Fachhochschule Brandenburg
Fachbereich Informatik und Medien

Bachelorarbeit

Funktionale Programmierung mit F# und Microsoft Accelerator am Beispiel Bildverarbeitung

Autor: Tobias Schacht
Abgabe: 27.10.2010
Erstbetreuer: Prof. Dr. sc. techn. Harald Loose
Zweitbetreuer: Dipl.-Inf. Ingo Boersch

zum
Erlangen des akademischen Grades

BACHELOR OF SCIENCE
(B.Sc.)

Zusammenfassung

Die Arbeit präsentiert Funktionale Programmierung im Allgemeinen durch Schilderung deren grundlegender Konzepte und im Speziellen durch Präsentation der Funktionsweise der Sprache F#. Zusammen mit Vorstellung der Bibliothek Accelerator, die ein gleichartiges Programmiermodell verfolgt, und der Umsetzung von Beispielen aus Bildverarbeitung auf niedrigerer Ebene bei Ausnutzung von Datenparallelität, wird die Anwendung dieser Techniken in einer Demonstrationssoftware vorgezeigt.

Abstract

This thesis presents Functional Programming generally by describing it's basic concepts and specifically by presenting the language F#. Together with introduction of the library Accelerator which pursues a similar programming model and by implementing examples of low-level image processing operations while utilizing data-parallelism, the application of these technologies is shown in a piece of demonstration software.

Inhaltsverzeichnis

1 Einleitung

1.1 Thema und Motivation

Die Funktionale Programmierung stellt eine interessante Ergänzung zu den heute am meisten verbreiteten imperativen und objektorientierten Programmiersprachen dar. Der Gewinn an Popularität zeigt sich an einigen in den letzten Jahren neu erschienenen sowie sich in aktiver Entwicklung befindenden funktionalen Programmiersprachen[1] mit zugehörigen, publizierten Büchern. F# ist eine seit 2002 bei Microsoft Research[2] entwickelte, vornehmlich funktionale Sprache für das .NET-Framework. Mit dem Erscheinen von Visual Studio 2010 ist die Sprache offizielles Microsoft Produkt.

Anhand des Gegenstandsbereichs der Bildverarbeitung (BV) und Implementation einfacher Basisalgorithmen soll die Funktionale Programmierung mit F# beschrieben werden. Die stark an mathematischen Modellen orientierte Bildverarbeitung stellt ein für funktionale Sprachen spannendes Anwendungsgebiet dar, denn die Funktionale Programmierung, als Programmierparadigma entstanden aus einem mathematischen Modell, eignet sich mit ihrer Arbeitsweise mit Funktionen im mathematischen Sinne gut für einen Versuch der Verknüpfung beider Themen.

Die Operationen der Bildverarbeitung stellen sich oft als sehr aufwendig, aber häufig inhärent stark parallel dar, indem bestimmte Operationen in gleicher Weise auf jedes Bildelement unabhängig angewendet werden. Die Möglichkeit der untereinander unabhängigen Anwendung von Operationen auf alle Elemente eines Datums wird als *Datenparallelität* [BFRR95] (S. 4) bezeichnet. Für dieses Spezialgebiet existieren eine Reihe von Frameworks[3], die das Ausführen von Algorithmen auf der ebenfalls stark parallel arbeitenden Hardware von Grafikkarten ermöglichen. Dieser Ansatz wird mit dem Begriff *GPGPU* (General-Purpose Graphics Processing Unit)[4] überschrieben. Ein solches Framework ist das als .NET-Bibliothek entwickelte Projekt *Accelerator*[5] [TPO06]. Accelerator stellt allgemeine datenparallele Operationen bereit, die auf einen speziellen Datentyp angewendet werden. Die Bibliothek ist so konzipiert, dass diese Operation später zur Laufzeit des Programms in entsprechenden *Shader-Code* zur Verarbeitung auf der Grafikkarte kompiliert werden. Dazu werden die Operationen auf Variablen dieses speziellen Datentyps mit der normalen Syntax der jeweilig verwendeten .NET-Programmiersprache in Programmcode aufgeschrieben. Zur Ausführung werden die Variablen dieses Typs einer vorher erstellten Abstraktionsinstanz der Hardware, einem sogenanntem *target*, übergeben. Die Bibliothek übernimmt die Kompilierung der Operationen sowie Übertragung der Daten und Ergebnisse zwischen Hauptprozessor und Grafikkarte.

[1] bspw. Clojure, Scala, Haskell
[2] `http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/default.aspx`
[3] bspw. *OpenVIDIA mit NVidia CUDA, AMD CTM, GPUCV, Sh, Brook*
[4] `http://gpgpu.org/`
[5] `http://research.microsoft.com/en-us/projects/accelerator/`

1.2 Zielstellung

Es soll die Funktionale Programmierung im Allgemeinen und konkret anhand der Sprache F# präsentiert werden.

Es ist nicht Ziel, das Paradigma im Vergleich zu anderen Programmierparadigmen zu bewerten. Bzgl. der möglichen Problemlösung und Umsetzbarkeit von Algorithmen besteht zwischen funktionalem und imperativem bzw. objektorientiertem Paradigma kein Unterschied (auch nicht bei der Realisierung des praktischen Teils dieser Arbeit). Unterschiedlich ist der Stil der Umsetzung. Eine Bewertung ist höchstens in Form der Erhebung bestimmter Metriken zweier Programme möglich, welche dasselbe Problem mit denselben Algorithmen mit zwei unterschiedlichen Paradigmen umsetzen oder indem statistische Merkmale vieler Programme beider Paradigmen verglichen werden. Dieser Versuch wird nicht unternommen.

In dieser Arbeit sollen Eigenschaften und charakteristische Merkmale der Funktionalen Programmierung erläutert und anhand einer praktischen Umsetzung demonstriert werden. Mit Hilfe der Accelerator-Bibliothek sollen grundlegende Operationen der Bildverarbeitung umgesetzt und durch die Bibliothek auf der GPU berechnet werden.

1.3 Aufbau der Arbeit

Im folgenden Kapitel 2 wird die Funktionale Programmierung allgemein beschrieben, um einen Gesamteindruck durch Schilderung charakteristischer Eigenschaften und Erläuterung gängiger Begriffe zu vermitteln. Die verwendete Notation soll die Konzepte theoretisch darstellen. Sie orientiert sich zwar an den Schreibweisen der Programmiersprachen F# und Haskell, soll aber nicht unbedingt tatsächlichen Programmcode darstellen.

Kapitel 3 behandelt die konkrete Beschreibung der Sprache F# und die Umsetzung der beschriebenen Konzepte aus Kapitel 2. F# ist eine pragmatische, vornehmlich funktionale Sprache des .NET-Frameworks mit auch imperativen und objektorientierten Ausdrucksmöglichkeiten. Im Vordergrund sollen aber die funktionalen Aspekte stehen.

Das Accelerator Projekt wird in Kapitel 4 als spezialisierte Bibliothek zur Ausführung datenparalleler Operationen auf Grafikkartenprozessoren (GPU) vorgestellt.

Anhand des konkreten Beispiels von BV-Operationen wird in Kapitel 5 die gemeinsame praktische Anwendung beider Technologien demonstriert.

2 Funktionale Programmierung

2.1 Einführung

Die Funktionale Programmierung (FP) baut auf dem mathematischen Begriff der Funktion auf. In der Mathematik wird eine Funktion als Beziehung zwischen zwei Mengen definiert. Jedem Wert aus der Menge des Definitionsbereichs wird genau ein Element aus der Menge des Wertebereichs zugeordnet. Somit besteht ein funktionales Programm aus einer Reihe von Funktionsdefinitionen. Das Programm selbst ist geschrieben als Funktion, welche die Eingaben des Programms als Argument erhält und das Ergebnis als Ausgabe liefert. Dabei werden intern eine Vielzahl definierter Funktionen aufgerufen. Die *main*-Funktion ist mittels anderer Funktionen definiert, für welche wiederum dasselbe gilt. Charakteristisch ist die zur mathematischen Schreibweise oft sehr ähnliche Syntax und der sehr prägnante Programmcode.

Die Theoretische Basis funktionaler Sprachen bildet das λ-Kalkül von Church. Basierend darauf wurde 1958 von John McCarthy die funktionale und heute zweitälteste Programmiersprache LISP („List Processing“) am Massachusetts Institute of Technology (MIT) entwickelt. FORTRAN, die älteste Programmiersprache, ist der erste Vertreter der imperativen Programmiersprachen. Beide unterscheidet die Herangehensweise wie die zugrunde liegende Maschine abstrahiert wird. Imperative Sprachen bilden ausgehend vom Maschinencode, immer höhere Abstraktionsebenen, während funktionale Sprachen als Vertreter der deklarativen Sprachen von einem mathematischen Modell ausgehen und abwärts in Richtung Maschinencode wachsen.

2.2 Funktionaler Entwicklungsstil

Kennzeichnend ist die Komposition kleiner, spezialisierter und möglichst generischer Funktionseinheiten zur Lösungen eines größeren Gesamtproblems. Die Funktionalen Programmierung bietet mit ihren Mitteln einen eleganten Gestaltungsweg der hierarchischen Problemlösung und Modularisierung [1] an. Der Fokus auf den Funktionen und deren vielfältige Verbindungsmöglichkeiten zu neuen, komplexeren Funktionen, erlaubt ein hohes Maß an Abstraktion und führt zu ausdrucksstarkem Programmcode, bis hin zu effektiven domänenspezifischen Sprachen [Jon08].

Dies geschieht angeleitet durch ein Typensystem mit starker Typisierung[2] und *Typinferenz*[3], welches dem Entwickler hilft, Funktionen anhand der Typen derer Argumente und derer Ergebnisse als Endpunkte korrekt miteinander zu verbinden. Dabei werden unnötige Fehler früh, schon zur Zeit des Kompilierens abgefangen.

Die Funktionen stehen und gelten voneinander isoliert. Sie erhalten alle notwendigen Informationen zur Berechnung immer explizit durch Übergabe von Argumenten, anstatt implizit durch Modifikation einer Umgebung. Dadurch werden Abhängigkeiten zwischen Komponenten eines Systems konkret sichtbar. Iteration wird idiomatisch mit Anwendung rekursiver Funktionen

[1] [Hug90] beschreibt Vorteile, welche die Funktionalen Programmierung zur Modularisierung bereithält.
[2] Jedoch ist nicht jede funktionale Sprache stark typisiert.
[3] siehe Unterabschnitt 2.3.3 Typensystem

realisiert, ohne Verwendung imperativer Konstrukte. Funktionen können an *Funktion höherer Ordnung* [4] wie Variablen übergeben werden, was eine effektive Kodierung häufig auftretender Muster wie Iteration in Funktionen erlaubt. (Typische Beispiele sind die Funktionen *map* & $fold$ und werden später noch erläutert.)

Im Gegensatz zum objektorientierten Ansatz, bei dem vorwiegend die Bildung von Hierarchien im Vordergrund steht, fokussieren funktionale Sprachen eher auf den Fluss von Daten und deren Transformation. Die Beschreibung und Definition von Daten und Datentypen übernimmt das Typensystem. Die Transformationen und Operationen auf diesen Typen werden mit Funktionen umgesetzt, deren Argumente und Ergebnisse entsprechende Datentypen tragen. Dies führt zu einer Trennung von Operationen auf Daten und deren Definition. In der objektorientierten Welt werden beide Konzepte mit Objekten verwirklicht. Zusätzlich gibt es Konzepte[5] zur Modellierung von (u.a.) Programmzustandsänderungen, welche wiederum bei objektorientierten Sprachen implizit durch Veränderung des Objektzustandes geschehen. Diese saubere Trennung mit isolierten, *referenziell transparenten*[6] Funktionen, ermöglicht ein hohes Maß an Wiederverwendbarkeit und Modularisierung. Diese Philosophie, der referenziell transparenten Funktionen, findet sich teilweise im *Domain-Driven Design* wieder [Eva03] (S. 175ff. „side-effect-free functions“). Dies bedeutet, dass Ideen der Funktionalen Programmierung nicht unbedingt eine funktionale Sprache voraussetzen. Es kann auch in imperativen Sprachen in diesem Stil programmiert werden. Der Unterschied besteht darin, wie einfach es eine Sprache macht, den funktionalen Konzepten zu folgen. Dies sieht in imperativen Sprachen sehr ungewöhnlich aus.

Aus der elementaren mathematischen Definition der Funktion ergeben sich Implikationen, welche charakteristisch für die Funktionale Programmierung gelten. Es gibt in rein funktionalen Sprachen kein Anweisungen oder veränderbare Variablen. Stattdessen gibt es nur Ausdrücke, die zu einem Wert eines bestimmten Typs evaluieren.

2.3 Grundlagen

In diesem Abschnitt sollen wichtige Grundlagen und Begriffe der Funktionalen Programmierung beschrieben und anhand einer *Pseudosyntax* veranschaulicht werden. Die gewählte Pseudosyntax orientiert sich an der Notation von Caml[7] und Haskell, stellt aber nicht unbedingt funktionierenden Programmcode dar. Die konkrete Umsetzung und Syntax von F# folgt in Kapitel 3. Ausführliche Erläuterungen zum Thema sind ansonsten u.a in [Hud00, Hut07, PH06] zu finden.

2.3.1 Zuweisung von Bezeichnern

Die Zuweisung (*binding*) von Funktions- oder Variablennamen entspricht einer *Unifikation*. Einmal gebundene Bezeichner sind im Laufe des Programms unveränderlich und liefern stets denselben Wert zurück. Insofern ist der Begriff *Variable* eigentlich falsch, er soll aber in dieser Arbeit weiterhin für die Bindung einzelner Werte an einen Bezeichner verwendet werden.

Ein Zuweisung erfolgt in der Regel mit dem Schlüsselwort *let* und einem zugehörigen Bezeich-

[4] siehe Unterabschnitt 2.3.5 Funktion höherer Ordnung
[5] siehe Unterabschnitt 2.3.12 Monaden
[6] siehe Unterabschnitt 2.3.2 Funktion
[7] F# ist eine Variante der Caml-Syntax und orientiert sich an der Caml-Implementierung „*Objective Caml*“

nernamen, bei Funktionen folgen zusätzlich die Variablennamen:

$$\begin{aligned} let \quad k \quad &= \quad 10 \\ let \quad f\ x \quad &= \quad x + k \end{aligned} \tag{2.3.1}$$

2.3.2 Funktion

Das Hauptausdrucksmittel ist die Definition von Funktionen. Sie sind *first-class* Objekte: D.h. sie können zur Laufzeit erzeugt, in Variablen und Datenstrukturen gespeichert und als Argumente übergeben sowie als Ergebnis zurückgegeben werden [Sco06] (S. 140ff.). Funktionen können ebenso wie Variablen als Werte angesehen werden.[8] Umgedreht stellen Variablen einen Grenzfall der Funktionen dar, indem eine Variable als konstante Funktion mit leerem Definitionsbereich angesehen werden kann.

Um der häufigen Verwendung von Funktionen in der Funktionalen Programmierung Rechnung zu tragen, wird in vielen funktionalen Sprachen[9] zur Applikation einer Funktion f auf ihr Argument[10] x, anstatt der geklammerten, mathematischen Schreibweise $f(x)$, ein links-assoziatives Leerzeichen verwendet:

$$f \quad x \tag{2.3.2}$$

Der einzige Effekt einer Funktion soll die Berechnung des Ergebnisses anhand der übergebenen Argumente sein. Ein möglicher Seiteneffekt kann als unterschlagenes Ergebnis gedeutet werden. Das Ergebnis einer Funktion soll nur von dessen Argumenten abhängen und bei gleichen Argumenten jederzeit dasselbe Ergebnis liefern. D.h. die Funktion darf weder Seiteneffekte produzieren, noch von denselben abhängig unterschiedliche Ergebnisse liefern.

Diese Unabhängigkeit vom Zeitpunkt der Evaluierung wird als *Referenzielle Transparenz* bezeichnet. Durch die Referenzielle Transparenz können Ausdrücke zu jeder Zeit evaluiert werden und resultieren immer im selben Ergebnis. Die Reihenfolge der Ausführung spielt keine Rolle. Wesentlicher Vorteil daran ist, dass durch diese Reproduzierbarkeit eine große Quelle von Fehlern eliminiert wird, denn dem Entwickler bleibt es erspart, komplizierten Kontrollfluss und Zustandsänderungen zu verfolgen. Der gerade vorliegende Programmcode steht und funktioniert für sich allein, isoliert vom Rest des Programms. Er ist somit besser zu verstehen und zu manipulieren, da der Rahmen seines Geltungsbereiches, um den sich der Entwickler Sorgen machen muss, größtmöglich reduziert ist.

Durch die sich ergebene Möglichkeit der Substitution können Programme transformiert werden, sodass komplizierte durch einfache Ausdrücke ersetzt werden. Somit wachsen die Funktionen eines funktionalem Programm aus elementaren Definitionen zu Aktionsbeschreibungen mit komplexem Umfang. Durch *Referenzielle Transparenz* werden Programme sogar beweisbar, weil so komplexe

[8] Dies kann man sich daran klar machen, dass in der Mathematik Funktionen auch als Wertetabellen aufgeschrieben werden. Eine Funktion kann nicht nur als Berechnungsvorschrift, sondern auch als Datenstruktur gesehen werden. Damit erscheint die Verwendung als *first-class*-Objekt plausibler, weil Tabelle und Funktion konzeptionell identisch sind (siehe auch [PH06] Kapitel 15).

[9] bspw. Haskell, F#

[10] zur Übergabe mehrerer Argumente siehe Unterabschnitt 2.3.6 Partielle Applikation von Funktionen und Currying

auf einfachere Ausdrücke zurückgeführt werden können.

Im Extremfall enthalten funktionale Programme gar keine Seiteneffekte. In der Realität entfernen sich viele funktionale Sprachen unterschiedlich stark von diesem Idealbild. Ein Schlüsselfaktor ist die Behandlung jener Seiteneffekte, denn ohne sie sind nützliche Programme schwer vorstellbar. Seiteneffekte werden entweder durch das Konstrukt der *Monade*[11] eingefangen oder sind schlicht erlaubt und es kommt auf die Disziplin des Entwicklers an, Überraschungen zu vermeiden. Wünschenswert und entscheidend ist die funktionale Schnittstelle. Lokale und isolierte Seiteneffekte innerhalb einer Funktion sind wenig problematisch.

2.3.3 Typensystem

Durch das Typensystem wird jeder Variable, Funktion und jedem Wert, jedem Ausdruck, ein bestimmter Typ zugewiesen. Die Bestimmung dieser Typen erfolgt vor der Evaluation von Ausdrücken durch einen Prozess der *Typinferenz* genannt wird. Aus den elementaren Grundtypen und Typen von Unterausdrücken kann der Compiler mit dem sogenannten *Hindley-Milner-Algorithmus* den Typ eines Gesamtausdrucks automatisch bestimmen, ohne dass explizite Kennzeichnungen im Programmcode nötig sind. Die statische Prüfung der Typisierung durch den Compiler ist eine Art Programmverifikation, dadurch ist das erfolgreich übersetzte Programm *typensicher*, d.h. frei von Typenfehlern.

Der Typ einer Funktion beschreibt die Art und Anzahl ihrer Argumente und ihres Ergebnisses. An der *Signatur* einer Funktion f lässt sich erkennen, welcher Definitionsbereich D in welchen Wertebereich W abgebildet wird. Diese Abbildung wird (wie in der Mathematik) mit einem Pfeil dargestellt:

$$f \;:\; D \to W \tag{2.3.3}$$

Definition- und Wertebereich müssen nicht aus elementaren Typen bestehen. Die möglichen Arten zusammengesetzter Typen sind *Tupeltypen* bzw. *Produkttypen* (kartesisches Produkt in der Mengenlehre) $(X_1, \ldots, X_n)$, *Funktionstypen* $(X \to Y)$, *Summentypen* (2.3.5) und *rekursive Typen*, wie z.B. Listen $[X]$ oder Sequenzen[12], als Kombination von Summen- und Produkttypen.

Die Signatur einer *Funktion höherer Ordnung*[13], welche ein Tupel mit einem elementaren Typ X und einem Listentyp $[Y]$ als Argument erhält sowie eine Funktion $(B \to C)$ als Ergebnis liefert, demonstriert die Schreibweise von Produkt- und Funktionstypen:

$$\begin{aligned} f &: (X, [Y]) \to (B \to C) \\ f &: (X, [Y]) \to B \to C \end{aligned} \tag{2.3.4}$$

Die Signatur zeigt die Assoziativität des Pfeils von Funktionstypen nach rechts, was die Weglassung der Assoziativitätsklammern erlaubt. Die Signatur kann unterschiedlich gelesen werden: entweder liefert f eine Funktion mit Typ $(B \to C)$ als Ergebnis oder f wird eine Funktion mit Typ $((X, [Y]) \to B)$ als Argument übergeben und liefert ein Ergebnis vom Typ C. Aufgrund von

[11] siehe Unterabschnitt 2.3.12 Monaden
[12] siehe Unterabschnitt 2.3.9 Listen
[13] siehe Unterabschnitt 2.3.5 Funktion höherer Ordnung

Currying[14] sind beide Leseweisen äquivalent.

Von *Polymorphie* wird gesprochen, wenn anstatt eines speziellen Typs eine *Typenvariable* verwendet wird. Diese Typenvariable wird in der Signatur von Ausdrücken benutzt, um identische Typen mit selber Typenvariable zu identifizieren, ohne aber den konkreten Typ angeben zu müssen. Ein einfaches Beispiel ist die Liste: Um nicht zwischen Listen von bspw. Buchstaben und Gleitkommazahlen unterscheiden zu müssen, wird einfach eine Typenvariable a verwendet. In der Signatur einer auf Listen arbeitenden Funktion kann so, anstatt explizit $[Char]$ und $[Float]$ in zwei ansonsten gleichen Funktionsdefinition, allgemein $[a]$ geschrieben werden. Eine *polymorphe* Funktion kann unabhängig vom konkreten Typen auf Listen aller Typen angewendet werden. Eine solche polymorphe Funktion ist z.B. *map* (2.3.8).

Mit Hilfe von *Summentypen* können eigene Datentypen definiert werden, welche aus mehreren Varianten bestehen. Dazu wird ein eigener *Typenkonstruktor* T als Bezeichner für den neuen Typ mit zugehörigen Typ-Varianten $T_1 \dots T_n$ definiert:

$$type\ T \quad = \quad T_1 | \dots | T_n \tag{2.3.5}$$

$T_1 \dots T_n$ sind konstante *Datenkonstruktoren* oder *Konstruktorfunktionen*, die mit übergebenen Typ-Variablen einen Typ als Ergebnis zurückgeben. Dies kann rekursiv geschehen, wie das nächste Beispiel zeigt.

Die Konstruktorfunktion erhält *Typenkonstruktoren* mit *Typvariablen* als Argumente. Sie ist aber nicht mit einer normalen Funktion mit *Werten* als Argumente zu verwechseln. Ein Beispiel einer Listenstruktur:

$$type\ liste\ a \quad = \quad Leer \mid Cons\ a * (liste\ a) \tag{2.3.6}$$

Leer und *Cons* sind die beiden Datenkonstruktoren, wobei *Leer* einen konstanten Datenkonstruktor repräsentiert und *Cons* eine Konstruktorfunktion, welche ein Produkttyp (Tupel) als Argument übergeben wird. Die zwei Bestandteile des übergebenen Tupels stellen die polymorphe Typenvariable a und der Typenkonstruktor *liste* mit Typenvariable a dar. Das Ergbnis der Konstruktorfunktion ist der Typ *liste*.

Der Typenkonstruktor $(liste\,a)$ kann mit Typenvariablen ebenfalls Argumente erhalten, um mit diesen später konkrete Typen instanziieren zu können: Die Typenkonstruktorfunktion erhält als Argument einen konkreten Typ und liefert einen konkreten Typ (mit gebundener Typenvariable) zurück (z.B. *liste int*). Der Typ *liste* im Beispiel wird als *Typ höherer Ordnung* bezeichnet. Weiterhin handelt es sich um eine *rekursive* Typendefinition.

2.3.4 Patternmatching

Patternmatching ist eine Methode um Daten in deren Bestandteile zu *dekonstruieren*. Häufig wird Patternmatching bei der Prüfung von Argumenten eingesetzt, um Variablen zu binden und, abhängig von einer erfolgreichen Musterübereinstimmung, Code zu evaluieren. Diese Methode kann auf jeden beliebigen Datentyp angewendet werden. Das *Matching* erfolgt in der Regel anhand von *Datenkonstruktoren*. Diese können sich ggf. hinter Syntaxerweiterungen („syntaktischer

[14] siehe Unterabschnitt 2.3.6 Partielle Applikation von Funktionen und Currying

Zucker“) verstecken. Bspw. werden Listen häufig mit eckigen Klammern in der Form $[0, 1, 2]$ geschrieben. Dies entspricht analog zu (2.3.6) $Cons\ 0\ (Cons\ 1\ (Cons\ 2\ Leer))$. Konkrete Werte können als eine Art Datenkonstruktoren angesehen werden, da sie trivialer Weise Elemente eines Datentyps konstruieren, weswegen Werte in Musterausdrücken erlaubt sind. Zusätzlich gibt es nachgestellte Prädikate (sogenannte *guards*), die weitere Einschränkungen auferlegen, welche erfolgreich gebundene Variablen im Musterausdruck erfüllen müssen. Eine genauere Beschreibung der spezifischen Syntax für F# folgt in Unterabschnitt 3.1.4.

2.3.5 Funktion höherer Ordnung

Funktionen höherer Ordnung sind Funktionen, die andere Funktionen als Argument akzeptieren oder eine Funktion als Ergebnis zurückgeben. Sie machen die besondere Eleganz der Funktionalen Programmierung aus. Das Abstrahieren von Code wird vereinfacht, indem Funktionalität als Argument übergeben und in andere Funktionen injiziert werden kann.

Ein typisches Beispiel ist die Funktion *map*, die das Muster der Anwendung einer Funktion f auf jedes Element einer Folge xs beschreibt:

$$map \quad f\ xs \tag{2.3.7}$$

Sie hat folgende Signatur:

$$map : (a \to b) \to [a] \to [b] \tag{2.3.8}$$

2.3.6 Partielle Applikation von Funktionen und Currying

Als Applikation von Funktionen wird die Anwendung der Funktion auf ihre Argumente bezeichnet. In der Funktionalen Programmierung können Funktionen auf weniger Argumente als übergeben angewendet werden. Das Ergebnis ist eine neue Funktion, welche nur noch auf die restlichen Argumente angewendet werden kann und die ersten mit konkreten Werten intern gebunden hat.

Dies wird durch das sogenannte *Currying* erreicht. Currying macht sich die Möglichkeit der Rückgabe von Funktionen als Ergebnis zunutze, indem schrittweise immer ein Argument angewendet und eine Funktion, welche die restlichen akzeptiert, zurück gegeben wird. Deutlich wird dies an der geklammerten Schreibweise einer Funktion mit drei Argumenten und schrittweiser Applikation:

$$\begin{array}{llll} ((f & x_1) & x_2) & x_3 \\ (f' & x_2) & x_3 & \\ f'' & x_3 & & \end{array} \tag{2.3.9}$$

Deshalb wird die Übergabe mehrerer Argumente nicht mit Tupeln in der Form $f(x_1, \ldots, x_n)$ notiert, sondern mit einfacher Trennung durch Leerzeichen. Die Klammern in (2.3.9) können weggelassen werden, da das Leerzeichen links-assoziativ ist. Die Signatur von f wird ebenfalls anstatt in Tupel-Schreibweise in *Curry-Notation* mit Pfeilen geschrieben (Assoziativitätsklammern können aufgrund der Rechts-Assoziativität des Pfeils weggelassen werden.):

$$\begin{aligned} f &: X \to (X \to (X \to X)) \\ f &: X \to X \to X \to X \end{aligned} \tag{2.3.10}$$

2.3.7 Komposition von Funktionen

Komposition von Funktionen entspricht in der Funktionalen Programmierung der mathematischen Definition und ist ein weiteres Mittel zur Verbindung mehrerer Funktionen und Modularisierung des Programmcodes. Durch Komposition können bspw. drei Funktionen f, g und h zu einer neuen Funktion k verbunden werden. Die Schreibweise zur Bindung eines Funktionsnamens mit dem Schlüsselwort *let*, geklammert und mit Kompositionsoperator, gestaltet sich so:

$$\begin{aligned} let \quad k\ x &= f \quad (g \quad (h \quad x)) \\ let \quad k\ x &= (f \circ g \circ h) \quad x \end{aligned} \tag{2.3.11}$$

Je nach funktionaler Sprache existieren dafür unterschiedliche Operatoren. Folgende Schreibweise unter Weglassung der Variable x wird als *pointfree* bezeichnet und gilt als besonders elegant: (vorausgesetzt f, g und h akzeptieren jeweils nur ein Argument)

$$let \quad k \quad = \quad f \circ g \circ h \tag{2.3.12}$$

2.3.8 λ-Ausdrücke

λ-Ausdrücke sind eine besondere Form der Notation zur Funktionsdefinition mit Ursprung in Church's λ-Kalkül. Mit ihr können Funktionen definiert und verwendet werden, ohne einen expliziten Namen vergeben zu müssen. Diese Notation wird oft verwendetet, wenn Funktionen als Argument übergeben werden. Diese *anonymen Funktionen* werden mit Hilfe von λ bzw. des jeweiligen Operators oder Schlüsselwortes der gerade verwendeten Sprache wie folgt verwendet:

$$\lambda x_1 \ldots x_n \to e \tag{2.3.13}$$

Die Variablen $x_1 \ldots x_n$ stehen direkt hinter λ, gefolgt vom Funktionsrumpf e, in welchem sie gebunden werden. Entsprechend sind diese beiden Schreibweisen äquivalent:

$$\begin{aligned} let \quad f &= \lambda x_1 \ldots x_n \to e \\ let \quad f\ x_1 \ldots x_n &= e \end{aligned} \tag{2.3.14}$$

2.3.9 Listen

Listen (oder allgemein Folgen) spielen in der FP ein besondere Rolle, da es diese Datenstrukturen sind, auf welche funktionale Sprachen besonders zugeschnitten sind. Daher würdigt die Sprache LISP diesen Umstand in ihrem Namen.

Diese Datenstrukturen sind genau wie der in (2.3.6) entworfene Typ definiert. Daraus folgt, dass aus Gründen der Effizienz häufig mit dem Kopf der Liste gearbeitet wird und die Liste konzeptionell als Zusammenschluss von Kopf (*head*) und Rest der Liste (*tail*) gesehen wird, bis der Rest der Liste nur noch aus der leeren Liste besteht.

Diese rekursive Definition macht die Liste zu einer *funktionalen* Datenstruktur, indem das Hinzufügen von Elementen die Ursprungsliste unberührt lässt und keinen Seiteneffekt benötigt. Viele oft verwendete Funktionen, wie bspw. *map* zur Anwendung einer Funktion auf jedes Element der Liste, *fold/reduce* zum Reduzieren der Liste durch fortlaufende Anwendung einer binären Funktion, *filter*, *concat* zum Verknüpfen von Listen, der *cons*-Operator zum Hinzufügen eines Elements am Kopf oder *zip* zum Vereinigen zweier Listen mit einer binären Funktion vergleichbar mit einem Reißverschluss, basieren auf dieser rekursiven Eigenschaft.

Diese elementaren Funktionen bilden in der FP eine wichtige Grundlage, vergleichbar mit Schleifen in der imperativen Programmierung. Häufig werden andere Datenstrukturen an der Oberfläche mit einer entsprechenden Syntax und Semantik ausgedrückt, auch wenn intern nicht unbedingt mit einer Liste gearbeitet wird. Bspw. existiert die Funktion *fold/reduce* auch für assoziative Arrays, Mengen oder noch exotischere Datenstrukturen wie bspw. Finger-Trees[15].

2.3.10 Auswertungsstrategien *eager* & *lazy*

Programmiersprachen unterscheiden sich darin, welche der zwei wesentlichen Arten von Auswertungsstrategien *eager* und *lazy* angewendet wird. Als *eager* wird die *strikte* Auswertungsstrategie bezeichnet (auch *innermost reduction*). Hier werden die Argumente einer Funktion zuerst ausgewertet. Diese Strategie ist typisch für imperativen Sprachen. Die nicht-strikte Auswertung (auch *outermost reduction*) hingegen ist häufig bei funktionalen Sprachen anzutreffen. Sie wertet Argumente nur bei Bedarf aus, wenn diese tatsächlich für eine Berechnung benötigt werden. Mit dieser Strategie lassen sich *unendliche* Datenstrukturen definieren, welche stets ein Folgeelement liefern, wenn dies angefordert wird. Der Rest einer Liste ist als unendlich definiert und wird nicht ausgewertet. Die Auswertung passiert erst bei Zugriff auf das Element am Kopf der Liste. Diese Idee unendlicher Objekte mit nicht-strikter Auswertungsstrategie wird in den Büchern [PH06] (Kapitel 2), [Hut07] (Kapitel 12) und [Hud00] (Kapitel 14) ausführlicher diskutiert.

2.3.11 Comprehension-Notation

Die *comprehension*-Notation wird zu Generierung von Listen und anderen Folgen benutzt. Sie ähnelt der aus der Mathematik bekannten *beschreibenden Darstellung* von Mengen in der Form, bspw.: $\{x^y \mid x \in \{1,2,3,4\}, x \text{ ist gerade}, y \in \{2,3\}\}$. Eine entsprechende *list comprehension* erzeugt die Liste $[4,8,16,64]$ und wird wie folgt geschrieben:

$$[x^y \mid x \leftarrow [1,2,3,4], even\ x, y \leftarrow [2,3]] \tag{2.3.15}$$

even ist eine Bool'sche Funktion als Prädikat, auch *guard* genannt. Der Ausdruck $x \leftarrow [1,2,3,4]$ wird als *Generator* bezeichnet. Eine comprehension kann mehrere getrennte Generatoren enthalten. Die Generatoren werden von links nach rechts tiefer geschachtelt, d.h. Generatoren rechts erzeugen häufiger Elemente. Hier werden für jedes x jeweils alle y erzeugt, darunter umgekehrt:

[15] Diese interessante *funktionale* Datenstruktur erlaubt u.a. einen effizienten Zugriff auf beide Enden. [HP06]

$$
\begin{aligned}
& [(x, y) \mid x \leftarrow [1, 3], y \leftarrow [2, 4]] \\
= \; & [(1, 2), (1, 4), (3, 2), (3, 4)] \\
& [(x, y) \mid y \leftarrow [1, 3], x \leftarrow [2, 4]] \\
= \; & [(2, 1), (4, 1), (2, 3), (4, 3)]
\end{aligned}
\tag{2.3.16}
$$

Dadurch sind gebundene Variablen von weiter links stehenden Generatoren in tiefer geschachtelten Generatoren weiter rechts verfügbar:

$$
\begin{aligned}
& [y \mid x \leftarrow [[1, 2], [3, 4]], y \leftarrow x] \\
= \; & [1, 2, 3, 4]
\end{aligned}
\tag{2.3.17}
$$

In x werden einfache Liste als Elemente einer Liste-von-Listen gebunden. Im hinteren Generator wird y mit Elementen aus x (einfache Zahlen) gebunden. y wird jeweils im vorderen Teil der Comprehension als Element der neuen Liste zurückgegeben.

2.3.12 Monaden

Monaden sind ein Konzept, welches ursprünglich aus der Kategorientheorie stammt. Es wurde entdeckt, dass bestimmte Aspekte vieler Sprachkonstrukte und Vorgehensweisen Monaden sind [Wad95]. Dazu gehören z.B. E/A-Operationen (bestimmte Abfolge der Operationen, da in der FP diese durch *Referenzielle Transparenz* nicht gefordert ist, siehe auch *Referenzielle Transparenz* im Abschnitt 2.3.2), Exception-Handling oder Operationen mit nicht determiniertem Ergebnis (bspw. die Abfrage eines assoziativen Arrays oder eine unbestimmte Anzahl von Ergebnissen in Form einer Liste).

Monaden können pragmatisch als ein *Interface* angesehen werden, welches diesen allgemeinen Aspekt abstrahiert. Das Ziel ist die Möglichkeit der Verkettung (analog zur Applikation einer Funktion) monadischer Operation (Funktionen die ausgehend von einem Wert vom Typ a ein Ergebnis vom Typ b mit Kontext M liefern: $a \to M\ b$, wobei a und b Typenvariablen sind.). Eine monadische Operation unterscheidet sich von einer reinen Funktion, indem sie ein Ergebnis zusätzlich mit einem bestimmten Kontext liefert (wie Nicht-Determinismus, Seiteneffekte, Möglichkeit des Auftretens einer Exception, etc.).

Als Beispiel für die Verkettung reiner Funktionen soll der Infix-Operator *ap* als umgedrehter *Applikationsoperator* (2.3.2) dienen:

$$
\begin{aligned}
& ap \;\; : \;\; a \to (a \to b) \to b \\
let \;\; & ap \;\; x \; f \;\; = \;\; f \;\; x
\end{aligned}
\tag{2.3.18}
$$

Mit *ap* ist es möglich, die Funktionen f, g und h folgendermaßen mit einem Argument x zu

verketten (darunter analog die geklammerte Schreibweise):

$$\begin{array}{llllllll} x & ap & f & ap & g & ap & h \\ h & (g & (f & x)) \end{array} \tag{2.3.19}$$

Der Operator *ap* erinnert nicht aus Zufall an den *UNIX Pipe-Operator*, nur dass „|“ zusätzlich zu den „reinen“ Werten vom *Basistyp* a im FIFO-Puffer Seiteneffekte im Hintergrund zulässt. Diese Seiteneffekte im Hintergrund entsprechen einem monadischen Kontext M. Der Typ einer Monade ist ein Typ höherer Ordnung und wird notiert als:

$$M\ a \tag{2.3.20}$$

Zu diesem Typ gehören die elementaren Standardfunktionen *bind* und *return* (*return* wird in der Literatur auch mit dem Namen *unit* oder $lift$ bezeichnet.) oder alternativ *map* und *join* (auch $flatten$). Die Paare lassen sich jeweils durch das andere Paar ausdrücken [PH06] (*11.1.3 Monaden* S. 220).

Die Funktion *return* dient zum Einbetten eines Basistyps in einen monadischen Kontext:

$$return \ : \ a \to M\ a \tag{2.3.21}$$

Am Beispiel einer Liste bedeutet dies die Bildung einer einelementigen Liste $[a]$ mit dem Wert a.

Mit dem Operator *bind* ist es möglich (analog zu (2.3.19)) monadische Operationen zu verketten (hier am Beispiel der Funktion f', g' und h' vom Typ $(a \to M\ a)$):

$$\begin{array}{rlll} & bind & : & M\ a \to (a \to M\ b) \to M\ b \\ (return & x) \ bind & & f' \ bind \ g' \ bind \ h' \end{array} \tag{2.3.22}$$

bind implementiert die Behandlung des aus der vorherigen monadischen Operation entstandenen Kontextes M und leitet den Wert des Ergebnisses (vom Basistyp a) an die nächste monadische Operation weiter (Funktionen-Argument auf der rechten Seite des Operators).

Für die Funktion *map* und *join* soll an dieser Stelle nur die Signatur angegeben werden:

$$\begin{array}{lll} map & : & (a \to b) \to M\ a \to M\ b \\ join & : & M\ (M\ a) \to M\ a \end{array} \tag{2.3.23}$$

Die Funktion *join* für Listen wurde vom Prinzip her übrigens schon in (2.3.17) gezeigt. Aus einer Liste von Listen wird eine einfache Liste erzeugt.

Entsprechend zur Infix-Notation mit dem *bind*-Operator, existiert in funktionalen Sprachen mit Monaden oft eine Blockschreibweise [16], in der die monadischen Operationen *untereinander*

[16] Haskell: *„do-Notation“*, F#: *„Computation Expressions“*

in einem Codeblock geschrieben werden. Diese Notation ist übersichtlicher und entspricht der Schreibweise eines imperativen Programms. Eine Beschreibung der spezifischen Syntax für F# und einem Beispiel folgt in Abschnitt 3.2 Computation Expressions.

Ein weitere Notation ist die *comprehensions*-Schreibweise, die vor allem als Schreibweise zur Generierung von Listen und Sequenzen benutzt wird[17]. Diese Analogie von Monaden und comprehensions-Schreibweise geht aus der Möglichkeit hervor, die Elementarfunktionen *join*, *map* und *return* in comprehensions-Schreibweise ausdrücken zu können [Wad92].

Monaden abstrahieren somit einen bestimmten gleichartigen Kontext von Berechnungen mit einem Basistyp. Mit Implementation von *return* und *bind* durch eine Monaden-Instanz, können monadischen Berechnungen sauber wie Funktionen durch Applikation miteinander verkettet werden. Die Details werden unter Anwendung des *bind*-Operators oder der jeweiligen Syntax im Hintergrund von der Instanz der Monade erledigt. Die Art des Kontextes wird explizit am Monaden-Typ deutlich gemacht. Monaden stellen somit ein weiteres Werkzeug zur Strukturierung und Komplexitätsreduzierung eines Programms dar, ohne von der mathematischen Definition von reinen Funktionen zur Erzeugung von Seiteneffekten oder Kontext abweichen zu müssen.

Neben den genannten Büchern finden sich zu diesem Thema in [Yor09] vertiefende Erläuterungen.

[17] siehe Unterabschnitt 2.3.11 Comprehension-Notation

3 Die Sprache F#

F# ist eine Programmiersprache mit Unterstützung mehrerer Paradigmen. Der Fokus liegt vornehmlich auf der Funktionalen Programmierung. Die Sprache unterstützt als .NET-Sprache aber auch die objektorientierte Programmierung mit den üblichen Konstrukten. Die Einbindung von Modulen anderer .NET-Sprachen ist möglich. Umgekehrt ist durch Nutzung der objektorientierten Konstrukte von F# eine Erzeugung von Modulen mit entsprechend objektorientierter Schnittstelle realisierbar, sodass diese aus anderen .NET-Sprachen heraus genutzt werden können.

Zunächst sollen in Abschnitt 3.1 grundsätzliche Syntaxelemente der Sprache vorgestellt werden. In Abschnitt 3.2 folgt die Erläuterung des fortgeschrittenen Konzepts der *Computation Expressions*. Abschnitt 3.3 gibt einen Überblick, wie objektorientierte Programmierung in F# ermöglicht wird. Schließlich stellt Abschnitt 3.4 pragmatische Aspekte im Umgang mit der Sprache vor und Abschnitt 3.5 fast kurz nicht tiefer behandelte Konstrukte zusammen.

3.1 Grundsätzliche Syntaxelemente

3.1.1 Ausdrücke und Bindungen

Da es in funktionalen Sprachen keine Anweisungen wie in imperativen Sprachen gibt, besteht die Syntax exklusiv aus *Ausdrücken*, welche zu *einem* Wert eines bestimmten Typs evaluieren. Funktionen evaluieren zum zuletzt angegebenen Wert, der somit den Rückgabe-Typ bestimmt. Wird von einem Ausdruck ein Seiteneffekt ausgelöst (Dies ist bei F# möglich, entspricht aber nicht „reiner“ FP.), ist der Rückgabewert vom Typ `unit` mit einzig möglichem Wert `()` (leeres Tupel). Jedes Syntax-Konstrukt formt eine bestimmte Art Ausdruck, auch das Fallunterscheidungskonstrukt `if...then...else` ist insgesamt ein Ausdruck. Dies bedeutet, dass die Werte in beiden Zweigen den selben Typ haben müssen, besonders darf der `else`-Zweig nicht weggelassen werden. Wird abhängig in einem Zweig ein Seiteneffekt erzeugt, muss der andere, wie in Programmcode 3.1 gezeigt, ebenfalls zu *unit* evaluieren. Der `then`-Zweig soll hier eine Seiteneffekt auslösende Funktionen mit Rückgabewert `unit` sein.

```
1 if approved() then launchMissiles() else ()
```

Programmcode 3.1: *if-then-else*-Ausdruck

Insofern sind in F# Seiteneffekte pragmatischer Weise erlaubt, die eingangs genannten Bedingungen für Ausdrücke gelten aber trotzdem. Seiteneffekte können weiterhin mit Hilfe von *Computation Expressions*[1] explizit gemacht werden.

[1] siehe Abschnitt 3.2 Computation Expressions

Programmcode 3.2 zeigt die Zuweisung von Funktionen und Variablen mit dem Schlüsselwort `let` und `=`.

```
1 let k = 10 // Variable
2 let f x = x + k // Funktion
3 let f = (fun x -> x + k) // äquivalente Funktion als Lambda
```

Programmcode 3.2: Binden von Ausdrücken in F#

Hierbei handelt es sich um eine Unifikation von Name und Ausdruck, deswegen wird `=` auch zur Prüfung auf Gleichheit als Operator benutzt. Es können eigene Operatoren definiert werden [Micl]. Anstelle des Funktionsnamens werden bestimmte Zeichen in runde Klammern gesetzt. Die Zeichen bilden den gewünschten Operatornamen. Unäre Prefix- und binäre Infix-Operatoren können auf diese Weise definiert werden.

Obwohl F# automatisch mit unveränderlichen Variablen arbeitet, gibt es trotzdem zwei Konstrukte, um mit veränderlichen Variablen zu arbeiten. Zum einen können Variablen mit dem Schlüsselwort `mutable` explizit als veränderlich deklariert werden. Zum anderen gibt es sogenannte *Reference Cells*, die auf einen veränderlichen Speicherort zeigen. Sie werden erzeugt, indem das Schlüsselwort `ref` vor einen Wert geschrieben wird und können mit Schreiben des Operators `!`[2] dereferenziert werden. Ausführliche Beschreibungen zur Anwendung und Syntax der Reference Cells bietet [Mico].

Um Zuweisungen vorzunehmen, wird `<-` verwendet. Bei Reference Cells wird der Operator `:=` verwendet. Weiterhin ist zu beachten, dass Arrays in F#, im Gegensatz zu Listen und Sequenzen, grundsätzlich veränderlich sind. Programmcode 3.3 zeigt Beispiele zum Umgang mit veränderlichen Variablen in F#.

```
183 let a = ref 1
184 let mutable b = 2
185 let c = [|3|]
186
187 a := !a + 1
188 b <- b + 1
189 c.[0] <- c.[0] + 1
190
191 printfn "a:␣%d,␣b:␣%d,␣c:␣%d" !a b c.[0] // Ausgabe: a: 2, b: 3, c: 4
```

Programmcode 3.3: Konstrukte für veränderliche Werte in F#

Neben dem Schlüsselwort `let` können Objekte, welche `System.IDisposable` implementieren, mit dem Schlüsselwort `use` gebunden werden. Damit wird am Ende des Geltungsbereiches der Variable, wenn diese nicht *null* ist, automatisch die `Dispose()`-Methode zum Freigeben der durch dieses Objekt verwendeten Ressourcen aufgerufen.

3.1.2 Whitespace-Signifikanz

Sogenannter *Whitespace* (Leerzeichen, Tabulator-Zeichen, Einrückungen) hat in F# auf zwei Weisen signifikante Bedeutung. Wie bei funktionalen Sprachen üblich, werden Argumente an

[2] Zur Negation von Werten wird in F# die Funktion `not` verwendet.

Funktionen ungeklammert und getrennt durch ein Leerzeichen übergeben.

Die zweite signifikante Bedeutung ist die Einrückung von Ausdrücken zur Bildung von Blöcken. Für Einrückungen muss mindestens ein Leerzeichen verwendet werden, Tabulatorzeichen sind nicht erlaubt. Eine eingerückte Zeile signalisiert, äquivalent zu einer geöffneten geschweiften Klammer in *Java* oder *C*, den Beginn eines geschachtelten Blockes und die umgekehrte Einrückung das Ende. Deswegen muss der Programmcode richtig eingerückt sein, um kompiliert werden zu können. Dieses System wird mit den Begriffen *layout* oder *offside rule* bezeichnet.

3.1.3 Typensystem

Das Typensystem von F# besteht einerseits aus den primitiven Typen, wie bspw. Booleans, Interger und Floating Point Typen in unterschiedlichen Größen, die zu entsprechenden .NET-Typen korrespondieren. Andererseits stellt die Sprache Tupel, Listen, Arrays, Sequenzen, Records und sogenannte *Discriminated Unions* als Typen bereit. Hinzu kommt die Möglichkeit der Definition von Klassen in F#-Syntax. Tabelle 3.1 listet besondere Typen in F# mit deren Bezeichnungen in Signaturen sowie der Syntax ihrer Anwendung auf. *Konstruktion* meint die Konstruktion von Werten des Typs. Eine vollständige Übersicht bietet [Micj].

Typ	Signatur	Konstruktion	Beispiele
Typ-Variable	`'a`	—	`seq<'a>`, `'a -> 'b`
Unit	`unit`	`()`	`()` (Es gibt nur diesen einen Wert.)
Tupel	`'a * 'b {* ...}`	`(x,...)`	`(3.0, "abc")`, `(2L, [], 255uy)`
Array	`'a [] {[] ...}`	`[||]`, `[|x;...|]`	`[|1..7|]`, `[|3;5|]`, `[|[|7|]|]`
rechteckiges Array	`'a [,{,...}]`	Bibliotheksfunktionen	`Array3D.create 2 3 4 0L`
Liste	`'a list {list ...}`	`[]`, `[x;...]`, `x::[...]`	`[1..3]`, `[1;2;3]`, `1::2::3::[]`
Sequenz	`seq<'a>`	`seq {...}`, `seq [...]`	`seq {1..3}`, `seq [1;2;3]`
Discriminated Unions	`'a <TK>`	`<DK> x`	`Some "F#"`, `None`

TK = Typkonstruktorname, *DK* = Datenkonstruktorname

Tabelle 3.1: Übersicht F#-Typen

Das Typensystem beherrscht Typinferenz, sodass die explizite Angabe von Typen und Signaturen unterlassen werden kann. Der Compiler versucht dabei einen möglichst generischen Typ abzuleiten. Wenn der Typ aber spezifiziert werden soll oder für den Compiler nicht genügend Informationen verfügbar sind, um einen Typ ableiten zu können, wird das Symbol : zur expliziten Annotation von Typen verwendet. Programmcode 3.4 zeigt eine Funktion zur Wiederholung eines Strings mit explizit angegebenen Typen für Argumente und Rückgabewert der Funktion. (`fold` ist eine Funktion, die über eine Liste iteriert und alle Elemente mit einer übergebenen Funktion zu einem Wert akkumuliert.)

```
173 // Funktionsdefiniton
174 let repeat (s : string) (x : int) : string =
175     List.fold (fun (a : string) (b : string) -> a + b)
176         "" [ for i in 1 .. x -> s ]
177
178 // Anwendungsbeispiel
179 repeat "F#" 3 // Ergebnis: "F#F#F#"
```

Programmcode 3.4: Beispiel explite Typen-Annotation

Ein Beispiel eines Summentyps ist der in Programmcode 3.5 analog zu (2.3.6) definierte Typ. Die Definition erfolgt mit Hilfe des als *Discriminated Union* bezeichneten Konstrukts [Mici].

```
24 type 'a liste = Leer | Cons of 'a * 'a liste
25 let l = Cons (1, Cons (2, Cons (3, Leer))) // l : int liste = Cons (1,Cons (2,
       Cons (3,Leer)))
```

Programmcode 3.5: Beispiel eines Summentyps *liste*

Listen in F# (Typ `list`) sind entsprechend aufgebaut: `Leer` wird geschrieben als `[]` und `Cons (x, <Rest>)` als `x :: <Rest>`. Entsprechend wird die Liste 1 bis 3 so geschrieben: `1 :: 2 :: 3 :: []` oder mit "syntaktischem Zucker" `[1;2;3]`.

Ein anderes Beispiel ist der `'a option`-Typ. Dieser ist entweder konstant `None` (kein Wert) oder `Some x` (ein Wert `x` vom Typ `'a`). Dieser Typ wird z.B. für Operationen benutzt, die in imperativen Sprachen in bestimmten Situationen *null* zurückgeben würden (bspw. das Abfragen eines assoziativen Arrays mit nicht vorhandenem Schlüssel). Diese zusätzliche Eigenschaft des Ergebniswertes (ein Kontext) wird durch den Typ gekapselt.

3.1.4 Patternmatching

Das in Unterabschnitt 2.3.4 beschriebene Patternmatching erfolgt in F# mit der in Abbildung 3.1 gezeigten Syntax [Micm].

```
match <Ausdruck> with
| <Muster> [ when <Prädikat> ] -> <Ergebnisausdruck>
```

Abbildung 3.1: *Pattern Matching*-Syntax

Das einfachste Muster ist das einer Variable. Ausdruck und Variablenname werden stets unifiziert und die Variable steht im Ergebnisausdruck zur Verfügung. Wird die Variable nicht weiter verwendet, wird diese mit Hilfe von `_` als *Wildcard*-Muster verworfen. Damit wird stets eine positive Musterübereinstimmung erzeugt, ohne die Variable konkret zu binden. Das Wildcard-Muster wird gebraucht, um alle möglichen, nicht betrachteten Muster-Übereinstimmungen abzufangen. Wildcard und Variable können als Teil eines Musters herangezogen werden, wenn einem bestimmten Element des untersuchten Ausdrucks entsprochen werden soll. Den Einsatz der erwähnten Möglichkeiten zeigt Programmcode 3.6.

```
16 let f (x : char list) =
17     match x with
18     | ['F'; '#'] -> printfn "konkreter␣Wert"
19     | firstchar :: _ -> printfn "Kopf␣als␣%c␣gebunden" firstchar
20     | any -> printfn "nicht␣abgedecktes␣Muster:␣%s" (any.ToString ())
```

Programmcode 3.6: *Pattern Matching*-Beispiel 1

Programmcode 3.7 zeigt Patternmatching angewendet auf den in Programmcode 3.5 definierten Listentyp. Die definierte rekursive Funktion *map* verwendet die beiden Datenkonstruktoren `Leer` und `Cons of 'a * 'a liste` zum *dekonstruieren* der möglichen Werte dieses Typs.

```
29 let rec map (f : 'a -> 'b) (l : 'a liste) =
30     match l with
31     | Leer -> Leer // Ende der Liste erreicht
32     | Cons (x, rest) -> Cons (f x, map f rest) // f auf x anwenden, Liste
           wieder zusammensetzen und mit rest fortfahren
33
34 // l ist die Variable aus vorherigem Beispiel
35 let l' = map (fun x -> x*x) l // l' : int liste = Cons (1,Cons (4,Cons (9,Leer)
       ))
```

Programmcode 3.7: *Pattern Matching*-Beispiel 2: *map*-Funktion angewendet auf den Typ *liste*

Neben den genannten Möglichkeiten gibt es weitere Musterausdrücke, welche in [Micm] ausführlicher beschrieben sind.

3.1.5 Iteration

Die in Programmcode 3.7 gezeigte Funktion *map* ist ein typisches Beispiel der Umsetzung von Iteration in funktionalen Sprachen. Ein weiteres ist die in Programmcode 3.8 gezeigte Funktion *fold* (auch häufig als *reduce* bezeichnet). Sie iteriert ebenso wie *map* über alle Elemente eines Summentyps, *„faltet“* die Elemente jedoch *paarweise* mit einer binären Funktion, sodass ein einzelner Wert *akkumuliert* wird. Beide Funktionen nutzen Rekursivität und kodieren das Muster in einer Funktion höherer Ordnung. Rekursive Funktionen müssen mit dem Schlüsselwort `rec` definiert werden.

```
39 let rec fold (f : 'a -> 'a -> 'a) (acc : 'a) (l : 'a liste) =
40     match l with
41     | Leer -> acc // Akkumulator als Ergebnis
42     | Cons (x, rest) -> fold f (f acc x) rest // f auf Akkumulator und x
           anwenden, mit restlicher Liste fortfahren
43
44 let produkt = fold (fun x y -> x*y) 1 l' // l' aus vorherigem Beispiel
45 // Ergebnis: s : int = 36
```

Programmcode 3.8: *Iteration*-Beispiel: *fold*-Funktion angewendet auf den Typ *liste*

Als Anfangswert für den Akkumulator der Funktion `fold` wird ein bzw. *das* neutrale Element übergeben, denn diese Funktion existiert theoretisch für alle algebraische Strukturen, die einen *Monoid* bilden. Monoiden sind gerade durch ein neutrales Element und einer assoziativen binären

Funktion auf diese Elemente definiert. Für Listen ist dies die Konkatenation als binäre Funktion sowie die leere Liste als neutrales Element. Ein anderer Monoid ist die Addition über natürliche Zahlen mit neutralem Element 0. Je nach Monoid wird der *fold*-Funktion ein anderes Paar aus binärer Funktion und neutralem Element übergeben.

Die übergebene Liste kann wiederum theoretisch durch jede algebraische Struktur ersetzt werden, für die eine *map*-Funktion existiert, die es erlaubt jedes Element zu „besuchen“ und mit einer Funktion zu transformieren. Dazu gehören Listen, Sequenzen oder Mengen. Diese Arten von „Container“-Strukturen werden als *Funktor* bezeichnet. Ausführliche Erläuterungen zur Verwendung dieser und anderer Strukturen in der FP sind in [Yor09] am Beispiel der Sprache *Haskell* tiefer gehend erläutert.

Neben der Rekursion gibt es `for`- und `while`-Schleifen. Die Syntax der verschiedenen Schleifentypen ist in Abbildung 3.2 zusammengefasst.

```
for <Iterationsvariable> = <Start> [ to | downto ] <Ende> do
    <Schleifenkörper-Ausdruck>

for <Musterausdruck> in <enumerable-Ausdruck> do
    <Schleifenkörper-Ausdruck>

while <Prädikat> do
    <Schleifenkörper-Ausdruck>
```

Abbildung 3.2: Schleifen-Syntax

Als Alternative zu über Listen, Sequenzen oder Arrays (allgemein Folgen) iterierende Schleifen, die Seiteneffekte im Schleifenkörper erzeugen (welcher zum Wert `()` (`unit`) evaluiert), gibt es die spezifischen Bibliotheksfunktionen `iter` und `iteri`. Sie sind u.a. für die Typen Liste, Array und Sequenz verfügbar. Sie kapseln dieses typische Muster, indem eine Seiteneffekt erzeugenden Methode als Funktion höherer Ordnung übergeben wird (entsprechend mit Rückgabetyp `unit`). `iteri` übergibt an diese Funktion zusätzlich zum aktuellen Wert der Folge noch den aktuellen Index des Typs. Bei Iteration über ein Array können Operationen auf dessen konkrete Elemente ausgeführt werden.

3.1.6 Applikation & Komposition von Funktionen

Die Applikation von Argumenten kann in F# mit den sogenannten *Pipe*-Operatoren vorgenommen werden. Sie existieren in Form von `|>` (vorwärts Pipe) und `<|` (rückwärts Pipe). Der vorwärts Pipe-Operator entspricht dem in (2.3.18) und (2.3.19) präsentierten *ap*-Operator. Das in Programmcode 3.9 dargestellte Beispiel der Umrechnung von Dezimal- zu Dualsystem zeigt die Betonung des *Datenflusses* bei Verwendung des (vorwärts) Pipe-Operators. Dieser erlaubt das Weglassen der Klammerung bei der Verkettung mehrerer Funktion.

(Die Funktion `unfold` ist die Umkehrung von `fold`, indem aus einem Ausgangswert eine Sequenz erzeugt wird. Durch mehrmalige Anwendung einer Funktion jeweils für Folgewert und Wert, welcher in die Sequenz eingetragen wird.)

```
195 let toBinaryNumStr (d : int) : string =
196     d
197     |> Seq.unfold (fun x -> if x > 0 then Some (x%2, x/2) else None)
198     |> Seq.toList // Liste mit Koeffizienten der Zweierpotenzen
199     |> List.rev // Liste umkehren
200     |> List.fold (fun i i' -> i.ToString () + i'.ToString ()) "" // String
        erzeugen
201
202 let bins = List.map toBinaryNumStr [1..10] // bins : string list = ["1"; "10";
    "11"; "100"; "101"; "110"; "111"; "1000"; "1001"; "1010"]
```

Programmcode 3.9: Beispiel *Pipe*-Operator: Umrechnung Dezimal- zu Dualsystem

Der zu (2.3.12) äquivalente Operator in F# heißt `<<`. Eine Variante in umgekehrter Kompositionsrichtung ist mit `>>` ebenfalls verfügbar.

3.1.7 Listen, Sequenzen & Auswertungsstrategie

In F# unterscheiden sich Listen und Sequenzen in der Auswertungsstrategie. In F# werden Ausdrücke generell strikt evaluiert, so auch Listen. Sequenzen dagegen basieren auf dem Interface `IEnumerable` und sind *lazy*, d.h. sie werten Elemente nur bei Bedarf aus und nicht komplett die ganze Sequenz. Diese und andere Datenstrukturen besitzen im Namensraum `Microsoft.FSharp.Collections` entsprechende Module mit statischen Funktionen, um sie zu verarbeiten. Beispiele wie `List.fold` oder `Seq.unfold` wurden in vorherigen Programmbeispielen schon verwendet.

3.2 Computation Expressions

Computation Expressions, alias *Workflows*, bezeichnen das F#-Syntax-Konstrukt zum Umgang mit in Monaden gekapselten Berechnungen. Abbildung 3.3 zeigt den Aufbau der Workflow-Syntax.

```
<Builder> { <Ausdrücke> }
```

Abbildung 3.3: *Computation Expressions*-Syntax

Das *Builder*-Schlüsselwort verweist auf den Namen der Instanz des verwendeten Workflow-Typs. Häufig gebraucht werden `seq`[3] und `async`[4]. Es können eigene Workflow-Typen durch Erzeugen einer *Builder-Klasse* definiert werden.

Innerhalb des geklammerten Blocks können neben normalen Ausdrücken die speziellen Workflow-Schlüsselwörter zum Erzeugen der gewünschten Seiteneffekte verwendet werden. Sie rufen die speziellen, den Ablauf steuernde Methoden des verwendeten Workflows auf. Zu diesen Methoden gehören u.a. die Operatoren `Bind`, aufgerufen durch das Schlüsselwort `let!` und `do!` sowie `Return`, aufgerufen durch das Schlüsselwort `return`. Diese Methoden geben jeweils monadische Werte vom Typ `M<'a>` zurück, wobei `M` den jeweiligen speziellen Workflow-Typ bezeichnet, z.b. `Async<'a>`. Diese Workflow-Objekte werden innerhalb des Workflows als monadische Werte zwischen den Methoden weitergereicht und auf diese Weise zu einem gesamten Workflow verknüpft. Eine

[3] siehe Unterabschnitt 3.2.1 Sequence Expressions
[4] siehe Unterabschnitt 3.2.2 Asynchronous Workflows

komplette Auflistung von Workflow-Methoden und zugehörigen Schlüsselworten ist in [Mic10] zu finden.

Ein Beispiel zur Demonstration der Arbeitsweise von Computation Expressions ist der im Quellcode-Listing A.1 abgebildete *timed*-Workflow. Dieser misst die Ausführungszeiten von Funktionen innerhalb eines Workflows mit einer *monadischen* Hilfsfunktion `clock`, welche zu einem Wert immer noch den Kontext der Zeitmessung zur Berechnung desselben hinzufügt. Die Messergebnisse werden im Hintergrund durch den Workflow in einer Liste gesammelt.

Folgend werden in dieser Arbeit Computation Expressions nur noch in der Form von `async` und `seq` verwendet.

3.2.1 Sequence Expressions

Sequence Expressions sind Ausdrücke zur Generierung von Folgen wie Sequenzen (`seq {...}`), Listen (`[...]`) oder Arrays (`[|...|]`). Als F#-Äquivalent der Comprehension-Notation[5] sind sie eine Spezialform der Computation Expressions. Programmcode 3.10 zeigt ein Beispiel einer Folge von Quadratzahlen analog zu (2.3.15).

```
8  seq { for x in 1.0 .. 4.0 do
9        for y in 2.0 .. 3.0 do
10           if x % 2.0 = 0.0 then
11               yield x ** y }
12 // seq<float> = seq [4.0; 8.0; 16.0; 64.0]
```

Programmcode 3.10: Quadrat-Folge als *Sequence Expression*

`..` ist ein binärer Operator zur Generierung von Sequenzen mit einer automatischen Schrittweite von 1. Die Schrittweite kann optional mit `<Start> .. <Schrittweite> .. <Ende>` angegeben werden. Der Operator wird auch in Schleifenkonstrukten zur Angabe von Bereichen der Iterationsvariablen verwendet.

3.2.2 Asynchronous Workflows

Mit *Asynchronous Workflows* [Micg] werden zur asynchronen Ausführung gedachte Berechnungen in Form eines Workflows definiert (Builder-Schlüsselwort `async`). Damit wird ein Objekt vom Typ `Async<'a>` erzeugt, welches diese Berechnungen repräsentiert. Mit Hilfe der Klasse `Async`[6] und dessen verschiedenen Auslöser-Funktionen werden die asynchronen Berechnungen schließlich gestartet. Beispiele sind `Async.StartImmediate`, zum Starten der Berechnung im aktuellen Thread, oder `Async.StartAsTask`, um die Berechnung als *Task* in einem *Thread Pool* zu starten.[7]

Innerhalb des Workflow-Blocks wird zwischen synchronen und asynchronen Berechnungen unterschieden, indem Ergebnisse von asynchronen Berechnungen (asynchrone Elementarroutinen) vom Typ `Async<'a>` mit `let!` gebunden werden. Bereits asynchrone Berechnungen werden auf diese Weise in den aktuelle Workflow eingebunden (*bind*-Operation (2.3.22)). Um asynchrone Elementarroutine selbst zu definieren, wird der Körper einer Funktion komplett in einen `async`-Block eingeschlossen.

[5] siehe Unterabschnitt 2.3.11 Comprehension-Notation
[6] s. a. `Microsoft.FSharp.Control.FSharpAsync`
[7] Task und Thread Pool sind Bestandteile der *Task Parallel Library (TPL)* [Micw]

Beispiele zum Umgang mit Asynchronous Workflows sind im Anhang unter Programmcode A.2 und in [Sym10] zu finden.

3.3 Objektorientierte Programmierung

F# ist eine vollwertige objektorientierte Sprache. Es kann sowohl mit .NET-Objekten interagiert als auch eigene Klassen definiert werden. [Ng10] präsentiert eine Zusammenfassung der gängigen objektorientierten Konstrukte als Gegenüberstellung von Varianten in F# und C#. Da im Rahmen dieser Arbeit nur eine sehr simple Klasse implementiert wurde, soll an dieser Stelle nur auf die genannte Gegenüberstellung sowie auf die ausführlicheren Erläuterungen der entsprechenden Kapitel in [Smi10, SGC10, Pic09] verwiesen werden.

3.4 Entwicklung mit F#

3.4.1 Allgemeines

Zuletzt soll auf einige Feinheiten der Entwicklung mit F# eingegangen werden.

Ein wichtiges Instrument ist *F# Interactive* (FSI). Dabei handelt es sich um eine Art REPL (read-eval-print-loop Konsole), in die Ausdrücke geladen werden können, um diese interaktiv zu verwenden und auszuprobieren. Im Unterschied zur REPL der Sprache LISP, werden die Ausdrücke nicht nur evaluiert, sondern tatsächlich vom Compiler übersetzt und ausgeführt. Zu beachten ist, dass dies erst nach Terminierung von Eingaben durch `;;` passiert. FSI kann direkt durch Ausführen der Datei `Fsi.exe` gestartet werden oder man nutzt die Integration in VS. Eine einfache Methode ist, Programmteile zu markieren und diese durch drücken von `Alt+Enter` in FSI zu laden. Die in dieser Arbeit präsentierten kurzen Programmbeispiele stammen aus der Datei `scribbles.fsx` und können auf die eben beschriebene Weise ausprobiert werden.

Explizite Typen-Annotation zu schreiben ist selten nötig, da Typen vom Compiler automatisch abgeleitet werden. Sie trotzdem zu schreiben hilft beim Verständnis des Programmcodes und sie bilden eine Art Dokumentation. Bei der eigentlichen Entwicklung kann darauf verzichtet werden und man benutzt die VS-Funktionaliät, mit der vom Compiler abgeleitete Typensignaturen beim Führen der Maus über einen Wert automatisch anzeigt.

Bei der Verwendung von Objekten-Methoden aus dem .NET-Framework innerhalb von F# kann die Klammerung von einem einzelnen Argument weggelassen werden (bei Methoden, die nur ein Argument akzeptieren). Bei mehreren Argumenten werden diese als Tupel übergeben. Das bedeutet, dass tatsächlich ein F#-Tupel übergeben wird und nicht aus Syntax-Gründen geklammert wird. Methoden ohne Argumente wird das leere Tupel `()` übergeben (`unit`). Wenn jegliche Übergabe von Werten unterlassen wird, kann die Methode wie auch Funktionen innerhalb von F# als Wert behandelt und z.B. anderen Funktionen übergeben werden. Programmcode 3.11 illustriert ein Beispiel.

```
211 let ceil : float -> float = Math.Ceiling
212 ceil 2.3 // float = 3.0
213
214 List.map (Math.Ceiling : float -> float) [1.0 .. 10.0]
215 // float list = [1.0; 2.0; 3.0; 4.0; 5.0; 6.0; 7.0; 8.0; 9.0; 10.0]
```

Programmcode 3.11: Verwendung einer .NET-Methode als Funktionen-Variable

Um die exakte Überladung der Methode auszuwählen, musste der Typ der Funktionen-Variable explizit angegeben werden.

3.4.2 Bibliotheken

F# bringt eine eigene Kernbibliothek mit [Micd]. Diese ergänzt die Kernsprache um häufig benutzte Konstrukte wie die erwähnten asynchronen Workflows. *F# PowerPack* ist eine extern angebotene Bibliothek, die ebenfalls vom F#-Team stammt [Micr].

3.5 Nicht behandelte Konstrukte

Zum Schluss soll noch auf nicht tiefer behandelte Konstrukte hingewiesen und diese kurz eingeordnet werden.

- *Quotations* [Mich] sind ein Mittel um F#-Code programmatisch zu generieren und zu manipulieren.
- *Agents* [Mice] (Klasse `MailboxProcessor` der F#-Bibliothek) sind eine Umsetzung des *Aktoren-Modells*, welches Nebenläufigkeit durch elementare Prozesse (Aktoren) modelliert, die auf Nachrichten beliebig vieler schreibender Prozesse reagieren. Die funktionale Sprache *Erlang* [Arm07], die sich auf Nebenläufigkeit und Programmierung verteilter Systeme konzentriert, macht von diesem Muster stark Gebrauch.
- *Records* [Micn] sind ein Typ zur Aggregation verschiedener Werte ähnlich einem Tupel, der aber mit zusätzlichen Schlüsselwörtern Zugriff auf einzelne Elemente erlaubt. Programmcode 3.12 zeigt ein simples Beispiel.

```
206 type Punkt3D = {x : float; y : float; z : float}
207 let ursprung : Punkt3D = {x = 0.0; y = 0.0; z = 0.0}
```

Programmcode 3.12: *Record*-Beispiel

- *Object Expressions* [Mick] sind ein Mittel zur dynamischen Erzeugung von Objekten (ähnlich *anonymer Klassen*).
- *Type Extensions* [Micp] erlauben das Hinzufügen von erweiternden Methoden zu bestehenden Typen. Diese sind mit der Punktnotation bei Werten dieses Typs verfügbar.
- *Active Patterns* [Micf] sind eine Art Kurzform der *Discriminated Union* und erlauben das Partitionieren von Daten, um diese beim *Patternmatching* entsprechend zu dekonstruieren.
- *Units of Measure* [Micq] erlauben die Deklaration von (naturwissenschaftlichen) Größen. Auf diese Weise kann bspw. zwischen einem Gleitpunktwert, der eine bestimmte Längengröße indiziert, und einem Gleitpunktwert, der Volumen anzeigt, unterschieden werden. Die korrekte arithmetische Verknüpfung der Werte wird dabei vom Compiler überprüft.

4 Datenparallelität mit Accelerator

Accelerator ist eine .NET-Bibliothek zur Ausführung von Array-Operationen auf einer GPU oder Mehrkern-CPU. Die Bibliothek stellt eine Programmierschnittstelle zur Verfügung, mit der *datenparallele* Operationen definiert und später auf einem bestimmten Zielsystem (sog. *target*) ausgeführt werden.

4.1 Begriffsabgrenzung

Bei der parallelen Verarbeitung wird zwischen verschiedenen Modellen unterschieden. Auf Prozessorebene wird das Aufteilen von Maschinenbefehlen in parallel ausführbare Teilaufgaben zur Steigerung des Durchsatzes als *Pipelining* bezeichnet. Dieses Prinzip wird in äquivalenter Weise als Entwurfsmuster verwendet. Teilaufgaben eines Prozesses werden durch Puffer miteinander verbunden, sodass Teilaufgaben parallel verarbeitet werden können [Micu].

Eine andere Form von paralleler Verarbeitung ist die *Prozessparallelität*, bei der unterschiedliche Kontrollflüsse nebenläufig sind und parallel auf verschiedenen Prozessoren ausgeführt werden können. Dies wird als *asynchrone Parallelverarbeitung* bezeichnet [BFRR95] (S. 3). Im Fokus steht die mögliche Parallelität durch unabhängige, nebenläufige Prozesse mit jeweils eigenem Kontrollfluss. Der Zugriff von verschiedenen Prozessen auf gemeinsame Daten muss daher *synchronisiert* werden. Typische Beispiele sind nebenläufige E/A-Operationen oder Arbeitsthreads im Hintergrund einer grafischen Anwendung.

Um Prozessparallelität im Rahmen von .NET anzuwenden, gibt es die *Task Parallel Library (TPL)* [Micw]. In Kombination mit den in Unterabschnitt 3.2.2 beschriebenen Asynchronen Workflows können Hintergrundprozesse definiert und gesteuert werden.

Von *Datenparallelität* wird hingegen gesprochen, wenn Daten auf mehrere Prozessoren in *Partitionen aufgeteilt* werden und in gleicher Weise parallel verarbeitet werden. Zur Ausführung wird ein SIMD-System oder eine Mehrkern-CPU benötigt, welche dieselben Instruktionen parallel auf verschiedene Datenströme anwenden. Damit gibt es nur einen sequentiellen Kontrollfluss. Der Fokus liegt hier auf der Parallelität der Daten. Diese Form wird nach [BFRR95] als *synchron parallel* bezeichnet, da *keine* explizite Synchronisation benötigt wird. Typische Beispiele sind lokale Operation oder Punktoperation auf Pixel eines digitalen Bildes.

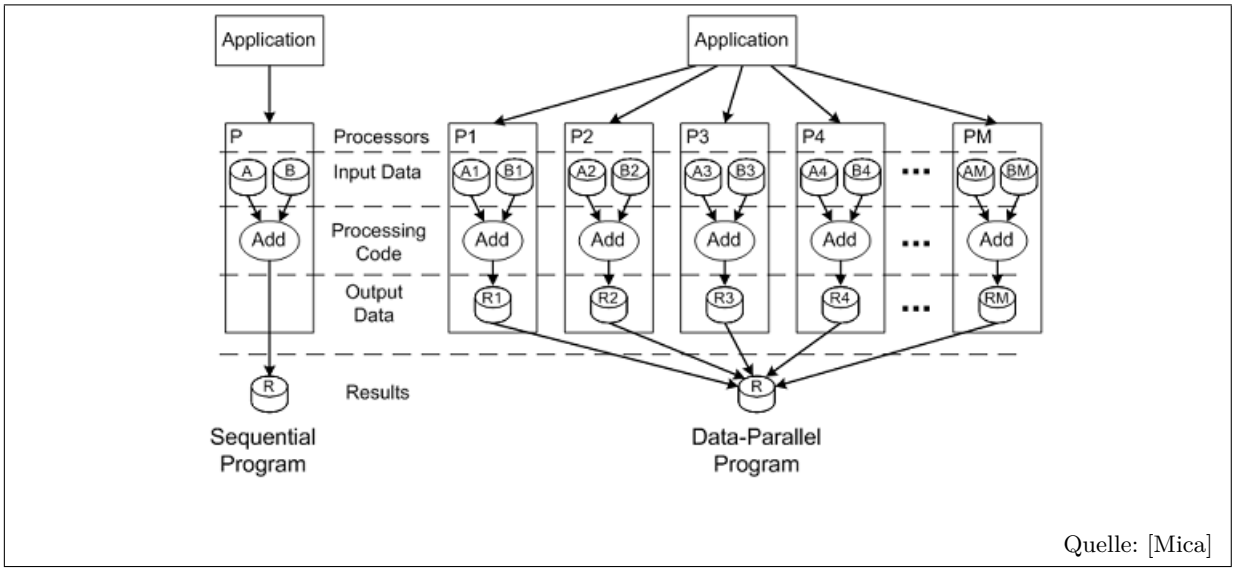
Zum Thema Datenparallelität existiert im .NET-Umfeld neben Accelerator noch das *Parallel LINQ*-Framework [Mics]. Das Framework arbeitet mit `IEnumerable`-Objekten und deren paralleler Verarbeitung auf Mehrkern-CPUs. Mit dem *PSeq*-Modul der *F# PowerPack*-Bibliothek [Micr] existiert eine Programmierschnittstelle, welche die PLINQ-Operationen ähnlich zum Umgang mit F#-Sequenzen in einem eigenen Namensraum[1] in F# integriert. Zuletzt stellt die TPL parallele Schleifen bereit, welche imperative Konstrukte darstellen.[2]

[1] `Microsoft.FSharp.Collections.PSeqModule`
[2] `System.Threading.Tasks.Parallel` [Mict, Micc]

4.2 Vorstellung Accelerator-Bibliothek

4.2.1 Datenparallele Programmierung mit GPUs

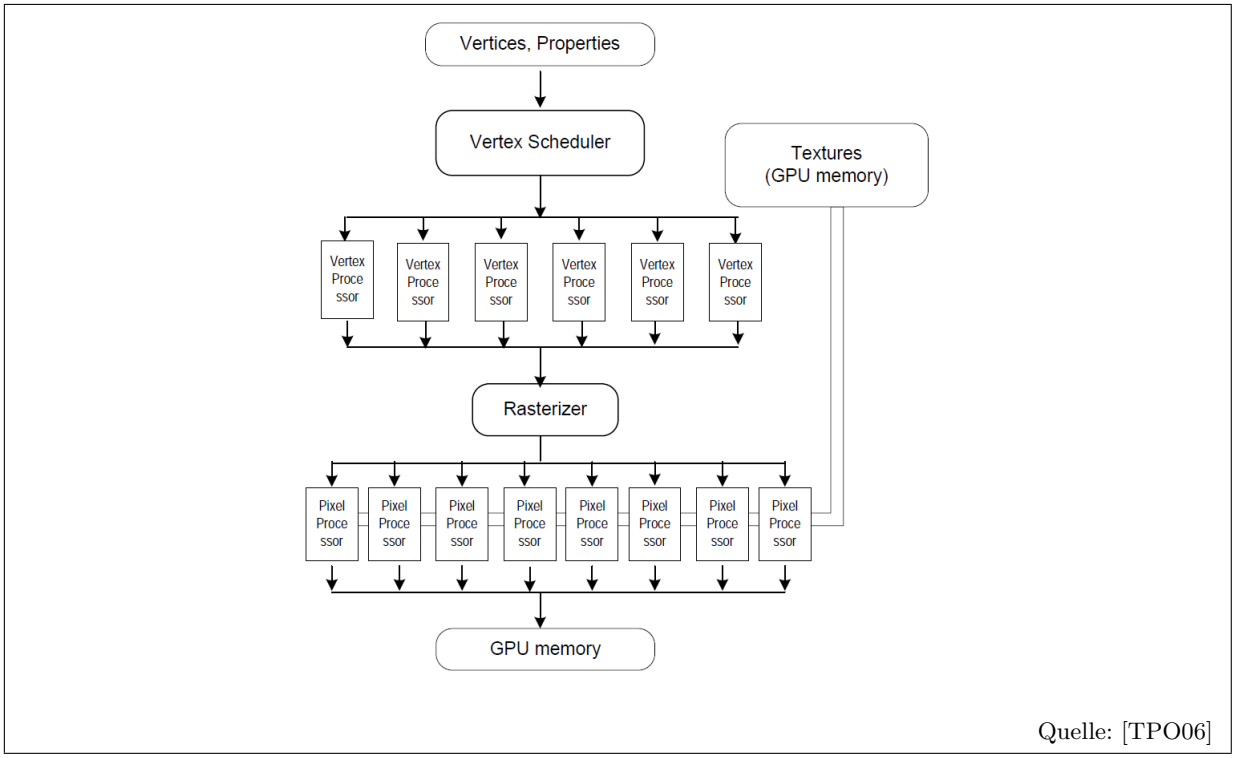
Die datenparallele Programmierung involviert zunächst den Schritt der Aufteilung der Daten in Partitionen und deren Zuweisung zum zugehörigen Prozessor. Auf jedem Prozessor wird dieselbe Operation auf dessen lokalen Daten ausgeführt. Schließlich werden die individuellen Ergebnisse zu einem Endergebnis zusammengefügt. Abbildung 4.1 illustriert diesen Ansatz im Vergleich zu einem sequenziellen Programm.



Quelle: [Mica]

Abbildung 4.1: Vergleich: datenparalleles & sequenzielles Programm

Im Falle der Programmierung mit GPUs bedeutet dies die Umsetzung der parallelen Operation in *Pixel-Shader*-Programme. Diese dienen ursprünglich der Berechnung der endgültigen Pixelwerte eines digitalen Bildes. Bei der allgemeinen Programmierung mit GPUs werden Array-Operationen mit diesen Pixel-Shader-Einheiten auf der GPU umgesetzt. Jedes Element eines Ergebnis-Arrays wird so unabhängig von den anderen berechnet. Eingabedaten werden im RAM der Grafikkarte als Texturen gespeichert. Abbildung 4.2 zeigt den schematischen Aufbau einer GPU mit Pixel-Shader-Prozessoren. Die Pixel-Shader-Einheiten greifen auf diesen Speicher ausschließlich lesend oder schreibend zu. D.h. die selbe Textur kann nicht gleichzeitig gelesen und geschrieben werden. Die Pixel-Shader-Programme können nur durch Schreiben in die jeweiligen Ausgaberegister Ergebnisse produzieren. Einschränkend hinzu kommt, dass Shader-Programme keine Konstrukte für Iteration oder Programmverzweigung vorsehen oder diese zumindest negative Auswirkungen auf die Leistung bedeuten und deshalb vermieden werden sollten. Eine andere Einschränkung stellen die verwendbaren Datentypen dar, indem i.d.R. nur Gleitkommazahlen mit 32-Bit-Genauigkeit oder weniger unterstützt werden.



Quelle: [TPO06]

Abbildung 4.2: Schematischer Aufbau einer GPU

Zur Programmierung der Shader existieren verschiedene Programmierschnittstellen wie *DirectX* oder *OpenGL* mit jeweils spezialisierten Programmiersprachen in unterschiedlich hohen Abstraktionsebenen (vergleichbar mit *Assembler* oder *C*).

4.2.2 Programmiermodell

Accelerator abstrahiert die in Unterabschnitt 4.2.1 zusammengefasste Vorgehensweise in einem Typ *paralleles Array*, welcher aus .NET-Projekten heraus genutzt werden kann. Konzeptionell unterscheidet sich dieser Typ wenig von normalen Arrays. Wie normale Arrays haben alle Elemente denselben elementaren Typ und mehrdimensionale Varianten sind möglich. Im Unterschied zu normalen Arrays ist keine Zugriff auf individuelle Elemente möglich. Nach dem SIMD-Prinzip kann dieses Array nur als Ganzes, mit derselben Operation auf alle Elemente, verarbeitet werden. Variablen dieses Typs sind *funktionaler* Natur. D.h. jede Operation generiert ohne Erzeugung von Seiteneffekte stets ein neues paralleles Array. Die Eingabe-Arrays ändern sich nicht.

Die möglichen Operationen werden als statische Methoden der Klasse `ParallelArrays` und überladene Operatoren der Subtypen von `ParallelArray`[3] zur Verfügung gestellt. Mit ihnen werden Operationen auf parallelen Arrays definiert. Um ein Ergebnis dieses Typs in ein normales Array umzuwandeln, wird ein paralleles Array einer Zielinstanz übergeben. Diese abstrahiert die verwendete Zielhardware (*target*). Neben einem GPU-Target gibt es einen Instanz-Typ für Mehrkern-CPUs. Zukünftig sollen weitere Hardware wie z.B. FPGAs unterstützt werden. Im Rahmen dieser Arbeit wird `DX9Target` verwendet, um die Berechnungen auf die Grafikkarte auszulagern. Der Aufruf der `toArray(1|2)D`-Methode dieser Zielinstanz löst die Ausführung der Operationen sowie die Umwandlung in ein normales Array aus. Die Accelerator-Operationen werden zur Laufzeit in Pixel-Shader-Programme übersetzt.

[3] zu beachten ist das fehlende ‚s‘

Berechnungen mit parallelen Arrays werden nicht sofort realisiert. Der Aufruf von Methoden mit parallelen Arrays erzeugt im Hintergrund einen gerichteten azyklischen Graphen, welcher die geforderten Operationen repräsentiert. Erst mit der Auslösung des Targets wird dieser Graph von der Bibliothek evaluiert und in Pixel-Shader-Code übersetzt und ausgeführt. Dies erlaubt eine effiziente Übersetzung der Operationen. Ausführliche Erläuterungen sind in [TPO06] zu finden.

Programmcode 4.1 demonstriert ein einfaches Anwendungsbeispiel. Zunächst wird eine `DX9Target`-Instanz erzeugt. Sie repräsentiert eine *DirectX*-kompatible Grafikkarte. Anhand von zwei normalen Arrays werden die parallelen Arrays erstellt. Mit Hilfe von statischen Methoden der Klasse `ParallelArrays` und überladenem Operator wird eine Berechnung definiert. Schließlich wird das parallele Ergebnis-Array mit Hilfe der Zielinstanz in eine normales Array umgewandelt.

```
130 // ..\Accelerator v2\bin\x86\Debug\Accelerator.dll muss in das Verzeichnis von
    Fsi.exe kopiert werden
131 #r @"C:\Program␣Files\Microsoft\Accelerator␣v2\bin\Managed\Release\Microsoft.
    Accelerator.dll"
132 open Microsoft.ParallelArrays
133 // Abkuerzungen oft benutzter Typen
134 type PA = ParallelArrays
135 type FPA = FloatParallelArray
136 type BPA = BoolParallelArray
137
138 // Zielinstanz erzeugen
139 let target = new DX9Target()
140 // normale Ausgangsarrays erzeugen
141 let arr1 : float32 [,] = Array2D.create 3 3 1.4f
142 let arr2 : float32 [,] = Array2D.create 3 3 2.3f
143 // parallele Arrays erzeugen
144 let parr1 = new FPA (arr1)
145 let parr2 = new FPA (arr2)
146
147 // Operation mit parallen Arrays
148 let pResultArr : FPA = PA.Floor(parr1) - PA.Ceiling(parr2)
149
150 // Ergebnis wieder in normales Array umwandeln (Berechnung ausloesen)
151 let resultArr = target.ToArray2D(pResultArr)
152 // resultArr : float32 [,] =
153 // [[-2.0f; -2.0f; -2.0f]
154 // [-2.0f; -2.0f; -2.0f]
155 // [-2.0f; -2.0f; -2.0f]]
```

Programmcode 4.1: Accelerator: Einfache Beispieloperation (1)

4.2.3 Architektur

Der Aufbau der Bibliothek soll anhand der folgenden Abbildungen geschildert werden. Zentrale Bestandteile bilden die statische Klasse `ParallelArrays` (Abbildung 4.4), der abstrakte Typ `ParallelArray` mit dessen Subtypen (Abbildung 4.5) und mögliche `Target`-Typen (Abbildung 4.3).

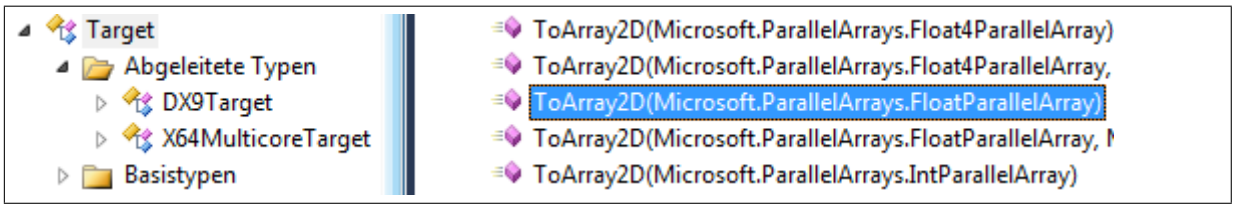


Abbildung 4.3: Accelerator: mögliche Zielinstanzen

Die Methoden der Klasse `ParallelArrays` stellen die eigentliche Funktionalität der Bibliothek dar. Dort sind die mit parallelen Arrays mögliche Operationen definiert. Für jede Methode gibt es verschiedene Überladungen, je nachdem welcher konkrete parallele Array-Typ verwendet wird. Wesentliche Typen sind `FloatParallelArray` und `BoolParallelArray`. Die Typen bieten in Form überladener Operatoren weitere Funktionalität an.

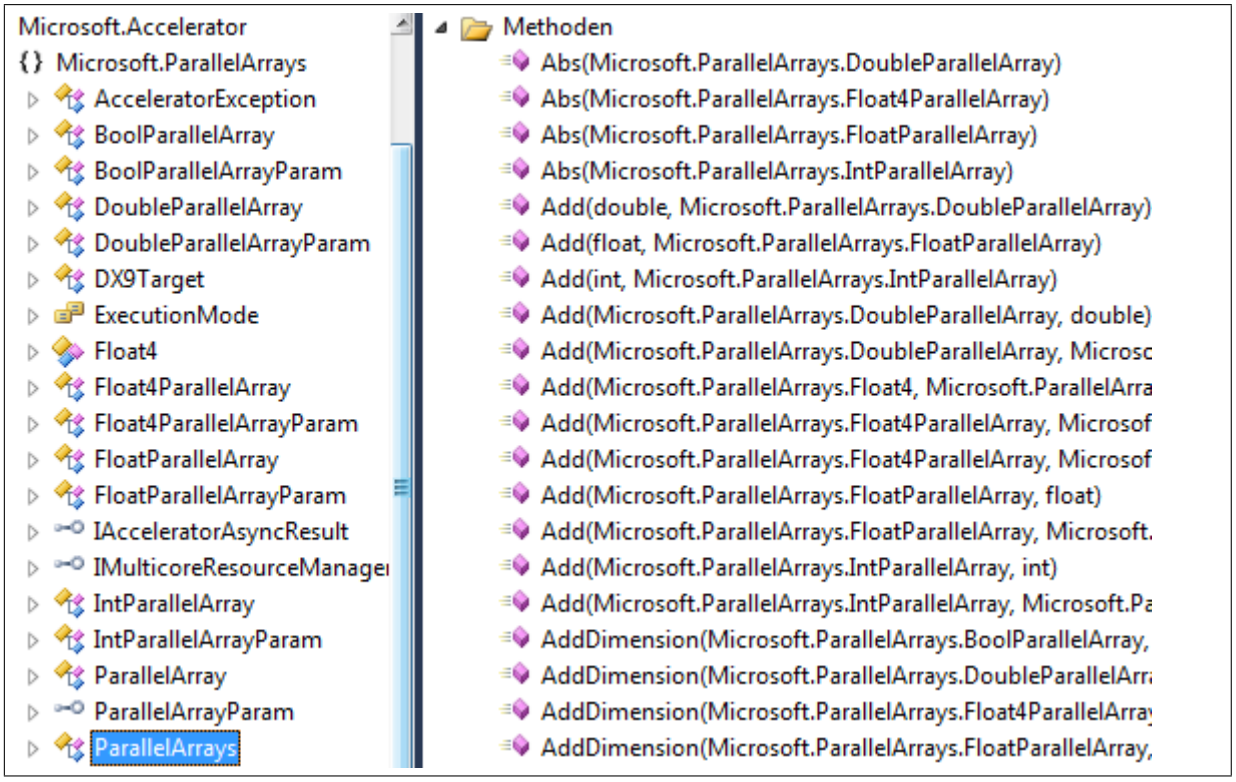


Abbildung 4.4: Accelerator: zentrale Klasse *ParallelArrays* mit statischen Methoden

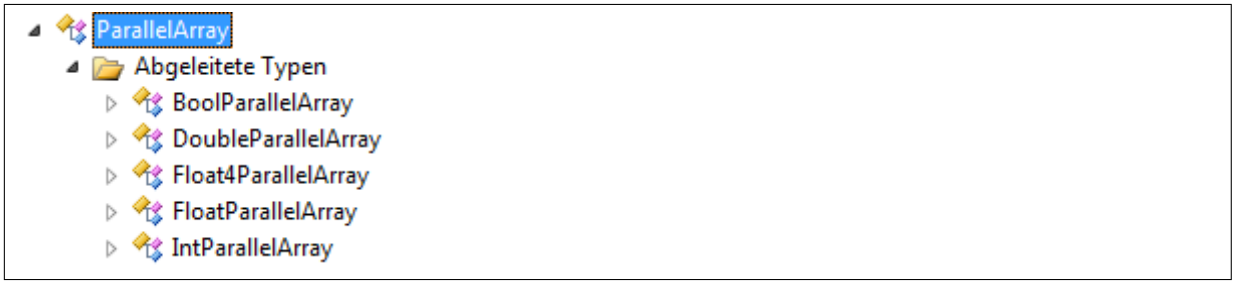


Abbildung 4.5: Accelerator: Hierarchie paralleler Array-Typen

Zu beachten ist, dass zum derzeitigen Stand noch nicht alle Typen fertig implementiert sind (Bspw. ist `IntParallelArray` derzeit noch ohne Funktionalität). Weiterhin funktioniert der Typ `BoolParallelArray` intern zwar korrekt, aber dessen Umwandlung in ein normales Array ist noch fehlerhaft. Deswegen sollte zur Repräsentation von Daten hauptsächlich mit dem Typ `FloatParallelArray` gearbeitet werden.

Die Verwendung von `BoolParallelArray` illustriert Programmcode 4.2. Mit dessen Hilfe wird eine parallele Vergleichsoperation vorgenommen.

```
159 let parr3 = new FPA ([|1.0f .. 5.0f|]) // Werte 1 bis 5
160 let value = new FPA (2.3f, parr3.Shape) // 5 mal der Wert 2.3
161 let ones = new FPA (1.0f, parr3.Shape)
162 let zeros = new FPA (0.0f, parr3.Shape)
163 // Vergleich von value und parr3 erzeugt ein BoolParallelArray
164 // als Maske mit korrespondierenden boolschen Werten
165 let mask : BPA = PA.CompareLess(parr3, value)
166 // Erzeugung eines FloatParallelArray anhand der Maske
167 let pResultArr2 : FPA = PA.Cond(mask, ones, zeros)
168 let resultArr2 = target.ToArray1D(pResultArr2)
169 // Ergebnis: resultArr2 : float32 [] = [|1.0f; 1.0f; 0.0f; 0.0f; 0.0f|]
```

Programmcode 4.2: Accelerator: Einfache Beispieloperation (2)

Es wird zunächst ein Array `parr3` mit zu testenden Werten definiert. Das Array `value` enthält den Vergleichswert. Zu beachten ist, dass beide Arrays die selbe Dimension und Größe haben (`Shape`). Eine Vergleichsoperation kann mit der statischen Methode `CompareLess` vorgenommen werden. Diese liefert eine Maske in Form eines `BoolParallelArray`. Um das Ergebnis in ein normales Array zu konvertieren, wird wie folgt verfahren: Die Methode `Cond` erstellt anhand dieser Maske und zwei anderer Arrays (`ones` & `zeros`) ein neues `FloatParallelArray`, um eine korrekte Rückumwandlung zu erreichen. Entscheidend ist, dass unterschiedlich zum vorhergehenden Beispiel die Methode `toArray1D` der Zielinstanz benutzt wird. Der Entwickler muss sicherstellen, dass die Dimensionen der Arrays korrekt zueinander passen. Diese Fehler werden vom Typensystem *nicht* erkannt und führen ggf. zu Laufzeitfehlern. Ein- und zweidimensionale parallele Arrays haben den selben konkreten `ParallelArray`-Subtyp (bspw. `FloatParallelArray`). [Mica, Micb] zeigen weitere ausführlichere Beispiel im Umgang mit Accelerator.

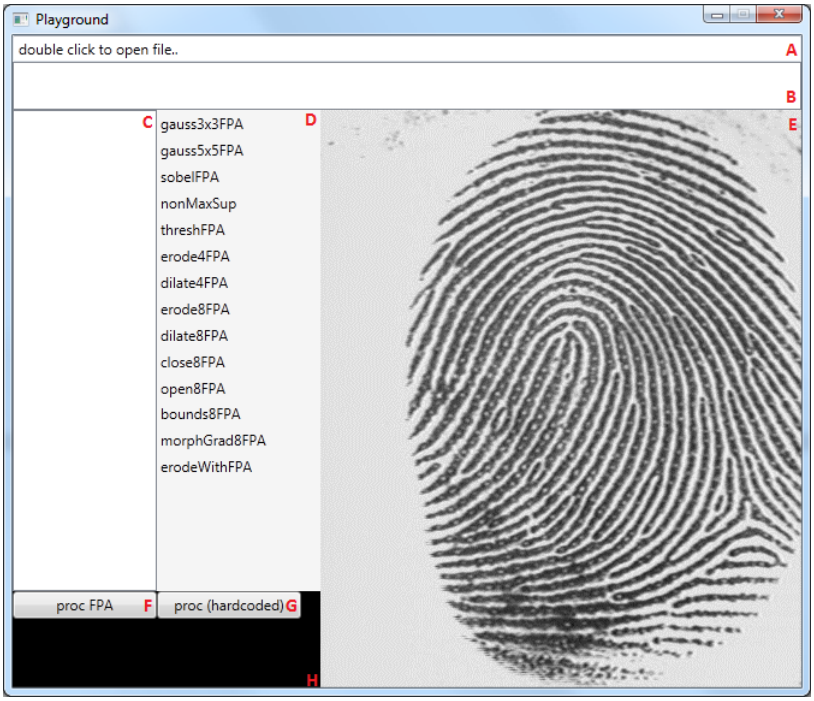
5 Praktische Anwendung

5.1 Beschreibung der Demonstrationssoftware

Die Software ermöglicht die Ausführung der implementierten BV-Operationen. An das Programm werden folgende Anforderungen gestellt: Es sollen morphologische Operationen, der Sobel-Operator und darauf aufbauend eine Non-Maximum-Suppression umgesetzt werden. Die Programmoberfläche soll es ermöglichen, Bilder zu laden und darauf BV-Operationen auszuführen. Die Berechnung soll mit Hilfe der GPU einer Grafikkarte unter Ausnutzung von Datenparallelität erfolgen. Da BV-Operationen nicht für sich allein stehen, sondern oft miteinander verknüpft werden, soll die Kombination in der Oberfläche ermöglicht werden. Da BV-Operationen tendenziell eine längere Berechnungszeit benötigen, sollen diese im Hintergrund erfolgen, um die Oberfläche nicht zu blockieren. Berechnungszeiten sollen generell erfasst und ausgegeben werden, um die genaue Dauer der Berechnungen mit und ohne GPU sowie mit verschieden großen Bilden nachvollziehen zu können. Die nächsten Abschnitte beschreiben Oberfläche und Bedienung der Software sowie den Aufbau des Quellcodes.

5.1.1 Oberfläche & Bedienung

Abbildung 5.1 zeigt die grafische Oberfläche der Demonstrationssoftware. Elemente der Oberfläche sind mit den Buchstaben *A* bis *H* gekennzeichnet und sollen folgend erläutert werden.



Bildquelle: [GW08b] (Fig. 7.36 (a))

Abbildung 5.1: Demonstrationssoftware: grafische Oberfläche

Element *A* dient dem Laden von Bilddateien. Auf Doppelklick öffnet sich ein entsprechendes Dateiauswahlfenster. Der Pfad der gerade geöffneten Datei wird in diesem Feld angezeigt. Element *B* dient der Textausgabe in Form eines fortlaufenden Protokolls. Die Bereiche *C* und *D* bilden eine Ziehen- und Ablegen-Konstellation: Die Textelemente in *D* repräsentieren die verfügbaren BV-Operationen. Diese können in das Feld *C* gezogen und kombiniert werden. Durch Doppelklick auf einzelne Elemente in *C* können diese entfernt werden. Schließlich löst die Schaltfläche *F* die zusammengestellte Bildoperation aus. Diese wird intern zu *einer* Bildoperation verknüpft und als *Ganzes* mit Hilfe von Accelerator auf der Grafikkarte ausgeführt. Die Dauer der Operation wird gemessen und in *B* ausgegeben. Schaltfläche *G* ist im Programm fest mit der naiven, ohne Accelerator implementierten Variante der *Sobel*-Operation verbunden. Die Dauer dieser Operation wird ebenfalls gemessen. Zuletzt wird im Bereich *H* ein Histogramm sowie Miniaturbild der zuvor ausgeführten Bildoperation gezeigt. In *E* wird das Ergebnis der Bildoperation in Originalgröße dargestellt.

5.1.2 Architektur

Die Software ist, wie Abbildung 5.2 zeigt, in die insgesamt sieben Module `Helper`, `Accelerator`, `Representation`, `BV`, `BVaccelerator`, `UI` und `App` aufgegliedert.

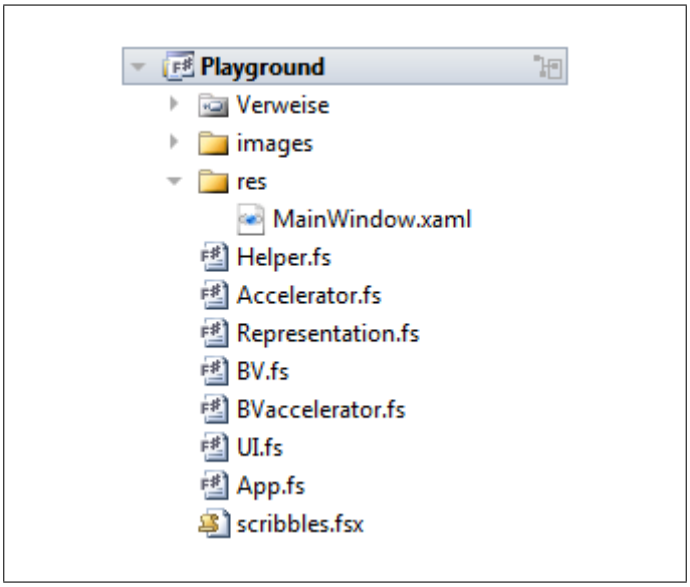


Abbildung 5.2: Demonstrationssoftware: Module

Das Modul `Helper` enthält allgemeine Hilfsfunktionen. `Accelerator` definiert Hilfsfunktionen sowie Operatoren zum Umgang mit Datentypen und Operation der Accelerator-Bibliothek. `BV` beinhaltet allgemeine Hilfsfunktionen sowie Typendefinition zum Themenbereich Bildverarbeitung. Dazu gehört die Umsetzung der *Sobel*-Operation mit sequentiellem Programmcode und ohne Ausnutzung von Datenparallelität. Das Modul `BVaccelerator` umfasst die Umsetzung der BV-Operation mit Accelerator sowie einige spezifische Hilfsfunktionen. In `Representation` werden Funktionen zur Konvertierung von verschiedenen Datentypen und Formaten definiert. Dazu gehört die Konvertierung zwischen Bitmaps, Byte-Arrays und parallelen Arrays der Accelerator-Bibliothek. Schließlich beinhaltet das Modul `App` den Einstiegspunkt für die Anwendung. Hier werden benötigte Datenstrukturen initialisiert sowie die grafische Oberfläche aufgebaut. Der Entwurf der Oberfläche ist in der Datei `res/MainWindow.xaml` definiert. Diese ist als *Ressource* in das übersetzte Programm eingebunden. Im Rahmen der Initialisierung wird diese geöffnet und die grafischen Elemente werden mit entsprechender Funktionalität hinterlegt. Dazu werden Hilfsfunktionen aus dem Modul `UI` verwendet, welches spezifische Methoden zur Manipulation der Oberfläche enthält. Zusätzlich ist in diesem Modul die Klasse `AppState` definiert. Sie dient der Vorhaltung von Ergebnissen der BV-Operationen sowie verschiedener Repräsentationen des aktuellen Bildes, die für die entsprechenden Berechnungen gebraucht werden.

5.1.3 Detailerläuterungen zur Funktionsweise

Kombination der Accelerator-BV-Operationen

Ziel der Oberflächenelemente C und D ist es, einzelne BV-Operationen beliebig kombinieren zu können. Dabei dient das Ergebnisbild einer Operation als Eingabe für die direkte Folgeoperation. Um unnötige Umlaufzeiten und Konvertierungen zwischen GPU und CPU zu vermeiden, sollen alle Operationen in einem Durchlauf als Ganzes auf der GPU ausgeführt werden.

Dies geschieht anhand einer assoziativen Struktur `cmdMap`, welche Zeichenfolgen als Schlüssel (Kommandos) auf Funktionen vom Typ `(FPA -> FPA)` abbildet. Die Textelemente im Oberflächenelement C werden mit der Funktion `mkCmds` zunächst in eine Liste von Schlüssel-Zeichenketten übersetzt. Die Funktion `mkImProcFpaFunc` erstellt daraus eine Liste von Accelerator-Funktionen vom Typ `(FPA -> FPA) list` und faltet diese mit der binären Funktion `>>` (vorwärts-Komposition) und der Identitätsfunktion `id` als neutrales Element zu einer einzelnen Funktion vom Typ `(FPA -> FPA)`.

Verwendung *asynchroner Workflows* und *Task*-Objekte

Die Operationen der Schaltflächen F und G sind jeweils als asynchrone Workflows definiert. Für die Berechnung mit Accelerator wird die eben beschriebene Funktion der Workflow-Definition als Argument übergeben. Der Berechnung ohne Accelerator wird die naive Variante der *Sobel*-Operation direkt übergeben. Im Workflow wird jeweils die Ausführungszeit der BV-Operation gestoppt sowie ein Histogramm erstellt.

Die *Callbacks* der Schaltflächen starten die asynchronen Operationen als *Task*-Objekt im Hintergrund. Sie behandeln die Präsentation der Ergebnisse des Tasks durch Darstellung des Ergebnisbildes, des Histogramms und Ausgabe der gemessenen Zeit.

Klasse `AppState` und Event Handling

Diese Klasse dient der Vorhaltung des Originalbildes in verschiedenen Repräsentationen (Arrays). Bei erneuter Auslösung einer BV-Operation muss das Originalbild nicht erneut eingelesen und in entsprechende Formate konvertiert werden. Weiterhin könnten hier andere Zwischenergebnisse wie bspw. Histogramme vorgehalten werden, um diese als Eingabe für weitere BV-Operationen zu nutzen. Das Ergebnisbild wird ebenfalls vorgehalten, jedoch nicht weiter benutzt. Denkbar wäre, dieses als Eingabe für weitere BV-Operationen zu nutzen. Im vorliegenden Programm wird durch dessen Zuweisung nur ein entsprechendes Event ausgelöst, um das Ergebnisbild in der Oberfläche darzustellen.

Während die Ausgabe der Zeitmessung und die Darstellung des Histogramms direkt im *Callback* der Schaltflächen durch Aufruf entsprechender Hilfsmethoden realisiert ist, wird die Darstellung des Ergebnisbildes *Event-basiert* vorgenommen. Ein entsprechendes Event wird bei Zuweisung des Ergebnisbildes im `AppState`-Objekt ausgelöst. Auf dieses Event wird mit Berechnung des Miniaturbildes sowie der Darstellung beider Bildern reagiert.

Asynchrone & Event-basierte Programmierung mit *Reactive Extensions* (Rx)

Die Reaktionen auf Events sowie auf Abschluss von Tasks werden mit der *Reactive Extensions*-Bibliothek (Rx) [Micv] realisiert. Diese setzt einen Benachrichtigungsmechanismus mit Hilfe

von den in .NET 4.0 eingeführten Interfaces `IObservable<T>` und `IObserver<T>` um. Dabei handelt es sich um *beobachtbare Collections* (`IObservable<T>`), welche *Pushbenachrichtigungen* an Beobachter senden.

5.2 Morphologische Bildverarbeitung

In der morphologischen Bildverarbeitung werden Objekte in Bildern als Mengen von Pixeln dargestellt. Morphologische Operationen modifizieren und analysieren die Form dieser Objekte durch Mengenoperationen mit Bildpunkten. Diese Mengenoperationen arbeiten mit einer Menge aller Pixel im Bild und einer Menge von Pixel eines *strukturierendes Element* (SE). Das SE bestimmt die Wirkung der Operation. Das SE bestimmt relativ zu einem Referenzpixel (in der Regel das Zentrum) und dessen maskierten, benachbarten Pixeln welchen Wert ein Referenzpixel zugewiesen bekommt. Dazu wird das SE über alle Pixel der Bildmenge gelegt.

Im einfachsten Fall werden morphologische Operationen auf Binärbilder angewendet, dann kann ein Pixel zu einem Objekt nur hinzugefügt oder entfernt werden. Je nach Konvention sind weiße oder schwarze Pixel Elemente einer Menge. Bei Grauwertbildern tragen Pixel einer Menge zusätzlich einen Intensitätswert. Theoretisch kann das SE aus Grauwerten bestehen [GW08a] (S. 665). Im Rahmen dieser Arbeit wird aber von einem binären SE ausgegangen. Einsen markieren Pixel, die Teil der SE-Menge sein sollen.

Wenn alle durch das SE maskierten, benachbarten Pixel des Referenzpixels in der Mitte mit Pixel aus der Bildmenge besetzt sein sollen, um den Referenzpixel in die neue Bildmenge aufzunehmen, wird dies als *Erosion* bezeichnet. D.h. die Menge des SE muss komplett in der Bildpixelmenge enthalten sein, um den Referenzpixel in die neue Bildpixelmenge aufzunehmen. Die Bildpixelmenge wird verkleinert. *Dilatation* vergrößert diese Menge, indem der referenzierte Pixel in die neue Bildmenge aufgenommen wird, falls nur *ein* maskierter Nachbarpixel Teil der Bildmenge ist, d.h. es reicht wenn die Schnittmenge zwischen beiden nicht leer ist. Erosion und Dilatation werden bei Binärbildern durch UND- (Erosion) und ODER-Operationen (Dilatation) auf maskierte Pixel vorgenommen. Bei Grauwertbildern und einem binären SE entspricht dies der MIN-Operation für Erosion und der MAX-Operation für Dilatation [GW08a] (Kapitel 9.6.1). Die beiden Operatoren stellen elementare Operationen der morphologischen Bildverarbeitung dar. Viele komplexere Operationen werden aus diesen beiden primitiven Operatoren erzeugt.

Im Programm sind sie in jeweils drei Formen für Binär- und Grauwertbilder definiert. Die drei Varianten unterscheiden sich im verwendeten SE: Die erste Variante verwendet alle Pixel in 4er-Nachbarschaft mit Distanz eins zu einem Pixel (für Grauwertbilder `erode4FPA` & `dilate4FPA`), das SE entspricht einem 3x3-Kreuz. Die zweite Variante (`erode8FPA` & `dilate8FPA`) benutzt die 8er-Nachbarschaft eines Pixels in Form eines 3x3 Quadrats (verwendet in Abbildung 5.3). Die letzte Form (`erodeWithFPA` & `dilateWithFPA`) verallgemeinert alle Varianten, indem ein zweidimensionales binäres Array als SE übergeben wird.

Der Zugriff auf Nachbarpixel erfolgt datenparallel durch Schieben des gesamten Bildes. Alle linken Nachbarpixel eines Bildes erhält man durch Schieben des Bildes um einen Pixel nach links. Für die allgemeine Form von Erosion und Dilatation wird die Funktion `shiftToFPA` verwendet, die das SE in entsprechende Schiebeoperation übersetzt und eine Sequenz von Nachbarbildern zurückgibt. Zur Erzeugung des Ergebnisbildes werden diese Nachbarbilder mit der Funktion *fold* und einer binären Funktion UND oder ODER für Binärbilder, bzw. MIN oder MAX für Grauwertbilder, verknüpft. Das neutrale Element für diese Operation ist das Ausgangsbild selbst.

(a) Erosion (b) Dilatation

Abbildung 5.3: Bildoperation: Erosion & Dilatation mit 3x3-Quadrat-Maske

Aus den beiden beschriebenen fundamentalen Operationen wurden aufbauend die vier Operationen *Ränder-Extraktion* (`bounds8FPA`), *Öffnen* und *Schließen* (`open8FPA` & `close8FPA`) sowie die Berechnung des *Morphologischen Gradienten* zur Erstellung eines Kantenbildes für Grauwertbilder (`morphGrad8FPA`) implementiert. Durch Definition von entsprechenden (Mengen-) Operatoren und Verwendung der Funktionen für Erosion und Dilatation, ist die Implementierung dieser aufbauende Funktionen im Prinzip durch Abtippen der entsprechenden Formeldefinitionen[1] zu erreichen. Das Thema Morphologische Bildverarbeitung ist ansonsten in [GW08a] (Kapitel 9) näher erläutert.

5.3 Diskrete Faltung mit dem Sobeloperator

Der Sobeloperator erstellt ein Kantenbild durch Berechnung von Gradientenbildern. Hier wird ebenfalls eine Maske über alle Bildpunkte geschoben. Diese enthält Koeffizienten mit der die korrespondierenden Nachbarpixel multipliziert werden. Der neue Wert des Referenzpixels in der Mitte ergibt sich aus der Summe aller mit den Koeffizienten gewichteten Nachbarpixel. Dieser Vorgang wird als *diskrete Faltung* bezeichnet (nicht zu verwechseln mit der Faltung von Listen aus der FP mit der *fold*-Funktion).

[1] z.B. in [GW08a] Tabelle 9.1

-1	-2	-1
0	0	0
1	2	1

(a) x-Richtung

-1	0	1
-2	0	2
-1	0	1

(b) y-Richtung

Abbildung 5.4: Sobeloperator: Filtermasken

Mit den in Abbildung 5.4 dargestellten Sobel-Filtermasken für das Gradientenbild in x- und y-Richtung wird eine diskrete Faltung vorgenommen. Die beiden entstehenden Gradientenbilder g_x und g_y werden mit der Formel aus Gleichung 5.3.1 zu einem Kantenintensitätsbild (Gradientenbild) kombiniert.

$$M(x,y) \quad = \quad \sqrt{g_x^2 + g_y^2} \tag{5.3.1}$$

Die Berechnung von g_x und g_y erfolgt mit den spezialisierten Funktionen `conv_sobel_3x3_x` und `conv_sobel_3x3_y`. Diese Funktionen nutzen die *lineare Separierbarkeit* der Masken aus. Das Bild wird mit dem Vektor $(\,1\;2\;1\,)$ bzw. $(\,1\;2\;1\,)^T$ multipliziert, d.h. Multiplikation der beiden Nachbarbilder sowie des Originalbildes mit den entsprechenden Koeffizienten. Danach wird linkes und rechts bzw. oberes und unteres Nachbarbild des Ergebnisbildes voneinander subtrahiert (Vektorkomponente $(\,-1\;0\;1\,)^T$ bzw. $(\,-1\;0\;1\,)$ der Filtermasken). Anstatt 8 werden auf diese Weise nur 4 Schiebeoperationen benötigt.

Die gesamte parallele Implementierung des Sobel-Operators ist in der Funktion `sobelFPA` im Modul `BVaccelerator` umgesetzt. Als Vergleich ist im Modul `BV` eine naive Variante mit der Funktion `sobel` implementiert, welche mit normalen zweidimensionalen Arrays arbeitet und mit Array-Indizes auf benachbarte Pixel zugreift.

5.4 Non-Maximum Suppression

Die Non-Maximum Suppression setzt auf das vom Sobel-Operator erstellte Kantenbild auf und versucht nicht maximale Kanten zu eliminieren (zu unterdrücken). Die Funktion `nonMaxSup` im Modul `BVaccelerator` implementiert dieses Verfahren. In Kombination mit vorheriger Faltung mit einer *Gauß*-Filtermaske bildet sie einen Teil des *Canny*-Algorithmus.

Die Gauß-Filtermasken können durch Übergabe von σ-Wert und gewünschter Größe der Maske mit der im Modul `BV` definierten Funktion `mkGauss` erstellt werden. Die Funktionen `gauss3x3FPA` und `gauss5x5FPA` im Modul `BVaccelerator` benutzen die in `BV` vordefinierten Masken und setzen die Faltung um.

Nachdem das Bild geglättet wurde, wird eine Variante der Sobel-Funktion angewendet (`sobelMAlphaFPA`), die neben dem Kantenbild (siehe 5.5a) zusätzlich die Winkel der Gradienten-Vektoren nach Gleichung 5.4.1 berechnet.

$$\alpha(x,y) \quad = \quad \tan^{-1}\left(\frac{g_y}{g_x}\right) \tag{5.4.1}$$

Diese Informationen werden benutzt, um das Gradienten-Bild in dessen Teile mit horizontalen, vertikalen und den zwei diagonalen Kantenarten aufzuspalten. Dies wird durch Verwendung einer `BoolParallelArray`-Maske erreicht (analog zu Programmcode 4.2). Durch entsprechende Schiebeoperation dieser Bilder orthogonal zur Kante (entlang der Kanten-Normalen/des Gradienten-Vektors) wird das Maximum bestimmt. Nicht maximale Pixel werden mit schwarzen unterdrückt und die vier Bilder schließlich zu *einem* Kantenbild mit ausgedünnten Kanten zusammengefügt, wie in 5.5b dargestellt.

(a) Sobel mit vorheriger Gauß-Filterung (b) Ergebnisbild

Abbildung 5.5: Bildoperation: Non-Maximum Suppression

6 Fazit

Der Gegenstandsbereich der Bildverarbeitung involviert viele aufeinander aufbauende Transformationen von Bilddaten. Sie lassen sich ausgehend von elementaren Operationen zu komplexeren Algorithmen aufbauen. Dieser hierarchische Ansatz passt zum Stil der Funktionalen Programmierung, indem sich komplexere Funktionen aus kleineren zusammensetzen. Die Abhängigkeiten zwischen den isolierten Operationen werden explizit durch deren Signaturen gekennzeichnet. Dies schafft ein gewisses Vertrauen in die korrekte Arbeitsweise aller anderen Funktionseinheiten des Programms, isoliert von einer gerade betrachteten.

Tatsächlich werden globaler Applikationszustand und Seiteneffekte erst in Verbindung mit einer grafischen Oberfläche benötigt. Hier kommen die Stärken des objektorientierten Paradigmas zum Tragen, während das funktionale Paradigma für die Umsetzung der Operationen einer stark mathematisch geprägten Fachdisziplin angewendet wird.

Das Programm zeigt, wie sich Software mit Hilfe des Paradigmas der FP unter Anwendung der Sprache F# und in Kombination mit wenigen objektorientierten Elementen umsetzen lässt. Die Accelerator-Bibliothek vereinfacht die Auslagerungen von Operationen auf die GPU, indem die verwendete Sprache und Entwicklungsumgebung nicht verlassen werden muss.

BV-Operationen sind nicht immer mit Verkettung einfacher Funktionen zu lösen. Operationen der höheren Bildverarbeitung stellen größere Anforderungen an Effizienz und Art der Verknüpfung. Verschiedene Algorithmen wollen individuell parametrisiert werden und können von Ergebnissen unterschiedlicher anderer Operationen abhängen. Die Verkettung mehrerer Operationen zu *einer* Funktion vom Typ `(FPA -> FPA)`, wie im vorgestellten Programm mit BV-Operationen auf niedriger Ebene, ist auf höherer Ebene nicht mehr so einfach möglich. Weiterhin bedeutet eine subjektiv elegante Lösung eines Problems nicht unbedingt, dass allen Leistungsanforderungen entsprochen wird. Durch Einhaltung von Prinzipien wie Isolation und Referenzieller Transparenz, wird das Ableiten von effizienteren aber komplexeren Implementierungen vereinfacht.

Wie vorteilhaft die Umsetzung mit F# ist, lässt sich schwer nachweisen und ist letztendlich eine subjektive Entscheidung. Wie eingangs erwähnt, müsste dazu dasselbe Projekt mit einer imperativen bzw. objektorientierten Sprache wie C# identisch umgesetzt und anhand bestimmter Metriken bewertet werden.

Literaturverzeichnis

[Arm07] Armstrong, Joe: *Programming Erlang – Software for a Concurrent World.* The Pragmatic Programmers, 2007

[BFRR95] Bräunl, Thomas ; Feyer, Stefan ; Rapf, Wolfgang ; Reinhardt, Michael: *Parallele Bildverarbeitung.* Addison-Wesley, Bonn, 1995

[Eva03] Evans, Eric: *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2003

[GW08a] Gonzalez, Rafael C. ; Woods, Richard E.: *Digital Image Processing.* 3/e. Pearson, 2008

[GW08b] Gonzalez, Rafael C. ; Woods, Richard E.: *Digital Image Processing 3/e – Book Images Downloads.* `http://www.imageprocessingplace.com/DIP-3E/dip3e_book_images_downloads.htm`. Version: 2008. – [Online; Stand 14. Oktober 2010]

[HP06] Hinze, Ralf ; Paterson, Ross: Finger trees – a simple general-purpose data structure. (2006). – `http://www.soi.city.ac.uk/~ross/papers/FingerTree.html` [Online; Stand 23. Oktober 2010]

[Hud00] Hudak, Paul: *The Haskell School of Expression.* Cambridge University Press, 2000

[Hug90] Hughes, John: Why Functional Programming Matters. (1990). – `http://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf` [Online; Stand 23. Oktober 2010]

[Hut07] Hutton, Graham: *Programming in Haskell.* Cambridge University Press, 2007

[Jon08] Jones, Mark P.: Experience Report: Playing the DSL Card - A Domain Specific Language for Component Configuration. (2008). – `http://web.cecs.pdx.edu/~mpj/pubs/playing.html` [Online; Stand 23. Oktober 2010]

[Mica] Microsoft: *Accelerator Introduction.* `http://research.microsoft.com/en-us/projects/accelerator/accelerator_intro.docx`. – [Online; Stand 30. September 2010]

[Micb] Microsoft: *Accelerator Programmer's Guide.* `http://research.microsoft.com/en-us/projects/accelerator/accelerator_programmers_guide.docx`. – [Online; Stand 30. September 2010]

[Micc] Microsoft: *Data Parallelism (Task Parallel Library).* `http://msdn.microsoft.com/en-us/library/dd537608.aspx`. – [Online; Stand 26. Juni 2010]

[Micd] Microsoft: *F# Core Library Reference.* `http://msdn.microsoft.com/en-us/library/ee353567.aspx`. – [Online; Stand 15. Oktober 2010]

[Mice] Microsoft: *F# Core Library Reference – Control.MailboxProcessor<'Msg> Class.* http://msdn.microsoft.com/en-us/library/ee370357.aspx. – [Online; Stand 15. Oktober 2010]

[Micf] Microsoft: *F# Language Reference – Active Patterns.* http://msdn.microsoft.com/en-us/library/dd233248.aspx. – [Online; Stand 26. Juni 2010]

[Micg] Microsoft: *F# Language Reference – Asynchronous Workflows.* http://msdn.microsoft.com/en-us/library/dd233250.aspx. – [Online; Stand 26. Juni 2010]

[Mich] Microsoft: *F# Language Reference – Code Quotations.* http://msdn.microsoft.com/en-us/library/dd233212.aspx. – [Online; Stand 26. Juni 2010]

[Mici] Microsoft: *F# Language Reference – Discriminated Unions.* http://msdn.microsoft.com/en-us/library/dd233226.aspx. – [Online; Stand 26. Juni 2010]

[Micj] Microsoft: *F# Language Reference – F# Types.* http://msdn.microsoft.com/en-us/library/dd233230.aspx. – [Online; Stand 26. Juni 2010]

[Mick] Microsoft: *F# Language Reference – Object Expressions.* http://msdn.microsoft.com/en-us/library/dd233237.aspx. – [Online; Stand 26. Juni 2010]

[Micl] Microsoft: *F# Language Reference – Operator Overloading.* http://msdn.microsoft.com/en-us/library/dd233204.aspx. – [Online; Stand 26. Juni 2010]

[Micm] Microsoft: *F# Language Reference – Pattern Matching.* http://msdn.microsoft.com/en-us/library/dd233181(v=VS.100).aspx. – [Online; Stand 26. Juni 2010]

[Micn] Microsoft: *F# Language Reference – Records.* http://msdn.microsoft.com/en-us/library/dd233184.aspx. – [Online; Stand 26. Juni 2010]

[Mico] Microsoft: *F# Language Reference – Reference Cells.* http://msdn.microsoft.com/en-us/library/dd233186.aspx. – [Online; Stand 26. Juni 2010]

[Micp] Microsoft: *F# Language Reference – Type Extensions.* http://msdn.microsoft.com/en-us/library/dd233211.aspx. – [Online; Stand 26. Juni 2010]

[Micq] Microsoft: *F# Language Reference – Units of Measure.* http://msdn.microsoft.com/en-us/library/dd233243.aspx. – [Online; Stand 26. Juni 2010]

[Micr] Microsoft: *The F# PowerPack.* http://fsharppowerpack.codeplex.com/. – [Online; Stand 26. Juni 2010]

[Mics] Microsoft: *Parallel LINQ (PLINQ).* http://msdn.microsoft.com/en-us/library/dd460688.aspx. – [Online; Stand 26. Juni 2010]

[Mict] Microsoft: *Parallel Programming with Microsoft .NET – Parallel Loops.* http://msdn.microsoft.com/en-us/library/ff963552.aspx. – [Online; Stand 26. Juni 2010]

[Micu] Microsoft: *Parallel Programming with Microsoft .NET – Pipelines.* http://msdn.microsoft.com/en-us/library/ff963548.aspx. – [Online; Stand 26. Juni 2010]

[Micv] Microsoft: *Reactive Extensions for .NET (Rx).* http://msdn.microsoft.com/en-us/devlabs/ee794896.aspx. – [Online; Stand 12. Oktober 2010]

[Micw] Microsoft: *Task Parallel Library*. http://msdn.microsoft.com/en-us/library/dd460717.aspx. – [Online; Stand 26. Juni 2010]

[Mic10] Microsoft: *F# Language Reference – Computation Expressions*. http://msdn.microsoft.com/en-us/library/dd233182.aspx. Version: 2010. – [Online; Stand 26. Juni 2010]

[Ng10] Ng, Tim: *F# Quick Guides: Object Oriented Programming*. http://blogs.msdn.com/b/timng/archive/2010/04/05/f-object-oriented-programming-quick-guide.aspx. Version: 04 2010. – [Online; Stand 14. September 2010]

[PH06] Pepper, Peter ; Hofstedt, Petra: *Funktionale Programmierung*. Springer, 2006

[Pic09] Pickering, Robert: *Beginning F#*. Apress, 2009. – alias Foundations of F# 2nd Ed.

[Sco06] Scott, Michael L.: *Programming language pragmatics*. 2. Elsevier, 2006

[SGC10] Syme, Don ; Granicz, Adam ; Cisternino, Antonio: *Expert F# 2.0*. Apress, 2010

[Smi10] Smith, Chris: *Programming F#*. O'Reilly, 2010

[Sym10] Syme, Don: *Async and Parallel Design Patterns in F#: Parallelizing CPU and I/O Computations*. http://blogs.msdn.com/b/dsyme/archive/2010/01/09/async-and-parallel-design-patterns-in-f-parallelizing-cpu-and-i-o-computations.aspx. Version: 1 2010. – [Online; Stand 26. Juni 2010]

[TPO06] Tarditi, David ; Puri, Sidd ; Oglesby, Jose: Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. (2006). – http://research.microsoft.com/apps/pubs/default.aspx?id=70250 [Online; Stand 23. Oktober 2010]

[Wad92] Wadler, Philip: Comprehending Monads. (1992). – http://homepages.inf.ed.ac.uk/wadler/topics/monads.html [Online; Stand 23. Oktober 2010]

[Wad95] Wadler, Philip: Monads for functional programming. (1995). – http://homepages.inf.ed.ac.uk/wadler/topics/monads.html [Online; Stand 23. Oktober 2010]

[Yor09] Yorge, Brent: The Typeclassopedia. In: *The Monad.Reader* Issue 13 (2009), S. 17–68. – http://www.haskell.org/haskellwiki/The_Monad.Reader/Previous_issues [Online; Stand 23. Oktober 2010]

Abbildungsverzeichnis

Tabellenverzeichnis

Formelverzeichnis

Programmcodeverzeichnis

Glossar

BV
Bildverarbeitung. 2, 30–32, 37

FP
Functional Programming - Funktionale Programmierung. 9–11, 14, 19, 34, 37

FPGA
Field Programmable Gate Array - programmierbarer Integrierter Schaltkreis. 26

FSI
F# Interactive - interaktive Umgebung zur Ausführung von F#-Code (siehe REPL). 22, 71

GPGPU
General-Purpose computation on GPUs - Verwendung von Grafikprozessoren für allgemeine Berechnungen. 1

GPU
Graphics Processing Unit - Grafikprozessor. 2, 24–26, 30, 32, 37

LISP
List Processing - Familie von Programmiersprachen, ursprünglich 1958 von John McCarthy entwickelt. 3, 9, 22

REPL
read-eval-print loop - interaktive Umgebung ähnlich einer Konsole zur Auswertung von programmatischen Ausdrücken. 22

SE
strukturierendes Element, auch Strukturelement (morphologische Bildverarbeitung). 33

SIMD
Single Instruction, Multiple Data - Rechnerarchitektur zur gleichzeitigen Ausführung von Operationen. 24, 26

TPL
Task Parallel Library - Teil des .NET Frameworks Version 4. 24

VS
Visual Studio - in der Regel ist Visual Studio 2010 gemeint. 22, 71

A Anhang

A.1 Programmcodeauflistungen

A.1.1 Beispiele

```
49 // Typ kapselt aktuelles Ergebnis und Messungen
50 type 'a TimedResult = TR of ((int64 * string) list * 'a)
51
52 // misst die Funktion "f"
53 let clock (f : unit -> 'a) (label : string) : 'a TimedResult =
54     let stw = new System.Diagnostics.Stopwatch ()
55     stw.Start ()
56     let result = f () // "f" ausfuehren
57     stw.Stop ()
58     // Messungen und Ergebnis in "TimedResult"-Typ zusammenfuegen
59     // (Messungen als Kontext des Ergebniswertes)
60     TR ([stw.ElapsedMilliseconds, label], result)
61
62 // Definition des Workflow-Typs
63 type TimedBuilder() =
64     member self.Return value =
65         TR ([], value)
66     // Tupel-Argument (kein Currying) aus vorherigem Ergbnis und naechster
        Funktion
67     // vorherige Ergbnis wird durch "TimedResult"-Datenkonstruktor "TR"
        gematched/dekonstruiert
68     member self.Bind ( (TR (m, value)) : 'a TimedResult, (f : 'a -> 'b
        TimedResult) ) : 'b TimedResult =
69         // Ausfuehrung der Funktion erzeugt neue Messung und Wert
70         // neue und alte Messung konkatenieren und wieder als "TimedResult"
            zurueckgeben
71         match (f value) with
72         | TR (nextM, nextValue) -> TR (nextM @ m, nextValue) // "@" ist
            Konkatenationsoperator
73
74 // Workflow-Instanz erzeugen
75 let timed = new TimedBuilder ()
76
77 let messungen : float TimedResult =
78     timed { // Workflow-Instanz anwenden
79         let f v = // eine willkuerliche Funktion
80             List.fold (fun p q -> q - p) 0.0
81                 (List.map (fun x -> x ** v) [1.0 .. 100000.0])
82         // Die Lambda-Ausdruecke verzoegern die Ausfuehrung von "f", indem
```

```
83        // erst noch das Argument "()" (unit) uebergeben werden muss.
84        // So wird die Funktion erst innerhalb von "clock" ausgefuehrt.
85        let! x = clock (fun () -> f 5.0) "A"
86        let! x = clock (fun () -> f 7.0) "B"
87        let! x = clock (fun () -> f 11.0) "C"
88        let! x = clock (fun () -> f 13.0) "D"
89        let! x = clock (fun () -> f 17.0) "E"
90        let! x = clock (fun () -> f 19.0) "F"
91        let! x = clock (fun () -> f 23.0) "G"
92        // Jedes "x" ist abwaerts eine neue, tiefer geschachtelte Variable.
93        // So entsteht der Eindruck eines imperativen Programms.
94        return x // Rueckgabe des letzten (tiefesten) "x" als Ergebniswert
95        }
```

Programmcode A.1: Beispiel *timed*-Workflow

```
messungen : float TimedResult =
  TR
    ([(53L, "G"); (43L, "F"); (46L, "E"); (40L, "D"); (38L, "C"); (43L, "B");
      (41L, "A")], 5.000575e+114)
```

Abbildung A.1: mögliches Ergebnis des *timed*-Workflows

```
99  // Definition asynchroner Elementarroutinen
100 let asyncf1 : Async<int> = async { return 42 }
101 let asyncf2 : Async<int> = async { return 23 }
102 let asyncf3 : Async<unit> = async { return () }
103
104 // Verkettung der Funktionen in Workflows
105
106 let asyncWF1 =
107     async {
108         let! v1 = asyncf1
109         let! v2 = asyncf2
110         do! asyncf3
111         printfn "%d" v1
112         printfn "%d" v2
113     }
114 asyncWF1 |> Async.StartImmediate // im aktuellen Thread starten
115
116 let asyncWF2 =
117     async {
118         let! v1 = asyncf1
119         let! v2 = asyncf2
120         do! asyncf3
121         return v1 + v2
122     }
123 let comp = asyncWF2 |> Async.StartAsTask // als Task im Thread-Pool starten
124 while not comp.IsCompleted do
```

```
125     ()
126 printfn "%d" comp.Result
```

Programmcode A.2: Beispiel *async*-Workflow

A.1.2 Demonstrationssoftware

```
module Playground.Helper

// 2D-Array als Sequenz (lazy)
let flatten (a : 'T [,]) =
    let rs, cs = Array2D.length1 a, Array2D.length2 a
    seq {
        for r in 0 .. rs - 1 do
            for c in 0 .. cs - 1 do
                yield a.[r,c] }

// 2D-Array als Sequenz von 3-Tupeln (mit Index Reihe/Spalte)
let flatteni (a : 'T [,]) =
    let rs, cs = Array2D.length1 a, Array2D.length2 a
    seq {
        for r in 0 .. rs - 1 do
            for c in 0 .. cs - 1 do
                yield r,c,a.[r,c] }
```

Programmcode A.3: Modul `Helper`

```
module Playground.Accelerator

open Microsoft.ParallelArrays

// kuerzere Typ-Synonyme
type PA = ParallelArrays
type FPA = Microsoft.ParallelArrays.FloatParallelArray
type BPA = Microsoft.ParallelArrays.BoolParallelArray
type IPA = Microsoft.ParallelArrays.IntParallelArray

// 2D-FPA mit einzelner Zahl mit entsprechender Groesse (shape) erstellen
// (3 Varianten fuer int, double, single)
let mkFPAi (n : int) (shape : int []) : FPA =
    new FPA (Array2D.create shape.[0] shape.[1] (float32 n))
let mkFPAf (n : float) (shape : int []) : FPA =
    new FPA (Array2D.create shape.[0] shape.[1] (float32 n))
let mkFPA (n : float32) (shape : int []) : FPA =
    new FPA (Array2D.create shape.[0] shape.[1] n)

// haeufig verwendet
let mkWhite = mkFPAi 255
let mkBlack = mkFPAi 0
let mkZeros = mkBlack // synonym
let mkOnes = mkFPAi 1

// wiederholte Anwendung einer Funktion auf ein Bild
let rec apply (n : int) (f : (FPA -> FPA)) (image : FPA) : FPA =
    match n with
```

```
29      | _ when n <= 0 -> image
30      | _ -> apply (n-1) f (f image)
31
32  // nuetzliche Schiebeoperationen
33
34  let shift (si : int) (sj : int) (image : FPA) : FPA =
35      PA.Shift (image, [|si; sj|]) // def.: A(i-si),(j-sj)
36
37  let shiftBPA (si : int) (sj : int) (image : BPA) : BPA =
38      PA.Shift (image, [|si; sj|]) // def.: A(i-si),(j-sj)
39
40  let shiftD : (FPA -> FPA) = shift 1 0
41  let shiftU : (FPA -> FPA) = shift -1 0
42  let shiftL : (FPA -> FPA) = shift 0 1
43  let shiftR : (FPA -> FPA) = shift 0 -1
44
45  let shiftDL : (FPA -> FPA) = shift 1 1
46  let shiftUL : (FPA -> FPA) = shift -1 1
47  let shiftDR : (FPA -> FPA) = shift 1 -1
48  let shiftUR : (FPA -> FPA) = shift -1 -1
49
50  // nuetzliche Operatoren
51
52  let (&&.) (a : FPA) (b : FPA) = PA.Min (a, b)
53  let (||.) (a : FPA) (b : FPA) = PA.Max (a, b)
54
55  let (&&&) (a : BPA) (b : BPA) = PA.And (a, b)
56  let (|||) (a : BPA) (b : BPA) = PA.Or (a, b)
57  let (!!!) (a : BPA) = PA.Not a
58  let (---) (a : BPA) (b : BPA) = a &&& !!!b
59  let (>=<) (a : BPA) (b : BPA) = (a ||| b) &&& !!!(a &&& b) // xor
60  let (<=>) (a : BPA) (b : BPA) = (a &&& b) ||| !!!(a ||| b) // xnor, equiv
61
62  let (=?) (a : FPA) (b : FPA) : BPA = PA.CompareEqual (a, b)
63  let (>?) (a : FPA) (b : FPA) : BPA = PA.CompareGreater (a, b)
64  let (>=?) (a : FPA) (b : FPA) : BPA = PA.CompareGreaterEqual (a, b)
65  let (<?) (a : FPA) (b : FPA) : BPA = PA.CompareLess (a, b)
66  let (<=?) (a : FPA) (b : FPA) : BPA = PA.CompareLessEqual (a, b)
67  let (<>?) (a : FPA) (b : FPA) : BPA = PA.CompareNotEqual (a, b)
68
69  let (<?>) (b : BPA) (f : FPA * FPA) : FPA = PA.Cond (b, fst f, snd f) // wie
        ternaere Operator
70  let (<~>) (a : FPA) (f : FPA * FPA) : FPA = PA.Select (a, fst f, snd f) //
        Schlange wie *S*elect
71
72  // Reduzierung von BPA um eine Dimension durch
73  // Verkettung (der Spalten bei 2D) mit entsprechender boolschen Operation
74  let all (a : BPA) : BPA = PA.All a
75  let any (a : BPA) : BPA = PA.Any a
```

```
76
77 // Arrays I, J mit korrespondierenden Indexen als Werte
78 // Beispiel 2x2 Array:
79 // mkIndexArraysRange 0 1 0 1;;
80 // I: [[0; 0]
81 // [1; 1]]
82 // J: [[0; 1]
83 // [0; 1]]
84 let mkIndexArraysRange fromRow toRow fromCol toCol =
85     let rs = toRow - fromRow + 1
86     let cs = toCol - fromCol + 1
87     let is = Array2D.init rs cs (fun i _ -> i + fromRow)
88     let js = Array2D.init rs cs (fun _ j -> j + fromCol)
89     [| new IPA (is); new IPA (js) |]
90
91 let mkIndexArrays (width : int) (height : int) =
92     let is = Array2D.init height width (fun i _ -> float32 i)
93     let js = Array2D.init height width (fun _ j -> float32 j)
94     new FPA (is), new FPA (is)
95
96 let cropFPA fromRow toRow fromCol toCol (paa : FPA) =
97     PA.Gather (paa, mkIndexArraysRange fromRow toRow fromCol toCol)
98
99 let capFPA (image : FPA) : FPA =
100     let max = mkWhite image.Shape
101     image &&. max
102
103 let powFPA (exp : int) (pa : FPA) =
104     PA.Pow (pa, mkFPAi exp pa.Shape)
105
106 let sqrtFPA (pa : FPA) =
107     PA.Sqrt pa
108
109 let tanFPA (pa : FPA) =
110     PA.Sin pa / PA.Cos pa
111
112 // Liste von String-Kommandos mit Hilfe der Map in _eine_ Funktion umwandeln (
    FPA -> FPA)
113 let mkImProcFpaFunc (cmdMap : Map<string, FPA -> FPA>) (cmds : string list) :
    FPA -> FPA =
114     List.map (fun cmd -> Map.find cmd cmdMap) cmds
115     |> List.fold (>>) id
```

Programmcode A.4: Modul `Accelerator`

```
1 module Playground.Representation
2
3 open Playground.Accelerator
4 open System // Exceptions
5 open System.IO // MemoryStream
```

```
6  open System.Drawing // Bitmap
7  open System.Drawing.Imaging // PixelFormat
8  open Microsoft.ParallelArrays
9  // PSeq, schoen um daten-parallel konvertieren zu koennen aber schwierig mit 2D
       -Arrays (via flatten?)
10 //open Microsoft.FSharp.Collections
11 //open Playground.Helper
12
13 type Red = R of byte
14 type Green = G of byte
15 type Blue = B of byte
16 type Alpha = A of byte
17 type Hue = H of byte
18 type Saturation = S of byte
19 type Intensity = I of byte
20
21 type PixelRGB = RGB of Red * Green * Blue
22 type PixelHSI = HSI of Hue * Saturation * Intensity
23
24 type ImageGrey = byte [,]
25 type ImageGreyF = float32 [,] // wie ImageGrey nur als floats fuer Accelerator
26 type ImageRGB = PixelRGB [,]
27
28 // "type extensions"
29 type PixelRGB with
30     member self.RedByte =
31         match self with
32         | RGB(R r, _, _) -> r
33     member self.GreenByte =
34         match self with
35         | RGB(_, G g, _) -> g
36     member self.BlueByte =
37         match self with
38         | RGB(_, _, B b) -> b
39     member self.ToGrey =
40         match self with
41         | RGB (R r, G g, B b) -> byte (0.3 * float r + 0.59 * float g + 0.11 *
               float b)
42
43 // BMP:
44 // - erste 54 bytes sind headerdaten (0..53)
45 // - width = byte 18-22
46 // - height = byte 22-26
47 // - pixel jeweils 4 bytes: BGRA in .NET default (PixelFormat.Format32bppArgb)
48 let BMP_OFFS = 54
49 let N_CH = 4 // Format32bppArgb
50
51 let bitmap_from_file (filePath : string) =
52     use bm = new Bitmap (filePath)
```

```
53      // return copy to unlock file
54      // (apparently Format32bppArgb = standard pixel format for new bitmaps)
55      new Bitmap (bm)
56
57  let bitmap_from_bytes (bs : byte[]) =
58      let ms = new MemoryStream (bs)
59      new Bitmap (ms)
60
61  // Pixel-Bytes aus Bitmap extrahieren
62  let bytes_from_bitmap (bm : Bitmap) =
63      match bm.PixelFormat with
64      | PixelFormat.Format32bppArgb -> ()
65      | _ -> raise <| ArgumentException("Pixel␣format␣not␣supported.␣Need␣
            PixelFormat.Format32bppArgb.")
66      use ms = new MemoryStream ()
67      bm.Save (ms, ImageFormat.Bmp)
68      ms.GetBuffer ()
69
70  // 1D-Byte-Array mit Konvertierungsfunktion in 2D-Array umwandeln
71  let parse_from_bytes w h (bs : byte[]) (f : int -> byte [] -> 'T) : 'T [,] =
72      let ps : 'T [,] = Array2D.zeroCreate h w
73      let lastPixel = BMP_OFFS + N_CH * w * h - 1
74      let bs' = bs.[BMP_OFFS .. lastPixel] // nur die Bild-Bytes ohne Header
75      let refpos = ref 0
76      // ueber das Zielarray iterieren und jedes Element mit uebergebener
            Funktion berechnen
77      Array2D.iteri (fun r c _ ->
78          let pos = !refpos // Positions eines Pixels (4 Bytes)
79          ps.[r,c] <- f pos bs' // Berechnung des aktuellen Elements im Zielarray
80          refpos := pos + N_CH) ps
81      ps
82
83  // 2D-Array mit "PixelRGB"-Elementen aus 1D-Byte-Array
84  let rgb_from_bytes w h (bs : byte[]) =
85      parse_from_bytes w h bs (fun (i : int) bs' -> // Konvertierungsfunktion als
            Lambda
86          RGB (R bs'.[i+2], G bs'.[i+1], B bs'.[i])) // 'out of memory' bei sehr
               grossen Bildern
87
88  // 2D-Array mit byte-Elementen aus 1D-Byte-Array
89  let grey_from_bytes w h (bs : byte[]) =
90      parse_from_bytes w h bs (fun (i : int) bs' -> // Konvertierungsfunktion als
            Lambda
91          byte (0.3 * float bs'.[i+2] + 0.59 * float bs'.[i+1] + 0.11 * float bs
               '.[i]))
92
93  let greyf_from_grey (grey : ImageGrey) : ImageGreyF =
94      Array2D.map (fun b -> float32 b) grey
95
```

```
96  // 2D-Array mit Konvertierungsfunktion wieder in 1D-Byte-Array umwandeln
97  let parse_to_bytes (ps : 'T [,]) (f : 'T -> int -> byte [] -> unit) : byte [] =
98      let (rs, cs) = (Array2D.length1 ps, Array2D.length2 ps)
99      let bs : byte [] = Array.zeroCreate (BMP_OFFS + rs * cs * N_CH)
100     // Dummy-Header erzeugen (PixelFormat.Format32bppArgb)
101     let bmpHeader : byte [] = (bytes_from_bitmap (new Bitmap (cs, rs))).[..
            BMP_OFFS-1]
102     // Dummy-Header schreiben
103     Array.Parallel.iteri (fun i b -> bs.[i] <- b) bmpHeader
104     // es folgen die Pixel-Bytes, iterieren ueber Eingabearray ps
105     let refpos = ref BMP_OFFS
106     Array2D.iter (fun p ->
107         let pos = !refpos // Positions eines Pixels (4 Bytes)
108         f p pos bs // Funktion setzt den aktuelle Pixel p als Bytes im Zielarray
                bs
109         refpos := pos + N_CH) ps
110     bs
111
112 // 2D-Array mit PixelRGB Elementen in 1D-Byte-Array umwandeln
113 let bytes_from_rgb (ps : PixelRGB [,]) =
114     parse_to_bytes ps (fun p pos bs ->
115         bs.[pos] <- p.BlueByte
116         bs.[pos+1] <- p.GreenByte
117         bs.[pos+2] <- p.RedByte
118         // ignore alpha
119         )
120
121 // 2D-Array mit Byte-Elementen in 1D-Byte-Array umwandeln
122 let bytes_from_grey (ps : ImageGrey) =
123     parse_to_bytes ps (fun p pos bs ->
124         bs.[pos] <- p // Byte direkt setzen
125         bs.[pos+1] <- p
126         bs.[pos+2] <- p
127         // ignore alpha
128         )
129
130 let greyf_from_fpa (target : Target) (image : FPA) : ImageGreyF =
131     target.ToArray2D image
132
133 let fpa_from_gref (image : ImageGreyF) : FPA =
134     new FPA (image) // out of memory bei sehr grossen Bildern
135
136 // Histogramm als Bitmap
137 let bitmap_from_hist (hist : int []) : Bitmap =
138     let height = 150
139     let height' = float (height - 1) // fuer Normierung
140     let max = Array.max hist
141     let bm = new Bitmap (256, height)
142     for k in 0 .. 255 do // k = Key, Helligkeit als Schluessel fuer Histogramm
```

```
143         let v = hist.[k] // v = Value, Anzahl
144         if v > 0 then
145             let mutable lastY = -1
146             for v' in 0 .. v - 1 do // Pixel 0 bis Anzahl im Bitmap zeichnen (
                    Balken)
147                 // normieren
148                 let y = int ((float v') * (height' / float max)) + 1
149                 if y <> lastY then
150                     bm.SetPixel (k, height - y, Color.Aquamarine)
151                     lastY <- y
152     bm
```

Programmcode A.5: Modul `Representation`

```
1  module Playground.BV
2
3  open Playground.Representation
4  open System // exceptions, Math
5
6  let delta = 0.00001f
7
8  let Pi = Math.PI
9  let Pi2 = Pi / 2.
10 let Pi4 = Pi / 4.
11 let Pi8 = Pi / 8.
12
13 let sobelY = matrix [[-1.; 0.; 1.];
14                     [-2.; 0.; 2.];
15                     [-1.; 0.; 1.]]
16
17 let sobelX = matrix [[-1.; -2.; -1.];
18                     [ 0.; 0.; 0.];
19                     [ 1.; 2.; 1.]]
20
21 let scharrY = matrix [[ -3.; 0.; 3.];
22                      [-10.; 0.; 10.];
23                      [ -3.; 0.; 3.]]
24
25 let scharrX = matrix [[-3.; -10.; -3.];
26                      [ 0.; 0.; 0.];
27                      [ 3.; 10.; 3.]]
28
29 // erstellt Gauss-Maske anhand gewuenschter Groesse und Sigma
30 let mkGauss (size : int) (sigma : float) : matrix =
31     let c = floor (float size / 2.) + 1.
32     let sum = ref 0.
33     let templ =
34         Array2D.mapi (fun i j _ ->
35             let i', j' = float (i + 1), float (j + 1)
36             let tval = exp -( ((j' - c) ** 2. + (i' - c) ** 2.) / (2. * sigma **
```

```
                2.) )
37            sum := !sum + tval
38            tval) (Array2D.zeroCreate size size)
39    Matrix.ofArray2D (Array2D.map (fun i -> i / !sum) templ)
40
41 // haeufig benutzte Gauss-Masken
42 let gauss3 = mkGauss 3 0.5
43 let gauss5 = mkGauss 5 1.0
44
45 // Konvertierung zwischen Grad und Bogenmass
46 let deg_from_rad r = r * (180. / Pi)
47 let rad_from_deg d = Pi * d / 180.
48
49 // strukturierende Elemente fuer morphologische Operationen
50 let block8 = Array2D.create 3 3 true
51 // TODO: alle 4 Rotationen, "don't care conditions" (HMM!)
52 let block2BL = Array2D.create 3 3 false // 2-block bottom-left
53 block2BL.[1,0] <- true
54 block2BL.[1,2] <- true
55 block2BL.[2,0] <- true
56 block2BL.[2,2] <- true
57 let teeT = Array2D.create 3 3 false // Tetris-T in "Top row"
58 teeT.[0,0] <- true
59 teeT.[0,1] <- true
60 teeT.[0,2] <- true
61 teeT.[1,1] <- true
62 let brickL = Array2D.create 3 3 false // vertikaler Balken links
63 brickL.[0,0] <- true
64 brickL.[1,0] <- true
65 brickL.[2,0] <- true
66
67 // naiver, einfacher Schwellwert
68 let thresh (thresh : byte) (image : ImageGrey) : ImageGrey =
69     Array2D.iteri (fun r c p ->
70         if p <= thresh then
71             image.[r,c] <- byte 0
72         else
73             image.[r,c] <- byte 255) image
74     image
75
76 // Histogrammerstellung
77 let mkHistF (image : ImageGreyF) : int [] * float [] =
78     let len = float image.Length
79     let hist : int [] = Array.create 256 0
80     let histn : float [] = Array.create 256 0.
81     Array2D.iter (fun i ->
82         hist.[int i] <- (hist.[int i]) + 1) image
83     Array.iteri (fun i c ->
84         histn.[i] <- float c / len) hist
```

```
85      hist, histn // absolute und normalisiert
86
87  let mkHist (image : ImageGrey) : int [] * float [] =
88      let len = float image.Length
89      let hist : int [] = Array.create 256 0
90      let histn : float [] = Array.create 256 0.
91      Array2D.iter (fun i ->
92          hist.[int i] <- (hist.[int i]) + 1) image
93      Array.iteri (fun i c ->
94          histn.[i] <- float c / len) hist
95      hist, histn
96
97
98  // Faltung eines Pixels mit einer Maske
99  let convolve (kernel : matrix) (image : ImageGrey) rs cs r c : float =
100     let refacc = ref 0.
101     Array2D.iteri (fun i j k ->
102         let (nr',nc') = (r - 1 + i, c - 1 + j) // Nachbar
103         // 'out of index' Nachbarn abfangen
104         let nr =
105             match nr' with
106             | _ when nr' >= rs -> rs - 1
107             | _ when nr' < 0 -> 0
108             | _ -> nr'
109         let nc =
110             match nc' with
111             | _ when nc' >= cs -> cs - 1
112             | _ when nc' < 0 -> 0
113             | _ -> nc'
114         let p = image.[nr,nc]
115         refacc := !refacc + float p * k) (kernel.ToArray2D())
116     let acc = !refacc
117     acc
118
119 // Sobel-Operation
120 let sobel (image : ImageGrey) : ImageGrey =
121     let rs, cs = Array2D.length1 image, Array2D.length2 image
122     let imOut : ImageGrey = Array2D.zeroCreate rs cs
123     // ueber Bildpixel iterieren
124     Array2D.iteri (fun r c _ ->
125         let gx = convolve sobelX image rs cs r c // Gradient in X-Richtung
126         let gy = convolve sobelY image rs cs r c // Gradient in Y-Richtung
127         let p : byte =
128             let s = sqrt(gx ** 2. + gy ** 2.)
129             match s with // Wertgrenzen deckeln
130             | _ when s >= 255. -> byte 255
131             | _ when s < 0. -> byte 0
132             | _ -> byte s // sonst
133         imOut.[r,c] <- p
```

```
134         ) image
135     imOut
```

Programmcode A.6: Modul BV

```
1  module Playground.BVaccelerator
2
3  open Playground.Helper
4  open Playground.Representation
5  open Playground.Accelerator
6  open Playground.BV
7  open Microsoft.ParallelArrays
8  open System
9
10 // Region einer 8er-Nachbarschaft als Sequenz von entsprechenden
       Schiebeoperationen (inkl. Zentrum)
11 let region8 (dist : int) (image : FPA) : seq<FPA> = seq {
12     for r in -dist .. dist do
13         for c in -dist .. dist do
14             yield shift r c image }
15
16 // 8er-Nachbarschaft als Sequenz von entsprechenden Schiebeoperationen (ohne
       Zentrum)
17 let neighb8 (dist : int) (image : FPA) : seq<FPA> = seq {
18     for r in -dist .. dist do
19         for c in -dist .. dist do
20             if r <> 0 || c <> 0 then
21                 yield shift r c image }
22
23 let neighb8BPA (dist : int) (image : BPA) = seq {
24     for r in -dist .. dist do
25         for c in -dist .. dist do
26             if r <> 0 || c <> 0 then
27                 yield shiftBPA r c image }
28
29 // Sequenz von Schiebeoperationen anhand eines strukturienden Elements
30 let shiftToBPA (se : bool [,]) (image : BPA) =
31     let rs, cs = Array2D.length1 se, Array2D.length2 se
32     // Liste mit 3-Tupel aus Reihe, Spalte, Wert
33     let seListAbs = Seq.toList (flatteni se)
34     // Liste zu relativen Positionen wandeln, so dass:
35     // b.[0,0] true -> Identitaet des Bildes (shift um 0)
36     // b.[0,c] true -> Bild um c Spalten schieben (negative/positiv -> nach
           links/rechts)
37     // b.[r,0] true -> Bild um r Reihen schieben (negative/positiv -> nach oben
           /unten)
38     let seListRel = List.map (fun (r,c,v) -> (r - (rs/2), c - (cs/2), v))
           seListAbs
39     // Schiebeoperationen vornehmen und als Sequenz von FPAs sammeln
40     let rec doshiftto (rel : (int * int * bool) list) (ns : BPA list) =
```

```
41          // relative Position
42          match rel with
43          | [] -> ns // Liste ns (Nachbarn) zurueckgeben
44          | (r,c,v) :: tail ->
45              if v then // Boolscher Wert gibt an, ob dieser Nachbar inkludiert
                    wird
46                  // Bild zu aktuellem Index schieben und in Sequenz einfuegen
47                  doshiftto tail ((shiftBPA -r -c image) :: ns)
48              else
49                  doshiftto tail ns
50      doshiftto seListRel []
51
52  // uebersetzen des SE erst in Schiebeoperationen, dann in eine Liste von
        Nachbarbildern
53  let shiftToFPA (se : bool [,]) (image : FPA) =
54      let rs, cs = Array2D.length1 se, Array2D.length2 se
55      let seListRel = List.map (fun (r,c,v) -> (r - (rs/2), c - (cs/2), v)) (Seq.
            toList (flatteni se))
56      let rec doshiftto (rel : (int * int * bool) list) (ns : FPA list) =
57          match rel with
58          | [] -> ns
59          | (r,c,v) :: tail -> if v then doshiftto tail ((shift -r -c image) ::
                ns) else doshiftto tail ns
60      doshiftto seListRel []
61
62  // Sobel-Faltung in Y-Richtung
63  let conv_sobel_3x3_y (image : FPA) : FPA =
64      let col = (shiftD image) + 2.f * image + (shiftU image) // (1 2 1)^T * I
65      (shiftL col) - (shiftR col) // [-1 0 1]
66
67  // Sobel-Faltung in X-Richtung
68  let conv_sobel_3x3_x (image : FPA) : FPA =
69      let row = (shiftR image) + 2.f * image + (shiftL image) // (1 2 1) * I
70      (shiftU row) - (shiftD row) // [-1 0 1]^T
71
72  let sobelFPA (image : FPA) : FPA =
73      // benutzen die Separierbarkeit der Sobel-Masken -> weniger
            Schiebeoperationen als mit "region8"-Funktion
74      let gx = conv_sobel_3x3_x image
75      let gy = conv_sobel_3x3_y image
76      sqrtFPA ((powFPA 2 gx) + (powFPA 2 gy)) // Intensitaet berechnen (intensity
            aka. magnitude)
77      |> capFPA
78
79  // einfacher Schwellwert
80  let threshFPA (thresh : float32) (image : FPA) : FPA =
81      image - (thresh + delta) <~> (mkWhite image.Shape, mkBlack image.Shape)
82      // pedantische Berechnung mit "delta", weil weisse Pixel /ueber/ dem
            Schwellwert liegen sollen (Eq. 10.3-1 dip/3e gonzalez/woods)
```

```
83      // PA.Select selektiert schon ab 0, also bei Differenzbildung /auf/ den
            Schwellwert, nicht darueber
84
85  // Erosion und Dilatation mit 4er-Nachbarschaftskreuz
86  // strukturierendes Element: #
87  // ###
88  // #
89
90  let erode4FPA (image : FPA) : FPA =
91      image &&. (shiftL image) &&. (shiftR image) &&. (shiftD image) &&. (shiftU
            image)
92
93  let dilate4FPA (image : FPA) : FPA =
94      image ||. (shiftL image) ||. (shiftR image) ||. (shiftD image) ||. (shiftU
            image)
95
96  // Erosion und Dilatation mit 8er-Nachbarschaftsblock
97  // strukturierendes Element: ###
98  // ###
99  // ###
100
101 let erode8FPA (image : FPA) =
102     let ns = neighb8 1 image
103     Seq.fold (&&.) image ns
104
105 let dilate8FPA (image : FPA) =
106     let ns = neighb8 1 image
107     Seq.fold (||.) image ns
108
109 let erode8BPA (image : BPA) : BPA =
110     let ns = neighb8BPA 1 image
111     Seq.fold (&&&) image ns
112
113 let dilate8BPA (image : BPA) : BPA =
114     let ns = neighb8BPA 1 image
115     Seq.fold (|||) image ns
116
117 // allgemeine Erosion und Dilation
118
119 let erodeWithFPA (b : bool [,]) (image : FPA) : FPA =
120     let ns = shiftToFPA b image
121     List.fold (&&.) image ns
122
123 let dilateWithFPA (b : bool [,]) (image : FPA) : FPA =
124     let ns = shiftToFPA b image
125     List.fold (||.) image ns
126
127 let erodeWithBPA (b : bool [,]) (image : BPA) : BPA =
128     let ns = shiftToBPA b image
```

```
129     List.fold (&&&) image ns
130 
131 let dilateWithBPA (b : bool [,]) (image : BPA) : BPA =
132     let ns = shiftToBPA b image
133     List.fold (|||) image ns
134 
135 // aus Erosion und Dilation abgeleitete morphologische Operationen
136 
137 let open8FPA : (FPA -> FPA) = erode8FPA >> dilate8FPA
138 
139 let close8FPA : (FPA -> FPA) = dilate8FPA >> erode8FPA
140 
141 let bounds8FPA (image : FPA) : FPA =
142     image - (erode8FPA image)
143 
144 let morphGrad8FPA (image : FPA) =
145     (dilate8FPA image) - (erode8FPA image)
146 
147 let gauss3x3FPA (image : FPA) : FPA =
148     let g3 : seq<float32> = Seq.map float32 gauss3 // Maske
149     let r3 : seq<FPA> = region8 1 image // korrespondierende Bildregion (
        Distanz 1)
150     Seq.map2 (fun (fpa : FPA) (s : float32) -> fpa * s) r3 g3 // beide
        Koeffizienten multiplizieren
151     |> Seq.fold (fun (a : FPA) (b : FPA) -> a + b) (mkZeros image.Shape) // und
         aufsummieren
152 
153 let gauss5x5FPA (image : FPA) : FPA =
154     let g5 : seq<float32> = Seq.map float32 gauss5
155     let r5 : seq<FPA> = region8 2 image
156     Seq.map2 (fun (fpa : FPA) (s : float32) -> fpa * s) r5 g5
157     |> Seq.fold (fun (a : FPA) (b : FPA) -> a + b) (mkZeros image.Shape)
158 
159 // Target muss hier ausgeloest werden, eher unschoen weil hin und her auf
    relativ langsamen Bus zwischen GPU und CPU
160 
161 let atanFPA' (target : Target) (pa : FPA) : FPA =
162     let a : float32 [,] = target.ToArray2D(pa)
163     new FPA (Array2D.map (float >> Math.Atan >> float32) a) // alpha -> (-pi/2;
        pi/2)
164 
165 // zusaetzlich zur Sobel-Operation auch korrespondierende Winkel der Normalen
    berechnen
166 let sobelMAlphaFPA (target : Target) (image : FPA) : FPA * FPA =
167     let gx = conv_sobel_3x3_x image
168     let gy = conv_sobel_3x3_y image
169     let M = sqrtFPA ((powFPA 2 gx) + (powFPA 2 gy)) |> capFPA // M wie '
        magnitude'
170     let alpha = atanFPA' target (gy / gx) // Winkel der Kanten-Normalen (
```

```
        Gradient) im Bogenmass
171     M, alpha
172
173 // Non-Maximum Suppression
174 let nonMaxSup (target : Target) (image : FPA) : FPA =
175     let shape = image.Shape
176     let black = mkBlack shape
177
178     let pi = mkFPAf Pi shape
179     let pi2 = mkFPAf Pi2 shape
180     let pi4 = mkFPAf Pi4 shape
181     let pi8 = mkFPAf Pi8 shape
182
183     // Sobeloperation inklusive Berechnung der Normalenwinkel (vorher mit Gauss
            glaetten)
184     let M, alpha = sobelMAlphaFPA target (gauss5x5FPA image)
185
186     // Vorsicht: 0-Grad-Gradienten sind /vertikal/ (also horizontale Kanten im
        Bild)
187     // d.h. der Winkelkreis beginnt senkrecht/unten
188
189     // horizontaler Gradient
190     let maskH = ( (alpha >=? -pi2) &&& (alpha <? -pi2 + pi8) ) ||| ( (alpha >?
        pi2 - pi8) &&& (alpha <=? pi2) )
191
192     // verticaler Gradient
193     let maskV = (alpha >? -pi8) &&& (alpha <? pi8)
194
195     // aufwaerts diagonaler Gradient
196     let maskDD = (alpha >=? pi8) &&& (alpha <=? pi2 - pi8)
197
198     // abwaerts diagonaler Gradient
199     let maskDU = (alpha <=? -pi8) &&& (alpha >=? -pi2 + pi8)
200
201     // Maximum entlang der Kanten-Normalen/des Gradienten-Vektors
202     let maxV = (shiftU M) ||. (shiftD M) ||. M
203     let maxH = (shiftL M) ||. (shiftR M) ||. M
204     let maxDU = (shiftDL M) ||. (shiftUR M) ||. M
205     let maxDD = (shiftUL M) ||. (shiftDR M) ||. M
206
207     // unterdruecken wenn 'M < max'
208     let supV = M - maxV <~> (( maskV <?> (M, black) ), black)
209     let supH = M - maxH <~> (( maskH <?> (M, black) ), black)
210     let supDU = M - maxDU <~> (( maskDU <?> (M, black) ), black)
211     let supDD = M - maxDD <~> (( maskDD <?> (M, black) ), black)
212
213     // Bilder der verschiedenen Normalen-Richtungen wieder zusammenfuegen
214     List.fold (||.) black [supV; supH; supDU; supDD]
```

Programmcode A.7: Modul `BVaccelerator`

```
1  module Playground.UI
2
3  open Playground.Representation
4  open System.Drawing // Bitmap
5  open System.Drawing.Imaging // PixelFormat/ImageFormat
6  open System.IO // MemoryStream
7  open System.Reflection // ressources
8  open System.Windows // Window
9  open System.Windows.Controls // Grid, Label
10 open System.Windows.Media.Imaging // ImageSource > BitmapSource > BitmapImage
11
12 // GUI-Hilfsfunktionen
13
14 // Operator zum extrahieren von WPF-Elementen
15 let (@?) (name : string) (container : Control) : 'T =
16     container.FindName name :?> 'T // down cast
17
18 // Bitmap im uebergebenen Control rendern
19 let render (imgCtrl : Controls.Image) (bm : Bitmap) : unit =
20     // Bitmap als MemoryStream speichern
21     use ms = new MemoryStream ()
22     bm.Save(ms, ImageFormat.Bmp)
23     // verknuepft Control und Bitmap in ms
24     let bmImage = new BitmapImage ()
25     bmImage.BeginInit ()
26     //bmImage.StreamSource <- new MemoryStream (ms.ToArray()) // call by val (
           neues Array)?
27     bmImage.StreamSource <- new MemoryStream (ms.GetBuffer ()) // call by ref (
           Referenz auf Array)?
28     bmImage.EndInit ()
29     imgCtrl.Source <- bmImage
30
31 // Text einem ListBox-log hinzufuegen
32 let log_to (text : string) (log : ListBox) =
33     let logEntry = new ListBoxItem () // neues Listenelement
34     logEntry.Content <- text // Text setzen
35     log.Items.Add logEntry |> ignore // neues Element hinzufuegen
36     log.ScrollIntoView logEntry
37
38 // uebersetzt Listenelemente in eine String-Liste
39 let mkCmds (lb : ListBox) =
40     List.map (fun (i : ListBoxItem) -> i.Content :?> string) // cast zu String
41         [for i in 0 .. lb.Items.Count - 1 ->
42             lb.Items.GetItemAt(i) :?> ListBoxItem] // ListBox-Elemente als Liste
                   sammeln
43
44 // Klassendefinitionen
45
46 // Aufzaehlung von Aktionsmarkern fuer EventArgs
```

```
47 type AppStateAction = BmResChanged
48
49 // EventArgs speichern Event-Daten (z.B. Ergebnis einer Berechnung im
       Hintergrund)
50 type AppStateEventArgs<'a>(value : 'a, action : AppStateAction) =
51     inherit System.EventArgs ()
52     member self.Action = action // Aktionsmarker
53     member self.Value = value // Event-Daten
54
55 // Delegatetyp fuer Event-Quellen
56 type AppStateDelegate<'a> = delegate of obj * AppStateEventArgs<'a> -> unit
57
58 // Klassendefinition, kapselt/speichert den Applikationszustand
59 type AppState(bm, bmRes, grey, greyF, rgb) =
60     let mutable bm : Bitmap = bm
61     let mutable bmRes : Bitmap = bmRes // Ergebnis einer Bildverarbeitung
62     let mutable grey : ImageGrey = grey
63     let mutable greyF : ImageGreyF = greyF
64     let mutable rgb : ImageRGB = rgb
65
66     // Event erzeugen
67     let bmResChanged = new Event<AppStateDelegate<Bitmap>, AppStateEventArgs<
           Bitmap>>()
68
69     // Konstruktor
70     new () =
71         // eingebettete Ressourcen laden
72         let assembly = Assembly.GetExecutingAssembly()
73         let stream = assembly.GetManifestResourceStream("finger.tif") //
               eingebettetes Bild als Startbild
74         let tmp = new Bitmap (new Bitmap (stream)) // sonst Format24bppArgb
               anstatt Format32bppArgb
75         AppState(tmp, tmp, Array2D.zeroCreate 0 0, Array2D.zeroCreate 0 0,
               Array2D.zeroCreate 0 0)
76
77     // Getter/Setter als .NET-Properties
78     member self.GreyF with get () = greyF
79                       and set newGreyF = greyF <- newGreyF
80     member self.Grey with get () = grey
81                       and set newGrey = grey <- newGrey
82     member self.RGB with get () = rgb
83                       and set newRGB = rgb <- newRGB
84     member self.Bm with get () = bm
85                       and set newBm = bm <- newBm
86     member self.BmRes with get () = bmRes
87                       and set newBmRes =
88                             bmRes <- newBmRes
89                             // Event ausloesen
90                             bmResChanged.Trigger (self, new AppStateEventArgs<
```

```
                                Bitmap> (newBmRes, BmResChanged))
91
92      // Events publizieren
93      member self.BmResChangedEvent = bmResChanged.Publish
94
95      // wandelt lokales Bitmap in verwendbare Formate um, direkt nach Laden
            eines neuen Bildes -> Vorberechung, spart Zeit
96      member self.computeImage =
97          let bs = bytes_from_bitmap bm
98          grey <- grey_from_bytes bm.Width bm.Height bs // byte Array
99          greyF <- greyf_from_grey grey // float Array
100         // zurzeit nicht gebraucht, da Operationen nur auf Grauwertbildern
101         //rgb <- rgb_from_bytes bm.Width bm.Height bs
```

Programmcode A.8: Modul `UI`

```
1  module Playground.App
2
3  open Playground.Accelerator
4  open Playground.BV
5  open Playground.BVaccelerator
6  open Playground.Representation
7  open Playground.UI
8  open Microsoft.ParallelArrays
9  open Microsoft.Win32 // OpenFileDialog
10 open System
11 open System.Drawing // Bitmap
12 open System.Concurrency // IScheduler, Scheduler
13 open System.Linq // Observable.ObserveOn (Rx)
14 open System.Reflection // ressources
15 open System.Threading.Tasks
16 open System.Windows // Window
17 open System.Windows.Controls // Grid, Label
18 open System.Windows.Markup //XamlReader
19
20 // XAML-Ressource einlesen und als Window-Control zurueckgeben
21 // Vorsicht: diese Methode /muss hier in App.fs/ stehen. Die selbe in UI.fs
       fuehrt zu Abstuerzen durch 'ListBox'en..
22 let create_window =
23     let load_xaml =
24         let assembly = Assembly.GetExecutingAssembly ()
25         let stream = assembly.GetManifestResourceStream "MainWindow.xaml"
26         XamlReader.Load stream
27     load_xaml :?> Window
28
29 [<EntryPoint; STAThread>]
30 let main(args : string[]) =
31     let target = new DX9Target ()
32     let mainWindow = create_window
33
```

```
34      // GUI Elemente aus XAML extrahieren
35      let imgCtrl : Controls.Image = "imageCtrl" @? mainWindow
36      let miniCtrl : Controls.Image = "miniCtrl" @? mainWindow
37      let histCtrl : Controls.Image = "histCtrl" @? mainWindow
38      let logBox : ListBox = "log" @? mainWindow
39      let pathBox : TextBox = "imagePathBox" @? mainWindow
40      let funcMan : ListBox = "funcMan" @? mainWindow
41      let funcItms : StackPanel = "funcItms" @? mainWindow
42      let btnFPA : Button = "btnFPA" @? mainWindow
43      let btnNoFPA : Button = "btnNoFPA" @? mainWindow
44
45      // beinhaltet die Bildverarbeitungsfunktion
46      let cmdMap =
47          new Map<string, FPA -> FPA>([ ("threshFPA", threshFPA 200.f); //
                einfachheitshalber feste Parametrisierung
48                                        ("sobelFPA", sobelFPA);
49                                        ("nonMaxSup", nonMaxSup target);
50                                        ("erode8FPA", erode8FPA);
51                                        ("erode4FPA", erode4FPA);
52                                        ("erodeWithFPA", erodeWithFPA block8); //
                                            einfachheitshalber feste
                                            Parametrisierung
53                                        ("dilate8FPA", dilate8FPA);
54                                        ("dilate4FPA", dilate4FPA);
55                                        ("open8FPA", open8FPA);
56                                        ("close8FPA", close8FPA);
57                                        ("bounds8FPA", bounds8FPA);
58                                        ("morphGrad8FPA", morphGrad8FPA);
59                                        ("gauss3x3FPA", gauss3x3FPA);
60                                        ("gauss5x5FPA", gauss5x5FPA);
61                                    ])
62
63      let st = new AppState ()
64      // initiales Bild rendern
65      render imgCtrl st.Bm
66      // und vorberechnen
67      st.computeImage
68
69  // Event-Handling
70
71      let wpfDispatcher = Scheduler.Dispatcher // context/dipatcher mit Zugriff
            auf WPF-Elemente
72
73      // wird nach "BmResChangedEvent" ausgefuehrt
74      let bmResChangedCallback = fun (args : AppStateEventArgs<Bitmap>) ->
75          let bm = args.Value // Bitmap aus Event-Daten extrahieren
76          // Miniaturbild erstellen
77          let miniW = 256
78          let miniH = int (float miniW / (float bm.Width / float bm.Height))
```

```
79          let mini = new Bitmap (miniW, miniH)
80          let miniRect = new Rectangle (0, 0, miniW, miniH)
81          use g = Graphics.FromImage(mini)
82          g.SmoothingMode <- Drawing2D.SmoothingMode.AntiAlias
83          g.InterpolationMode <- Drawing2D.InterpolationMode.HighQualityBicubic
84          g.CompositingQuality <- Drawing2D.CompositingQuality.HighQuality
85          g.DrawImage(bm, miniRect, 0, 0, bm.Width, bm.Height, GraphicsUnit.Pixel)

86          // Bitmap und Miniaturbild rendern
87          render imgCtrl bm
88          render miniCtrl mini
89
90      // (Rx) Observer fuer "BmResChangedEvent" auf WPF-Dispatcher erstellen und
            mit eben definiertem Callback verknuepfen
91      let bmResObsr = st.BmResChangedEvent.ObserveOn(wpfDispatcher).Subscribe(
            bmResChangedCallback)
92
93  // GUI-Funktionalitaet
94
95      // oberes Textfeld mit Dateipfad
96      pathBox.MouseDoubleClick.Add(fun _ ->
97          let dlg = new OpenFileDialog ()
98          dlg.Multiselect <- false
99          dlg.Filter <- "images␣(*.jpg;*.png;*.bmp;*.tif;*.gif)|*.jpg;*.png;*.bmp
                ;*.tif;*.gif"
100         dlg.CheckFileExists <- true
101         dlg.CheckPathExists <- true
102         dlg.DereferenceLinks <- true
103         dlg.ValidateNames <- true
104         if dlg.ShowDialog().Value then
105             pathBox.Text <- dlg.FileName
106             st.Bm <- bitmap_from_file dlg.FileName
107             st.Bm |> render imgCtrl
108             st.computeImage)
109
110     // Drag-Drop-Feld zum verknuepfen der einzelnen Accelerator-
            Bildverarbeitungsfunktionen
111     funcMan.Drop.Add(fun evt ->
112         if evt.Data.GetDataPresent(DataFormats.Text) then // wenn Drop-Event
                Textdaten enthaelt
113             evt.Effects <- DragDropEffects.Copy
114             let itm = new ListBoxItem ()
115             itm.Content <- evt.Data.GetData(DataFormats.Text) // Caption-Text
                    extrahieren und in neues Listenelement schreiben
116             funcMan.Items.Add(itm) |> ignore
117             // bei Doppelklick Listenelement entfernen
118             itm.MouseDoubleClick.Add(fun _ -> funcMan.Items.Remove(itm))
119         else
120             evt.Effects <- DragDropEffects.None)
```

```
121
122     // Text der Textbloecke extrahieren und als Drag-Event-Daten benutzen
123     for chld in funcItms.Children do
124         let txBlk = chld :?> TextBlock // Kinder der Listenelemente sind
                Textbloecke
125         txBlk.MouseDown.Add(fun _ -> DragDrop.DoDragDrop(txBlk, txBlk.Text,
                DragDropEffects.Copy) |> ignore)
126
127     // Definition der asynchronen Operation fuer Buttons
128     let imProcFpaAsync (imProcFpaFunc : FPA -> FPA) =
129         async {
130             let stw = new System.Diagnostics.Stopwatch () // (Thread)-lokale
                    Stopuhr
131             // Bildverarbeitungszeit messen
132             stw.Start ()
133             let procRes =
134                 st.GreyF
135                 |> fpa_from_gref
136                 |> imProcFpaFunc // gesamte verknuepfte BV-Funktion
137                 |> greyf_from_fpa target
138             stw.Stop ()
139             // Histogram erstellen
140             let histBm =
141                 procRes
142                 |> mkHistF
143                 |> fst
144                 |> bitmap_from_hist
145             // BV-Ergbnis in Bitmap umwandeln
146             let resBm =
147                 procRes
148                 |> Array2D.map (fun f -> byte f)
149                 |> bytes_from_grey
150                 |> bitmap_from_bytes
151             return resBm, histBm, stw.ElapsedMilliseconds
152         }
153     let imProcAsync (imProcFunc : ImageGrey -> ImageGrey) =
154         async {
155             let stw = new System.Diagnostics.Stopwatch ()
156             stw.Start ()
157             let procRes =
158                 st.Grey
159                 |> imProcFunc
160             stw.Stop ()
161             let histBm =
162                 procRes
163                 |> mkHist
164                 |> fst
165                 |> bitmap_from_hist
166             let resBm =
```

```
167             procRes
168             |> bytes_from_grey
169             |> bitmap_from_bytes
170         return resBm, histBm, stw.ElapsedMilliseconds
171     }
172
173 // Buttons mit definierten asynchronen Operationen verknuepfen
174 btnFPA.Click.Add(fun _ ->
175     // Task-Objekt erstellen und im Hintergrund starten
176     let workerTsk : Task<Bitmap * Bitmap * int64> =
177         let cmdStrings = mkCmds funcMan // String-Kommandos extrahieren
178         let imProcFunc = mkImProcFpaFunc cmdMap cmdStrings // Kommandos in
                BV-Funktion uebersetzen
179         imProcFpaAsync imProcFunc |> Async.StartAsTask // BV-Funktion
                uebergeben
180     // wird nach Beendingung des Tasks ausgefuehrt
181     let workerResCallback = fun (resBm, histBm, timeEl) ->
182         log_to ("FPA␣elapsed:␣" + string timeEl + "ms␣for␣'" + pathBox.Text
                + "'") logBox
183         render histCtrl histBm
184         st.BmRes <- resBm // Ergebnis-Bitmap setzen (loest Event zum rendern
                aus)
185     // Rx (Reactive Extensions):
186     // Task beobachten und mit Callback verknuepfen
187     let workerObsl = TaskObservableExtensions.ToObservable(workerTsk)
188     let workerObsr = workerObsl.ObserveOn(wpfDispatcher).Subscribe(
            workerResCallback)
189     () // unit zurueckgeben
190     )
191 btnNoFPA.Click.Add(fun _ ->
192     let workerTsk : Task<Bitmap * Bitmap * int64> =
193         imProcAsync (sobel) |> Async.StartAsTask // Sobel-Funktion hart
                kodiert
194     let workerResCallback = fun (resBm, histBm, timeEl) ->
195         log_to ("elapsed:␣" + string timeEl + "ms␣for␣'" + pathBox.Text + "'
                ") logBox
196         render histCtrl histBm
197         st.BmRes <- resBm
198     let workerObsbl = TaskObservableExtensions.ToObservable(workerTsk)
199     let workerObsr = workerObsbl.ObserveOn(wpfDispatcher).Subscribe(
            workerResCallback)
200     ()
201     )
202
203 // Anwendung starten
204 let app = new Application ()
205 app.Run mainWindow
```

Programmcode A.9: Modul App

```
-->
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Playground"
        SizeToContent="WidthAndHeight"
        ResizeMode="CanMinimize">
    <!-- Client area (for content) -->
    <StackPanel>
        <TextBox Name="imagePathBox" IsReadOnly="True">
            double click to open file..
        </TextBox>
        <ListBox Name="log" Height="40" />
        <DockPanel Background="Black">
            <StackPanel>
                <DockPanel>
                    <ListBox Name="funcMan" Width="120" Height="400" AllowDrop="
                        True"/>
                    <StackPanel Name="funcItms" Background="WhiteSmoke">
                        <TextBlock Padding="3">gauss3x3FPA</TextBlock>
                        <TextBlock Padding="3">gauss5x5FPA</TextBlock>
                        <TextBlock Padding="3">sobelFPA</TextBlock>
                        <TextBlock Padding="3">nonMaxSup</TextBlock>
                        <TextBlock Padding="3">threshFPA</TextBlock>
                        <TextBlock Padding="3">erode4FPA</TextBlock>
                        <TextBlock Padding="3">dilate4FPA</TextBlock>
                        <TextBlock Padding="3">erode8FPA</TextBlock>
                        <TextBlock Padding="3">dilate8FPA</TextBlock>
                        <TextBlock Padding="3">close8FPA</TextBlock>
                        <TextBlock Padding="3">open8FPA</TextBlock>
                        <TextBlock Padding="3">bounds8FPA</TextBlock>
                        <TextBlock Padding="3">morphGrad8FPA</TextBlock>
                        <TextBlock Padding="3">erodeWithFPA</TextBlock>
                    </StackPanel>
                </DockPanel>
                <DockPanel Width="256">
                    <Button Name="btnFPA" Width="120" Height="23"
                        HorizontalAlignment="Left">proc FPA</Button>
                    <Button Name="btnNoFPA" Width="120" Height="23"
                        HorizontalAlignment="Left">proc (hardcoded)</Button>
                </DockPanel>
                <StackPanel>
                    <Image Name="histCtrl" Stretch="None"/>
                    <Image Name="miniCtrl" Stretch="None"/>
                </StackPanel>
            </StackPanel>
            <Image Name="imageCtrl" Stretch="None" HorizontalAlignment="Left"
                VerticalAlignment="Top"/>
        </DockPanel>
    </StackPanel>
```

```
47 </Window>
48 <!--
```

Programmcode A.10: Oberflächenentwurf `MainWindow.xaml`

A.2 Installation vom beiliegenden Datenträger

Es muss installiert sein:
(Die Dateien in Klammern sind auf dem beiliegenden Datenträger enthalten.)

- Windows 7 und .NET 4
- VS inkl. F#
- Accelerator (`Accelerator.msi`)
- F# PowerPack (`InstallFSharpPowerPack.msi`)
- Reactive Extensions (`Rx_Net4.msi`)

Die `.zip`-Datei enthält das VS-Projekt. Im entpackten Unterordner muss, bevor das Projekt geöffnet wird, ein Ordner `bin/` angelegt werden. In diesen muss jeweils aus dem Installationsverzeichnis von Accelerator der Inhalt der Ordner `../Accelerator v2/bin/Managed/` und `../Accelerator v2/bin/x64/` bzw. `../Accelerator v2/bin/x86/` kopiert werden, sodass `bin/` aus den zwei Unterverzeichnissen `Debug/` und `Release/` besteht, die die entsprechenden `.dll`-Dateien enthalten (`Accelerator.dll` & `Microsoft.Accelerator.dll`).

Das Projekt kann mit VS durch Öffnen der `.fsproj`-Datei geladen werden. Beim Erstellen erscheint gegebenenfalls ein Dialog zum Speichern einer `.sln`-Datei. Diese kann im selben Projektordner gespeichert werden. Außerdem wird VS einige zusätzliche Dateien von selbst anlegen.

Durch Betätigung von `F5` innerhalb von VS sollte die Demonstrationssoftware gestartet werden können.

Um Accelerator mit FSI verwenden zu können, muss `Accelerator.dll` in das Verzeichnis von `Fsi.exe` kopiert werden (`c:/Program Files/Microsoft F#/v4.0/`).

Zusätzlich befindet sich diese Arbeit in elektronischer Form als `.pdf`-Datei und als LaTeX-Quellen auf dem Datenträger.

B Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit zum Thema

Funktionale Programmierung mit F# und Microsoft Accelerator am Beispiel Bildverarbeitung

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Brandenburg/Havel, 24. Oktober 2010

_____

Unterschrift