Eine datenflußorientierte funktionale Programmiersprache für die Echtzeitdatenverarbeitung

A. Knoll, A. Schweikard und M. Freericks

Technische Universität Berlin
Institut für Technische Informatik, FR 2-2
D-1000 Berlin 10

I Einführung

Funktionale Programmiersprachen weisen gegenüber den bekannten imperativen Programmiersprachen, wie beispielsweise den Sprachen der ALGOL-Familie, eine Reihe von Vorteilen auf: Die Abstraktion von Maschineneigenheiten und algorithmischen Details ist wesentlich höher, deshalb kann die Spezifikation eines Programmes wesentlich näher am Problem orientiert sein. Als Folge davon sind in funktionalem Stil geschriebene Programme häufig wesentlich kürzer als ihre imperative Entsprechung; sie sind einfacher zu lesen, besser zu verstehen, einfacher zu verfassen und daher weniger fehleranfällig.

Es hat in letzter Zeit mehrere Ansätze zur Konzeption funktionaler Echtzeitprogrammiersprachen gegeben, die auf spezielle Einsatzfälle abzielen oder für theoretische Untersuchungen gedacht sind. Die Sprachen Lustre[1] und Signal[2] basieren auf der nichtprozeduralen Sprache Lucid[3] und eignen sich für die Programmierung synchroner Systeme. Silage[4] ist eine funktionale Programmiersprache für digitale Signalverarbeitung, die jedoch keine Rekursion zuläßt. Arctic[5] kommt aus dem Bereich der Klangerzeugung bzw. elektronische Musik, Ruth[6] ist datenflußorientiert und bietet die Möglichkeit, Zeiten über während der Laufzeit auszuwertende „Zeitbäume" einzubeziehen.

Die Programmiersprache ALDiSP[7] ist demgegenüber eine funktionale Programmiersprache für allgemeine Anwendungen in der Echtzeitdatenverarbeitung. Die Mächtigkeit von ALDiSP ist der von Scheme[8] oder ML[9] vergleichbar, ALDiSP kann auch für über den engeren Kreis der Echtzeitdatenverarbeitung hinausgehende Aufgabenstellung verwendet werden. Im folgenden werden wir uns jedoch auf die für die Echtzeitdatenverarbeitung erforderlichen Eigenschaften beschränken. Die in diesem Zusammenhang wichtigen Ziele bei der Entwicklung der Sprache waren:

+ Einfache Darstellung und Manipulation von Datenflüssen
+ Bereitstellung von Konstrukten zur zeitlichen Einplanung von Prozessen in Abhängigkeit von synchron und asynchron auftretenden Ereignissen
+ Verzicht auf Variablen
+ Einengung des Sprachumfangs auf solche Konstrukte, die in effizienten Code übersetzt werden können
+ Mechanismen für synchrone und asynchrone Ein-/Ausgabe
+ Reiche Auswahl vordefinierter parametrierter Datentypen samt mächtiger Operatoren
+ Möglichkeiten zur Behandlung von Ausnahmen
+ Leistungsfähiges Modulkonzept, Möglichkeiten zur Einbeziehung von in anderen Sprachen geschriebenen, ausgetesteten Code-Modulen

II Spracheigenschaften

Programm- und Kommunikationsstruktur
Ein ALDiSP-Programm besteht im allgemeinen aus verschiedenen Modulen; jedes dieser Module enthält Funktionen, die die auf die Datenobjekte anzuwendenden Algorithmen spezifizieren. Es gibt ein Hauptmodul, das Funktionen aus allen anderen Modulen importieren kann; dieses Hauptmodul enthält einen *Netzwerk-Teil,* in dem beschrieben wird, wie die einzelnen importierten Funktionen zusammenarbeiten, indem sie Daten von anderen Funktionen empfangen bzw. an sie senden.

Ein grundlegender Datentyp der Sprache ist der Datentyp *stream.* Ströme stellen potentiell unendlich lange Listen dar, deren Elemente erst bei Bedarf berechnet werden. In ALDiSP dienen sie zur Darstellung von zeitdiskreten (Prozeß-)Größen und zur Kommunikation zwischen Funktionen. Die Elemente des Stroms werden von einem Strom-Generator erzeugt, der bei jeder neuen Nachfrage durch eine stromkonsumierende Funktion jeweils ein neues Element erzeugt. Zu jedem beliebigen Zeitpunkt ist nur eine endliche Menge von Stromelementen (nämlich die bereits erzeugte) vorhanden, während die (unendliche) Menge der folgenden Elemente erst noch berechnet werden muß. Die den Strom-Generator realisierende Funktion bleibt inaktiv, bis eine stromkonsumierende Funktion ein neues Element anfordert (Call-by-Need-Semantik). Auf diese Art und Weise können potentiell unendlich lang andauernde Signale bequem manipuliert werden.

Das folgende ALDiSP-Programm illustriert die Benutzung von Funktionen und Strömen. Es erzeugt als Ausgabe einen Strom von Zahlen, die die Lösung des Problems von Hamming [10] darstellen: Zu produzieren ist die geordnete und wiederholungsfreie Folge natürlicher Zahlen, die durch keine anderen Primzahlen als 2, 3 und 5 teilbar sind. Das Problem kann in einer Sprache wie ALDiSP, die komfortable Stromverarbeitung samt automatischer Speicherverwaltung zur Verfügung stellt, gelöst werden, indem ein Strom generiert wird, der zunächst nur ein Initialelement 1 enthält. Dieser Strom wird sowohl ausgegeben, wie auch in drei Prozesse eingespeist, die die Stromelemente mit 2, 3 und 5 multiplizieren. Die von diesen drei Prozessen erzeugten Zahlen werden durch einen weiteren Prozeß (hier mit merge3Streams bezeichnet) sortiert, zu einem einzigen Strom verschmolzen und von mehrfachem Auftritt derselben Zahl befreit. Der vom Prozeß merge3Streams ausgegebene Strom stellt den Ausgabestrom dar, dessen Elemente wieder in die Multiplizierer eingespeist werden. Bild 1 zeigt graphisch, wie der Algorithmus arbeitet; Bild 2 zeigt das zugehörige ALDiSP-Programm.

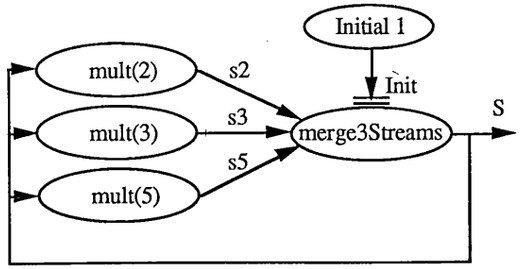

Bild 1: Netzwerk zur Lösung des Problems von Hamming

Die Topographie des Netzwerkes, d.h. die Angabe, wie der Datenfluß zwischen den Funktionen mult() und merge3Streams organisiert werden soll, spiegelt sich in dem mit net eingeleiteten Programmteil wider. Der Ausgabestrom S kann von anderen Funktionen

weiterverwendet werden. Das dargestellte Programm spezifiziert den Algorithmus komplett; es sind keine zusätzliche Speicherverwaltungsmaßnahmen, Zuweisungen, etc. erforderlich bzw. möglich.

```
func mult (n) (aStream) =  n*head(aStream) :: delay mult(n) (tail(aStream))

func mergeStreams (strA, strB) =
if   head(strA) < head(strB) then head(strA) :: delay mergeStreams(tail(strA), strB))
|    head(strA) > head(strB) then head(strB) :: delay mergeStreams(tail(strB), strA))
|    head(strA) = head(strB) then mergeStreams(tail(strA), strB) \ ---- suppress multiples
endif

func merge3Streams (strA, strB, strC) =
      mergeStreams (strA, mergeStreams (strB, StrC))

net
S = 1 :: delay merge3Streams(s2, s3, s5)
s2 = mult (2) (S)
s3 = mult (3) (S)
s5 = mult (5) (S)
in
  S
endnet
```

Bild 2: ALDiSP-Programm zur Realisierung des Netzwerkes von Bild 1

Die Funktionsresultate werden auf Ströme geschickt, indem die Funktion einem stream-Bezeichner zugewiesen wird. Sämtliche Maßnahmen, die erforderlich sind, um einen Strom zu erzeugen, ihm die Funktionsresultate zuzuordnen und auf ihm zu transportieren, verbergen sich hinter dem als Zuweisungssymbol verwendeten Gleichheitszeichen. Eine Funktion, die für eine Zahl definiert wurde, kann auch auf einen Strom angewendet werden. In diesem Fall wird automatisch dafür gesorgt, daß die Funktion auf jedes Element des Stromes angewendet wird. In ALDiSP funktioniert diese automatische Abbildung nicht nur bei Strömen, sondern auch bei homogenen Datentypen. Deren wichtigste Repräsentanten sind Felder (Arrays). Hier wird die Funktion auf alle Elemente des homogenen Datentyps angewendet. Die (zweistufige) Abbildung von Funktionen auf Ströme und Felder werde im folgenden Beispiel demonstriert:

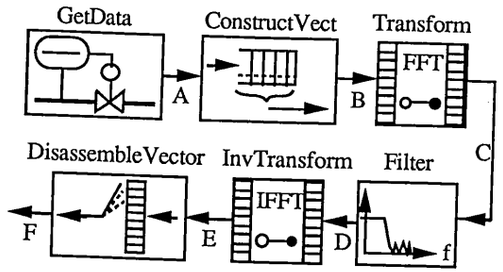

Bild 3: Netzwerk zur Auswertung einer Prozeßgröße

Das in Bild 3 dargestellte Netzwerk dient zur Analyse einer Prozeßgröße „Durchflußmenge". Dazu soll die in bestimmten Zeitabständen abgetastete Prozeßgröße zu Blöcken (Vektoren) gruppiert werden, eine blockweise Transformation in den Frequenzbereich erfolgen und im Frequenzbereich eine Tiefpaßfilterung der (nun komplexwertigen) Vektoren vorgenommen werden. Nach Rücktransformation wird wieder ein Strom bestehend aus skalaren Elementen erzeugt. Die Filterung im Frequenzbereich werde derart vorgenommen, daß die Amplitude der transformierten Vektoren mit der Filterfunktion multipliziert werde, während die Phasen unverändert bleiben: Jede Komponente des die Amplitudenwerte des Spektrums enthaltenden Vektors werde dazu mit dem korrespondierenden Dämpfungsfaktor der Filterfunktion multipliziert. Es werde angenommen, daß die Funktionen

```
GetData () = ...          \ Data acquisition
FFT (aVector) = ...       \ Input: Real vector, Output:Two real vectors
IFFT (twoVectors) = ...   \ Inverse FFT
AttnCoeffs (n) = ...      \ Generate n spectral coefficients (transfer function)
```

bereits innerhalb eines Moduls mit dem Namen FreqDomain definiert wurden. Dann nimmt das ALDiSP-Hauptmodul die in Bild 4 dargestellte Form an.

```
Imports GetData, FFT, IFFT, AttnCoeffs from FreqDomain
net
   A = GetData()
   B = VectorizeScalarStream (A, 256)
   (Ampl, Phase) = FFT (B)
   D = (Ampl * AttnCoeffs(256), Phase)      \ Apply " * " to all elements of both vectors
   E = IFFT (D)
   F = ListifyVector (E)                    \ Now we have a stream of scalars again
in
   F
endnet
```

Bild 4: ALDiSP - Programm für das Netzwerk nach Bild 3

Man beachte, daß FFT zwei Ströme erzeugt (strukturieres Ergebnis), nämlich für Amplitude und Phase. Wenn diese beiden Ströme getrennt gehalten werden, ist die Manipulation der Amplitude allein sehr einfach. Jede ALDiSP-Funktion kann multiple Ergebniswerte erzeugen, damit ist es nicht erforderlich, getrennt zu haltende Funktionsresultate auf einen Strom zu „multiplexieren" und vor Weiterbenutzung in der konsumierenden Funktion zu demultiplexieren. Von der automatischen Abbildung einer Funktion auf homogene Datenstrukturen wird in der Programmzeile in Bild 4 Gebrauch gemacht, in der die Filterung vorgenommen wird: Der Operator "*" wird auf alle Elemente der Blöcke/Vektoren Ampl und AttnCoeffs angewendet und hat als Resultat einen neuen Vektor mit skalaren Elementen. Die Sprache verfügt darüberhinaus über eine Reihe vordefinierter Funktionen zur Manipulation von Feldern (siehe [7]); es wird daher nur sehr selten erforderlich sein, Feldelemente einzeln zu behandeln.

Dieser Beispiel zeigt deutlich, daß der funktionale Ansatz ideal mit dem „Black-Box-Denken" des Systemanalysten und Regelungstechnikers harmoniert: Die Möglichkeiten zur Manipulationen von Strömen, die automatische Abbildung von Funktionen auf Ströme und insbesondere von Funktionen auf Felder macht die Transformation von

Block-Diagrammen in Programme extrem einfach. Der Programmierer muß komplexe Datenstrukturen nicht zunächst auseinandernehmen, ihre Komponenten behandeln und danach wieder zusammensetzen. Er kann diese Datenstrukturen stattdessen als Ganzes behandeln und sich darauf verlassen, daß der Abbildungsprozeß auf die einzelnen Elemente vom Compiler korrekt vorgenommen wird.

Wie viele moderne funktionale Programmiersprachen ist ALDiSP streng typisiert und polymorph. Das bedeutet, daß Funktionen mit Parametern unterschiedlichen Typs aufgerufen werden können (siehe dazu das folgende Beispiel). Der Übersetzer erkennt Typinkonsistenzen automatisch, indem er den Typ der Operation, die auf einen bestimmten Parameter angewendet wird, analysiert und feststellt, ob Operation und Operand zueinander typkompatibel sind. Sowohl Funktionen wie auch Operatoren können überladen werden, d.h. für vollständig unterschiedlich geartete Operanden definiert werden. Operatoren können in Post-, Prä- oder Infix-Notation verwendet werden, je nachdem, was besser lesbar oder leichter anzuwenden ist. Die Operator-Präzedenz ist in sechs verschiedenen Stufen einstellbar.

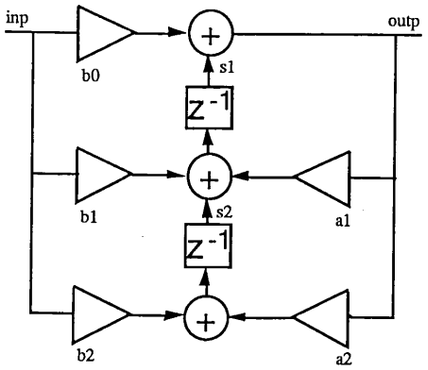

Bild 5: Filternetzwerk

```
func IIR (b0,b1,b2,a1,a2: number) (inp: stream) =
let
   outp = b0*inp + s1
   s1 = 0 :: delay  b1*inp + a1*outp + s2
   s2 = 0 :: delay  b2*inp + a2*outp
in
   outp
endlet
```

Bild 6: Implementierung des Filternetzwerks nach Bild 5

ALDiSP erlaubt ferner die Definition von Funktionen höherer Ordnung, d.h. Funktionen, die andere Funktionen als Eingabeparameter tragen oder Funktionen als Resultate abliefern. Ströme können rekursiv definiert werden und sie können um eine beliebige Anzahl von Elementen verschoben bzw. verzögert werden. Deshalb kann das Stromkonzept auch ideal in der Regelungstechnik zur Spezifikation von digitalen Reglern verwendet werden.

Als Beispiel werde das Filternetzwerk nach Bild 5 betrachtet (siehe z.B. [11]): Dieses Netzwerk kann durch die ALDiSP-Funktion nach Bild 6 direkt realisiert werden. Der Strom outp ist das Resultat der elementweisen Addition der Ströme inp (jedes Stromelement gewichtet mit b0) und s1. Der Strom s1 besteht aus der Konkatenation des Elementes "0" mit der gewichteten Summe aller anderen Ströme, die jeweils um ein Element verzögert wurden. Man beachte, daß die Funktion IIR Integer-Zahlen jedes Typs sowie auch Real- und Complex-Zahlen filtert, da die Operatoren "+", "*" und "::" für alle diese Datentypen definiert sind. Ein kürzere Spezifikation für das Filter als die in Bild 6 gezeigte ist wohl nur schwerlich vorstellbar.

Asynchrone Kommunikation und Ein-/Ausgabe

Ströme sind ein adäquates Mittel zur Repräsentation von Datenflüssen im synchronen Fall, in welchem die permanente Verfügbarkeit des jeweils nächsten angeforderten Stromelements garantiert werden kann. Für Situationen, in denen nicht angegeben werden kann, ob und wann ein nächstes Stromelement verfügbar wird, ist dieses Konzept weniger geeignet. Es ist zwar vorstellbar, daß ein Strom-Generator im Falle der Nichtverfügbarkeit eines Datenobjektes ständig Stromelemente erzeugt, die der konsumierenden Funktion die Nichtverfügbarkeit von Nutzinformation mitteilen, das aber wäre höchst ineffizient. Aus diesem Grunde wurden in ALDiSP die sogenannten *pipes* eingeführt. Syntaktisch sind pipes äquivalent zu streams, sie sind jedoch nicht nachfrage-, sondern datengetrieben. Eine pipe kann als Puffer angesehen werden, der Datenobjekte aufnimmt und speichert, sobald sie verfügbar werden. Eine konsumierende Funktion kann Daten aus pipes in genau der gleichen Weise wie bei streams benutzen. Wenn eine Funktion Daten aus der pipe entnimmt, wird diese elementweise geleert. Wenn ein Konsument auf eine leere pipe zugreifen will, so wird er *suspendiert*. Auf pipes ist ein Prädikat isAvailable definiert, welches den Wahrheitswert true annimt, sobald sich mindestens ein Element in der pipe befindet.

Sowohl streams wie pipes können direkt für Ein-/Ausgabe verwendet werden. Streams werden für synchrone Eingabe verwendet, bei der einzugebende Daten ständig verfügbar sind. Für asynchrone E/A werden pipes benutzt. Mit pipes und streams sind auf einfache Art und Weise Systeme realisierbar, die unterschiedliche Ankunftsraten aufweisen (multi-rate-systems). Logische Geräte, auf die ständig nichtblockierend zugegriffen werden kann werden als *register* bezeichnet, während blockierende Geräte *port* heißen. Normalerweise werden register mit streams assoziiert und ports mit pipes. Damit ist eine vollständige Verbindung des Programmsystems mit der Umwelt bzw. Peripherie möglich. Weil pipes und streams syntaktisch gleich und im Gebrauch ähnlich sind, ist es sehr einfach, eine mit einem E/A-Gerät verbundene pipe durch einen von einem Strom-Generator gespeisten Strom zu ersetzen. Durch die polymorphe Struktur der Sprache kann der Strom-Generator sogar in vielen Fällen Datenobjekte eines anderen Typs als die pipe liefern, ohne daß die konsumierende Funktion geändert werden muß. Diese Ersetzung kann in der Testphase eines Programms sehr nützlich sein, wenn nämlich zunächst Peripheriedaten vollständig reproduzierbar erzeugt werden sollen, bevor das Programmsystem mit der Umwelt verbunden wird.

Die Einbeziehung von Zeiten und das suspend-Konstrukt

In ALDiSP ist sowohl synchrone wie asynchrone Prozeßeinplanung möglich. Bei synchroner Einplanung wird eine Aktion zu bestimmten, äquidistanten Zeitpunkten ausgeführt, während bei asynchroner Einplanung ein Prozeß für eine gewisse Zeit suspendiert bleibt, bis sich entweder ein externer Zustand ändert oder ein Zeitüberwachungsmechanismus den Prozeß aktiviert. In ALDiSP gibt es nur ein einziges Konstrukt, das alle Möglichkeiten der Zeiteinplanung umfaßt, das suspend-Konstrukt:

suspend expr1 **until** expr2 **within** timeExpr1, timeExpr2

Wenn eine Funktion aufgerufen wird, die einen suspendierten Ausdruck enthält, wird ein anonymer Prozeß kreiert. Dieser Prozeß wird sofort suspendiert und bleibt inaktiv, bis expr2 wahr wird. Sobald dies der Fall ist, wird garantiert, daß der Prozeß *nach* Ablauf der Zeit timeExpr1 und *vor* Ablauf der Zeit timeExpr2 wieder aktiviert wird. Nach seiner Aktivierung wertet der Prozeß expr1 aus und terminiert. Ein einfaches Beispiel möge den Gebrauch von suspend veranschaulichen:

suspend OpenValve() **until** OverPressure **within** 0 ms, 0.5 ms

Hier wird ein Sicherheitsventil, das von einer Funktion OpenValve gesteuert wird, in einem Zeitintervall von 0,5 ms nach dem Auftreten von Überdruck geöffnet. Mit suspend sind Makros für alle anderen wichtigen Fälle zeitlicher Einplanung leicht zu schreiben, so z.B.:

```
\ ---- Synchrone Verzögerung
expr after time ≡ suspend expr until true within time, time

\ ---- Asynchron, Aktion wird sofort ausgeführt
expr1 when expr2 ≡ suspend expr1 until expr2 within 0 sec, 0 sec

\ ---- Asynchron mit Zeitüberwachung
when (expr1, expr2, timeoutPeriod) ≡
let
   StartTime = SystemClock()
in
   suspend expr1 until (expr2 or SystemClock - StartTime > timeoutPeriod) within 0 sec, 0
sec
endlet
```

Die timeExpressions dürfen zur Laufzeit auswertbare, dynamische Ausdrücke sein; sind sie jedoch bereits zur Übersetzungszeit bekannt, kann ein statischer Zeiteinteilungsplan bereits vom Übersetzer erstellt werden.

Mit suspend und dem isAvailable-Prädikat können asynchrone pipes sehr elegant manipuliert werden. So nimmt beispielsweise die wichtige Operation des Zusammenführens zweier pipes (von denen jede Daten verfügbar haben kann) die folgende Form an:

```
proc merge (pipe1, pipe2) =
suspend
   if isAvailable(p1) then head(p1) :: merge(p2,p1)
   else head(p2) :: merge(p1,p2)
   endif
until
   isAvailable(p1) or isAvailable(p2)
within 0 sec, 0 sec
```


Dies ist ein leichtverständliches Gegenstück zum ALT-Konstrukt aus Occam [8], das dazu benutzt wird, Daten, die durch unterschiedliche Kanäle bei verschiedenen Ankunftsraten ankommen, zusammenzuführen. Es ist offensichtlich, daß die Zusammenführung synchroner Ströme noch einfacher aufschreibbar ist. Mit Hilfe des suspend-Konstrukts sind auch Unterbrechungsbehandler leicht in ALDiSP zu formulieren.

Das Typsystem der Sprache

Das Typsystem von ALDiSP ist auf Prädikaten und nicht auf Termen aufgebaut: Eine beliebig gewählte Menge von Werten kann einen Typ darstellen. Auch Funktionen können zur Typvereinbarung herangezogen werden. Zum Beispiel kann ein Typ definiert werden, dessen Wertebereich alle Vielfache von 3 umfaßt:

type mult3(x) = **if** isInt (x) **then** x **mod** 3 = 0 **else** false **endif**

In den meisten Fällen sind solche komplexe Definitionen natürlich nicht erforderlich. Eine umfassende Menge von Grunddatentypen ist bereits vordefiniert, die meisten Grunddatentypen sind darüberhinaus parametriert. Vordefinierte Typklassen sind:

- Basistypen zum Aufbau des Typsystems
- Atomare Typen: Unstrukturiert, alle Arten von Zahlen
- Felder: n-dimensionale Reihungen von Objekten desselben Typs
- Produkttypen zur Darstellung von Listen
- Summentypen (Records)
- Maschinentypen: Register, Ports, Interrupts

Beipiele für atomare Typen sind Zahlen: nBitInteger (n), nBitCardinal (n), FixInt (n,m), ShortReal, Real, Longreal, etc. Alle gängigen Operationen auf diesen Typen sind vorhanden. Für Felder sind eine ganze Reihe weiterer Operationen vordefiniert, damit der Zugriff auf ein einzelnes Element praktisch nie erforderlich wird:

- Automatische Abbildung: [1,2,3] + 10 → [11,12,13]
- Selektive Änderung: UpdVector ([1,2,3], 1, 5) → [1,5,3]
- Reduktion: Reduce ("-", [1,2,3,4]) → ((1 - 2) - 3) - 4
- Erzeuge Untervektor: SubVector ([1,2,3], 0, 2) → [1,2]

Ähnliche Funktionen sind für zweidimensionale Felder verfügbar; ferner sind Funktionen vordefiniert, die Spalten und Zeilen ganzer Matrizen manipulieren, Diagonalen extrahieren, Determinaten bestimmen und Matrizen reduzieren.

Ausnahmebehandlung

Ausdrücke und Funktionen können bewacht werden, d.h. während ihrer Auswertung zu erwartende Fehler können spezifiziert und im Falle ihres Auftritts abgefangen werden. Falls ein Fehler auftritt, wird ein vordefinierter oder benutzerdefinierter Ausnahmebehandler aufgerufen. Prinzipiell hat ein bewachter Ausdruck die folgende Form:

```
guard expr
on   ExceptionDefinition
:
on   ExceptionDefinition
endguard
```

Wenn der Fehler auftritt, wird die entsprechende Behandlungsroutine aufgerufen. Nach der Auswertung des Behandlers wird die Auswertung von expr nicht fortgesetzt, sondern der im Behandler definierte Wert wird zurückgeliefert. Sollte es dennoch erforderlich sein, die Auswertung fortzusetzen, kann dies durch Aufruf der Funktion continue ebenfalls geschehen. Beispiel:

```
func reciprocalPlus3Vers1 (x) =  guard (1/x) + 3
                                 on DivisionByZero = 42
                                 endguard

func reciprocalPlus3Vers2 (x) =  guard (1/x) + 3
                                 on DivisionByZero = continue(42)
                                 endguard
```

Wird die erste Funktion mit dem Wert 0 aufgerufen, liefert sie als Resultat 42 ab. Die zweite Funktion liefert bei einem entsprechenden Aufruf den Wert 45, da mit der Auswertung hinter der Aufrufstelle des Ausnahmebehandlers fortgefahren wird.

Das Konzept der Ausnahmebehandlung wird auch verwendet für die Definition der Rundungsmodi im Falle von Zahlenüberläufen. Diese Rundungsmodi sind benutzerdefinierbar, die wichtigsten für Integer-Zahlen vordefinierten sind: Signaling, Ignoring, Realing, Complexing, Saturated, Wrapping. Vordefinierte Rundungsmodi für Real-Zahlen sind: RoundToZero, RoundToPlus, RoundToMinus, RoundToEven, RoundToOdd.

III Zusammenfassung und Ausblick

Es wurde eine mächtige polymorphe funktionale Programmiersprache für die Echtzeitdatenverarbeitung vorgestellt, die über Funktionen höherer Ordnung verfügt, die eine automatische zweistufige Abbildung von Funktionen auf unendliche und endliche Datenstrukturen vornimmt und die über universelle Möglichkeiten zur zeitlichen Einplanung von Aktionen verfügt. Ferner stellt sie die üblichen Konstrukte zur Modularisierung samt überladbarer Operatoren zur Verfügung. Sowohl synchrone wie asynchrone Datenströme können komfortabel gehandhabt werden, dies macht die Programmierung von Systemen mit unterschiedlichen Datenflußraten einfach. Direkter Maschinenzugriff und Hardware-Unterbrechungsbehandlung sind möglich. Ein sehr flexibles Typsystem, ein leistungsfähiger Ausnahmebehandlungsmechanismus und eine große Anzahl vordefinierter Funktionen ermöglichen die Bildung komplexer, der Problemstellung angemessenen Datenstrukturen und ihre Manipulation als ganzes, ohne sich in algorithmischen Details zu verlieren. Mit dieser Sprache können Probleme auf sehr hohem Niveau spezifiziert werden, oft als direkte Entsprechung ihrer mathematischen Beschreibung. Das bedeutet auf der anderen Seite, daß ein großer Teil des Aufwandes, den normalerweise der Programmierer übernimmt (beispielsweise für die Organisation der Speicherverwaltung) nunmehr vom Übersetzer übernommen werden muß. Demensprechend ist es wesentlich schwieriger, Code für Standard-Prozessoren zu erzeugen, als dies von imperativen Sprachen bekannt ist. Andererseits ist es zumindest prinzipiell wesentlich einfacher, aus einer funktionalen Beschreibung Code für Multiprozessorsysteme zu erzeugen, als sequentialisierten Code einer imperativen Sprache zu re-parallelisieren. Mit fortschreitender Compiler-Technologie wird der Unterschied in der Code-Qualität zwischen den beiden Sprachklassen zunehmend

geringer werden, die Vorteile der höheren Abstraktion von der Maschine und der parallelen Ausführbarkeit funktionaler Programme werden jedoch bestehen bleiben.

IV Schrifttum

[1] **P. Caspi, N. Halbwachs, D. Pilaud, J.A. Plaice**
Lustre, a Declarative Language for Programming Synchronous Systems
Proc. 14th ACM Symposium on Principles of Programming languages, München, 1987

[2] **A. Benveniste, P. Bournai, T. Gautier, P. LeGuernic**
Signal: A Dataflow Oriented Language for Signal Processing
INRIA, Rapport No. 378, März 1985

[3] **E. Ashcroft, W. Wadge**
Lucid, a Nonprocedural Language with Iteration
Comm. ACM, Vol. 20, No. 7, Juli 1977

[4] **European Development Center**
Silage Compiler Reference Manual
Brüssel, 1989

[5] **R. Dannenberg**
Arctic: A Functional Language for Real-Time Control
1984 ACM Symp. on Lisp and Functional Programming, Austin, Texas

[6] **D. Harrison**
Ruth: A Functional Language for Real-Time Programming
LNCS 259, Parallel Arch. and Languages Europe, Springer-Verlag, 1987

[7] **M. Freericks, A. Knoll**
ALDiSP - Eine applikative Programmiersprache für Anwendungen in der Echtzeitdatenverarbeitung und der digitalen Signalverarbeitung
Technischer Bericht, Techn. Univ. Berlin, FB 20 Nr. 90-9

[8] **H. Abelson et. al.**
Revised^3 Report on the Algorithmic Language Scheme
MIT AI Memo 848a, Sept. 1986

[9] **R. Milner**
A Proposal for Standard ML
1984 ACM Symp. on Lisp and Functional Programming, Austin, Texas

[10] **P. Pepper, G. Egger**
The Opal Project
Interner Bericht, TU Berlin, 1987

[11] **Leonhard, W.**
Digitale Signalverarbeitung in der Meß- und Regelungstechnik
Stuttgart: Teubner, 1989

[12] **INMOS Ltd.**
Occam 2 Reference Manual
Prentice Hall, 1988