alma project, or how first order logic can help us in imperative programming. In *Correct System Design*, Nummer 1710 in Lecture Notes in Computer Science, 89–113. Springer-Verlag.

Ariola, Z. M. und Sabry, A. (1998): Correctness of monadic state: An imperative call-by-need calculus. In *Proceedings of 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 62–73.

Backhouse, R., Jansson, P., Jeuring, J. und Meertens, L. (1999): Generic programming — an introduction. In *Lecture Notes in Computer Science*, volume 1608, 28–115. Springer-Verlag. Revised version of lecture notes for AFP'98.

Backus, J. (1978): Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641.

Barthe, G., Dybvier, P., Pinto, L. und Saraiva, J. (Hrsg.) (2002): *Applied Semantics*. Nummer 2395 in Lecture Notes in Computer Science. Springer-Verlag.

Benton, N., Hughes, J. und Moggi, E. (2002): *Applied Semantics*, Kapitel Monads and Effects. Nummer 2395 in Lecture Notes in Computer Science. Springer-Verlag.

Berlin-Brandenburgische Akademie der Wissenschaften (2003): Das digitale Wörterbuch der deutschen Sprache des 20. Jahrhunderts. World Wide Web. `http://www.dwds.de`, zuletzt geprüft: 2003-07-28.

Boehm, H.-J. (1982): A logic for expressions with side-effects. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 268–280. ACM Press.

Breitinger, S., Loogen, R., Ortega-Mallén, Y. und Peña, R. (1996): The Eden coordination model for distributed memory systems. In Chakravarty u.a. (1996), 25–34.

Brus, T., van Eekelen, M. C. J. D., van Leer, M., Plasmeijer, M. J. und Barendregt, H. (1987): Clean - a language for functional graph rewriting. In Kahn (Hrsg.), *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '87), Portland, Oregon, USA*, Nummer 274 in Lecture Notes in Computer Science, 364–384. Springer-Verlag.

Budd, T. A. (1991): Blending imperative and relational programming. *IEEE Software*, 8 (1).

Budd, T. A. (1995): *Multiparadigm Programming in Leda*. Addison-Wesley.

Budd, T. A. (1998): Functional programming and the fragile base class problem. *ACM SIGPLAN Notices*, 33(12):66–71.

Budd, T. A. (2002): The return of Jensen's device. In Striegnitz u.a. (2002), 45–63.

Budd, T. A., Justice, T. P. und Pandey, R. K. (1995): *General-purpose multiparadigm programming languages: An enabling technology for constructing complex systems*. Technischer Bericht 95-60-04, Oregon State University, Department of Computer Science.

Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., Lopez, P. und Puebla, G. (2002): *The Ciao prolog system*. Technischer Bericht CLIP 3/97.1, The CLIP Group, School of Computer Science, Technical University of Madrid. `http://clip.dia.fi.upm.es/Software/Ciao/ciao.pdf`, zuletzt geprüft: 2003-02-24.

Burstall, R. M., MacQueen, D. B. und Sannella, D. T. (1980): Hope: An Experimental Applicative Language. In *Conference Record of the 1980 LISP Conference*, 136–143, Stanford University, Stanford, California, August 25–27. ACM Press.

Chakravarty, M. M. T., Guo, Y. und Ida, T. (Hrsg.) (1996): *Multi-paradigm Logic Programming*, Nummer 96–28 in Forschungsberichte des Fachbereichs Informatik. Technische Universität Berlin.

Chakravarty, M. M. T., Guo, Y., Köhler, M. und Lock, H. C. R. (1997): Goffin: Higher-order functions meet concurrent constraints. *Elsevier Science*, 29. Special issue of "Science of Computer Programming".

Colmerauer, A. (1990): An introduction to Prolog III. *Communications of the ACM*, 33(7): 69–90.

Czarnecki, K. und Eisenecker, U. W. (2000): *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley.

Dahl, O.-J. und Nygaard, K. (1966): SIMULA – an Algol-based simulation language. *Communications of the ACM*, 9(9):671–678.

Diaz, D. (2002): GNU Prolog: A native Prolog compiler with constraint solving over finite domains. World Wide Web. `http://pauillac.inria.fr/~diaz/gnu-prolog/manual/index.html`, zuletzt geprüft: 2003-02-24.

Felleisen, M. (1988): $\lambda$-v-CS: an extended $\lambda$-calculus for Scheme. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, 72–85. ACM Press.

Frühwirth, T., Herold, A., Küchenhoff, V., Le Provost, T., Lim, P., Monfroy, E. und Wallace, M. (1993): *Constraint logic programming: An informal introduction.* Technischer Bericht ECRC-93-5, ECRC.

Gelernter, D. und Carriero, N. (1992): Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107.

Gifford, D. K. und Lucassen, J. M. (1986): Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, 28–38.

Goldberg, A. und Robson, D. (1983): *Smalltalk-80: The Language and its Implementation.* Addison-Wesley.

Grabmüller, M. (2003): *Constraint Imperative Programming.* Diplomarbeit, Technische Universität Berlin.

Grabmüller, M. und Hofstedt, P. (2003): Turtle: A Constraint Imperative Programming Language. In *Twenty-third SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence*, Nummer XX in Research and Development in Intelligent Systems. British Computer Society, Springer-Verlag.

Guo, Y. und Pull, H. (1993): *FALCON: Functional And Logic language with CONstraints–language definition.* Technischer Bericht, Imperial College London.

Hailpern, B. (1987): Design of a multiparadigm language. In *Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences*, 239–246.

Hanus, M. (1990): Compiling logic programs with equality. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, Nummer 456 in Lecture Notes in Computer Science, 387–401. Springer-Verlag.

Hanus, M. (1997): A unified computation model for functional and logic programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, 80–93.

Hanus, M., Hofstedt, P., Abdennadher, S., Frühwirth, T. und Wolf, A. (Hrsg.) (2002): *MultiCPL'02: Workshop on Multiparadigm Constraint Programming Languages and RCoRP'02: Fourth Workshop on Rule-Based Constraint Reasoning and Programming.* Workshop Proceedings.

Hanus, M., Kuchen, H. und Moreno-Navarro, J. J. (1995): Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, 95–107.

Haridi, S., Van Roy, P. und Smolka, G. (1996): An overview of the design of Distributed Oz. In Chakravarty u.a. (1996), 13–24.

Hill, P. M. und Lloyd, J. W. (1994): *The Gödel Programming Language*. MIT Press.

Jaffar, J. und Maher, M. (1994): Constraint logic programming: A survey. *Journal of Logic Programming 19/20*, 503–581.

Jaffar, J. und Lassez, J.-L. (1987): Constraint logic programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, volume 14, 111–119. ACM Press.

Janson, S. (1994): *AKL—A Multiparadigm Programming Language*. PhD thesis, Uppsala University. Also available as SICS Dissertation Series 14.

Janson, S. und Haridi, S. (1991): Programming paradigms of the Andorra kernel language. In Saraswat, V. und Ueda, K. (Hrsg.), *Logic Programming, Proceedings of the 1991 International Symposium*, 167–186, San Diego, USA. MIT Press.

Jansson, P. (2000): *Functional Polytypic Programming*. PhD thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden.

Jouvelot, P. und Gifford, D. (1991): Algebraic reconstruction of types and effects. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 303–310. ACM Press.

Kelsey, R., Clinger, W., Rees, J. u.a. (1998): Revised$^5$ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(6):26–76.

Kiczales, G., Rivières, J. und Bobrow, D. (1991): *The Art of the Metaobject Protocol*. MIT Press.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M. und Irwin, J. (1997): Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*.

Knuth, D. E. (1983): *The WEB system of structured documentation*. Technischer Bericht CS980, Stanford University, Stanford, CA.

Kobayashi, N., Marin, M., Ida, T. und Che, Z. (2002): Open CFLP: An open system for collaborative constraint functional logic programming. In *Proc. of the 11th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2002)*, Research Report UDMI/18/2002/RR. University of Udine.

Krishnamurthi, S., Felleisen, M. und Friedman, D. P. (1998): Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming*.

Launchbury, J. (1993): Lazy imperative programming. In *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, DK, SIPL '92*, 46–56.

Leroy, X. und Weis, P. (1991): Polymorphic type inference and assignment. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 291–302. ACM Press.

Lloyd, J. W. (1995): *Declarative programming in Escher*. Technischer Bericht CSTR-95-013, Department of Computer Science, University of Bristol.

Lopez, G., Freeman-Benson, B. und Borning, A. (1994): Kaleidoscope: A constraint imperative programming language. In Mayoh, B., Tyugu, E. und Penjaam, J. (Hrsg.), *Constraint Programming: Proc. 1993 NATO ASI Parnu, Estonia*, 305–321. Springer-Verlag.

Lucassen, J. M. und Gifford, D. K. (1988): Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 47–57. ACM Press.

Miller, D. (1991): A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536.

Milner, R., Tofte, M., Harper, R. und MacQueen, D. (1997): *The Definition of Standard ML*. MIT Press. Überarbeitete Auflage.

Moreno-Navarro, J. J. und Rodriguez-Artalejo, M. (1992): Logic programming with functions and predicates: The language Babel. *Journal of Logic Programming*, 12:191–223.

Müller, M., Müller, T. und Van Roy, P. (1995): Multiparadigm programming in Oz. In Smith, D., Ridoux, O. und Van Roy, P. (Hrsg.), *Visions for the Future of Logic Programming: Laying the Foundations for a Modern Successor of Prolog, A Workshop in association with ILPS'95*.

Nadathur, G. und Miller, D. (1988): An overview of lambda prolog. In Bowen, K. A. und Kowalski, R. A. (Hrsg.), *Proceedings of Fifth International Logic Programming Conference*, 810–827, Seattle, Washington. MIT Press.

Odersky, M., Rabin, D. und Hudak, P. (1993): Call by name, assignment, and the lambda calculus. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 43–56. ACM Press.

Okasaki, C. (1999): *Purely Functional Data Structures*. Cambridge University Press.

Pepper, P. (2003): *Funktionale Programmierung in OPAL, ML, HASKELL und GOFER*. Springer-Verlag, zweite Auflage.

Peyton Jones, S. (Hrsg.) (2003): *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press. Auch erhältlich von: `http://www.haskell.org/definition/`, zuletzt geprüft: 2003-06-23.

Peyton Jones, S. (2001): *Engineering theories of software construction*, Kapitel Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. NATO ASI.

Placer, J. (1992): Integrating destructive assignment and lazy evaluation in the multiparadigm language G-2. *ACM SIGPLAN Notices*, 27(2):65–74.

Placer, J. (1993): The promise of multiparadigm languages as pedagogical tools. In *Proceedings of the 1993 ACM Conference on Computer Science*, 81–86. ACM Press.

Plasmeijer, R. und van Eekelen, M. (2001): *Clean language report version 2.0.* Technischer Bericht, University of Nijmegen.

Puget, J.-F. (1994): A C++ Implementation of CLP. In *Proceedings of the Second Singapore International Conference on Intelligent Systems*, Singapore.

Radensky, A. (1990): Toward integration of the imperative and logic programming paradigms: Horn-clause programming in the Pascal environment. *ACM SIGPLAN Notices*, 25(2):25–34.

Sabry, A. (1998): What is a purely functional language? *Journal of Functional Programming*, 8(1):1–22.

Sahlin, D. (1996): The concurrent functional programming language Erlang – an overview. In Chakravarty u.a. (1996), 9–12.

Saraswat, V. A. (1993): *Concurrent Constraint Programming.* MIT Press.

Semmelroth, M. und Sabry, A. (1999): Monadic encapsulation in ML. In *International Conference on Functional Programming*, 8–17.

Simonyi, C. (1995): *The death of computer languages, the birth of intentional programming.* Technischer Bericht MSR-TR-95-52, Microsoft Research.

Siskind, J. M. und McAllester, D. A. (1993): Nondeterministic lisp as a substrate for constraint logic programming. In Fikes, R. und Lehnert, W. (Hrsg.), *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Menlo Park, California. AAAI Press.

Smetsers, S., Barendsen, E., van Eekelen, M. und Plasmeijer, R. (1994): Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In Schneider und Ehrig (Hrsg.), *Proc. of Graph Transformations in Computer Science, International Workshop, Dagstuhl Castle, Germany*, Nummer 776 in Lecture Notes in Computer Science, 358–379. Springer-Verlag.

Smolka, G. (1995): The Oz programming model. In van Leeuwen, J. (Hrsg.), *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, 324–343. Springer-Verlag, Berlin.

Somogyi, Z., Henderson, F. und Conway, T. (1995): Mercury: an efficient purely declarative logic programming language. In *ASCS'95, Glenelg, Australia.*

Spinellis, D., Drossopoulou, S. und Eisenbach, S. (1995): Using objects for structuring multiparadigm programming environments. *Journal of Object-Oriented Programming*, 8(1): 33–38.

Steele, G. L. (1990): *Common Lisp: The Language*. Digital Press, Badford, Massachusetts, zweite Auflage.

Stefik, M. J., Bobrow, D. G. und Kahn, K. M. (1986): Integrating access-oriented programming into a multiparadigm environment. *IEEE Software*, 3(1):10–18.

Striegnitz, J., Davis, K. und Smaragdakis, Y. (Hrsg.) (2002): *Multiparadigm Programming with Object-Oriented Languages (MPOOL)*, NIC Series. John von Neumann Institute for Computing (NIC).

Sun Microsystems (2002): Reflection. World Wide Web. `http://java.sun.com/j2se/1.3/docs/guide/reflection`.

Sussman, G. J. und Steele, G. L. (1980): Constraints: A language for expressing almost-hierarchical descriptions. *Artificial Intelligence 14*, 1–39.

Swarup, V., Reddy, U. und Ireland, E. (1991): Assignments for applicative languages. In Hughes, R. J. M. (Hrsg.), *Conference on Functional Programming and Computer Architecture*, Nummer 523 in Lecture Notes in Computer Science. Springer-Verlag.

Vranić, V. (2000): *Towards multi-paradigm software development*. Technischer Bericht, University of Bratislava.

Wadler, P. (1997): How to declare an imperative. *ACM Computing Surveys*, 29(3):140–263.

Westbrook, D. S. (1999): A multiparadigm language approach to teaching principles of programming languages. In *29th ASEE/IEEE Frontiers in Education*.

Zave, P. (1989): A compositional approach to multiparadigm programming. *IEEE Software*, 6(5):15–25.

# Glossar

Dieser Glossar erläutert die wichtigsten in diesem Bericht verwendeten Begriffe. *Kursiv* hervorgehobene Begriffe sind Verweise auf andere Einträge des Glossars. Der Index ab Seite 73 verzeichnet die Verwendungen der Begriffe im Hauptteil.

**access-orientierte Programmierung** Instrumentalisieren der Lese- und Schreibzugriffe auf Daten, um dadurch weitere Programmaktionen auszulösen.

**algebraische Datentypen** Datentyp-Definitionen, die eine Algebra induzieren.

**allgemeinster Unifikator** *Unifikator*, aus dem alle Unifikatoren zweier Terme durch *Substitution* abgeleitet werden können.

**aspekt-orientierte Programmierung** Trennung verschiedener Sichtweisen auf ein Programmierproblem und separate Lösung der Teilprobleme durch unterschiedliche Programmiermittel.

**Backtracking** Programmiertechnik, um mehrere alternative Programmabläufe auszuführen. Ein Ablauf wird solange verfolgt, bis er fehlschlägt, dann wird mit der nächsten Alternative am letzten *Wahlpunkt* fortgefahren.

**call-by-need** Siehe *Verzögerte Auswertung*.

**call-by-value** Siehe *Strikte Auswertung*.

**call-by-name** Parameter einer Funktion werden nicht beim Aufruf ausgewertet, sondern erst, wenn der Wert des entsprechenden formalen Parameters benötigt wird.

**choice point** Siehe *Wahlpunkt*.

**Constraint-Programmierung** Programmierung durch die Spezifikation von Randbedingungen, die für die gesuchte(n) Lösung(en) gelten müssen.

**continuation-passing-style** Programmiertechnik, bei der jeder Funktion als zusätzliches Argument der Rest des Programmablaufs in Form einer Funktion übergeben wird. Statt einen Wert zurückzugeben, wird die *Continuation* mit dem Ergebnis als Argument aufgerufen.

**Continuation** Eine Continuation ist eine Funktion, welche den Rest des Programmablaufs darstellt. Continuations werden u.a. verwendet, um die Aufrufreihenfolge von Funktionen festzulegen.

**eager evaluation** Siehe *strikte Auswertung*.

**eindeutige Typen** (engl. *unique types*) Eindeutige Typen beschreiben Datenobjekte, die nicht dupliziert werden können, sondern nur genau einmal in einem Kontext verwendet werden können. Es ist z.B. nicht möglich, ein Objekt mit einem eindeutigen Typ mehrfach in einer Datenstruktur abzulegen oder gleichzeitig an mehrere Funktionen als Parameter zu übergeben.

**funktionale Programmierung** Programmierung durch die Definition und Komposition von Funktionen. Oft in Verbindung mit *verzögerter Auswertung* und *algebraischen Datentypen*.

**generische Programmierung** Bei der generischen Programmierung werden Berechnungen über Algebren abstrahiert, so dass Funktionen durch die Struktur ihrer Parameter definiert werden können. Auch *polytypische Programmierung* genannt.

**geteilter Speicher** Zustand, der von verschiedenen Programmstellen so verändert werden kann, dass die Änderungen sofort an allen anderen Programmstellen, die Zugriff auf den geteilten Speicher haben, sichtbar werden (engl. *shared data*).

**imperative Programmierung** Anweisungsbasierte Programmierung, bei der der interne Zustand des Programms durch aufeinanderfolgende Anweisungen solange modifiziert wird, bis der Zustand das gewünschte Ergebnis repräsentiert.

**Kleisli-Identität** Neutrales Element bzgl. der *Kleisli-Komposition*.

**Kleisli-Komposition** Komposition zweier Funktionen, die inkompatible Typen haben, durch einen Zwischenschritt.

**Komponente** Teil eines Software- oder Hardwaresystems, das über definierte Schnittstellen mit seiner Umgebung, z.B. anderen Komponenten kommuniziert.

**lazy evaluation** Siehe *verzögerte Auswertung*.

**lineare Typen** Siehe *eindeutige Typen*.

**logische Programmierung** Programmierung durch den Beweis von Formeln durch Fakten und die Ableitung von Regeln. Das Berechnungsverfahren heißt *Resolution*.

**logische Variable** Können sowohl ungebunden (ohne Wert) oder an einen Wert gebunden sein. In *logischen Programmiersprachen* werden durch *Unifikation* Werte für logische Variablen berechnet.

**Memoization** Speichern einmal berechneter Zwischenergebnisse, um eine eventuelle Neuberechnung zu vermeiden. Siehe auch *Verzögerte Auswertung*.

**Monade** Konzept aus der Kategorientheorie, das v.a. in der *funktionalen Programmierung* eingesetzt wird. Eine Monade besteht aus einem Datentyp und zwei Operationen, für die bestimmte Gesetze gelten: die *Kleisli-Komposition* und die *Kleisli-Identität*.

**Multiparadigmen-Programmierung** Anwendung mehrerer Programmierparadigmen zur Lösung eines Problems bzw. einer Kombination von Teilproblemen.

**Multiparadigmensprache** Sprache, die die Programmierung mit unterschiedlichen Paradigmen erlaubt.

**Multiparadigmen-Umgebung** Programmierumgebung, die die Programmierung mit Programmiersprachen unterschiedlicher Paradigmen in einem Programm erlaubt, gewöhnlich realisiert durch einen Mechanismus zur Verwendung von Funktionen, Relationen und Methoden aus anderen Paradigmen.

**Nachricht** (1) Ansammlung von Daten, die zwischen *Komponenten* eines Systems verschickt wird, (2) Aufforderung an ein *Objekt*, ein bestimmtes *Verhalten* zu zeigen.

**Narrowing** Berechnungsverfahren funktional-logischer Programmiersprachen. Dabei wird das *Pattern-Matching* funktionaler Sprachen durch *Unifikation* ersetzt.

**Objekt** Zusammenfassung von *Verhalten* und *Zustand*.

**objekt-orientierte Programmierung** Programme werden als Ansammlung von Objekten betrachtet, wobei jedes Objekt einen individuellen Zustand hat und mit anderen Objekten durch den Austausch von *Nachrichten* kommunizieren kann.

**Paradigma** Musterbeispiel. In der Programmierung eine Sichtweise auf ein Problem sowie eine passende Lösungsstrategie.

**Pattern-Matching** Prüfen der gemeinsamen Struktur zweier Terme mit Bindung von Variablen an Subterme. In der funktionalen Programmierung zur Definition von Funktionen benutzt.

**polytypische Programmierung** Siehe *generische Programmierung*.

**Programmierparadigma** Sichtweise auf ein Problem zur Lösung mittels einer Programmiersprache.

**Programmzustand** Siehe *Zustand*.

**prozedurale Programmierung** Erweiterung der *Imperativen Programmierung*, bei der Teile eines Programms in Unterprogrammen zusammengefasst werden, die an verschiedenen Stellen durch Unterprogrammaufrufe verwendet werden können.

**Reduktion** Berechnungsverfahren funktionaler Programmiersprachen. Terme werden durch die Substitution von Teil-Termen durch passende Regeln reduziert, bis keine weiteren Ersetzungsschritte mehr möglich sind.

**Residuation** Berechnungsverfahren funktional-logischer Programmiersprachen. Funktionsaufrufe werden solange verzögert, bis die Parameter ausreichend gebunden sind, um eine Regelauswahl zu treffen.

**Resolution** Berechnungsverfahren logischer Programmiersprachen. Terme werden ausgewertet, indem durch *Unifikation* passende Prädikate ausgewählt werden, durch deren Definition der Term nach Ersetzung der formalen durch die aktuellen Parameter ersetzt werden kann.

**Seiteneffekt** Ein Ausdruck hat Seiteneffekte, wenn er, neben der Erzeugung des Ergebniswertes, den Zustand in einer Art verändert, die die Werte anderer Ausdrücke im Kontext beeinflusst.

**single-threaded** Ein Datenobjekt ist single-threaded, wenn es nicht dupliziert wird. Siehe auch *eindeutige Typen*.

**strikte Auswertung** Teilterme werden sofort ausgewertet, wenn sie als Parameter an Funktionen übergeben oder an Namen gebunden werden. Auch *call-by-value* und *eager evaluation* genannt

**Substitution** Funktion, die in einem Term Variable durch Terme ersetzt.

**synchronisierte Ströme** Ein Programm modelliert Interaktionen dadurch, dass es eine Funktion ist, welche einen Strom (verzögerte Liste) als Eingabe und einen anderen als Ausgabe besitzt. Jedes Element der Eingabe resultiert in einem Element der Ausgabe, dadurch sind die Ströme synchronisiert.

**Trail** Datenstruktur, um *Wahlpunkte* zu speichern, solange sie aktiv sind. Dient der Implementierung logischer und constraint-logischer Sprachen.

**überschreiben** (1) Verändern einer *Variablen* durch das Ersetzen ihres Wertes, (2) Umdefinieren von geerbtem *Verhalten* in der *objekt-orientierten Programmierung*.

**Unifikation** Gleichmachen zweier Terme (z.B. arithmetische Ausdrücke oder Konstruktorterme) durch passende Instanziierung der Variablen in den Termen.

**Unifikator** Substitution, die zwei Terme gleichmacht.

**unique type** Siehe *eindeutige Typen*.

**Variable** (1) Änderbarer Speicherbereich in der imperativen Programmierung, (2) Name für einen Wert oder Term in der deklarativen Programmierung.

**Vererbung** Wiederverwendung von existierendem *Verhalten* und *Zustand* von bereits definierten *Klassen* oder *Objekten*.

**Verhalten** Programmcode, der die Reaktion eines Objektes auf eine *Nachricht* beschreibt.

**verteilte Programmierung** Programmierung mit einer *verteilten Programmiersprache*.

**verteilte Programmiersprache** Programmiersprache, welche die verteilte Ausführung eines Programms auf mehreren Rechnern bzw. Prozessoren erlaubt.

**Verteiltheit** Ein Programm ist verteilt, wenn verschiedene Teile des Programms konzeptionell auf verschiedenen Prozessoren bzw. Rechnern ablaufen.

**verzögerte Auswertung** Die Auswertung von Teiltermen eines Programms wird solange verzögert, bis das Ergebnis zur weiteren Ausführung benötigt wird. Oft kombiniert mit *Memoization*.

**Wahlpunkt** Punkt im Ablauf eines Programms, zu dem im Fall einer fehlschlagenden Berechnung zurückgekehrt wird, um einen anderen Programmpfad abzuarbeiten.

**Zustand** Menge aller Variablen eines Programms oder eines *Objekts*.

**zustandsbehaftet** Eine Programmiersprache ist zustandsbehaftet, wenn sich der Zustand eines Programms in dieser Sprache durch Veränderung bestehender Daten verändert.

**zustandsfrei** Eine Programmiersprache ist zustandsfrei, wenn Daten nicht verändert, sondern nur neu erzeugt werden können.

**Zustandsänderung** Veränderung des *Programmzustandes* durch Anweisungen.

**Zuweisung** Ändern einer *Variablen* durch *Überschreiben*.

# Index

Dieser Index verzeichnet alle wichtigen Begriffe, Programmiersprachen und Autoren, die im Haupttext dieses Berichts verwendet werden. Wichtige Begriffsverwendungen (z.B. Definitionen) sind durch **fett** gedruckte Seitenzahlen markiert.