Technischer Report No. 90-9
ALDiSP-Sprachbeschreibung

Knoll, Freericks

6. September 1995

Zusammenfassung

Diese Arbeit beschreibt die Sprache ALDiSP. ALDiSP ist eine funktionale Sprache im Stil von *ML* oder *Scheme*. ALDiSP ermöglicht es, Algorithmen der digitalen Signalverarbeitung schnell und direkt zu formulieren. ALDiSP ist keine Sprache, die nur für spezielle Anwendungen geeignet ist, sondern eine sehr mächtige, allgemeine Sprache, die über Konstrukte zur Beschreibung von Zeit und Prozessen verfügt und sich besonders zur Arbeit mit numerischen Arrays eignet. ALDiSP kennt den Begriff des Nebeneffekts und ist so in der Lage, I/O sauber zu beschreiben. ALDiSP verfügt über

- eine breite Palette numerischer Datentypen, die sich beliebig erweitern läßt,
- die Möglichkeit, mathematische Operatoren in natürlicher Weise zu notieren,
- automatisches Mapping aller Funktionen auf Datenströme und Arrays,
- ein ausdrucksstarkes Typsystem,
- die Möglichkeit zur Definition abstrakter Datentypen,
- ein occam entlehnter Rendezvous-Mechanismus zur Interprozeßkommunikation,
- verschiedene Typen wie '`Interrupt`', '`Port`' und '`Register`', die die durch die Hardware vorgegeben I/O-Strukturen modellieren,
- Modularisierungskonstrukte und Blockstruktur und
- einen allgemeinen Exception-Mechanismus, der auf dynamisch gebundenen Funktionen beruht.

ALDiSP wird call-by-value ausgewertet; es ist zugleich möglich, explizit call-by-need (lazy evaluation) durchzuführen und Makro-Funktionen zu definieren.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Einordnung

Dieser Abschnitt soll ALDiSP in die Sprachlandschaft einordnen und die Position im Rahmen des CADiSP-Projekts klarmachen. CADiSP ist ein System zur interaktiven graphischen Entwicklung von Programmen für DSP-Prozessoren. Das CADiSP-System unterteilt sich in die graphische Umgebung (einen Editor, der ALDiSP-Programme generiert) und den ALDiSP-Interpreter/Compiler.

1.1.1 Digitale Signalverarbeitung

Digitale Signalverarbeitung ist, vor allem, wenn sie in Echtzeit erfolgen soll, ein Aufgabengebiet, das die Grenzen des technisch Machbaren ausreizt. Deswegen werden DSP-Systeme heutzutage auf Basis spezieller Prozessoren konstruiert, die aus Effizienzgründen in Assembler (bestenfalls noch in C) programmiert werden. Mit der wachsenden Komplexität von DSP-Algorithmen werden die Programme größer und unübersichtlicher. Da DSP-Algorithmen sich auf Strukturebene meist als Datenflußdiagramme darstellen lassen, scheint es sinnvoll, von imperativen Sprachen abzugehen und Datenfluß- oder funktionale Sprachen als Beschreibungsmittel einzusetzen. Zwei spezielle Probleme tauchen dabei auf: das der effizienten Compilation und das der Echtzeittauglichkeit der Sprachkonstrukte, d.h. der Möglichkeit, anzugeben, wie schnell Reaktionen auf asynchrone Ereignisse zu erfolgen haben bzw. wann synchrone Vorgänge geschehen sollen.

1.1.2 Warum eine funktionale Sprache?

Funktionale Sprachen haben eine lange Tradition. LISP, die erste Sprache, die funktional genannt werden kann (obwohl die Konzepte, die man heute unter dem Begriff 'funktional' sammelt, damals noch nicht explizit ausgearbeitet waren), entstand kurz nach der Entwicklung des ersten FORTRAN-Compilers. Funktionale Sprachen setzen direkt auf dem Funktionsmodell des Church'schen

Lambda-Kalküls auf, das deutlich mächtiger[1] ist als das Turing-Maschinen-Modell, das die theoretische Grundlage zum Bau herkömmlicher Maschinen und zur Entwicklung 'imperativer' Sprachen liefert. Während das Lambda-Kalkül 'Programme' als mächtige Funktionen begreift (in der Tat sind Funktions-Abstraktion und -Applikation die einzigen semantischen Elemente des Kalküls), geht der Turing-Maschinen-Ansatz davon aus, das ein Programm einen Speicherzustand in einen anderen Speicherzustand überführt, ohne in irgendeiner Weise von diesem Speicher zu abstrahieren. Imperative (Zustands-orientierte) Sprachen haben als 'Codierungshilfe' (*formula translation*) für von-Neumann-Maschinen angefangen und mit der Zeit immer mehr 'funktionale' Elemente übernommen. Die Implementierungsprobleme, die man in der Frühzeit mit funktionalen Sprachen hatte (vor allem der Speichermangel machte fortgeschrittene Techniken z.B. der Garbage Collection, d.h. der automatischen Speicherreorganisierung, erst spät möglich), brachten es mit sich, daß funktionale Sprachen das Ghetto der theoretischen Informatik erst spät verlassen haben.

Inzwischen sind viele Informatiker überzeugt, daß den funktionale Sprachen (und den relationalen wie PROLOG) die Zukunft, gerade im Bereich der großen Software-Systeme, gehört. Funktionale Sprachen sind einfacher automatisch verifizierbar als imperative[2] und zeigen bessere Möglichkeiten zur Modularisierung und zur Modul-Wiederverwertung (vor allem auf Grund der Möglichkeit, polymorphe Funktionen zu schreiben).

Man kann davon ausgehen, das die Produktivität eines Programmierers ungefähr proportional zum Abstraktionsgrad der benutzten Sprache und zum Interaktivitätsgrad der Entwicklungsumgebung ist[3]. Beides ist in modernen funktionalen Sprachumgebungen maximiert.

Elemente Funktionaler Sprachen

Wichtige Eigenschaften moderner funktionaler Sprachen sind:

- Polymorphie: Die Möglichkeit, allgemein definierte Funktionen auf Objekte verschiedenster Typen anwenden zu können, die implementierungstechnisch verschiedenartig sind.

- Modularisierungskonzepte und abstrakte Datentypen: hierdurch können große Projekte in sauber getrennte Teile separiert werden (ohne Einschränkungen wie z.B. Modula-2 [4] zu unterliegen).

[1] "Mächtiger" soll hier nicht im Sinne der Menge der dadurch berechenbaren Funktionen, sondern eher pragmatisch aufgefaßt werden. Modula-2 ist mächtiger als Pascal, Common Lisp ist mächtiger als Scheme.

[2] Heutzutage sind nur kleine imperative Programme verifizierbar, während die Größe bei – besser abstrahierenden – funktionalen Programmen keine so wichtige Rolle spielt.

[3] [1], pp. 134ff.

[4] In Modula-2 können nur solche Typen über Modulgrenzen exportiert werden, die sich in einer Registerbreite unterbringen lassen. Natürlich können auch Funktionen nur exportiert werden, wenn die Argumenttypen exportiert sind.

- Funktionen als First-Class-Objekte: Funktionen können Funktionen als Parameter und als Ergebnis haben.
- Orthogonalität: alle Typen haben 'gleiche Rechte' – jedes Objekt kann als Parameter über– und als Ergebnis zurückgegeben werden.
- Rekursion als einziges Iterationskonstrukt: Das `prog`–Konstrukt des frühen LISP ist überkommen; Die End-Rekursion (tail recursion) wird genauso effizient übersetzt wie eine imperative Schleife.
- Exceptions: die Möglichkeit, auf Ausnahmefälle gezielt zu reagieren.
- Lazy Evaluation: das Arbeiten mit potentiell unendlich großen Datenstrukturen.

Compilation Funktionaler Sprachen

Funktionale Sprachen sind, historisch gesehen, mit den unschmeichelhaften Attributen der Langsamkeit und Speicherineffizienz versehen. Zugleich jedoch sind funktional geschriebene Algorithmen gut zur Parallelisierung geeignet und mit formalen Methoden leicht analysierbar. Moderne Compiler zeigen, daß funktionale Sprachen sich durchaus so gut wie z.B. C übersetzen lassen [8].

1.2 Grundlegende Sprachkonzepte

ALDiSP ist eine Sprache, die an wenigen Konzepten orientiert ist. In dieser Einleitung sollen diese Konzepte kurz vorgestellt werden, damit die Übersicht beim Lesen des Reports nicht verlorengeht.

1.2.1 Prozesse und Zeit

Ein Programm läuft ab, indem parallele Prozesse ausgeführt werden, die Nebeneffekte erzeugen. Nur Nebeneffekte (= I/O) können zeitlich spezifiziert werden. Die Spezifikation erfolgt über ein einfaches Mittel: jeder Prozeß existiert in einem *virtuellen* Zeitpunkt; jegliche I/O, die ein Prozeß ausführt, vollzieht sich zu diesem Zeitpunkt. Ein Scheduler sorgt dafür, daß die virtuellen I/O-Zeiten mit den 'realen' Zeiten übereinstimmen. Es gibt ein zentrales Konstrukt namens `suspend`, das Prozesse erzeugt und mit Zeitvorgaben versieht. Bei der Prozeß-Erzeugung wird eine *Startbedingung* angegeben, die dem Scheduler sagt, welcher globale Zustand (bzw. welche Zustandsänderung) die Ausführung des Prozesses triggern soll und in welchem Zeitrahmen dies zu geschehen hat.

Ein typisches Beispiel für einen *asynchronen* Prozeß (das Dampfkesselbeispiel) könnte so aussehen:

```
suspend ÖffneVentil until ÜberdruckInterrupt within 0 ms, 10 ms
```

Nachdem dieser Ausdruck ausgewertet wurde, wartet ein Prozeß im Hintergrund darauf, daß der `ÜberdruckInterrupt` kommt. Ist dies der Fall, hat der Scheduler 10 Millisekunden Zeit, den Prozeß zu starten.

Typische *synchrone* Prozesse sind in DSP-Applikationen zu finden. Als einfachstes Beispiel soll ein System dienen, das in Echtzeit zwei Audio-Kanäle mischt:

```
\ Programm zum Mischen zweier Kan\"ale
\ kein Overflow-Test

proc misch() =
  writeRegister(readRegister(Kanal1)+readRegister(Kanal2))

proc misch44Khz() =
  suspend seq misch(); misch44Khz() endseq
  until true
  within 1/44000 sec, 1/44000 sec

net in
 misch44Khz()
endnet
```

Hier kann man sehen, daß der 'asynchrone' Anteil des Suspend-Konstrukts wegfällt (die triviale Bedingung `true` ist immer erfüllt); der 'synchrone' Anteil gibt an, daß genau 1/44000 Sekunde gewartet werden soll, bis es weitergeht.

1.2.2 Die Evaluation von Expressions

Das elementare Ausführungsmodell in ALDiSP ist die Evaluation von Expressions, d.h. die Auswertung von Ausdrücken. Ein Programm wird als Ausdruck verstanden, der aus Unter-Ausdrücken rekursiv zusammengesetzt ist. Das Programm wird ausgeführt, indem der dem Ausdruck entsprechende Wert berechnet wird. Die Semantik der Sprache wird angegeben, indem beschrieben wird, in welcher Weise der Wert von Ausdrücken bestimmt wird.

1.2.3 Objekte und Primitive

Die Werte in ALDiSP heißen *Objekte*. So ziemlich alles ist ein Objekt. Funktionen (selbst Objekte) bilden Objekte auf Objekte ab. Man muß zwischen *Datenobjekten* und *Maschinenobjekten* unterscheiden. Ein Datenobjekt ist ein berechenbares Ding; ein Maschinenobjekt ist die Repräsentation einer Hardware-Gegebenheit, z.B. eines D/A-Wandlers.

Die Funktionen, Typen und konstanten Objekte, die von der Sprache (in den Standard-Libraries) vorgegeben werden, heißen *Primitive*. ALDiSP wird

definiert, indem neben den Sprachkonstrukten, also den Regeln zur Bildung von Ausdrücken, die Primitiven beschrieben werden.

1.2.4 Funktionale

Funktionale (*higher-order functions*) sind Funktionen, die Funktionen als Argumente und/oder als Resultate haben. (Ein typisches Funktional ist `Map`, das eine Funktion auf alle Argumente einer Datenstruktur anwendet.) Funktionale werden oft als der implementierungstechnische Schwachpunkt funktionaler Sprachen angesehen, da sie eine Compilation stark erschweren.

ALDiSP arbeitet in starken Maße mit Funktionalen. Man kann ein Funktional als eine 'abstrakte Konstrollstruktur' verstehen — im gleichen Maße, in dem ein abstrakter Datentyp von einer aktuellen Datenrepräsentation abstrahiert, abstrahiert ein Funktional von einem aktuellen Rekursionsschema. Eine Sprache, die hinreichend viele Funktionale zur Verfügung stellt, kann auf Rekursion völlig verzichten! Gerade die Analyse der Rekursion aber (im allgemeinen, die Kontrollflußanalyse) stellt DSP-Compiler vor ein echtes Problem: wenn es zu beweisen gilt, daß ein gegebenes Stück Code in einer gegebenen Zeit fertig wird, darf es keine echte Rekursion geben[5] (oder die Sprungbedingungen müssen trivial analysierbar sein). Hat man nun vorgegebene Rekursionsstrukturen in Form von Funktionalen, so kann die Analyse vom Konstrukteur des Compilers übernommen werden. (Gibt es keine Funktionale, versucht der Compiler faktisch, herauszubekommen, welche Funktionale benutzt worden wären, wenn es sie denn gäbe.)

1.2.5 Nebeneffekte

Rein funktionale Sprachen haben es nicht nötig, sich mit I/O und anderen unliebsamen nicht leicht mathematisch beschreibbaren[6] Aspekten von Programmen zu beschäftigen. ALDiSP muß diese Aspekte sogar zeitlich spezifizieren; es tut dies, indem das Konzept der *Nebeneffekte* (worunter dasselbe wie "I/O" verstanden werden soll) in die Sprache integriert wird. ALDiSP kennt keine echten ("Von-Neumann"–) Variablen, auch gibt es ein echt funktionales Subset der Sprache; nur im semantischen Umfeld der Prozeßgestaltung und der I/O kommen diese 'unsauberen' Konstrukte ins Spiel. In einem 'normalen' Programm gibt es vielleicht 1% Nebeneffektbeschreibung; in den Libraries und Modulen gibt es so etwas überhaupt nicht[7]. Diese Isolierung der Problemstellen soll eine

[5] Das Halteproblem ist unlösbar, und die unkontrollierte Rekursion ist die Wurzel allen Übels.

[6] Man schaue sich die I/O in PROLOG an! Zugegeben, mit Streams und Continuations kann man einiges machen; aber eigentlich wird immer nur versucht, eine schöne Verkleidung zu konstruieren, hinter der sich die 'echte' I/O abspielt. Wen dies interessiert: in 4.3 wird beschrieben, wie dies in ALDiSP aussähe.

[7] D.h., es können zwar Prozeduren definiert werden, die, wenn sie später aufgerufen werden, Nebeneffekte erzeugen; es können jedoch keine Nebeneffekte *während der Definition* eines

übersichtliche und einfach zu programmierende Beschreibung von Prozessen in Echtzeit ermöglichen.

1.2.6 Modularisierungstechniken

Die zentralen Modularisierungstechniken, die in ALDiSP eingesetzt werden, sind die Unterteilung des Programms in voneinander unabhängige *Module* und die Benutzung abstrakter Datentypen. Module vereinfachen das Lesen des Programmtextes; Datentypen vereinfachen das Verständnis des Datenflusses. Die zentralen zum Einsatz kommenden Datentypen sind *Streams* und *Pipes*. Streams kann man als call-by-need-Listen auffassen, die potentiell unendlich sind (output-driven evaluation). Pipes sind das "call-by-availability"-Gegenstück zu Streams (input-driven computation). Beide repräsentieren Datenströme; sie unterscheiden sich nur im Kontrollfluß.

Aufbau eines Programms

Ein typisches DSP-Programm besteht aus Modulen und einer Netzbeschreibung. Die Netzbeschreibung definiert eine Datenfluß-Struktur, bei der die 'Verbindungslinien' durch Streams und Pipes, die 'black boxes' durch Funktionen repräsentiert werden. Die Funktionen werden, zusammen mit den Datentypen, über denen die Streams und Pipes deklariert sind, in den Modulen definiert. Die Module sind normalerweise in einzelnen Files abgelagert und vorübersetzt, so daß die Erstellung eines Programms im Normalfall (wenn keine neuen Funktionen zu schreiben sind) in der Netzdefinition besteht. Sind neue Funktionen notwendig, so können alte Module erweitert oder neue geschrieben werden, je nach Bedarf.

Streams und Pipes

Ein Stream stellt eine (potentiell nicht terminierende) Berechnung dar, die bei Bedarf beliebig 'weit' ausgeführt werden kann. Ein Stream ist für alle praktischen Belange eine Liste, deren erste n Elemente berechnet sind; der Rest der Liste existiert nur 'virtuell' und wird bei Bedarf erstellt. Ein typischer Stream ist die Liste aller natürlichen Zahlen:

```
let func count(i) = i :: DELAY count(i+1) IN count(0) endlet
```

evaluiert sich zum Stream `(0 1 2 3...)`[8]. Typische Klassen von Funktionen über Streams sind

Generatoren: Funktionen, die Streams erzeugen. `count` ist ein Generator.

Moduls auftreten.

[8] "::" ist der Infix-Listenkonstruktor, der oft auch durch den Punkt (".") notiert wird.

Konsumptoren: Funktionen, die Streams konsumiern. Ein Beispiel ist die Funktion, die die ersten n Elemente eines Streams addiert:

```
func addElements(n)(s) =
  if n=0 then 0
         else head(s)+addElements(n-1)(tail(s))
  endif
```

Mappings: Die typischste 'synchrone' Stream-Operation überhaupt ist die Funktion f', die eine Funktion f elementweise auf Streams anwendet. Mappings können ein- oder mehrstellig sein; es wird ein Stream generiert, der genauso lang ist wie der kürzeste[9] Eingabestream. Ein typisches Beispiel für ein zweistelliges Mapping, das über nichtterminierenden Number-Streams definiert ist, ist die Funktion, die zwei Streams elementweise addiert:

```
func addStreams(s1,s2) =
  head(s1)+head(s2) :: DELAY addStreams(tail(s1),tail(s2))
```

Weil Mappings so häufig auftreten, erfolgen sie in ALDiSP automatisch, d.h. die obige Definition könnte einfacher als

```
func addStreams(s1,s2) = s1+s2
```

beschrieben werden.

Das Problem mit Streams ist, daß sie zu jeder Zeit beliebig weit berechenbar sein müssen. Bei asynchronen Applikationen ist dies nicht der Fall: Dort muß man auf Ereignisse reagieren, d.h. warten, bis etwas (eine I/O) passiert. Diese Wartezeit ist nicht beeinflußbar. Das zu den Streams analoge Konstrukt für diesen Fall heißt *Pipe* und unterscheidet sich syntaktisch kaum von den Streams, die Semantik jedoch ist eine andere. Wenn im Rahmen der Auswertung eines Ausdrucks auf eine 'leere' Pipe zugegriffen wird, so wird der auswertende Prozeß schlafengelegt, bis der Input, der die Pipe mit Daten versorgt, ein neues Item erzeugt hat; dann wird der Prozeß aufgeweckt; die Daten werden verarbeitet, und der Prozeß legt sich wieder schlafen. Dies entspricht dem Standard-Datenflußmodell, in dem das Vorhandensein von Tokens einen Knoten aktiviert.

Es gibt zwei Arten, Funktionen über Pipes zu formulieren: die explizite und die implizite. Explizite Pipe-Funktionen benutzen das Suspension-Konstrukt, um die Aktivierungsbedinungen zu formulieren; implizite Pipe-Funktionen sehen aus wie Stream-Funktionen ohne `DELAY`; sie funktionieren, indem sie die Eigenschaft ausnutzen, daß der Zugriff auf ein nicht verfügbares Datenobjekt den zugreifenden Prozeß blockiert.

[9] Manchmal orientieren sich Mappings auch am längsten Eingabestream; die anderen, d.h. vorher endenden, Streams werden entsprechend sinnvoll verlängert.

Eine typische mit Streams schwer formulierbare Funktion ist der `Merge`-Knoten. Will man ihn mit Streams programmieren, muß man beide Eingänge aufrufen, nur um an einem Eingang ein 'not-available'-Token zu erhalten. (Dies läuft auf 'busy waiting' hinaus.) Arbeitet man mit Pipes, so formuliert man die Bedingung der Suspension, die das Schlafenlegen und Aufwecken des Funktionsknotens erledigt, derart, daß das Eintreffen eines Tokens auf einer der beiden Pipes den Knoten weckt[10]:

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

1.2.7 Typen, Polymorphie und Overloading

ALDiSP verfügt über ein ausdrucksstarkes Typisierungssystem, das zwei Anforderungen genügt: Einerseits ist es mächtiger als die Typisierungskalküle herkömmlicher statisch typisierter polymorpher Sprachen (wie z.B. ML): derartige Sprachen haben Probleme mit "Polymorphie zweiter Ordnung", d.h. der Übergabe von polymorphen Funktionen, die noch nicht mit Typen instanziiert sind, als Funktionsargumenten; andererseits ist es expressiver als schwach oder nicht getypte Sprachen wie z.B. LISP, in denen der Begriff des Typfehlers stark verwässert ist[11].

Das Typsystem von ALDiSP läßt sich sowohl operational als auch mengentheoretisch verstehen. Der mengentheoretische Ansatz wird in der Sprachbeschreibung und bei der Spezifikation komplexer Typen benutzt, der operationale Ansatz in der "Alltags-Typisierung". Der mengentheoretische Ansatz geht davon aus, daß es ein Universum (das durch den Typen `Obj` modelliert wird) von konstruierbaren Objekten gibt, deren Struktur uns momentan egal sein kann. Wichtig ist, daß es ein ausgezeichnetes (Daten-) Element $\perp$ (Bottom)und primitive Funktionen gibt. Jede primitive Funktion ist über einer Teilmenge (des n-stelligen kartesischen Produkts) von `Obj` definiert und hat eine Teilmenge (des n-stelligen kartesischen Produkts) von `Obj` als Wertebereich. Jede primitive Funktion bildet alle Tupel aus dem Nicht-Definitionsbereich nach $\perp$ ab.

[10] Natürlich muß es in dieser Implementierung einen Scheduler-Prozeß geben, der das Warten erledigt, aber der kann z.B. Interrupts dafür einsetzen; es wäre ungleich schwerer, die hierfür notwendigen Informationen aus einem äquivalenten, durch Streams implementierten, Modell zur extrahieren.

[11] Gerade in Common LISP [13] ist es möglich, daß die automatische Typumwandlung zur Laufzeit viele echte Programmierfehler schluckt, d.h. teilweise vernünftige Ergebnisse produziert.

Typen sind jetzt nichts weiter als Teilmengen von `Obj`. Sinnvollerweise gibt es einige wichtige disjunkte Teilmengen (genannt Int, Real, Bool, Char, String, usw.), die als "atomar" angesehen werden und im Typsystem vordefiniert sind. Alle anderen Typen (z.B. List, Vector, Map) sind komplexer und bedürfen der Parametrisierung [12].

Die Teilmengen von `Obj` lassen sich am einfachsten durch Prädikate charakterisieren - denn was ist eine Teilmenge anderes als ein einstelliges Prädikat über einer Menge von Objekten? Nimmt man diesen Ansatz ernst, so gibt es keinen Grund, nicht auch 'bizarre' Typen zuzulassen, die sich durch 'konstruktive' Typdeklarationsmethoden (Deklaration von Konstruktoren, Selektoren und Äquivalenzgleichungen) nicht oder nur sehr umständlich aufbauen lassen.

Eine Typ*deklaration* für ein Objekt, einen Ausdruck oder einen formalen Parameter ist nicht, wie in maschinenorientierteren Sprachen (z.B. PASCAL und C), primär eine Speicherverwaltungsdirektive, sondern eine *Zusicherung* (Assertion), daß ein Ausdruck ein bestimmtes Prädikat erfüllt. Bsp[13] :

```
func fak(n:nat):nat =
  if n=0 then 1 else n*fak(n-1) endif
```

ist äquivalent zu

```
func fak(n) =
 if isNat(n)
 then
   let
      tmp = if n=0 then 1 else n*fak(n-1) endif
    in
      if    isNat(tmp)
      then  tmp
      else  bottom
      endif
   endlet
 else  bottom
 endif
```

Dies ermöglicht die direkte Einbindung von Assertions, wie sie in der Programmverifikation benutzt werden, denn Assertions sind nichts weiteres als Prädikate (bzw. Ausdrücke, von denen der Programmierer zusichert, daß sie sich in jedem Laufzeitkontext zu '`true`' evaluieren).

Das im letzten Abschnitt vorgestellte fortgeschrittene Typkonzept bedeutet in der Praxis (also dort, wo 24-Bit-Integers und 512 Elemente große Vektoren

[12] Typen lassen sich sowohl mit anderen Typen als auch mit Konstanten, also Werten, parametrisieren. Beispiele: `List(Integer)` oder `SizedVector(16,Byte)`.

[13] Das Programm benutzt die ALDiSP–Konvention, daß es zu jedem Typen XY ein Prädikat isXY gibt.

gezähmt werden müssen) keine Abkehr von gewohnten Notationen, die genausogut in ML oder Pascal vorkommen könnten. Diese Notationen entsprechen dem Teil der Typen, der sich 'statisch' beschreiben und ohne großen Aufwand implementieren läßt. Insbesondere läßt sich bei diesen einfachen Typen zur Übersetzungszeit die Typkorrektheit großer Programmteile nachweisen.

Polymorphie ergibt sich, wenn einem Ausdruck keine expliziten Typinformationen beigegeben werden. In diesem Fall gibt es keine Typchecks; die einzige Typinformation ist die, die durch bereits getypte Teile des Ausdrucks gewonnen werden kann (d.h. durch die Typen von Konstanten, Primitiven und bereits definierten Funktionen.). In ALDiSP ist Polymorphie der Normalfall: jede Funktion ist polymorph, es sei denn, sie wird getypt.

Overloading, d.h. das Unterbringen mehrerer Funktionen unter einem Funktionsnamen, ist eine einfache Konsequenz der Bottom-Semantik von Primitiven. Wenn es zwei Funktionen gibt, deren Definitionsbereiche disjunkt sind, kann man die beiden Funktionen problemlos kombinieren (Wenn man sich eine Funktion als Tupelmenge vorstellt, ist die Kombination zweier Funktionen durch die Mengenvereinigung definiert. Solange das Ergebnis keine echte Relation ist, ist die Semantik eindeutig definiert.). Bei der Anwendung der überladenen Funktion wird, je nach Argumenttyp, eine der Ausgangsfunktionen angewandt.

In ALDiSP bedeutet dies, daß zwei oder mehr Funktionen unter dem gleichen Namen definiert werden können, solange sich keine Überschneidungen in den Definitionsbereichen ergeben. Dies wird durch ein einfaches Schlüsselword '`overloaded`' bei der Funktionsdefinition bewerkstelligt.

1.2.8 Verzögerte Auswertung

Das `delay`–Konstrukt ermöglicht es, die Auswertung eines Ausdrucks bis zu dem Zeitpunkt herauszuzögern, an dem der Wert des Ausdrucks wirklich benötigt wird. Dies macht es dem Programmierer möglich, Funktionen über unendlich großen Objekten (insbesondere *Streams*) zu spezifizieren (die dann in der Anwendung nur endliche Teil der Objekte verarbeiten).

1.2.9 Exceptions

ALDiSP stellt einen allgemeinen Exception-Mechanismus zur Verfügung, der durch dynamisch gebundene Funktionen, wie sie aus dem klassischen LISP (1.5) bekannt sind, implementiert wird. Diese Exceptions dienen nicht nur dem Abfangen von Fehlersituationen, sondern werden auch als Hintergrund für den parametrisierbaren Rundungsmechanismus eingesetzt.

1.2.10 Syntaktische Elemente

ALDiSP verfügt über die Möglichkeiten, *Operatoren* zu definieren. Dies ermöglicht klammerfreie Funktionsaufrufe. Im weiteren gibt es *Makro*–Funktionen,

die zur Übersetzunszeit expandiert werden und es ermöglichen, auch von solchen Ausdrücken zu abstrahieren, die verzögerte auszuwertende Sub-Ausdrücke enthalten.

1.3 Aufbau der Sprachbeschreibung

Die ALDiSP-Sprachbeschreibung ist in einen Report und einen Satz von Syntaxtabellen aufgeteilt. Zur weiteren Erläuterung gibt es eine Kurzbeschreibung der Entwicklungsumgebung und einen 'Rationals'-Teil, der alles enthält, was anderswo nicht untergebracht werden konnte.

1.3.1 Der Report

Der Report ist eine Sprachbeschreibung, die eine nicht-formale[14] Einführung in ALDiSP gibt, ohne dabei Anspruch auf die Vollständigkeit oder Präzision einer formalen Spezifikation zu erheben (obwohl er sich darum bemüht). Der Report erklärt alle syntaktischen Konstrukte, die elementaren Typen und die darauf definierten Funktionen. Das Wissen aus dem Report (und den Syntaxdiagrammen) muß ausreichen, um ALDiSP-Programme schreiben zu können. Report und Syntaxdiagramme sind die Teile des Reports, die für "den Anwender" gedacht sind. Alles, was im Report nicht oder nur unklar erklärt ist, sollte vom Benutzer als das genommen werden, was es ist – eine Spezifikationslücke, in der sich verschiedene Implementierungen der Sprache unterscheiden können. Einige dieser Lücken sind beabsichtigt, um den Implementoren Freiräume zu geben.

1.3.2 Die Entwicklungsumgebung

Dieser Abschnitt der Sprachbeschreibung erklärt die minimale Umgebung, die ein ALDiSP-System braucht. Er geht kurz auf CADiSP ein, um schließlich sinnvolle Compiler-Optionen aufzuzählen und die Unterschiede in der Semantik zwischen interpretiertem und compiliertem ALDiSP noch einmal aufzuarbeiten.

1.3.3 Die Rationals

Dieser Abschnitt geht näher auf umstrittene Entwurfsentscheidungen, kontraintuitive Aspekte und Implementierungsprobleme von ALDiSP ein. Alles, was in den anderen Abschnitten nur verwirren würde, ist hier untergebracht, zudem werden viele Einzelaspekte erläutert.

[14] Ich weigere mich, den Report 'umgangssprachlich' oder auch nur 'natürlichsprachlich' zu nennen - das Deutsch des Reports ist nicht 'natürlich'!

1.3.4 Die Anhänge

Es gibt zwei Syntax-Beschreibungen von ALDiSP. Die erste benutzt eine EBNF-Notation und ist nicht ganz vollständig (d.h., sie beschreibt ein Superset von ALDiSP); sie ist relativ gut lesbar und "für den Hausgebrauch" ausreichend. Die zweite ist eine yacc-Syntax, die alle Fragen von Operatorprorität und Assoziativität in ausreichender Weise regelt. Es folgen zwei Anhänge zum leidigen Thema 'primitive Funktionen und Typen'. Leidig ist das Thema deshalb, weil vollkommen unklar ist, wie groß der "funktionale Kern" einer Sprache zu sein hat. Manche Sprachen sehen keinen solchen Kern vor (was zu total inkompatiblen Programmen führt), andere haben einen aufgeblähten Satz von vordefinierten Funktionen, den vollständig zu implementieren eine größere Leistung ist (z.B. Common LISP). ALDiSP versucht, sich aus der Verantwortung zu drücken, indem es zwei "Kerne" vorgibt: einen minimalen, der in allen Implementierungen realisiert sein muß, und einen Satz vorgeschlagener Erweiterungen, die nicht Pflicht sind (wenn sie jedoch implementiert werden, dann mit den hier angegebenen Namen und Funktionalitäten).

Kapitel 2

ALDiSP Report

2.1 Syntaktische Form von Programmen

Ein ALDiSP–Programm ist ein (möglicherweise auf mehrere Files aufgeteilter) Text. Der zugrundeliegende Zeichensatz muß mindestens die sichtbaren ASCII–Zeichen umfassen; es ist von Vorteil, einige mathematische Sonderzeichen zur Verfügung zu haben. Jede Implementierung kann eine maximale Zeilenlänge vorgeben, die jedoch nicht kleiner als 80 sein darf. Ein Programm besteht aus einer Sequenz von Namen und Literalen, die durch Zwischenraumzeichen, Interpunktionszeichen und 'spezielle Namen' getrennt sind.

Kommentare

ALDiSP kennt nur zeilenorientierte Kommentare. Alles, was zwischen einem "\"-Zeichen (Backslash) und einem Zeilenende kommt (exklusive), wird ignoriert. Die Programme

```
net
 foo=19
endnet
```

und

```
net\ein kommentar kann beliebig nahe an ein token
foo\ herangehen.
\
=19 endnet \
```

sind, wenn auch von stark unterschiedlicher Lesbarkeit, gleichwertig. Da das Zeilenende nicht ignoriert wird, hat ein Kommentar die syntaktische Funktion eines Leerzeichens.

Reservierte Wörter

Es gibt eine Reihe reservierter Worte. Es sind dies die in syntaktischen Konstrukten eingesetzten Worte {**as func if delay else endguard endif endlet endmodule endnet endseq exception exports guard import in postfix let macro module net on overloaded then prefix proc seq suspend type until where within** }.

Prinzipiell gilt, daß reservierte Namen nicht für vom Benutzer definierte Objekte benutzt werden können. Die reservierten Wörter haben die Eigenschaft, daß bei ihnen nicht zwischen Groß- und Kleinschrift unterschieden wird ('`if`' = '`IF`' = '`If`' = '`iF`'), währnend bei 'normalen' Symbolen auf die Unterscheidung geachtet wird[1], d.h. '`bottom`' ≠ '`Bottom`'.

Spezielle Namen

'Spezielle Namen' sind aus nicht-alphanumerischen Zeichen zusammengesetzte Zeichensequenzen (Typische Exemplare sind `+`, `-^`, `->` und `::`). Eine Sequenz der Form 'Name, spezieller Name, Name' muß nicht durch Leerzeichen getrennt werden! Dies ermöglicht es, '`a+b`' statt '`a + b`' zu schreiben.

Zwischenräume

Als Zwischenraum (auch 'whitespace' oder 'atmosphere' genannt) dienen das Leerzeichen (BL), der Tabulator (TAB), das Zeilenende-Zeichen (CR), das Zeilenvorschub-Zeichen (LF) und das Seitenvorschub-Zeichen (FF).

Interpunktion

Als Interpunktionszeichen dienen die runden Klammern, die eckigen Klammern, die geschweiften Klammern, das Komma, der Punkt (in Dezimalzahl-Literalen), das Semikolon, der Doppelpunkt und das Gleichheitszeichen.

Alle anderen Symbole können in 'Special Identifier'n frei verwendet werden.

Grobaufbau eines Programms

Ein Programm ist eine Reihe von Deklarationen, denen eine in die Schlüsselworte `net` und `endnet` geschlossene Netzbeschreibung folgt. Diese Netzbeschreibung nimmt normalerweise die syntaktische Form einer Menge lokaler Variablendefinitionen an, kann aber auch aus einer einzigen Expression bestehen.

[1] Siehe hierzu auch 4.3.

2.2 Sprachkonstrukte

2.2.1 Literale

ALDiSP kennt Literale der Typen Zahl, String und Character.

Numerische Literale

Es gibt zwei Formen numerischer Literale: *ganze Zahlen* (Integers) und *Dezimalzahlen* (Reals). Per Default werden alle Zahlen als im Dezimalsystem notiert interpretiert; ganze Zahlen sind zu anderen Basen (zwischen 2 und 16) notierbar. Dies hat dann die Form *Basis*#*zahl*.

```
\ Beispiele f\"ur korrekte ganze Zahlen
0 1 00001 42 -42
898982478374875583535090893 5
16#ffe01 8#12763 2#00101101
```

Dezimalzahlen sind Zahlen mit Dezimalpunkt und optionalem Exponenten. Exponenten sind ganze Zahlen. Jeder Teil einer Dezimalzahl ist im Dezimalsystem notiert.

```
\ Beispiele f\"ur korrekte Dezimalzahlen
13.5
13.0 e -13
13.0 E9284235
-124141.141412e9
\ Beispiele f\"ur fehlerhafte Dezimalzahlen
13. e19
13 e19
16#1ef.2c3 16#1ef.16#2c3
.4
12.3 e 3.4
12,23
```

Literale sind vom Typ `Integer` bzw. `Real`. Will man Literale eines anderen Typs notieren, ist eine explizite Typumwandlung vorzunehmen.

String-Literale

Ein Literal vom Typ String hat die Form "*string*". Der String darf sichtbare Zeichen und Blanks enthalten, auch das Kommentarzeichen.

```
\ Beispiele f\"ur korrekte String-Literale.
"aha"
"jhgjsghsdhgDH\JFg\jksdhk\"O\"A\"U$%//$3452356/$%&!!'#'=?`^"
""      \ der leere String
```

2.2.2 Variablen

Jeder Name[2], der kein reserviertes Wort ist, denotiert eine Variable. Es gibt keine Unterscheidung zwischen Variablen, die Typen, Funktionen und 'normale' Werte denotieren. Syntaktisch gibt es zwei Arten von Namen: normale und spezielle.

identifier	$\longmapsto$	normalId | specialId
normalId	$\longmapsto$	Alpha AlphaNum*
specialId	$\longmapsto$	Special Special*
Alpha	$\longmapsto$	"`A-Z a-z _`" und lokale Umlaute
AlphaNum	$\longmapsto$	Alpha | "`0123456789`"
Special	$\longmapsto$	"`!$%&*@/?'^+_-~`" und verfügbare mathematische Symbole

```
\ einige syntaktisch korrekte Namen
a _b char2int
djhHKGhjdtfhJKHJfddf2635JSHFjdf__dsfjhdf_234
+ * ^ +#^ ? ! ++*+*+++ ?!?!

\ einige syntaktisch falsche Namen
number? set! C++ Modula-2 23skidoo
```

Will man, z.B. bei der Übertragung von Sprachen aus anderen Sprachen, Namen benutzen, die diesen Konventionen nicht genügen, kann man diese in Hochkommata setzen.

```
\ gueltige Namen sind
'a+b' 'set!' 'name mit leerzeichen' 'punk.tiert' '>?'
```

Das `'>?'`-Beispiel ist interresant, da es sich hierbei eigentlich um einen legitimen speziellen Namen handelt. Da aber `'>'` und `'?'` vordefinierte Infix- bzw. Präfix-Operatoren sind, würde der Parser `'>?'` in zwei Token aufzutrennen versuchen! Es gilt: der Parser versucht, von links nach rechts den größten Subnamen zu finden. Beispiel:

```
\ seien als operatoren definiert: + ++ * ** +**
+++      \ -> + ++
++***    \ -> ++ ** *
++++**   \ -> + ++ ** *, nicht ++ +** * !
```

[2] "Name" ≡ "Symbol" ≡ "Identifier"

Natürlich ist es stilistisch sehr schlecht, derartig obskure Operatoren überhaupt erst einzuführen! Sind derartige Operator-Ansammlungen unvermeidlich (z.B. bei Kombinationen von Funktionalen), sollten sie auf jeden Fall durch Leerzeichen getrennt werden, wenn die einzelnen Operatoren nicht "herausragend" genug sind.

Parser-Anweisungen

Parser-Deklarationen verändern den Status von Symbolen, die vom Parser, d.h. von der Syntaxphase des ALDiSP-Compilers, gelesen werden. In dieser Phase haben Symbole (Namen) noch keine Bedeutung; der Parser kann nur zwischen normalen und speziellen Symbolen und reservierten Worten (wie z.B. `if`) unterscheiden. Durch Parseranweisungen können Namen als Infix, Präfix oder Postfix deklariert werden. Dies ermöglicht gewohnte Notationen wie

```
a+b                 \ '+' infix
n!                  \ '!' postfix
sin 3               \ 'sin' pr\"afix
x elementOf set     \ 'elementOf' infix
```

Die ensprechenden Deklarationen müssen vor der erstmaligen Benutzung erfolgen. Wenn ein Name einmal als In/Prä/Postfix deklariert worden ist, kann er nicht mehr anders benutzt werden!

```
func elementOf(x,set) = ...definition...
... foo elementOf bar .... \ falsch
... elementOf(foo,bar) ... \ korrekt
INFIX "elementOf"
... foo elementOf bar .... \ korrekt
... elementOf(foo,bar) ... \ falsch
```

Jeder Name kann in Hochkommata gestellt werden. Dabei verliert er spezielle Parser-Attribute. In/Prä/Postfix–Notation kann auch in Funktionsdeklarationen auftreten!

```
POSTFIX "!"                     \ oder
FUNC n! =                       \ FUNC '!'(n) =
if n=0 then 1 else n*(n-1)!     \ if n=0 then 1 else n*'!'(n-1)
```

Wenn mehrere Parser-Deklarationen aufeinander treffen, kann es zu Konflikten kommen. Aus diesem Grunde gibt es Bindungsstärken und Assoziativitätsregeln. Die meisten Infix-Operatoren sind links-assoziativ; nur wenige (z.B. '::', das Infix-Cons) sind rechts-assoziativ. Es gibt 5 Bindungsstärken (1-5), 5 bindet am stärksten. Die Angabe von Bindungsstärken bei der Parserdeklaration ist optional, als Voreinstellung wird 3 angenommen. Die Standard-Funktionsanwendung bindet stärker als Postfix, Postfix bindet stärker als Präfix, Präfix bindet stärker

als Infix. Gleichstarke links- und rechtsassoziative Operatoren dürfen nicht gemischt werden.

```
INFIX "and","or" :1
INFIX "=",">","<":2
INFIX "+","-"    :3
INFIX "/","*"    :4
INFIX "^"        :5 \ Potenzierung

3>4 and a^b+c*d=42  \ (3>4) and (((a^b)+(c*d)) = 42)

POSTFIX "!"
PREFIX "sin"

sin 42 !            \ sin (42 !)
sin 3+4             \ (sin 3)+4
sin a(b)            \ sin (a(b))
sin a!(b)           \ sin ((42 !)(b))

R-INFIX "::" : 3

a::b::nil           \ a::(b::nil)
a-b-c               \ (a-b)-c
a-b-c::d::nil       \ Syntax Error
```

Parser-Deklarationen sind lokal zum File. Es gibt die Parser-Deklarationen `L-INFIX` (links-assoziativ infix), `R-INFIX` (rechts-assoziativ infix), `INFIX` (dasselbe wie `L-INFIX`), `PREFIX` und `POSTFIX`. Nur die Infix-Deklarationen können mit Bindungsstärken versehen werden. Nur die Ziffern '1','2','3','4' und '5' sind gültige Bindungsstärken. Die Namen der definierten Operatoren sind in Anführungszeichen zu setzen, wenn es sich um spezielle Namen handelt; bei normalen Bezeichnern sind die Anführungszeichen optional. Dies gilt auch für die Auflistung von Operatoren in Im- und Exportlisten.

2.2.3 Funktionsapplikation

Die Standard-Form der Funktionsanwendung hat die Form

$$expr(arg_1, \ \ldots \ , arg_n)$$

Die Semantik ist einfach: die *expr* und die arg_i werden in nicht festgelegter Reihenfolge, möglicherweise auch parallel, ausgewertet. Ist dies geschehen, wird das Ergebnis der Auswertung von *expr* auf die Argumente angewandt. Ist eine Anwendung nicht möglich (weil *expr* sich zu keiner Funktion, zu einer Funktion der falschen Stelligkeit oder eines falschen Typs ausgwertet hat), liegt eine Fehlersituation vor; das Ergebnis der Funktionsapplikation ist `bottom`.

Syntaktisch andere Formen der Funktionsapplikation (In/Prä/Postfix-Operatoren) verhalten sich semantisch nicht anders. Man kann sich vorstellen, daß ein komplexer Ausdruck mit nicht-Standard-Applikationen vom Compiler in einen Standard-Ausdruck umgewandelt wird, der dann zur Laufzeit ausgeführt wird:

`sin a+b!*42` $\Longrightarrow$ `'+'(sin(a),('*'('!'(b),42))`

Im Gegensatz zu anderen Sprachen sind die logischen Operatoren `and` und `or` normale Funktionen, deren Argumente alle ausgewertet werden.

2.2.4 Sequenzen

Eine Sequenz hat die Form

`seq` *stmt*$_1$ `;` ... *stmt*$_n$ `;` *expr* `endseq`

Eine Sequenz wird ausgewertet, indem die *stmts* der Reihe nach, von links nach rechts, ausgeführt werden; am Ende wird die *expr* berechnet und ihr Resultat als Ergebnis der Sequenzauswertung zurückgegeben. Eine Sequenz ist ein syntaktischer Block, d.h. innerhalb der Sequenz definierte Namen sind außerhalb der Sequenz nicht sichtbar. Sequenzen werden eingesetzt, um Nebeneffekte zu erzeugen:

```
proc readFileObj(name: String): Obj =
 seq
   fd=openFile(name);
   (obj,fd) =readFile(fd);
   closeFile(fd);
   obj
 endseq
```

Der Einsatz von Sequenzen ist dort notwendig, wo eine Auswertungsreihenfolge vorgegeben oder Parallelität verhindert werden muß. Dies sollte nur bei der Erzeugung von Nebeneffekten der Fall sein. Wie die *Statements*, die in der Sequenz auftreten, aussehen, erklärt der nächste Abschnitt.

Wertet sich ein Statement in einer Sequenz zu einer Suspension aus, so blockiert die Auswertung der Sequenz, bis die Suspension ausgewertet ist. Es gilt die Transformationsregel

`seq` *stmt* `;` *...rest...* `endseq`
$\equiv$ `suspend seq` *...rest...* `endseq until` *stmt* `within 0,0`

2.2.5 Statements

Es gibt zwei Arten von Statements: Prozeduraufrufe und sequentielle Definitionen. Letzere lassen sich durch 'normale', geschachtelte lokale Definitionen beschreiben; erstere sind nebeneffektbehaftete Expressions, können also auch außerhalb von Sequenzen auftreten.

Prozeduraufrufe

Der Aufruf einer Prozedur unterscheidet sich nicht vom normalen Funktionsaufruf. Es wird empfohlen, keine Prozeduren als In/Prä/Postfix-Operatoren zu vereinbaren und die Nebeneffektbehaftetheit der Prozeduren durch sinnfällige Namen (wie `Set`*XXX* oder `assign`*XXX* oder `Do`*XXX*) deutlich zu machen.

Sequentielle Definitionen

Eine einfache sequentielle Definition hat die Form *name* `=` *stmt*. Die Definition wird ausgeführt, indem das *stmt* ausgewertet wird und der entstehende Wert im aktuellen Environment an den Namen gebunden wird. Der Gültigkeitsbereich von Namen in Sequenzen ist die Restsequenz rechts von der Definition des Namens: In der Sequenz

```
seq fd=OpenOutputFile("bla",Byte);
    fd=writeToFile(fd,42);
    closeFile(fd)
endseq
```

erstreckt sich der Gültigkeitsbereich des ersten `fd` vom zweiten Statement bis zum Ende; im zweiten Statement wird ein neues `fd` definiert, das vom dritten Statement bis zum Ende gilt und das erste `fd` überdeckt. Aufgrund dieser Sichtbarkeitsregel ist es nicht möglich, mit sequentiellen Definitionen Rekursivität zu erreichen[3]. Sequentielle Definition werden benutzt, um die Ergebnisse von Prozeduraufrufen weiterverwenden zu können.

2.2.6 Das Konditional

Die einfachste Form des Konditionals ist die Expression der Form

$$\texttt{if } expr_{test} \texttt{ then } expr_{then} \texttt{ endif}$$

Bei der Auswertung dieses Ausdrucks wird zuerst $expr_{test}$ ausgewertet; ist das Ergebnis gleich dem Objekt `true`, so wird $expr_{then}$ ausgewertet und das Resultat als Ergebnis der Auswertung des Konditionals zurückgegeben. Wenn sich $expr_{test}$ zu `false` auswertet, ist das Ergebnis `bottom`. Wenn sich $expr_{test}$ zu etwas anderem als `true` oder `false` auswertet, liegt eine Fehlersituation vor; das Ergebnis ist `bottom`.

Die zweiteinfachste Form des Konditionals ist

$$\texttt{if } expr_{test} \texttt{ then } expr_{then} \texttt{ else } expr_{else} \texttt{ endif}$$

Die Auswertung ist dieselbe wie die im vorigen Fall; wenn sich allerdings $expr_{test}$ zu `false` evaluiert, wird $expr_{else}$ evaluiert und das Resultat als Ergebnis des Konditionals zurückgegeben.

Die allgemeine Form des Konditionals ist

[3] Wir wollen hier die Möglichkeit des Y-Kombinators außer acht lassen.

```
if expr_test1 then expr_then1
|  expr_test2 then expr_then2
           ...
|  expr_testn then expr_thenn
           else expr_thenn+1
endif.
```

Bei der Auswertung dieses Ausdrucks werden die $expr_{test_i}$ nacheinander ausgewertet, solange sich alle zu `false` auswerten, bis sich eins zu `true` evaluiert. Evaluiert ein $expr_{test_i}$ sich vorher zu einem Wert ungleich `false`, liegt eine Fehlersituation vor, und die Auswertung terminiert mit `bottom` als Ergebnis. Hat sich $expr_{test_k}$ zu `true` evaluiert oder gibt es nur $k-1$ *test*-Klauseln, aber eine *else*-Klausel, so wird $expr_{then_k}$ ausgewertet und das Ergebnis als Resultat des Konditionals zurückgegeben.

2.2.7 Die Abstraktion

Eine *Abstraktion* ist ein Ausdruck, von dem *abstrahiert* wird, d.h. der *parametrisiert* ist. Jeder Ausdruck kann als *Körper* einer Abstraktion auftreten. Das Ergebnis einer Abstraktion ist eine *Funktion*[4]. Ein *Header* definiert den Namen der Funktion und die Parameter. Optional können die Typen der Parameter und der Typ des Funktionsergebnisses genannt werden.

Eine Standard-Funktionsdefinition hat die Form

$$\texttt{func}\ name_0\texttt{(}\ name_1\texttt{,}\ \ldots\ \texttt{,}\ name_n\texttt{)} = expr.$$

Jeder der Namen $name_1$, ... , $name_n$ kann mit einer Typdeklaration der Form $:typ$ versehen werden.

Eine Funktionsdefinition wird ausgeführt, indem ein Objekt vom Typ `Function` erzeugt wird. Wenn dieses Objekt später im Rahmen einer Funktionsapplikation auf n Argumente arg_1 ... arg_n angewandt wird (nachdem diese ausgewertet sind), wird die *expr* in einem Environment ausgewertet, das dem Environment an der Stelle des Programms, an dem das Funktionsobjekt erzeugt worden ist, erweitert um das Environment, in dem jeder $name_i$ mit dem entsprechenden arg_i assoziiert ist, entspricht.

Dies entspricht den 'normalen' Scope-Regeln, wie sie z.B. aus PASCAL und C bekannt sind.

Wenn Parameternamen mit Typen attributiert werden, so verändert dies die Semantik des entstehenden Funktions/Prozedurobjekts dahingehend, daß der Definitionsbereich der Funktion/Prozedur für dieses Argument auf den entsprechenden Typen eingeschränkt wird, d.h. jeder Aufruf mit einem Argument "falschen" Typs eine Fehlersituation darstellt und `bottom` als Resultat liefert.

Eine syntaktische Variante der Typisierung der Einzelparameter ist die Typisierung der ganzen Funktion/Prozedur: nach der schließenden Klammer der

[4]Oder ein parametrischer Typ oder eine Prozedur.

Paramterliste kann ein : *typ* stehen, das die Typisierung des Resultats angibt. Die Typisierung des Resultats bringt eine der Typisierung von Argumenten ähnliche Semantik mit sich: resultiert eine Applikation des Funktions/Prozedurobjekts in einem Wert falschen Typs, so liegt eine Fehlersituation vor; der resultierende Wert ist `bottom`.

Weitere syntaktische Varianten von Funktionsdefinitionen sind Prä-, Post- und Infix-Operatoren, die auch in Funktionsdefinition so geschrieben werden müssen wie in der Applikation: statt

```
func '+'(a,b)=
  callAddRoutine(a,b)
func '!'(n)=
  if n=0 then 1 else n*(n-1)! endif
func 'cos'(x)=
  sin(x-pi/2)
```

kann es

```
func a+b =
  callAddRoutine(a,b)
func n! =
  if n=0 then 1 else n*(n-1)! endif
func cos x =
  sin(x-pi/2)
```

heißen. Will man dies dann typen, muss man den Typen der Funktion als Ganzes angeben. Die typisierten Varianten der o.a. Beispiele könnte so aussehen:

```
func a + b : (number,number -> number) =
  callAddRoutine(a,b)
func n! : (nat -> nat) =
  if n=0 then 1 else n*(n-1)! endif
func cos x : (real -> real) =
  sin(x-pi/2)
```

Funktionen höherer Ordnung

Eine Funktion *höherer Ordnung* (*higher-order function*) ist eine Funktion, deren Resultat eine Funktion ist [5]. ALDiSP erlaubt die Definition von Funktionen beliebiger Ordnung. Ein Beispiel ist die Kompositionsfunktion $f \circ g \equiv \lambda\, x.f(g(x))$:

```
func comp(f,g) =
```

[5] Präzise: eine Funktion, die eine Funktion der n-ten Ordnung zum Ergebnis hat, ist selbst von der Ordnung n+1. Daten sind 'Funktionen der Ordnung 0'. Aber so genau nimmt man es im Allgemeinen nicht; eine Sprache erlaubt entweder gar keine Funktionen höherer Ordnung, oder alle.

```
let
  func tmp(x)=
    f(g(x))
in
  tmp
endlet
```

Hier wird eine Funktion (`tmp`) erzeugt und als Resultat zurückgegeben. Eine äquivalente Formulierung erspart die, eigentlich unnötige, Benennung der Funktion:

```
func comp(f,g)(x) = f(g(x))
```

Es ist in ALDiSP nicht möglich, Funktionen variabler Stelligkeit zu notieren; daher kann keine allgemeine Kompositionsfunktion definiert werden.[6]

2.2.8 Parametrische Typen

Ein parametrischer Typ ist eine Funktion, die einen Typen zum Resultat hat. Typwertige Funktionen werden statt des des Schlüsselworts `func` mit `type` definiert. (Unter der operationalen Typ-Interpretation ist dies nicht notwendig, aber faktisch unterliegen Typterme eigenen Bildungsregeln. Auch ist der Dokumentationswert nicht zu unterschätzen.

```
type IntegerBetween(l,h)(x) =
 isInteger(x) and x>=l and x<h
```

Auch einfache Typgleichungen sind mit `type` einzuleiten:

```
type OddIntegers = Integer and isOdd
```

Bei diesen Beispielen zeigt sich wieder die Gleichwertigkeit von einstelligen Prädikaten und Typen.

2.2.9 Prozeduren

Eine *Prozedur* ist eine Funktion, die Nebeneffekte erzeugen kann. Eine Prozedur wird wie eine Funktion definiert, aber mit dem Schlüsselwort `proc` statt `func`. Auch der Aufruf erfolgt auf gleiche Weise. Beispiel:

[6] Es ist aber klar, wie eine Erweiterung aussähe: Man läßt einfach die Klammern bei der Applikation weg; eine Variable (bzw. eine Expression) steht dann für die ganze formale (bzw. aktuelle) Parameterliste: `func comp(f,g) x = f g x` Das syntaktische Problem dabei ist die Unterscheidung zwischen dem 1-Tupel und seinem Inhalt. `f(g x)` impliziert, das `g` genau einen Wert als Resultat liefert, kann aber auch als einfache Klammerung gesehen werden! Um diesen Problemen zu entkommen (und weil die Struktur eines Programms nicht gerade lesbarer wird, wenn mit variabelstelligen Funktionen gearbeitet wird), ist diese Erweiterung *nicht* übernommen worden.

```
\ Diese Prozedur schreibt einen Word-Wert byteweise in falscher
\ Reihenfolge in einen Port.
proc writeIntelWord(P:Port, V:Word) =
seq
  WriteToPort(P,[!Byte](V and 16#ff));
  WriteToPort(P,[!Byte](rshift(V, 8) and 16#ff));
endseq
```

2.2.10 Lokale Variablen

Ein Konstrukt der Form

```
let var_1 = expr_1
        ⋮
    var_n = expr_n
        in
    expr
endlet
```

definiert n lokale Variablen. Diese Deklaration wird ausgewertet, indem die n $expr_i$ in unbestimmter Reihenfolge, u.U. auch simultan, evaluiert werden und an die n var_i gebunden werden. Hierbei sind rekursive Funktionsdefinitionen möglich (Funktionen werden über das normalen `func`-Schlüsselwort deklariert), d.h. die var_i sind während der Evaluierung der *expr* sichtbar, aber mit unbestimmten Werten belegt.

Die Bindung der Variablen findet in einem neuen Environment statt und hat keinen Einfluß auf das Environment, in dem die lokale Definition stattfindet (d.h., ein neuer Scope wird aufgemacht); in diesem Environment wird dann die *expr* ausgewertet und ihr Wert als Ergebnis der lokalen Definition zurückgegeben.

Eine syntaktisch andere, aber semantisch äquivalente Ausprägung der lokalen Variablendefinition ist

```
let expr
        where
    var_1 = expr_1
        ⋮
    var_n = expr_n
endlet
```

Oft ist diese Formulierung lesbarer.

Die lokalen Variablen sind (trotz ihres Namens) keine 'echten' Variablen, deren Wert geändert werden kann, sondern 'single-assignment' Namen, also

'temporäre Namen' für Ausdrücke. Es liegt keine Verletzung der referentiellen Transparenz vor![7]

2.2.11 Multiple Rückgabewerte

Funktionen (eigentlich alle Ausdrücke) können mehr als einen Wert zum Resultat haben. Das Konstrukt

$$(expr_1 \ , \ \ldots \ , expr_n)$$

berechnet die n $expr_i$ in unspezifizierter Reihenfolge (oder auch parallel) und bildet einen n-stelligen multiplen Wert. Das einzige, was mit einem solchen multiplen Wert gemacht werden kann, ist die Rückgabe als Funktionswert und die Benutzung in einer n-stelligen Variablendeklaration. Es ist weder möglich, multiple Rückgabewerte direkt in Funktionsapplikationen einzusetzen, noch sie in Einzelvariablen zu speichern. Ein n-stelliger Wert muß in einem Schritt n Variablen zugewiesen werden!
Bsp:

```
(x,y) = (3,4)            \ ok

func foo(n)=
 (n+1,n-1)

(x,y)=foo(3)     \ x=4,y=2

z=foo(3)         \ Syntax Error

(a,b,c)=foo(3)   \ Syntax Error

func diff(a,b)
  abs(a-b)

diff(foo(3))     \ Syntax Error
```

2.2.12 Module

ALDiSP-Programme sind in unterschiedliche Module aufgeteilt. Ein Modul kapselt eine Reihe von Variablen- und Funktionsdeklarationen von der Umgebung ab. Jedes Modul verfügt über eine Schnittstelle in Form von `Import`- und `Export`-Deklarationen, die festlegen, welche Definitionen aus welchen anderen Module benutzt werden und welche Definitionen das Modul exportiert, d.h. öffentlich macht.

Die syntaktische Form eines Moduls ist

[7] Diese Erläuterung erfolgt auf besonderen Wunsch des Betreuers.

Module *ModulName*
Exports $name_{1_1}$, ... , $name_{1_i}$
⋮
Exports $name_{j_1}$ ... $name_{j_k}$
...Deklarationen...
EndModule *ModulName*

Innerhalb der Deklarationen können Importe auftreten (genauer: Importe sind Deklarationen). Eine Import-Deklaration hat die Form

Import $name_1$, ..., $name_n$ **From** *ModulName*

und ist gleichwertig mit Standard-Deklarationsform

$name_1$ = *ModulName*("$name_1$")
⋮
$name_n$ = *ModulName*("$name_n$")

Natürlich ist dieser "qualifizierte Zugriff" auch in einer Expression möglich. Aus Übersichtlichkeitsgründen wird dennoch empfohlen, die Standard Import/-Export-Deklarationen zu benutzen; sie sind mächtig genug und vergrößern die Übersichtlichkeit.

Sowohl in den Import- als auch in den Export-Listen können optionale Typdeklarationen der Form *:typ* und Umbenennungen der Form **as** *NeuerName* gemacht werden. Ersteres wird empfohlen, um die 'Signatur' der im- und exportierten Funktionen klar zu machen; außerdem ermöglicht es die so gewonnene Redundanz dem Compiler, mehr Fehler festzustellen und stärker zu optimieren. Letzteres ermöglicht es, Namensüberschneidungen zu vermeiden. Beispielfragment:

```
Module LongInt
 exports + as LongIntAdd: (LongInt,LongInt -> LongInt)
```

Module sind reine Deklarationsträger; in ihnen darf es keine imperativen Teile geben. So kann ein Modul wie `ScreenIO` kein bereits geöffnetes File wie `keyboard: InputFile` exportieren – das Öffnen eines Files ist ein Nebeneffekt.
reminder:
Das ändert sich vielleicht noch!

Die Deklarationen erfolgen 'parallel', d.h. es Rekursivität ist erlaubt. Module dürfen zwar Prozeduren definieren und exportieren, während der Modulauswertung dürfen aber keine Nebeneffekte auftreten.

Module entsprechen grob Files – ein Modul kann nicht auf mehrere Files aufgeteilt werden; meist gibt es daher eine 1:1-Relation zwischen Modulen und Files. Es ist aber auch möglich, mehrere Module (oder auch das ganze Programm) in ein File zu schreiben. Mehr dazu im Abschnitt 2.4. Der Name `main` ist als Modulname nicht erlaubt. Module sollten nicht geschachtelt werden.

2.2.13 Ausnahmefall-Vorbereitung

Eine *Ausnahmefall-Vorbereitung* (*exception handler*) ist ein Konstrukt, das erwartete Fehlersituationen handzuhaben in der Lage ist. Die grundlegende Idee dabei ist die des direkten Verlassens einer Berechnung, wenn ein Fehler auftritt. Das typische Beispiel für einen Fehler ist die Division durch Null: Wenn mit interaktiv eingegebenen Daten gearbeitet wird, sind fehlerhafte Daten aller Art jederzeit möglich. Es ist sehr aufwendig, alle Daten vor ihrer Verwendung auf ihre Korrektheit hin zu überprüfen. Sinnvoller scheint es, einen Mechanismus zu konstruieren, der dafür sorgt, daß, wenn ein Datensatz zu Fehlern führt, er neu eingelesen wird. Das sieht dann so aus:

```
proc do_interaktion() =
 seq
  daten = read_datensatz();
  guard verarbeite_datensatz()
     on DivisionByZero(n) =     \ im Lauf der Berechnung
        seq                     \ wurde n durch 0 geteilt.
          writeMessage("Division durch Null: "
                    ++ "Bitte Daten neu eingeben!");
          do_interaktion()
        endseq
     on NumericOverflow() =
        seq
          writeMessage("Numerischer \"Uberlauf: "
                    ++ "Bitte Daten neu eingeben!");
          do_interaktion()
        endseq
  endguard
```

Die Exceptions `DivisionByZero` und `NumericOverflow` sind vordefiniert (Sie werde im Pseudo-Modul `Numbers` definiert) und werden von numerischen Funktionen im Fehlerfall aufgerufen.

Das Konstrukt hat die allgemeine Form

`guard` *expr*
`on` *Exception-Definition*$_1$
⋮
`on` *Exception-Definition*$_n$
`enguard`

Jede *Exception-Definition*$_i$ ist dabei syntaktisch eine Prozedurdefinition, allerdings ohne der das Schlüsselwort `proc`. Die Prozeduren müssen verschiedenen Namens sein. Es ist *nicht* erforderlich, die Namen vorhandener Exceptions zu benutzen.

Wie wird eine Exception aufgerufen? Die Definition der Division könnte so aussehen:[8]

```
exception DivisionByZero(n) = \ standard-exception, sollte
 seq                          \ im top-level definiert sein
   writeMessage(NumberToString(n));
   writeMessage("sollte durch 0 geteilt werden");
   return(bottom)
 endseq

func '/'(a,b) =
  if b=0 then DivisionByZero(a)
         else divide(a,b)
  endif
```

Exceptions sind spezielle Funktionen. Wenn im Rahmen der Aufrufreihenfolge kein `guard`–Konstrukt aufgetreten ist, so wird die ursprüngliche Exception-Funktion aufgerufen. Ein `guard`–Konstrukt ersetzt diese Funktion durch eine andere. Dies gilt jedoch nur lokal zu dem Ausdruck, der durch das `guard` bewacht wird. Wenn `guard`–Ausdrücke geschachtelt werden, gilt immer der innerste Ausdruck.[9]

Nachdem eine Exception abgearbeitet ist, kehrt die Auswertung *nicht* automatisch zum Punkt des Aufrufs zurück, sondern macht 'hinter dem `guard`' weiter. Will man dort weiterfahren, wo der Fehler aufgetreten ist, so verrichtet die (Pseudo–) Funktion `return` diesen Dienst. Beispiel: Unter den Definitionen

```
func reziprok1(x) =
 guard 3+(1/x)
    on DivisionByZero(n) = 42
 endguard
func reziprok2(x) =
 guard 3+(1/x)
    on DivisionByZero(n) = return(42)
 endguard
```

gilt:

```
reziprok1(0)   ⟹   42
reziprok2(0)   ⟹   45
```

[8] Man kann sich jetzt fragen, warum '/' als `func` definiert werden kann, wenn es denn einen Nebeneffekt erzeugen kann. Es gibt mindestens drei Erklärungen dafür: a) Exceptions 'zählen nicht' als 'echte' Prozeduren, b) `writeMessage` erzeugt keinen 'echten' Nebeneffekt, c) da der zurückgegebene Wert $\perp$ ist, liegt eine Fehlersituation vor. Wenn aber der Programmierer explizit signalisiert, daß eine Berechnung fehlerhaft war, muß ihn der Interpreter/Compiler nicht noch darauf hinweisen... Keine von diesen Erklärungen ist die 'richtige'; fest steht: aus pragmatischen Gründen müssen auch Funktionen in der Lage sein, Fehlermeldungen abzugeben.

[9] Faktisch implementieren Exceptions das 'dynamic binding' früher LISPs.

2.2.14 Rundungs- und Überlaufmechanismen: Makros

ALDiSP ist stark numerisch orientiert. Der IEEE-Standard für Floating-Point-Zahlen sieht mindestens vier Rundungs-Modi vor ('round to nearest / to $+\infty$ / to $+\infty$ / to 0'), die durchaus noch weiter verfeinert werden können (Anzahl der zu rundenden Bits). Weitere Rundungsmethoden werden in DSP-Prozessoren und -Sprachen eingesetzt. Auch die Behandlung von Überläufen fällt unter den allgemeinen Rundungsbegriff ('Behandlung von Ergebnissen, die nicht in voller Präzision darstellbar sind').

All diese Rundungsmethoden müssen in ALDiSP darstellbar sein. Behufs diesen Zweckes gibt es einen generellen Mechanismus, der jeden denkbaren Rundungsmechanismus zu modellieren in der Lage ist.

Grundlage sind die Exceptions. Bei jeder numerischen Bereichsüberschreitung wird eine 'Overflow'-Exception aufgerufen (genauer gesagt, gibt es für jeden distinkten breitesten numerischen Typen eine derartige Excpeption, also auf jeden Fall für Integer, Fixpoint und Real). Diese bestimmt, was als Ergebnis geliefert werden soll. Per Default gilt:

- Die Integer- und Fixpoint- Typen melden einen Fehler und liefern `bottom`
- Die Real-Typen melden keinen Fehler und liefern einen der Unendlichkeits-Werte.

Die Behandlung der Rundung in Reals und bei Integer-Divisionen ist nicht so einfach, aber machbar: Es gibt eine `DivisionRoundException`, die als Argumente den Divisor und den Dividenden erhält und als Ergebnis +1 oder -1 liefert, je nachdem, ob auf- oder abgerundet werden soll.

Der häufigste Fall eines nicht-Standard-Überlaufverhaltens ist das Rechnen mit Saturierten Zahlen (MAXINT = $+\infty$). Für diesen Spezialfall gibt es spezielle saturierte Operatoren. In allen anderen Fällen muß jedoch eine Abweichung von der normalen Rundungsdisziplin explizit markiert werden. Eine Expression der Form

```
guard a+b
   on IntegerOverflow(X)  = SaturatedIntegerOverflow(X)
   on RealOverflow(X)     = SaturatedRealOverflow(X)
   on FixpointOverflow(X) = SaturatedFixpointOverflow(X)
endgaurd
```

wertet die Expression `a+b` in einer Umgebung aus, in der ein saturiertes Overflow-Verhalten herrscht. Dieser Ausdruck ist etwas unhandlich. Leider kann man, aufgrund der call-by-value Aufrufstrategie, keine Funktion schreiben, die das `guard` schachtelt. Es gibt daher, speziell für diesen Zweck, eine Funktionsart `macro`, die call-by-name ausgewertet wird, also durch textuelle Substitution. `macro`–Funktionen dürfen nicht rekursiv sein! Ein Beispiel:

```
macro Saturated(expr) =
  guard expr
    on IntegerOverflow(X)  = SaturatedIntegerOverflow(X)
    on RealOverflow(X)     = SaturatedRealOverflow(X)
    on FixpointOverflow(X) = SaturatedFixpointOverflow(X)
  endguard
```

Unter dieser Definition wird `Saturated(a+b)` zum o.a. Term ausgeweitet, ist also voll äquivalent. Man kann sich jetzt vorstellen, daß eine primitive Funktion wie "`+^`", also die saturierte Addition, folgendermaßen definiert ist:

```
macro '+^'(expr1,expr2) =
  Saturated(expr1+expr2)
```

2.2.15 Das Delay

Ein Konstrukt der Form

`delay` *expr*

wird *Delay* genannt. Es ermöglicht die *verzögerten Auswertung* (*lazy evaluation*) von Ausdrücken; eine Technik, die bei der Implementierung von *Streams* Verwendung findet.

Ein Delay evaluiert sich zu einem Objekt (genannt *Promise*[10]), das, wenn später durch eine primitive Funktion auf dessen Wert zugegriffen wird, den Wert annimmt, der bei der Auswertung von *expr* entstehen würde. Man kann sagen, daß *expr* erst dann ausgewertet wird, wenn das Ergebnis für den Fortgang des Auswertungsgeschehens gebraucht wird (daher auch die gebräuchliche Bezeichnung *call-by-need*. Die normale Auswertungsstrategie in ALDiSP ist *call-by-value*.). Der Aufwand bei der Promise-Bildung ist konstant. Es gibt keine Möglichkeit, einen 'verzögerten' Wert von einem 'ausevaluierten' Wert zu unterscheiden.

Als kleines Beispiel für die Benutzung von Delays folgt ein Programm, das die Summe der ersten n Primzahlen berechnet.

```
func nat(n) =           \ die Zahlen n...unendlich
  n :: delay nat(n+1)

func filter(x,s) =      \ Aus s werden alle Vielfache von x entfernt
  if mod(head(s),x)=0 then filter(x,tail(x))
                      else head(x) :: delay  filter(x,tail(x))
  endif
```

[10] Der begriff entspringt dem Scheme-Jargon [2][5]. Andernorts werden die entstehenden Objekte 'Suspensions' genannt; aber dieser Begriff ist in ALDiSP schon verbraucht. Funktionale Sprachen, die 'lazy' sind, brauchen keinen dementsprechenden Begriff, da jeder nicht-Konstante Ausdruck eine Promise (ein Apply-Knoten) ist.

```
func first(n,s) =       \ die ersten Elemente einer Liste
  if n=0 then EmptyList
         else head(s) :: first(n-1, tail(s))
  endif

func nPrims(n) =        \ die ersten n Primzahlen
  let
    func prim(s) = \ alle gegenseitigen Vielfachen aus s entfernen
      head(s) :: delay prim(filter(head(s),tail(s)))
  in
    first(n, prim(nat(2)))
  endlet

func summePrim(n) =     \ summe der ersten n Primzahlen
  reduce('+',nPrims(n))
```

2.2.16 Typ-Deklarationen

Jeder Ausdruck liefert bei der Auswertung ein Ergebnis. Es ist syntaktisch möglich, so ziemlich allen Ausdrücken Typen zuzuordnen. Eine solche Typzuordnung manifestiert die Intention des Programmierers, daß der Ausdruck sich zu Werten des angegebenen Typs auswertet. Es gibt in ALDiSP keinen Zwang, irgendwelche Typangaben zu machen; der Compiler/Interpreter inferiert alle Typen von selbst bzw. meldet sich mit Fehlermeldungen, wenn keine 'ordentliche' Typisierung gefunden werden kann. Es ist jedoch aus mehreren Gründen sinnvoll, selbst Typangaben zu machen:

- Typisierungen sind ein zentrales Software-Dokumentations-Konzept.
- Der Interpreter/Compiler hat die Möglichkeit, herauszufinden, ob die von ihm gefundene Typisierung (das ist die allgemeinste mögliche Typisiserung eines Ausdrucks) mit der Vorgabe übereinstimmt.
- Viele automatische Programm-Optimierungen sind nur möglich, wenn extensives Wissen über Wertebereiche (und nichts anderes sind Typen!) von Variablen und Parametern vorgegeben werden.

Die allgemeine Typdeklaration hat die Form [*typ*] *expr*. Die Laufzeit-Semantik ist die, daß nach der Evaluation von *expr* geprüft wird, ob das Ergebnis vom Typ *typ* ist. Ist dies nicht der Fall, liegt eine Fehlersituation vor, das Ergebnis ist `bottom`.

2.2.17 Typumwandlung

Typumwandlung (*Casting*) ist die Umwandlung eines Objekts in ein "äquivalentes" Objekt eines anderen Typs. Nicht zwischen allen Typen ist eine solche

Umwnadlung möglich. Die syntaktische Form der Umwandlung ist

$$[!\,typ]\,expr$$

Bei der Auswertung der Umwandlung wird zuerst die *expr* ausgwertet, danach wird sie zum äquivalenten Objekt des Typs *typ* gewandelt. Die Umwandlung geschieht über den Aufruf einer Umwandlungsfunktion. Eigene Umwandlungsfunktionen können leicht in das System integriert werden. Die global bekannte überladene Funktion `coerce` dient diesem Zweck. Hat man z.B. gerade einen neuen Typ `Real42` samt Umwandlungsfunktionen `Real32ToReal42` und `Real42ToReal64` definiert, so kann mit der Deklaration

```
module Real42
export Real32ToReal42 as Coerce : (Real32 -> Real42),
       Real42ToReal64 as Coerce : (Real42 -> Real64)
 ...
```

die Funktion Coerce im Definitionsbereich erweitern. Eigentlich ist dies keine korrekte Überladung (da normalerweise nur der Definitionsbereich überladen wird (ALDiSP ist nicht ADA!)), sondern eine globale Deklaration aus einem Modul heraus, die als Overloading getarnt ist, um nicht ein neues Syntax-Konstrukt einführen zu müssen.

Bei der Typwandlung sollte kein Informationsverlust stattfinden[11].

2.2.18 Saturierte Typumwandlung

Eine *saturierte Typumwandlung* ist die Umwandlung eines numerischen Objekts in ein numerisches Objekt eines anderen Typs, der über maximale und minimale Werte verfügt. Ist eine 'korrekte' Umwandlung nicht möglich, weil das umzuwandelnde Objekt zu groß oder zu klein ist, ist stattdessen dieser maximale bzw. minimale Wert Ergebnis. Eine saturierte Typumwandlung hat die Form

$$[\hat{}\,typ]\,expr$$

Beispiele:

```
[^Integer16]65000     ⟹    32767
[^Cardinal16]65000    ⟹    65000
[^Integer16]-65000    ⟹    -32768
[^Cardinal16]-65000   ⟹    0
[^Integer16] plusInfinity  ⟹  32767
```

[11] Siehe dazu: ??. Natürlich ist dies vom Compiler nicht nachprüfbar!

2.2.19 Typterme

Typterme sind die Argumente von Typdeklarationen und Typumwandlungen. Da Typen Datenobjekte sind, werden Typterme durch normale Ausdrücke geformt[12]. Die einfachsten Typterme sind die Namen der primitiven Typen wie `Bool`, `Int16` oder `Real`. Die komplexere Typen haben Argumente, wie z.B. `List(`*type*`)` oder `Array(`*size*`)`. Schließlich lassen sich Typen durch `and` (Schnittmenge zweier Typen), `or` (Vereinigungsmenge) und `not` (Komplementmenge eines Typs) kombinieren. Die allgemeinste Form eines Typterms ist die des Prädikattyps: Jede Expression, die sich zu einem einstelligen Prädikat auswertet, wird als der Typ verstanden, der alle Objekte enthält, auf die das Prädikat zutrifft.

2.2.20 Abstrakte Datentypen

Ein abstrakter Datentyp definiert sich über eine Menge von Konstruktoren und Selektoren. Die Elemente abstrakter Datentypen heißen `Terme`. Ein einfacher abstrakter Datentyp sind die natürlichen Zahlen:

```
abstype Nat = Null | Succ(Pred: Nat)

overloaded func '>'(x:Nat, y:Nat) =
 if x is Null and y is Null then false
  | x is Null and y is Succ then false
  | x is Succ and y is Null then false
  | x is Succ and y is Succ then Pred(x)>Pred(y)
 endif

overloaded func '+'(x:Nat, y:Nat) =
 if x is Null then y
              else Pred(x)+Succ(y)
 endif
```

Die Datentyp-Definition im Beispiel führt zwei Konstruktoren ein; einer davon ist nullstellig (ein 'unechter' Konstrutor), einer einstellig. Zu jedem n-stelligen 'echten' Konstruktor gibt es n Selektor-Funktionen, die implizit (durch die Angabe der Argumentnamen) mitdefiniert werden. Die Typisierung der Argumente ist optional. Die primitive `is`-Funktion stellt fest, ob ein Objekt von einem gegebenen Konstruktor erzeugt worden ist. Die Definition eines Typen *Foo* erzeugt zugleich ein Prädikat des Namens `is`*Foo*.

Datentypen können auch parametrisiert werden; es entstehen dann Typkonstruktoren:

[12] Eine Implementation darf jedoch Beschränkungen aussprechen, d.h. es kann syntaktisch und semantisch korrekt geformte Typen geben, die in bestimmten Kontexten, speziell in Umwandlungen und bei der Partialisierung von Funktion, nicht als Typen 'anerkannt' werden. Siehe dazu den Abschnitt 4.7.

```
abstype List(a) = Leer | Cons(Head:a, Tail:List(a))

func sumlist(l:List(Number)) =
  if l is Leer then 0
               else Head(l)+sumlist(Tail(l))
  endif
```

Die allgemeine Form der Typdefinition ist

typdef *TypName* = Konstruktor$_1$
| ⋮
| Konstruktor$_n$

Ein Konstruktor ist entweder eine Konstante oder eine n-stelliges Gebilde der Form

$$KonstruktorName(SelektorName_1, \ldots, SelektorName_n)$$

,wobei die Selektoren beliebig getypt werden können.

Konstruktoren sind Typen nicht eindeutig zugeordnet. Unter den Deklarationen

```
abstype farbe1 = rot | gruen | blau
abstype farbe2 = rot | gruen | blau | magenta | cyan | gelb
```

ist nicht feststellbar, ob `rot` ein Objekt der Typen `farbe1` oder `farbe2` ist.

Es sollte vermieden werden, gleich benannte Konstruktoren mit unterschiedlicher Stelligkeit zu definieren, da `is` nur über den Namen definiert ist und eine effiziente Implementierung große Probleme bereiten kann. Beispiel: Unter den (zulässigen) Deklarationen

```
abstype colour1 = colour(rot: Byte, gr\"un: Byte, blau: Byte)
abstype colour2 = colour(rot: Word, gr\"un: Word, blau: Word)
```

ist nicht feststellbar, ob `colour(0,0,0)` ein `colour1` oder ein `colour2` ist. Das ist an sich nichts schlimmes (es ist eben beides), nur läßt sich keine optimale Speicherplatzanforderung bestimmen – u.U. werden 3 nutzlose Bytes angefordert.[13]

[13]Diese Beispiele sehen, so wie sie hier stehen, dumm aus. Man fragt sich: Wer schreibt schon gleich benannte Konstruktoren? Als Antwort hierauf kann ich nur erwiedern, daß dies, gerade in großen Programmsystemen, oft vorkommt. Meist liegt dem ein Drang zugrunde, zu viele Namen unqualifiziert zu exportieren, so sich daß im TLE einige hundert Namen aus verschiedensten Modulen tummeln. Dann sind 'name clashes' unvermeidlich!

2.2.21 Überladung

Überladung (Overloading) kann auftreten, wenn es mehrere Definitionen einer Funktion (d.h., eines Namens) gibt. Wenn sich die Definitionsbereiche der einzelnen Definitionen überlagern, liegt ein Fehler vor; sind sie getrennt, so werden die Funktionen 'vereinigt', d.h. bei der Applikation der Funktion wird geprüft, welche Argumenttypen vorliegen; die partikulär passende Einzeldefinition wird dann angewandt. Ein Beispiel:

```
\ so k\"onnte die Funktion '+' definiert sein!

overloaded '+' = Integers("+")
overloaded '+' = Reals("+")
overloaded '+' = Rationals("+")

\ oder

overloaded import '+','-','*','/' from Integers

\ oder

import overloaded '+', overloaded '-',
       overloaded '*', overloaded '/' from Integers

\ oder

import overloaded AddInt as '+' from Integers

\ so schaut eine \"uberladene Funktionsdefinition aus

overloaded '*'(a: Vector, b: Vector) = \ Vektorprodukt
  if sizeOfArray(a)=sizeOfArray(b) then reduce('+',map('*',a,b))
                                   else bottom
  endif
```

Der `overloaded`–Indikator kann vor jede Form von Funktions-, Typkonstruktor- oder Prozedur-Deklaration gestellt werden. Funktionen dürfen nur mit Funktionen überladen werden, d.h. die 'Vereinigung' einer Prozedur mit einer Funktion muß als Prozedur notiert sein. Es ist nicht möglich, Exceptions zu überladen (Sei überdecken sich stets vollständig). Es wird nicht empfohlen, Typen zu überladen.

2.2.22 Die Suspension

Die Suspension[14] ist das zentrale Konstrukt zur Prozeßbildung und Zeitspezifikation. Sie hat die Form

suspend $expr_1$ **until** $expr_2$ **within** $time_1$ **,** $time_2$

Die Semantik der Suspension ist nicht ganz einfach. Wenn die Suspension ausgewertet wird, entsteht ein Prozeß, der der Auswertung von $expr_1$ gewidmet ist. Der Prozeß schläft, solange die Bedingung $expr_2$ nicht `true` ergibt. Der Augeblick, an dem $expr_2$ gilt, ist der Zeitpunkt Null für den Zeitrahmen [$time_1$ ... $time_2$], der ein Intervall der Länge ≥ 0 beschreibt dieses Zeitrahmens nun wird der schlafende Prozeß aufgeweckt und ausgewertet. Bei der Spezifikation des Auswertungszeitpunkts von Prozessen wird von der Vorstellung des *virtuellen Zeitpunktes* eines Prozesses ausgegangen: während ein Prozeß läuft, verändert sich seine Position in der Zeit nicht; nur durch Suspensions kann dies geschehen. Alle I/O eines Prozesses geschieht zum selben logischen Zeitpunkt (daher ist es meist sinnvoll, daß ein Prozeß während seiner Existenz maximal eine I/O-Operation vornimmt). Der Scheduler (also das Laufzeitsystem, daß das Einschläfern und Aufwecken von Prozessen erledigt) muß dafür sorgen, daß die realen Zeitpunkte der von den Prozessen vollführten I/O-Operationen mit der virtuellen Prozeßzeit synchronisiert werden. Zurück zur Suspension: nachdem der Prozeß erzeugt (und schlafengelegt) worden ist, evaluiert sich das Suspension-Konstrukt zu einem nicht näher spezifizierten Wert (vom Typ `Suspension`), den man sich grob als Mischung aus Prozeß-ID und Mailbox-Variable vorstellen kann. Wenn später der Prozeß gelaufen ist und $expr_1$ berechnet worden ist, dann überschreibt der Ergebniswert diese Suspension. Wird vorher auf die Suspension zugegriffen, so suspendiert sich der zugreifende Prozeß selbst, bis der Wert da ist, d.h. der Zugriff auf eine Suspension ist blockierend. Es gibt ein primitives Prädikat `isSuspended`, das feststellt, ob ein Wert eine Suspension, also ein schlafender Prozeß, ist, oder nicht. Nachdem der Prozeß gelaufen ist, kann man nicht mehr feststellen, daß der nun überschrieben Wert je eine Suspension war.

Einige offensichtliche Nebenbedingungen des Suspension-Konstrukts sind:

- $expr_2$ muß ein nebeneffektfreier Ausdruck sein, der sich zu `Bool` evaluiert.
- $expr_3$ muß eine Zeitangabe (Typ `Time`) ≥ 0 sein. Die Umwandlung von Number nach Time geschieht mit der Postfix-Funktion `sec`, die so definiert ist, daß `1 sec` genau eine Sekunde ist. Zeit ist, für alle praktischen Belange, ein Fixpunkt-Datentyp, dessen Auflösung durch die System-Konstante `MinTimeDelta` vorgegeben ist (sollte in der Größenordnung von 0.1 ms liegen).

[14] Das Konstrukt und die Begriffsbilsdung sind neu. Der Begriff 'Suspension' wird manchmal an Stelle von 'Promise' gesetzt. Im Unterschied zu einer Promise ist eine Suspension jedoch nicht beliebig auswertbar, sondern von einem Zustand abhängig (und zeitlich festlegbar).

- $expr_3$ muß ein 'echtes' Intervall sein, d.h. $t_1 \leq t_2$ muß gelten.
- $expr_2$ und $expr_3$ sollte zur Übersetzungszeit berechenbar sein, damit ein statischer Schedule angefertigt werden kann.

Nützliche Abkürzungen

Eine spezielle Suspension-Ausdrücke sind als "Abkürzungen" (sprich: Makros) definiert. Diese Makros haben den Status 'echter' Makro-Funktionen, es sind keine reservierten Bezeichner!

$$expr \texttt{ after } time \equiv \texttt{suspend } expr \texttt{ until true within } time,time$$

$$expr_1 \texttt{ when } expr_2 \equiv \texttt{suspend } expr_1 \texttt{ until } expr_2 \texttt{ within 0,0}$$

`After` spezifiziert ein konstante Wartezeit, d.h. eine Verschiebung des virtuellen Prozesses. `When` spezifiziert das Warten auf eine Zustandsänderung. Man kann sagen, daß `after` und `when` die 'synchrone' bzw. 'asynchrone' Seite der Suspension beschreiben.
$expr$ `all` $duration \equiv$

```
let
   StartTime = SystemClock()
   tmp() = suspend seq expr;
                       tmp()
                    endseq
            until (SystemClock()-StartTime) mod duration = 0,
                  (SystemClock()-StartTime) mod duration = 0
in tmp()
endlet
```

`All` ist ein harmlos aussehender Operator, der recht komplex arbeitet: Vom (virtuellen) Zeitpunkt des Aufrufs ausgehend wird alle *duration* Zeiteinheiten der Ausdruck *expr* ausgewertet. Der Ausdruck `writeRegister(A,1) all 10 sec`, aufgerufen um 12:34:17, bewirkt also, das um 12:34:27, 12:34:37,... eine `1` in das Register `A` geschrieben wird.

2.2.23 Die Netzbeschreibung

Ein Programm besteht aus einer Reihe von Definitionen und einer *Netzbeschreibung*. Die Netzbeschreibung ist eine Sammlung rekursiver Definitionen, die bestimmt, *was* das Programm zur Laufzeit 'macht' (Die Definitionen erklären, *wie* etwas gemacht wird). Ein einfaches Beispiel ist ein Programm, das Daten aus zwei Ports addiert. Die Synchronisation wird dabei durch die Ports vorgegeben.

```
func PipeFromPort(X) =
  readFromPort(X) :: delay PipeFromPort(X)

func PipeToPort(X,P) =
  seq
    writeToPort(X,head(P));
    PipeToPort(X,tail(P))
  endseq

net
 InputPipeA = PipeFromPort(A)
 InputPipeB = PipeFromPort(B)
in
 WriteToPort(C,Map('+^', InputPipeA, InputPipeB)
endnet
```

Eine Netzbeschreibung muß nicht unbedingt ein Datenflußnetz beschreiben – es wird aber in den meisten Fällen so sein.

Die Datenflußnetze ermöglichen es, Funktionen mit lokalem Zustand zu modellieren, indem die Zustandsvariablen als Pipes exportiert und importiert werden.

2.2.24 Das Top-Level-Environment

Man kann sich vorstellen, daß alle Definitionen auf der obersten Ebene des Programms in großen lokale Definition stehen, die von einer Expression gefolgt werden, die auszuwerten der eigentliche Sinn des Programms ist. Außerhalb dieser Definitionen ist nur das leere Environment, innerhalb sind alle Module und Objekte definiert. Der Programmierer schreibt den "unteren" Teil der Definition, d.h. es sind, für ihn unsichtbar, schon eine Reihe Definition vorgenommen worden. Diese Definitionen etablieren das Top-Level-Environment (TLE). Im nächsten Abschnitt 2.3 finden sich die Namen und Beschreibungen aller im TLE bekannten Module, Funktionen und Objekte. Es ist nicht möglich, im TLE definierte Namen zu verändern. (Funktionen können allerdings durch Überladung erweitert werden.)

2.3 Vorgegebene Typen und Funktionen

In diesem Abschnitt werden die vorgegebenen Typen und die auf ihnen definierten primitiven Funktionen beschrieben. Um die Beschreibung zu vereinfachen, wird vom Konzept des 'Subtyps' Gebrauch gemacht. Ein Subtyp erbt alle Eigenschaften seines Supertyps, außer denen, die explizit geändert sind. Einige Primitive sind als 'essentiell' gekennzeichnet. Dies bedeutet, daß jede ALDiSP–Implementierung diese Primitive enthalten muß. Für die anderen Primitiven gilt lediglich, daß sie, wenn sie unter den hier genannten Namen implementiert werden, auch die hier genannte Funktionalität aufweisen müssen.

2.3.1 Allgemeines

In ALDiSP ist jeder Typ ein Prädikat, und jedes Prädikat kann als Typ Verwendung finden. (Als Prädikat verstehe man hier jede Funktion, die beliebige Objekte auf die Menge {`true`, `false`} abbildet). Es gilt die Konvention, daß das einem Typ *Foo* entsprechende Prädikat `is`*Foo* heißt. Ist *Foo* ein parametrisierter Typ wie z.B. `ListOf`, so ist `is`*Foo* ein entsprechend parametrisiertes Typprädikat. In diesem Fall gibt es eine zweite Konvention, die als Namen für ein allgemeines Prädikat (entsprechend einer Parametrisierung mit `Obj`) `isA`*Foo* (oder `isAn`*Foo*) vorgibt. Semantisch gesehen ist ein Typ mit 'seinem' Typprädikat identisch (d.h., ein Typ hat keine über das Prädikat hinausgehende Bedeutung, etwa konstruktiver Art), aus Dokumentationsgründen werden sie jedoch getrennt. Das elementarste Typsystem, auf das sich alles zurückführen läßt, ist die Menge der Basis-Typen, verbunden mit der Möglichkeit, ein Prädikat zum Typen zu erklären. Syntaktisch und semantisch sind Typen normale Expressions, die jedoch weniger Bildungsmöglichkeiten haben können. Es gibt Typkonstanten und Typkonstruktoren ('typwertige Funktionen'), in deren Definitionen Typvariablen auftauchen.

2.3.2 Grobeinteilung der Typen: Typklassen

Vor allem aus Gründen der Übersichtlichkeit werden die vorgegebenen Typen in Typklassen aufgeteilt. Diese Klassen sind von keiner praktischen (semantischen) Bedeutung; die Aufteilung erfolgt nicht nach eindeutig nachvollziehbaren Kriterien.

Basistypen

Basistypen sind grundlegende Typen, die in den Sprachkonstrukten benötigt werden und es ermöglichen, ein vollständiges Typsystem aufzubauen[15].

[15]Natürlich stellen die drei Konstrukte Applikation, Abstraktion und Variablen-Lookup bereits ein dem λ-Kalkül äquivalentes System dar – mehr braucht es nicht. 'Aufbau eines Typsystems' meint daher 'Aufbau *innerhalb* der Sprache'. (Im Gegensatz zum λ-Kalkül, wo ein Interpretationsschritt notwendig ist – ($\lambda x.\lambda y.x$ ist nicht aus sich heraus "`true`".))

Es gibt eine fixe Zahl von Basistypen, die normalerweise durch ihre Namen benannt werden.

Atomare Typen

Atomar sollen die Typen heißen, die nicht "zerlegbar" sind. Ein Array ist offensichtlich zerlegbar, ein Bit nicht. Implementierungstechnisch gesehen ist ein atomarer Typ meist in der Wortbreite einem Maschinenregisters repräsentierbar[16]. Alle Zahlen (die eine komplexe Typhierarchie bilden) gehören in diese Klasse.

Die Menge der atomaren Typen ist offen; dies vor allem, weil gerade von den Zahlen eine unüberschaubare Menge möglicher Subtypen (Teilmengen) bestehen (natürlich ist jeder Subtyp eines atomaren Typs wieder ein atomarer Typ). Atomare Typen werden entweder über ihre Namen oder über durch Prädikate hergestellte Verbindungen zwischen anderen atomaren Typen definiert.

Arrays

Die wichtigste Klasse nicht-atomarer Typen ist die der Arrays. Ein Array ist eine n-dimensional geordnete Sammlung von Objekten des gleichen Typs. Arrays können in ihrer Größe fixiert sein oder sich dynamisch der Benutzung anpassen. Es gibt unzählige Spezialfälle von Arrays (Bit-Arrays, Dünne Arrays, Dreicks-Matrizen, ...), die auch nicht annähernd vollständig aufgezählt werden können. Array-Typen kommen vorgefertigt daher; nur wenige Paramter (meist Basis-Typ und Ausmaße) müssen instanziiert werden.

Produkttypen

Produkktypen ermöglichen die Bildung von n-Tupeln. Ein n-Tupel läßt sich als eindimensionales Array der festen Länge n interpretieren.

Summentypen

Ein Summentyp entspricht der Vereinigungsmenge mehrerer anderer Typen.

Funktionen

Funktionen sind Datenobjekte; daher haben sie Typen. `Function` ist ein Basistyp. Es ist i.A. nicht möglich, festzustellen, wie der "exakte" Typ einer Funktion aussieht (d.h., welches die Definitions- und Wertemengen sind). Es gibt allerdings einen speziellen Typ 'Predicate', der die Menge der Prädikate beschreiben soll.

[16] Natürlich ist diese Unterscheidung im Einzelfall nicht gerechtfertigt: ein 4∗4-ArrayOf(Bit) ist sicherlich maschinennäher als eine 80-Bit `LongReal`.

Maschinentypen

Maschinentypen erlauben den direkten Zugriff auf Gegebenheiten der konkreten Hardware, auf der ein Programm läuft. Der Abstraktionsgrad dieser Typen ist niedrig; auch lassen sie sich nicht vollständig funktional beschreiben (Dies erfordert lokale Zustände, also Zuweisungen).

2.3.3 Basistypen

`Bottom` $\longrightarrow$ *Type* Fehlertyp
`bottom` $\longrightarrow$ *Bottom* Fehlerkonstante

Das ausgezeichnete Element `bottom` ist der einzige Wert des Pseudo-Typs `Bottom`. `bottom` repräsentiert eine Berechnung, deren Wert (in der Terminologie der denotationellen Semantik) $\perp$ ist. Insbesondere ist `bottom` das Ergebnis von Funktionsaufrufen mit Argumenten, die nicht in der Definitonsmenge sind. `bottom` steht *nicht* für der Wert einer nichtterminierden Berechnung. `bottom` ist kein echtes Objekt; es ist vielmehr der 'am schwächsten definierte Wert', der durch jeden stärker definierten Wert (jedes echte Objekt) ersetzt werden kann.

`Obj` $\longrightarrow$ *Type* Basistyp

Der Typ `Obj` beschreibt die Menge aller Objekte, d.h. die Vereinigungsmenge aller Typen. Jede aktive oder passive Entität (bis auf `bottom`), die im Rahmen eines Programmlaufs auftreten kann, ist ein `Obj`.

`Type` $\longrightarrow$ *type* Basistyp

Der Typ `Type` beschreibt die Menge aller Typen.

($Type_1$ ... $Type_n$ `->` $Type_{n+1}$ ... $Type_m$) $\longrightarrow$ *Type* ess. Typkonstruktor

Der geklammerte Pfeil-Operator definiert einen Funktionstypen mit n Argumenttypen und $m - n$ Resultattypen. Die Klammern sind nicht optional. Funktionen können keine Argumente und/oder keine Resultate haben. Dieser Konstruktor beschreibt sowohl 'echte' Funktionen als auch Prozeduren und Typkonstruktoren.

`Function` $\longrightarrow$ *type* Basistyp

Der Typ `Function` beschreibt die Menge aller Funktionen. (Dies umfaßt sowohl 'echte' Funktionen als auch Prozeduren und Typkonstruktoren.)

`Bool` $\longrightarrow$ *type* Basistyp

Der Typ `Bool` beschreibt die Menge der Konstanten `false` und `true`.[17]

`Predicate` $\longrightarrow$ *type* Basistyp

[17] Was Konstanten sind, kann uns hier egal sein. Es reicht, zu wissen, das man sie auseinanderhalten kann.

Der Typ `Predicate` beschreibt die Menge aller einstelligen Prädikate, d.h. aller Funktonen des Typs `Obj`→`Bool`. Jedes Prädikat, das über `Obj` definiert ist, benennt einen Typen.

`eq` *(Obj,Obj)* $\longrightarrow$ *Bool* Basisfunktion

`eq` ist das Prädikat, das die Identität von Objekten festzustellen in der Lage ist. Es ist für alle Paare von Objekten[18] möglich, festzustellen, ob sie identisch sind.[19]

`identity` *(Obj)* $\longrightarrow$ *Obj* Basisfunktion

`identity` ist die Identitätsfunktion, d.h. die Funktion, die jedes Objekt auf sich selbst abbildet.

Bool `and` *Bool* $\longrightarrow$ *Bool* Basis-Infix-Funktion

Diese Infix-Operation implementiert das logische "$\wedge$".

Bool `or` *Bool* $\longrightarrow$ *Bool* Basis-Infix-Funktion

Diese Infix-Operation implementiert das logische "$\vee$".

`not` *Bool* $\longrightarrow$ *Bool* Basis-Prefix-Funktion

Diese Prefix-Operation implementiert das logische "$\neg$".

Function$_1$ `and` *Function*$_2$ $\longrightarrow$ *Function* Infix-Funktion
Function$_1$ `or` *Function*$_2$ $\longrightarrow$ *Function* Infix-Funktion
`not` *Function* $\longrightarrow$ *Function* Prefix-Funktion

Die logischen Funktionen sind auf die n-stelligen Prädikate erweitert.

Beispiel: Seien `F` und `G` zwei n-stellige Funktionen nach `Bool`. Dann ist `F and G` eine n-stellige Funktion nach Bool, die dadurch definiert ist, das gilt:

$$\texttt{F}(b_1, \ldots, b_n)\ \texttt{and}\ \texttt{F}(b_1, \ldots, b_n) \equiv (\texttt{F and G})(b_1, \ldots, b_n)$$

Die Anwendung von `and`, `or` und `not` auf Funktionen verschiedener Stelligkeit und/oder Funktionen, die nicht `Bool` zum Ergebnis haben, resultiert in `bottom` oder einer Funktion, die `bottom` liefert.

Es gibt einen wichtigen Spezialfall: wenn `F` und `G` Prädikate (bzw. Typen) sind, so stellt `F and G` den *Schnitt* der durch sie spezifizierten Objektmengen dar, `F or G` die *Vereinigung* (d.h, den *Summentypen*) und `not F` die *Komplementmenge*.

[18] Ausgenommen `bottom`

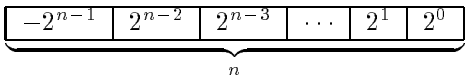
[19] `eq` ist nicht zu verwechseln mit der Gleichheitsfunktion "`=`", die feststellt, ob zwei numerische Objekte, die durchaus verschiedenen Typs sein können, die gleiche Zahl repräsentieren.

2.3.4 Numerische Typen

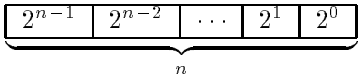
ALDiSP verfügt über eine Vielzahl numerischer Typen, die unter dem Supertype `Number` gesammelt sind. Grundlegend ist die Trennung in Ganzzahl-Typen (Integer und alle Subtypen), Fixpunkt-Typen (die ganze Zahlen, die um einen festen Betrag skaliert sind, beschreiben) und Floating-Point-Typen. Die Floating-Point-Typen orientieren sich am IEEE-754-Standard, der spezielle Werte für $+\infty$ und $-\infty$ sowie eine negative Null vorsieht. Statt einer negativ-Null gibt es in ALDiSP drei Nullen: die echte Null, $\frac{1}{+\infty}$ und $\frac{1}{-\infty}$.

Um zu ermitteln, von welchem Typ das Ergebnis einer Funktionsanwendung ist, werden die folgenden Tabellen eingesetzt, die die Typisierung von Integern, Cardinals, Integer- und Cardinal-Fixpunktzahlen zeigt. Zur Erläuterung:

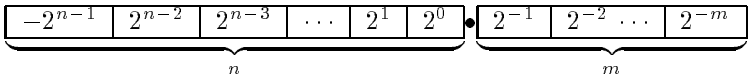
- Ein `Integer(`n`)` ist ein n Bit langes Objekt, das ganze Zahlen im Intervall $-2^{n-1}...2^{n-1}-1$ zu repräsentieren in der Lage ist.



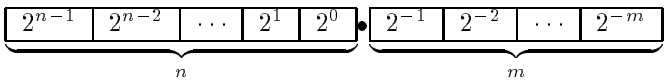
- Ein `Cardinal(`n`)` ist ein n Bit langes Objekt, das ganze Zahlen im Intervall $0...2^n-1$ zu repräsentieren in der Lage ist.



- Ein `IntegerFixpoint(`n,m`)` ist ein n+m Bit langes Objekt, in dem n Bits vor und m Bits nach dem Komma stehen. Es können Zahlen im Intervall $\frac{-2^{n+m-1}...2^{n+m-1}-1}{2^m}$ repräsentiert werden.



- Ein `CardardinalFixpoint(`n,m`)` ist ein n+m Bit langes Objekt, in dem n Bits vor und m Bits nach dem Komma stehen. Es können Zahlen im Intervall $\frac{0...2^{n+m}-1}{2^m}$ repräsentiert werden.



In den folgenden Tabellen gilt: $n'' = \max(n, n'), m'' = \max(m, m')$. Der Tabelleneintrag in der Zeile X und der Spalte Y definiert den Typen von $X \odot Y$, wobei $\odot$ die zu spezifizierende Funktion ist. $\bar{m}$ ist der maximale Wert für m, der im Kontext, in dem m steht, möglich ist. `I` steht für `Integer` und `IntegerFixpoint`, `C` steht für `Cardinal` und `CardinalFixpoint`.

Für die Addition gilt:

	I(n')	C(n')	I(n', m')	C(n', m')
I(n')	I($n''+1$)	I($n''+1$)	I($n''+1, m'$)	I($n''+1, m'$)
C(n')	-	C($n''+1$)	I($n''+1, m'$)	C($n''+1, m'$)
I(n', m')	-	-	I($n''+1, m''$)	I($n''+1, m''$)
C(n', m')	-	-	-	C($n''+1, m''$)

Für die Subtraktion gilt:

	$\mathrm{I}(n')$	$\mathrm{C}(n')$	$\mathrm{I}(n', m')$	$\mathrm{C}(n', m')$
$\mathrm{I}(n')$	$\mathrm{I}(n''+1)$	$\mathrm{I}(n''+1)$	$\mathrm{I}(n''+1, m')$	$\mathrm{I}(n''+1, m')$
$\mathrm{C}(n')$	-	$\mathrm{I}(n''+1)$	$\mathrm{I}(n''+1, m')$	$\mathrm{I}(n''+1, m')$
$\mathrm{I}(n', m')$	-	-	$\mathrm{I}(n''+1, m'')$	$\mathrm{I}(n''+1, m'')$
$\mathrm{C}(n', m')$	-	-	-	$\mathrm{I}(n''+1, m'')$

Für die Multiplikation gilt:

	$\mathrm{I}(n')$	$\mathrm{C}(n')$	$\mathrm{I}(n', m')$	$\mathrm{C}(n', m')$
$\mathrm{I}(n')$	$\mathrm{I}(n+n'-1)$	$\mathrm{I}(n+n')$	$\mathrm{I}(n+n'-1, m')$	$\mathrm{I}(n+n'+1, m')$
$\mathrm{C}(n')$	-	$\mathrm{C}(n+n')$	$\mathrm{I}(n+n', m')$	$\mathrm{C}(n+n', m')$
$\mathrm{I}(n', m')$	-	-	$\mathrm{I}(n+n'-1, m+m')$	$\mathrm{I}(n+n', m+m')$
$\mathrm{C}(n', m')$	-	-	-	$\mathrm{C}(n+n'+1, m+m')$

Für die Division gilt:

	$\mathrm{I}(n')$	$\mathrm{C}(n')$	$\mathrm{I}(n', m')$	$\mathrm{C}(n', m')$
$\mathrm{I}(n')$	$\mathrm{I}(n, \bar{m})$	$\mathrm{I}(n, \bar{m})$	$\mathrm{I}(n+m', \bar{m})$	$\mathrm{I}(n+m'+1, \bar{m})$
$\mathrm{C}(n')$	-	$\mathrm{C}(n, \bar{m})$	$\mathrm{I}(n+m', \bar{m})$	$\mathrm{I}(n+m', \bar{m})$
$\mathrm{I}(n', m')$	-	-	$\mathrm{I}(n+m', \bar{m})$	$\mathrm{I}(n+m', \bar{m})$
$\mathrm{C}(n', m')$	-	-	-	$\mathrm{C}(n+m'+1, \bar{m})$

Die Division stellt ein Problem dar, weil möglichst viele Nachkommastellen zur Verfügung gestellt werden sollten. Bei einer begrenzten Anzahl möglicher Typen (z.B. alles, was in 32 Bit unterzubringen ist) ist dies aber abhängig von der Anzahl der Vorkommastellen.

Da sich alle mathematischen Funktionen auf die vier Grundrechenarten zurückführen lassen, werden nur für diese explizite Ergebnistyptabellen angegeben. Ist eine Typisierung nicht durchführbar, weil ein entsprechend breiter Typ für das Ergebnis nicht zur Verfügung steht, geschieht das folgende:

Die Berechnung wird ausgeführt. Tritt ein Überlauf (oder Unterlauf) auf, wird eine entsprechende Exception aufgerufen, die ein Ergebnis erzeugt und u.U. eine Fehlermeldung absetzt. Ansonsten wird das Ergebnis im größten verfügbaren Typen dargestellt und zurückgegeben.

Für die Rundung gilt folgendes: Tritt eine Rundung auf (weil nicht genug Nachkommastellen zur Verfügung stehen oder weil eine Division ausgeführt wurde), wird eine Rundungs-Exception mit den Argumenten, dem ungerundeten (d.h, einfach abgeschnittenen) Ergebnis und dem ersten abgeschnittenen Bit aufgerufen; diese Exception gibt das gerundete Ergebnis zurück.

Für Fließkomma-Typen gilt eine einzige Regel: ist mindestens ein Argument einer primitiven Funktion eine Fließkommazahl, so ist das Ergebnis vom breitesten Fließkommatyp, der unter den Argumenten auftritt. Beispiele:

```
[ShortFloat]1+[Integer]1     ⟹     [ShortFloat]2
[ShortFloat]1+[LongFloat]1   ⟹     [LongFloat]2
```

Number

`Number` ⟶ *Type* numerischer Typ

`Number` ist der allgemeinste numerische Typ. Alle anderen als 'numerisch' gekennzeichnete Typen sind Subtypen von Number. Die allgemeinen numerischen Funktionen werden daher hier aufgeführt.

$Number_1$ `=` $Number_2 \longrightarrow Bool$	essentielle Infix-Funktion
$Number_1$ `<>` $Number_2 \longrightarrow Bool$	essentielle Infix-Funktion
$Number_1$ `>` $Number_2 \longrightarrow Bool$	essentielle Infix-Funktion
$Number_1$ `>=` $Number_2 \longrightarrow Bool$	essentielle Infix-Funktion
$Number_1$ `<` $Number_2 \longrightarrow Bool$	essentielle Infix-Funktion
$Number_1$ `=<` $Number_2 \longrightarrow Bool$	essentielle Infix-Funktion

Dies sind die elementaren Vergleichsoperationen. Wenn der Zeichensatz entsprechendes enthält, sollte "$\neq$" statt "`<>`" , "$\geq$" statt "`>=`" und "$\leq$" statt "`=<`" geschrieben werden.

Bei komplexen Zahlen bestimmt der Vergleich der Realteile das Ergebnis; sind diese gleich, so werden die Imaginärteile verglichen.

$Number_1$ `+` $Number_2 \longrightarrow Number$	essentielle Infix-Funktion
$Number_1$ `-` $Number_2 \longrightarrow Number$	essentielle Infix-Funktion
$Number_1$ `*` $Number_2 \longrightarrow Number$	essentielle Infix-Funktion
$Number_1$ `/` $Number_2 \longrightarrow Number$	essentielle Infix-Funktion

Dies sind die elementaren arithmetischen Funktionen. Die Typisierung des Ergebnisses ist nicht statisch festgelegt. Das Ergebnis wird von dem Typen sein, der das numerische Resultat am besten repräsentieren kann (und implementiert ist!).

`sqrt` $Number \longrightarrow Number$	Präfix-Funktion
`sin` $Number \longrightarrow Number$	Präfix-Funktion
`cos` $Number \longrightarrow Number$	Präfix-Funktion
`tan` $Number \longrightarrow Number$	Präfix-Funktion
`asin` $Number \longrightarrow Number$	Präfix-Funktion
`acos` $Number \longrightarrow Number$	Präfix-Funktion
`atan` $Number \longrightarrow Number$	Präfix-Funktion
`atan` $(Number,Number) \longrightarrow Number$	Funktion
`ln` $Number \longrightarrow Number$	Präfix-Funktion
`ld` $Number \longrightarrow Number$	Präfix-Funktion
`exp` $Number \longrightarrow Number$	Präfix-Funktion
$Number_1$ `**` $Number_2 \longrightarrow Number$	Infix-Funktion
`abs` $(Number) \longrightarrow Number$	essentielle Funktion

Diese Funktionen sind definiert durch:

`sqrt` $n \equiv \sqrt{n}$

Der Definitionsbereich von Sqrt hat eine Besonderheit: `sqrt(-1)` wertet sich zu einem Fehler aus; `sqrt(makeComplex(-1,0))` wertet sich zur komplexen Zahl i

aus.

`sin` $n \equiv \sin n$

Die trigonometrischen Funktionen sind im Bogenmaß skaliert. Sie sind nur über Fixpunkt- und Real-Typen definiert, nicht aber über Ganzzahltypen. Die Ergebnisse sind als Fixpunkt- oder Real-Zahlen getypt; die Implementierung ist darin frei. Es sollte sich aber um die präzisestmögliche Darstellung handeln; im Normalfall dürften dies Fixpunktzahlen sein.

`cos` $n \equiv \cos n$

`tan` $n \equiv \tan n$

Das Overflowverhalten der Tangensfunktion ist nicht explizit spezifizierbar. Da sich $\tan x = \frac{\sin x}{\cos x}$ definieren läßt und Sinus und Cosinus kein Overflow-Verhalten zeigen können, reicht es aus, das Overflowverhalten der Division zu spezifizieren.

`asin` $n \equiv \arcsin n$

Die inversen trigonometrischen Funktionen geben Ergebnisse im halboffenen Intervall $[0...2\pi[$ zurück.

`acos` $n \equiv \arccos n$

`atan` $n \equiv \arctan n$ `atan(`n`,`m`)` $\equiv \arctan n/m$

Der Arcus Tangens liefert Ergebnisse im Intervall $[-2\pi...2\pi]$.

`ln` $n \equiv \log_e n$

`ld` $n \equiv \log_2 n$

`exp` $n \equiv e^n$

n `**` $m \equiv n^m$

`abs(`m`)` $\equiv |n|$

Der Absolutwert einer komplexen Zahl ist ihr Betrag.

`max` *(Number$_1$, Number$_2$)* $\longrightarrow$ *Number*	essentielle Funktion
`min` *(Number$_1$, Number$_2$)* $\longrightarrow$ *Number*	essentielle Funktion
`max` *(Number$_1$, ..., Number$_n$)* $\longrightarrow$ *Number*	Funktion
`min` *(Number$_1$, ..., Number$_n$)* $\longrightarrow$ *Number*	Funktion

Diese Funktionen bestimmen das Maximum bzw. Minimum ihrer Argumente.

`signum` *Number* $\longrightarrow$ *Number*	Präfix-Funktion

Die Signum-Funktion bestimmt bei Skalaren das 'Vorzeichen', bei komplexen Zahlen die 'Richtung'.

$$\texttt{signum}(x) = \begin{cases} x & ,\text{falls} |x| = 0 \\ \frac{x}{|x|} & ,\text{falls} |x| \neq 0 \end{cases}$$

Number$_1$ `+^` *Number*$_2$ $\longrightarrow$ *Number* Infix-Funktion
Number$_1$ `-^` *Number*$_2$ $\longrightarrow$ *Number* Infix-Funktion

Dies sind 'saturierte' Funktionen. Sie sind über allen Zahlentypen außer den `Reals` definiert. Für letztere gibt es spezielle 'Unendlichkeitswerte'. Die saturierten Funktionen sind nur über Zahlen gleichen Typs definiert. Das Ergebnis dieser saturierten Funktionen hat den gleichen Typ wie die Argumente. Im Fall eines Über– oder Unterlaufs wird der maximale bzw. minimale Wert zurückgegeben. Bei der saturierten Division ist der Divident saturiert; Beispiele:

```
[Integer16]20000 +^[Integer16]30000    ⟹    32767
[Cardinal16]20000 -^[Cardinal16]30000  ⟹    0
```

Es gibt keine saturierte Multiplikation oder Division; stattdessen muß eine saturierte Typwandlung vorgenommen werden. Wenn der Zeichensatz dies zuläßt, sollten die saturierten Funktionen mit $\dot{+}$ und $\dot{-}$ oder $\hat{+}$ bzw $\hat{-}$ denotiert werden.

`WrapAround` *(expr)* $\longrightarrow$ *expr* Makro
`Saturated` *(expr)* $\longrightarrow$ *expr* Makro
`Ignore` *(expr)* $\longrightarrow$ *expr* Makro

Diese Makros sorgen dafür, das eine Expression unter einem bestimmten Overflow-Verhalten ausgewertet wird. Sie könnten so definiert sein:

```
macro WrapAround(expr) =
 guard expr
    on AddOverflow = WrapAroundAddOverflow
    on SubOverflow = WrapAroundSubOverflow
    on MultOverflow= WrapAroundMultOverflow
    on DivOverflow = WrapAroundDivOverflow
 endguard
```

Integer und seine Subtypen

`Integer` $\longrightarrow$ *Type* essentieller numerischer Typ
`MinInt` $\longrightarrow$ *Integer* essentielle numerische Konstante
`MaxInt` $\longrightarrow$ *Integer* essentielle numerische Konstante

`Integer` ist der 'Standard-Integer'. Die minimalen und maximalen Werte, die ein `Integer`-Objekt annehmen kann, sind über den Konstanten `MinInt` und `MaxInt` abrufbar. Typische Werte hierfür sind `MinInt`= -8388608, `MaxInt`= 8388607 (dies entspricht 24 Bit, Zweierkomplement). `Integer` muß 'genügend groß' sein. Was das heißt, bleibt undefiniert.

`nBitInteger` *(Cardinal)* $\longrightarrow$ *Type* numerischer Typkonstruktor
`Integer`*N* $\longrightarrow$ *Type* Namenskonvention numerischer Typen

`Integer`*N* (das Ergebnis von `nBitInteger`*N*) ist (per Namenskonvention) der Integer-Subset, der bei Zweierkomplementrepräsentation in *N* Bits unterzubringen ist. Es gibt normalerweise eine ganze Anzahl von `Integer`*N*-Typen (auf einem 68000-basierten System wahrscheinlich alles, was kleiner als `Integer32` ist).

`round` *(Number)* $\longrightarrow$ *Integer* essentielle numerische Funktion
`truncate` *(Number)* $\longrightarrow$ *Integer* essentielle numerische Funktion

`round` wandelt eine Zahl beliebigen Typs in den 'nächsten' entsprechenden Integer. Liegt die Zahl exakt zwischen zwei Integern, wird zum geraden Integer gerundet. `truncate` schneidet alles hinter dem Komma ab. Beispiele:

```
round(3.4)      ⟹    3
round(3.5)      ⟹    4
round(4.5)      ⟹    4
truncate(4.99)  ⟹    4
truncate(-4.99) ⟹   -4
```

Integer$_1$ `mod` *Integer*$_2$ $\longrightarrow$ *Integer* essentielle Funktion

`mod` ist die Modulo-Funktion.

`isEven` *(Integer)* $\longrightarrow$ *Bool* essentielle Funktion
`isOdd` *(Integer)* $\longrightarrow$ *Bool* essentielle Funktion

Diese Prädikate stellen fest, ob eine (ganze) Zahl gerade oder ungerade ist.

`Cardinal` $\longrightarrow$ *Type* essentieller numerischer Typ
`MaxCard` $\longrightarrow$ *Cardinal* essentielle numerische Konstante

`Cardinal` ist der 'Standard-Cardinal' in der gleichen Weise, wie `Integer` der 'Standard-Integer' ist. Ein typischer Wert für `MaxCard` ist 16777215 (24 Bit).

`nBitCardinal` *(Cardinal)* $\longrightarrow$ *Type* numerischer Typkonstruktor

`Cardinal`*N* ist (per Namenskonvention) der Cardinal-Subset, der in *N* Bits unterzubringen ist.

Reelle Zahlen

`Real` $\longrightarrow$ *Type* essentieller numerischer Typ
`ShortReal` $\longrightarrow$ *Type* numerischer Typ
`LongReal` $\longrightarrow$ *Type* numerischer Typ
`SmallestReal` $\longrightarrow$ *Real* numerische Konstante
`LargestReal` $\longrightarrow$ *Real* numerische Konstante
`SmallestShortReal` $\longrightarrow$ *ShortReal* numerische Konstante

`LargestShortReal` $\longrightarrow$ *ShortReal* numerische Konstante
`SmallestLongReal` $\longrightarrow$ *LongReal* numerische Konstante
`LargestLongReal` $\longrightarrow$ *LongReal* numerische Konstante

`Real`s sind Fließkomma-Zahlen nach IEEE 754. Es gibt drei `Real`-Varianten unterschiedlicher Genauigkeit. Der "Standard"-`Real` kann mit dem `ShortReal` oder dem `LongReal` identisch sein. Die Konstant `SmallestReal` beschreibt den kleinsten `Real` > 0, `LargestReal` den größten darstellbaren `Real` $< +\infty$.

Typische Längen für die drei Real-Typen sind 32 Bit (24+8), 64 Bit (48+16) und 80 Bit (64 + 16).

Uneigentliche Zahlen

`plusInfinity` $\longrightarrow$ *Real* numerische Konstante
`minusInfinity` $\longrightarrow$ *Real* numerische Konstante
`pointInfinity` $\longrightarrow$ *Real* numerische Konstante
`plusInfiniteSmall` $\longrightarrow$ *Real* numerische Konstante
`minusInfiniteSmall` $\longrightarrow$ *Real* numerische Konstante

Es gibt fünf 'uneigentliche' Zahlenkonstanten (Exceptional Numbers), die, respektive, für '$+\infty$', '$-\infty$', den 'Punkt Unendlich' der komplexen Zahlen, den 'unendlich kleinen positiven Wert' und den 'unendlich kleinen negativen Wert' ($\frac{1}{+\infty}$ und $\frac{1}{+\infty}$) steht. Die uneigentlichen Zahlen sind `Real`–Konstanten, die als Ergebnis von Overflow und Underflow auftreten können, nie aber als Ergebnis saturierter Funktionen.

Fixpunktzahlen

`ScaledFixpoint` *(Type,Cardinal$_2$)* $\longrightarrow$ *Type* numerischer Typ

Eine Fixpunktzahl ist ein skalierter Integer, d.h. ein Element der Menge $\{n \cdot k, n \in$ `Integer`$\}$. k ist dabei typischerweise ein ziemlich kleiner Wert, z.b. 0.01 (wenn man mit DN-Beträge repräsentieren will), 0.0005 (wenn eine 5ms-Clock die Zeit schlägt) oder 0.000000125 (8 Mhz).

`ScaledFixpoint(`A`,`B`)` definiert den Fixpunkttypen, der dadurch definiert wird, daß der Integer-Typ A als Vielfaches der (wahrscheinlich reellen) Zahle B interpretiert wird. So beschreibt `ScaledFixpoint(Cardinal16,0.01)` den Zahlenbereich zwischen 0 und 655.35, in Schritten zu 0.01 aufgeteilt.

`FixInt` *(Cardinal,Cardinal)* $\longrightarrow$ *Type* numerischer Typ
`FixCard` *(Cardinal,Cardinal)* $\longrightarrow$ *Type* numerischer Typ
`FixInt`*NM* $\longrightarrow$ *Type* Typ-Namenskonvention
`FixCard`*NM* $\longrightarrow$ *Type* Typ-Namenskonvention

Die am häufigsten auftretenden Fixpunkttypen haben die Form $\{\frac{a}{2^B}, a \in 0..2^{A+B}-1\}$. Genau diesen Typen definiert `FixCard(`A`,`B`)`. Ein typisches Beispiel ist `FixCard(0,16)`. Dieser Typ beschreibt die Zahlen zwischen 0 und 0.999984741, in Schritten von $1.52 * 10^{-5}$. Man kann sich vorstellen, daß `FixCard(`A`,`B`)`

eine Binärzahl mit A Stellen vor und B Stellen nach dem Komma ist. Eine ebenfalls häufige Variante ist der Typ `FixInt(`A`,`B`)`, der die durch $\{\frac{a}{2^B}, a \in -2^{A-1}..2^{A-1}-1>\}$ beschriebenen Objekte enthält. `FixInt(1,15)` beschreibt die Zahlen zwischen -1 und 0.999969482, in Schritten von $3.05*10^{-5}$. `FixInt(0,16)` beschreibt die Zahlen im Bereich $[-0.5...0.499984741]$, in Schritten von 2^{-16}, also 0.000015259. Kleinere Intervalle (etwas `Integer(-2,18)`) sind, obwohl denkbar, nicht erlaubt.

Als Namenskonvention gilt, daß ein Typ `FixCard`*NM* für `FixCard(N,M)` steht. Beispiele:

```
FixCard0016= FixCard(0,16)
FixInt0115 = FixInt(1,16)
FixCard44  = FixCard(4,4)
FixInt48 = FixInt(4,8)
```

Von besonderem Interesse sind die Fixpunktzahlen mit Beträgen ≤ 1, da dies die Standard-Ergebnistypen der trigonometrischen Funktionen sind. Wenn eine Implementierung Fixpunktzahlen von maximal 16 Bit implementiert, so wird z.B. `sin` Ergebnisse vom Typ `FixInt0116` liefern, d.h. Zahlen im Intervall $[-1...1-2^{-15}]$.[20] Gleichermaßen würde die `atan`–Funktion (die als Wertebereich $-\pi/2...\pi/2$ hat) auf `FixInt0216` abbilden.

Komplexe Zahlen

`Complex` *(Type)* $\longrightarrow$ *Type*	numerischer Typ
`makeComplex` *(Number*$_1$*, Number*$_2$*)* $\longrightarrow$ *Complex*	numerische Funktion
`realPart` *(Complex)* $\longrightarrow$ *Number*	numerische Funktion
`imagPart` *(Complex)* $\longrightarrow$ *Number*	numerische Funktion
`phase` *(Complex)* $\longrightarrow$ *Number*	numerische Funktion

Eine komplexe Zahl wird als Tupel zweier nicht-komplexer Zahlen gleichen Typs implementiert. `makeComplex(`a`,`b`)` bildet die komplexe Zahl $a+ib$. (Man könnte auch `a+sqrt(-1)*b` schreiben). `realPart` ermittelt den reellen Anteil einer komplexen Zahl, `imagPart` den imaginären Anteil. `phase` bestimmt den Winkel der Polarrepräsentation einer komplexen Zahl. (Der Betrag kann durch `abs` festgestellt werden.)

Zeit

`Duration` $\longrightarrow$ *Type*	essentieller Typ
n `sec` $\longrightarrow$ *Duration*	essentieller Typ
`MinDuration` $\longrightarrow$ *Number*	essentielle Konstante

`Duration` ist der Typ, der als Zeit-Argument für Suspensions benötigt wird. `sec` wandelt numerische Argumente in `Duration`–Größen um (`1 sec`$\equiv$ eine Sekunde). `MinDuration` ist die kleinste sinnvolle Zeiteinheit (für Suspensions).

[20] Daß die '1' in diesem Intervall nicht darstellbar ist, sollte nicht von Belang sein. Implementierer, die sich daran stören, können natürlich auch `FixInt0216` benutzen.

`Time` $\longrightarrow$ *Type* Typ
`SystemClock` *()* $\longrightarrow$ *Time* Prozedur
`Year` *(Time)* $\longrightarrow$ *Cardinal* Funktion
`Month` *(Time)* $\longrightarrow$ *Cardinal* Funktion
`Day` *(Time)* $\longrightarrow$ *Cardinal* Funktion
`Hour` *(Time)* $\longrightarrow$ *Cardinal* Funktion
`Minute` *(Time)* $\longrightarrow$ *Cardinal* Funktion
`Second` *(Time)* $\longrightarrow$ *Cardinal* Funktion
`makeTime` *(*$Cardinal_{\mathrm{Year}}$ *...* $Cardinal_{\mathrm{Second}}$*)* $\longrightarrow$ *Time* Funktion
`addTime` *(Time,Duration)* $\longrightarrow$ *Time* Funktion
$Time_1$ `-` $Time_2$ $\longrightarrow$ *Duration* Funktion

Der Funktionsaufruf `SystemClock()` liefert einen Wert vom Typ `Time`. Die sechs Selektor-Funktionen zerlegen ein Objekt dieses Typs in seine Bestandteile. Für die einzelnen Werte gelten folgende Wertebereiche:
`Year`(*Time*) ≥ 1990
$1 \geq$ `Month`(*Time*) ≥ 12
$1 \geq$ `Day`(*Time*) ≥ 31
$1 \geq$ `Hour`(*Time*) ≥ 24
$1 \geq$ `Minute`(*Time*) ≥ 60
$1 \geq$ `Second`(*Time*) ≥ 60

Ein `Time`–Objekt kann durchaus mehr repräsentieren (z.B. die aktuelle Zeitzone oder Zeitwerte bis in den Nanosekundenbereich)!

`makeTime` ist eine sechsstellige Funktion, die ein Objekt vom Typ Time erzeugt.

`addTime` addiert eine Zeitdauer zu einer Zeit;

Der "`-`"-Operator bestimmt die Länge eines durch zwei Zeitpunkte vorgegebenen Zeitraums. Es handelt sich hierbei um denselben (überladenen) Operator, der auch für normale Zahlen benutzt wird; nur sind Argumente und Resultat keine Zahlen (auch wenn sie so interpretiert werden können).

Strings
`String` $\longrightarrow$ *Type* Typ

Strings sind Sequenzen von Zeichen (was ein Zeichen ist, wird nicht erklärt). Es steht einer Implementierung frei, Strings als Vektoren, Listen oder wie auch immer zu implementieren. String-Konstanten sind in Anführungszeichen (") geschlossen.

`AsciiToString` *(Cardinal)* $\longrightarrow$ *String* Funktion

`AsciiToString`(n) erstellt einen String, der äquivalent zum ASCII-Zeichen n ist. Dies gilt auch für die Kontrollzeichen BS, CR, LF, FF und DEL.

String$_1$ `++` *String*$_2$ $\longrightarrow$ *String* Funktion

'`++`' verkettet zwei Strings.

`length` *(String)* $\longrightarrow$ *Cardinal* Typ

`length` bestimmt die Länge eines Strings.

`ldrop` *(Cardinal, String)* $\longrightarrow$ *String* Typ
`rdrop` *(Cardinal, String)* $\longrightarrow$ *String* Typ

`ldrop(`*n*,*s*`)` entfernt *n* Zeichen von links aus dem String *s* (`rdrop` analog von rechts). Ist *n*>`length(`*s*`)`, so ist der leere String das Ergebnis. Beispiele:

`NumberToString` *(Number)* $\longrightarrow$ *String* Typ

Diese Funktion erstellt die textuelle Repräsentation einer Zahl.

2.3.5 Arrays

`Array` $\longrightarrow$ *Type* essentieller Typ

`Array` ist die Obermenge aller Array-Typen egal welcher Dimension. Arrays sind Sammlungen von Objekten (meist gleichen Typs), die über Index-Tupel ausgewählt werden können. Als Indizes dienen Kardinal-Zahlen. Arrays sind (wie Listen) privil gierte Typen: wenn eine Funktion $\mathcal{F}$ auf ein Array angewandt wird, und $\mathcal{F}$ ist nicht über den entsprechenden Array-Typ definiert, so wird ein *automatisches Mapping* vorgenommen, d.h. $\mathcal{F}$ wird (in unspezifizierter Reihenfolge) auf alle Elemente des Arrays angewandt; die Ergebnisse werden in einem gleichgeformten Array gesammelt. Dies funktioniert für alle Funktionen (nicht nur primitive), egal welcher Stelligkeit. Beispiel:

`[3,4,5]+[10,20,30]` $\Longrightarrow$ `[13,24,35]`

Die Arrays müssen gleich groß sein:

`[1,2,3]*[3,4])` $\Longrightarrow$ *Fehlermeldung*

Wenn nur einige Argumente Arrays sind, werden die anderen automatisch zu entsprechenden Arrays gewandelt:

`[3,4,5]+10` $\Longrightarrow$ `[13,14,15]`

`MatrixLiteral([1,1,1,1],2,3,4)` $\Longrightarrow$

1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

Wenn einige Funktionsargumente Arrays sind, andere Listen, so müssen die Listen Listen von Arrays sein. (Sind sie es nicht, versagt das Auto-Mapping). Das Mapping erfolgt voll rekursiv:

`[[1,2],3,4]+[[10,20],0,30]` $\Longrightarrow$ `[[11,22],3,34]`

Zur Notation der Beispiele: Vektoren werden durch Aufzählung ihrer Elemente, geklammert in `[` und `]`, repräsentiert. Dies entspricht der Notation von Vektor-Literalen. Es gibt keine derartige Notation für Matrizen, deswegen wird eine Matrix direkt als Matrix dargestellt. Zur Klarheit wird sie in einem Rahmen gesetzt.

`sizeOf` *(Array)* $\longrightarrow$ *Cardinal* essentielle Funktion

`sizeOf(`*A*`)` stellt fest, wieviele Elemente *A* hat. Eine Matrix der Größe 4*5 hat 20 Elemente. Ein Vektor, der 4 Vektoren der Größe 5 enthält, hat 4 Elemente!

`Map` *(Function, Array)* $\longrightarrow$ *Array* essentielle Funktion
`DestructiveMap` *(Function, Array)* $\longrightarrow$ *Array* Funktion

Die Funktion (die vom Typ `(Obj -> Obj)` sein muß) wird auf jedes Element des Arrays angewandt. (Die Reihenfolge ist nicht definiert.) Die Resultate werden in einem gleichgeformten neuen Array gesammelt. **Imperativ!**

`DestructiveMap` ist die destruktive Version von `Map`: das Argument-Array wird überschrieben.

Vektoren

`Vector` $\longrightarrow$ *Type* essentieller Typ

Vektoren sind eindimensionale Arrays. Die Elemente eines Vektors der Größe n sind von $0 \ldots n-1$ durchnumeriert.

`CreateVector` *(Cardinal, Function)* $\longrightarrow$ *Vector* essentielle Funktion

`CreateVector(`*n,f*`)` konstruiert einen Vektor der Länge n, der an der Stelle i durch *f(i)* definiert (initialisiert) ist. f muß daher eine Funktion des Typs `Cardinal -> Obj` sein.

`[`Obj_1, ... , Obj_n`]` $\longrightarrow$ *Vector* essentielle Notation

Die eckigen Klammern sind keine 'Funktion', sondern eine Schreibhilfe zur Notation von Vektoren. Die Klammern können beliebig viele Argumente haben (u.U. durch eine Implementierungsschranke auf z.B. 255 beschränkt) und konstruieren aus ihnen einen Vektor.

Vector `(`*n*`)` $\longrightarrow$ *Object* essentielle Funktion

Vektoren sind ihre eigene Selektionsfunktion: ein Vektor ist, funktional gesehen, eine Funktion, die Indizes auf Werte abbildet. Also wird dies auch so geschrieben.

`ChangeVector` *(Vector, Cardinal, Object)* $\longrightarrow$ *Vector* essentielle Funktion
`DChangeVector` *(Vector, Cardinal, Object)* $\longrightarrow$ *Vector* Funktion

`ChangeVector(`*V,i,o*`)` liefert einen neuen Vektor, der mit V an allen Stellen außer i identisch ist; dort ist der Wert o. `DChangeVector` überschreibt V an der **Imperativ!**

Stelle i mit o. Das "D" steht für "destructive".

`Reduce` *(Function, Vector)* $\longrightarrow$ *Obj* essentielle Funktion
`RReduce` *(Function, Vector)* $\longrightarrow$ *Obj* essentielle Funktion

`Reduce`(f, V) wendet die zweistellige Funktion f von links nach rechts auf den Vektor V an. Beispiel:

`Reduce(", [1,2,3,4]) = ((1-2)-3)-4`

`RReduce` arbeitet wie `Reduce`, aber von rechts nach links. Die Vektoren müssen die Mindestgröße 2 haben.

`rev` *(Vector)* $\longrightarrow$ *Vector* Funktion

`rev` dreht einen Vektor um, d.h.

```
rev([1,2,3])   ⟹   [3,2,1]
```

`evens` *(Vector)* $\longrightarrow$ *Vector* Funktion
`odds` *(Vector)* $\longrightarrow$ *Vector* Funktion
`merge` *(Vector$_1$, Vector$_2$)* $\longrightarrow$ *Vector* Funktion

`evens`(V) liefert den Vektor $A\prime$, der nur aus den Elementen aus A besteht, die an 'geradzahligen' Positionen stehen. `odds` liefert entsprechend den 'ungeraden' Teilvektor. `merge` vermischt die beiden Vektoren zu ihrem Ausgangsvektor.

```
evens([10,20,30,40])   ⟹   [20,40]
 odds([10,20,30,40])   ⟹   [10,30]
merge([1,2,3],[4,5,6])   ⟹   [1,4,2,5,3,6]
        evens([10])   ⟹   []
         odds([])   ⟹   []
```

`subVector` *(Vector, Cardinal$_1$, Cardinal$_2$)* $\longrightarrow$ *Vector* Funktion
`compose` *(Vector$_1$, Vector$_2$)* $\longrightarrow$ *Vector* Funktion

`subVector`(A, i, j) bildet den Vektor, der aus den Elementen $A[i]...A[j-1]$ besteht.
`compose`(A, B) bildet den Vektor, der aus der Aneinanderreihung der Vektoren A und B besteht.

```
subVektor([1,2,3,4,5,6],0,3)   ⟹   [1,2,3]
  subVektor([1,2,3,4,5,6],3,4)   ⟹   [4]
   subVektor([1,2,3,4,5,6],3,3)   ⟹   []
subVektor([1,2,3,4,5,6],3,2)   ⟹   Fehlermeldung
subVektor([1,2,3,4,5,6],-3,2)   ⟹   Fehlermeldung
subVektor([1,2,3,4,5,6],3,20)   ⟹   Fehlermeldung
  compose([1,2,3],[4,5,6])   ⟹   [1,2,3,4,5,6]
```

Matrizen

`Matrix` $\longrightarrow$ *Type* essentieller Typ

Matrizen sind zweidimensionale Arrays. Die Elemente einer Matrix der Größe $m * n$ sind von $0, 0 \ldots m-1, n-1$ durchnumeriert. Die Reihenfolge ist

$$A = \begin{matrix} A[1,1] & \cdots & A[1,n] \\ \vdots & \ddots & \vdots \\ A[m,1] & \cdots & A[m,n] \end{matrix}$$

`CreateMatrix` *(Card*$_1$*, Card*$_2$*, Function)* $\longrightarrow$ *Matrix* essentielle Funktion

`CreateMatrix(`*m,n,f*`)` konstruiert eine Matrix der größe *m*n*, die an der Stelle i, j durch *f(i, j)* definiert (initialisiert) ist. *f* muß daher eine Funktion des Typs `Cardinal, Cardinal -> Obj` sein.

`MatrixLiteral` *(Vector*$_1$*, ... , Vector*$_n$*)* $\longrightarrow$ *Matrix* essentielle Funktion

Dies ist das genaue Gegenstück zum den eckigen Klammern, nur etwas unbeholfener geschrieben (Matrizen müssen seltener notiert werden, und es gibt kaum weitere Klammern). Die Matrix

1	2	3	4
5	6	7	8
9	10	11	12

wird durch die Expression

```
MatrixLiteral([1,2,3,4],
              [5,6,7,8],
              [9,10,11,12])
```

gebildet. Die Vektoren müssen gleich groß sein.

`hSize` *(Matrix)* $\longrightarrow$ *Number* Funktion
`vSize` *(Matrix)* $\longrightarrow$ *Number* Funktion

`hSize(`A`)` liefert die 'Breite' von A, `vSize(`A`)` die 'Höhe'.

`hSize(CreateMatrix(`m`,`n`,`f`))` $\Longrightarrow$ n
`vSize(CreateMatrix(`m`,`n`,`f`))` $\Longrightarrow$ m

Matrix `(`*Card, Card*`)` $\longrightarrow$ *Object* essentielle Funktion

`A(`*n,m*`)` Selektiert das Element in der *n*-ten Zeile und der *m*-ten Spalte aus der Matrix `A`. Genauso wie Vektoren einstellige Funktionen sind, sind Matrizen zweistellige Funktionen.

`ChangeMatrix` *(Matrix, Card, Card, Obj)* $\longrightarrow$ *Matrix* essentielle Funktion

`DChangeMatrix` *(Matrix, Card, Card, Obj)* $\longrightarrow$ *Matrix* Funktion

`ChangeMatrix`(*M,i,j,o*) liefert eine neue Matrix, die mit *M* an allen Stellen außer *i,j* identisch ist; dort ist der Wert *o*. `DChangeMatrix` überschreibt *A* an der Stelle *i,j* mit *o*. Das "`D`" steht für "destructive". **Imperativ!**

`UnitMatrix` *(Cardinal)* $\longrightarrow$ *Matrix* Funktion

`UnitMatrix`(*N*) liefert eine Einheitsmatrix der Größe *N*.

`UnitMatrix(3)` $\Longrightarrow$

1	0	0
0	1	0
0	0	1

`hrev` *(Matrix)* $\longrightarrow$ *Matrix* Funktion
`vrev` *(Matrix)* $\longrightarrow$ *Matrix* Funktion

`hrev` spiegelt einen Array an der horizontalen Mittelachse, `vrev` an der vertikalen.

`hrev(`

1	2	3	4
5	6	7	8
9	10	11	12

`)` $\Longrightarrow$

9	10	11	12
5	6	7	8
1	2	3	4

`vrev(`

1	2	3	4
5	6	7	8
9	10	11	12

`)` $\Longrightarrow$

4	3	2	1
8	7	6	5
12	11	10	9

`drev` *(Matrix)* $\longrightarrow$ *Matrix* Funktion
`orev` *(Matrix)* $\longrightarrow$ *Matrix* Funktion

`drev(`*Q*`)` spiegelt die Matrix *Q* an der Hauptdiagonalen. `orev(`*Q*`)` spiegelt die Matrix *Q* an der Nebendiagonalen

`drev(`

1	2	3	4
5	6	7	8
9	10	11	12

`)` $\Longrightarrow$

1	5	9
2	6	10
3	7	11
4	8	12

`orev(`

1	2	3	4
5	6	7	8
9	10	11	12

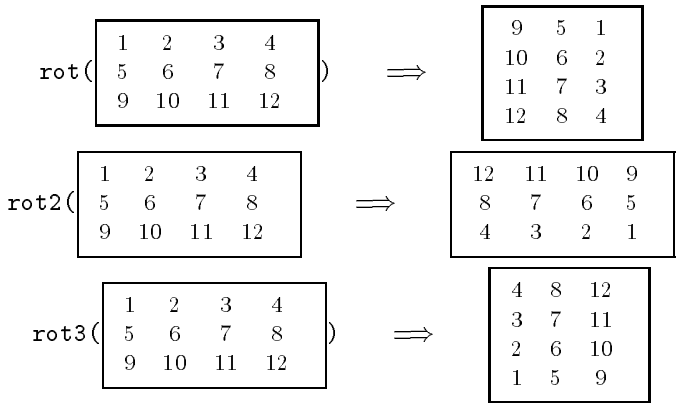
`)` $\Longrightarrow$

12	8	4
11	7	3
10	6	2
9	5	1

`rot` *(Matrix)* $\longrightarrow$ *Matrix* Funktion
`rot2` *(Matrix)* $\longrightarrow$ *Matrix* Funktion
`rot3` *(Matrix)* $\longrightarrow$ *Matrix* Funktion

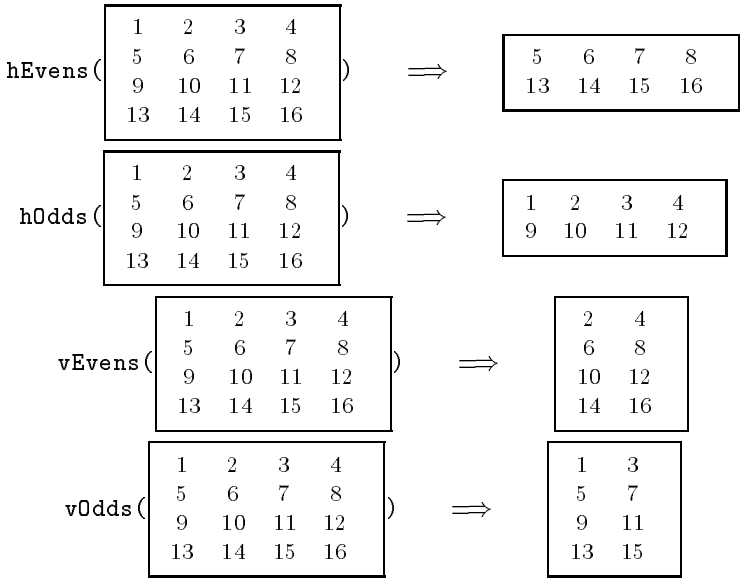
Die `rot`–Funktionen rotieren eine Matrix im Uhrzeigersinn um ihren Mittelpunkt um, respektive, ein Vierteldrehung, eine halbe Drehung und eine Drei-

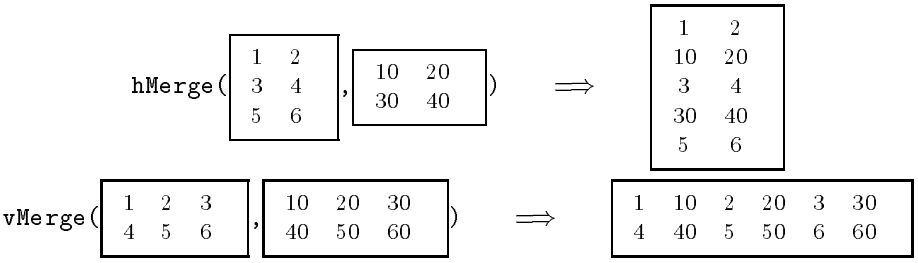
vierteldrehung.



`hEvens` *(Matrix)* $\longrightarrow$ *Matrix*	Funktion
`hOdds` *(Matrix)* $\longrightarrow$ *Matrix*	Funktion
`vEvens` *(Matrix)* $\longrightarrow$ *Matrix*	Funktion
`vOdds` *(Matrix)* $\longrightarrow$ *Matrix*	Funktion
`hMerge` *(*$Matrix_1$*,*$Matrix_2$*)* $\longrightarrow$ *Matrix*	Funktion
`vMerge` *(*$Matrix_1$*,*$Matrix_2$*)* $\longrightarrow$ *Matrix*	Funktion

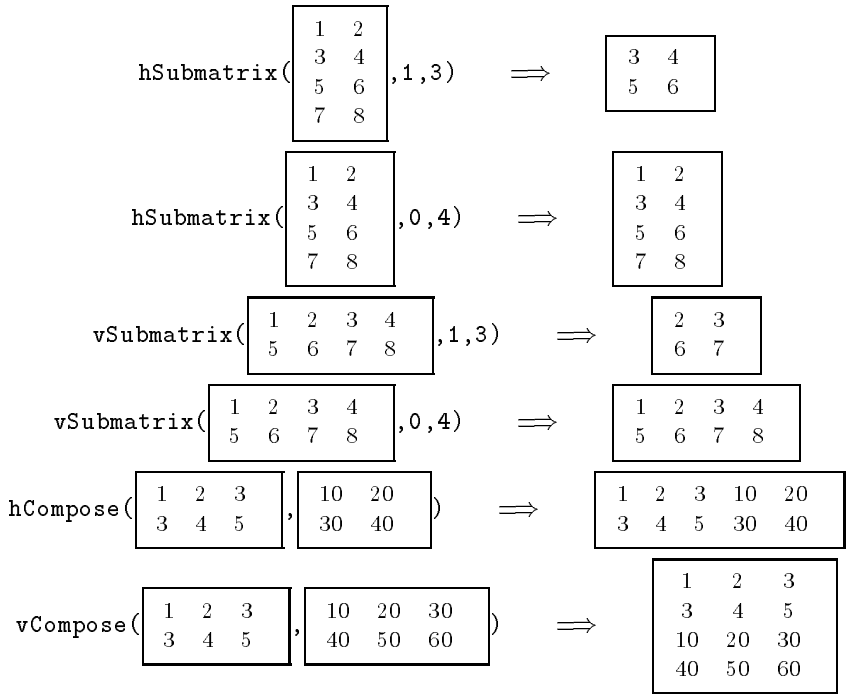
`hEvens(`A`)` liefert die Matrix $A\prime$, die aus den 'geraden Zeilen' von A besteht. `hOdds` liefert dementsprechend die 'ungeraden Zeilen', `vEvens` die 'geraden Spalten' und `vOdds` die 'ungeraden Spalten'. `hMerge` mischt zwei Arrays zeilenweise, `vMerge` spaltenweise. Die Merge-Operationen muß die zweite Matrix (der, der die 'odds' enthält, genauso groß oder eine Zeile/Spalte kleiner als die erste sein.





`hSubMatrix` *(Matrix, Card$_1$, Card$_2$)* $\longrightarrow$ *Matrix* Funktion
`vSubMatrix` *(Matrix, Card$_1$, Card$_2$)* $\longrightarrow$ *Matrix* Funktion
`hCompose` *(Matrix,Matrix)* $\longrightarrow$ *Matrix* Funktion
`vCompose` *(Matrix,Matrix)* $\longrightarrow$ *Matrix* Funktion

`hSubMatrix`(A, i, j) bildet die Matrix die aus den Zeilen $i \ldots j-1$ der Matrix A besteht.
`vSubMatrix` liefert entsprechend einen Zeilenbereich.
`hCompose`(A, B) liefert die Nebeneinanderreihung der Arrays A und B. A und B müssen die gleiche Zeilenzahl (=Höhe) haben.
`vCompose`(A, B) liefert die Übereinanderreihung der Arrays A und B. A und B müssen die gleiche Spaltenzahl (=Breite) haben.



`row` *(Matrix, Cardinal)* $\longrightarrow$ *Vector* Funktion

`col` *(Matrix, Cardinal)* $\longrightarrow$ *Vector* Funktion

`row` und `col` selektieren einzelne Zeilen bzw. Spalten einer Matrix.

`row(`
1	2	3	4
5	6	7	8
9	10	11	12

`,0)` $\Longrightarrow$ `[1,2,3,4]`

`col(`
1	2	3	4
5	6	7	8
9	10	11	12

`,0)` $\Longrightarrow$ `[1,5,9]`

`diag` *(Matrix, Cardinal)* $\longrightarrow$ *Vector* Funktion
`ndiag` *(Matrix, Cardinal)* $\longrightarrow$ *Vector* Funktion

`diag`(M, d) liefert eine zur Hauptdiagonalen parallele Diagonale von M als Vektor. d bestimmt, welche Diagonale dies ist: es wird der Schnitt der Matrix mit der Diagonalen [M(d,0),M(d+1,1),...] geliefert. Dies führt dazu, das $d = 0$ die Hauptdiagonale, $d > 0$ die d-te Diagonale 'unter' und $d < 0$ die d-te Diagonale 'über' der Hauptdiagonalen liefert. `ndiag`M, d liefert entsprechend zur Nebendiagonale parallele Diagonalen, die durch den Schnitt von M (M habe die Breite b_m) mit [M(d,b_m),M(d+1,b_m-1),...] definiert sind.

`diag(`
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

`,1)` $\Longrightarrow$ `[5,10,15]`

`diag(`
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

`,-2)` $\Longrightarrow$ `[3,8]`

`ndiag(`
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

`,1)` $\Longrightarrow$ `[8,11,14]`

`ndiag(`
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

`,-2)` $\Longrightarrow$ `[2,5]`

`ndiag(`
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

`,4)` $\Longrightarrow$ *Fehlermeldung*

`det` *(Matrix)* $\longrightarrow$ *Number* Funktion

`det`(A) berechnet die Determinanter der quadratischen numerischen Matrix A.

`hReduce` *(Function, Matrix)* $\longrightarrow$ *Vector* Funktion
`vReduce` *(Function, Matrix)* $\longrightarrow$ *Vector* Funktion

`hReduce`(F,M) wendet die zweistellige Funktion F auf die Zeilenvektoren von M an; jeder Zeilenvektor wird so auf ein Element des Ergebnisvektors abgebildet. `vReduce` tut dasselbe mit den Spaltenvektoren.

`hReduce('-',` $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$ `)` $\Longrightarrow$ `[-8,-16,-24,-32]`

`vReduce('-',` $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$ `)` $\Longrightarrow$ `[-26,-28,-30,-32]`

2.3.6 Listen, Streams und Pipes

Die Liste ist in der Signalverarbeitung *der* grundlegende Typ überhaupt. Listen modellieren *Signale*. Ein Signal ist eine geordnete Folge von Zahlenwerten, die u.U. in Blöcken (Vektoren oder Matrizen) gesammelt sind. In der DSP repräsentieren diese Zahlen synchrone Abtastwerte (*Samples*).

Pipes und *Streams* sind spezielle Formen von Listen – genauer gesagt, speziell *erzeugte* Listen. Ein Stream ist eine Liste, deren Auswertung durch den `delay`–Operator verzügert ist. Eine Pipe ist eine Liste, deren Auswertung durch eine Suspension verzögert ist.

Ein typisches Beispiel für einen Stream liefert das NOTCH-Filter-Programm: (das Beispiel ist dem *Silage Reference Manual*[21], pp. 1-4, entnommen)

```
func NOTCH(b0,b1,b2.a1,a2 :Number)(in :List)=
  let
     out = b0*in + s2
     s2  = 0 :: delay(b1*in + a2*out + s1)
     s1  = 0 :: delay(b2*in + a1*out)
  in
     out
  endlet
```

Ein '`0::delay` *tmp*'–Ausdruck entspricht dem '@'-Operator ('delay' genannt) von *Silage*, d.h. dem Rückgriff auf den 'vorigen' Sample. Die Notwendigkeit des `delay`s liegt in der Rekursivität der Streams: der `out`-Stream wird als Input

wiederverwendet. Da ALDiSP call-by-value auswertet, würde es hier zu einem Problem kommen.[21]

Im Gegensatz zu Streams tauchen Pipes selten explizit auf. Sie sind implizit durch die synchronen und asynchronen Input-Objekte gegeben. Als Beispiel soll die (primitive) Funktion `PipeFromPort` dienen, die so implementiert sein könnte:

```
func PipeFromPort(p:InputPort) =
  ReadFromPort(p) :: PipeFromPort(p)
```

Was in dieser Definition durch die Semantik der Ports implizit geschieht, sieht explizit so aus:

```
func PipeFromPort(p:InputPort) =
  suspend ReadFromPort(p) :: PipeFromPort(p)
    until isPortReadable(p)
   within 0,0
```

Suspensions 'pflanzen sich fort': der Zugriff auf eine Suspension suspendiert den Zugreifer. Daher liefert eine Funktion des Typs `(list -> list)`, die auf eine Pipe zugreift, eine Pipe als Ergebnis. Dies steht im Gegensatz zu den Streams!

`List` *(Type)* $\longrightarrow$ *Type*	Typ
`null` $\longrightarrow$ *List*	Konstruktor
`cons` *(Object, List)* $\longrightarrow$ *List*	Konstruktor
Object `::` *List* $\longrightarrow$ *List*	Konstruktor
`head` *(List)* $\longrightarrow$ *Object*	Selektor
`tail` *(List)* $\longrightarrow$ *List*	Selektor
`isEmpty` *(List)* $\longrightarrow$ *Bool*	Prädikat
`select` *(Cardinal,List)* $\longrightarrow$ *Object*	Funktion
`length` *(List)* $\longrightarrow$ *Cardinal*	Funktion
`reduce` *(Function,List)* $\longrightarrow$ *Object*	Funktional

Man kann sich Listen als den durch die folgenden Definitionen eingeführten Datentyp vorstellen:

```
abstype list = null
             | cons(head, tail: list)

r-infix '::'
func '::' = cons

func isEmpty(aList: list) =
```

[21] Oder auch nicht: Das `in`–Signal wird wahrscheinlich eine Pipe sein; die impliziten Mappings würden also irgendwann durch eine Suspension aufgehalten. Es ist trotzdem gut, hier Delays einzubauen, da man nicht im Voraus wissen kann, wie weit die `out`–Liste überhaupt gebraucht wird.

```
  aList is null

func isSingleton(aList: list) =
  not isEmpty(aList) and isEmpty(tail(aList))

func select(n,aList: list) =
  if isEmpty(aList) then bottom
   | n=0            then head(aList)
                    else select(n-1,tail(aList))
  endif

func length(aList: list) =
  if isEmpty(aList) then 0
                    else 1+length(tail(aList))
  endif

func reduce(f: Function, aList: list) =
  let
     r(akku, aList) =
        if isSingleton(aList)
                then f(akku,head(aList))
                else r(f(akku,head(aList)),tail(aList))
        endif
  in
     if length(aList)<2 then bottom
                        else r(head(aList),tail(aList))
     endif
  endlet
```

`isEmpty` testet, ob eine Liste leer ist. `isSingleton` testet, ob eine Liste einelementig ist. `select`(n,l) liefert das n-te der Liste l. `reduce` schließlich wendet eine zweistellige Funktion von links nach rechts auf die Elemente einer Liste an (die mindestens die Länge 2 haben muß). Beispiel:

`reduce('-',cons(a,cons(b,cons(c,null))))` $\Longrightarrow$ `(a-b)-c`

Es steht jeder Implementierung frei, die elementaren Listenfunktionen so zu definieren, wie es ihr gefällt, solange die Funktionalität, die durch die o.a. Definitionen vorgegeben ist, beibehalten bleibt.

Listen sind, wie Arrays, privilegierte Datenstrukturen: Wenn eine (beliebige) Funktion auf Argumente angewandt wird, über die sie nicht definiert ist, und mindestens eins der Argumente ist eine Liste, so werden aus den nicht-Listen-Elementen Listen gebildet (d.h, aus `1` wird die unendliche lange Liste, die nur Einsen enthält), und die Funktion wird auf die (Tupel der) Listenelemente angewandt. Die Listen dürfen endlich oder unendlich sein; die kürzeste Input-Liste bestimmt die Länge des Ergebnisses. Beispiel:

```
1::2::3::() + 10    ⟹    11::12::13::()
```

`ListToVector` *(List)* $\longrightarrow$ *Vector* Funktion
`VectorToList` *(Vector)* $\longrightarrow$ *List* Funktion

`ListToVector` wandelt eine (endliche) Liste in einen Vektor um. `VectorToList` tut das entgegengesetzte, wobei die enstehende Liste 'verzögert' sein sollte. Man kann sich die beiden Funktionen folgendermaßen definiert vorstellen:

```
func ListToVector(L:List) =
 let
   createVector(length(L),selectL)
 where
   func selectL(i) = select(i,L)
 endlet

func VectorToList(V:Vector) =
 let
   func RestList(i) =
     if i=sizeOf(V) then null
     else V(i) :: delay RestList(i+1)
     endif
 in
   RestList(0)
 endlet
```

2.3.7 Maschinentypen

Im Folgenden werden Typen beschrieben, die in der I/O benutzt werden. Dies sind keine 'Datentypen' im herkömmlichen Sinne, sondern Typen von 'Interface-Objekten'.

`Address` $\longrightarrow$ *Type* Typ
`CardinalToAddress` *(Cardinal)* $\longrightarrow$ *Address* Typ

Eine Adresse ist ein maschinenspezifischer Identifikator, über den Speicher adressiert wird. ALDiSP geht davon aus, das Maschinen-Objekte mit Speicherbereichen assoziiert werden.

Eine ALDiSP–Adresse muß nichts mit dem realen Adressierungsschema der Maschine zu tun haben – es wäre aber sinnvoll.

Anstelle der unhandlichen Funktion `CardinalToAddress` kann auch ein einfacher Cast `[!Address]` vorgenommen werden.

`Register` $\longrightarrow$ *Type* Typ
`InputRegister` *(Type)* $\longrightarrow$ *Type* Typ
`OutputRegister` *(Type)* $\longrightarrow$ *Type* Typ

`DeclareInputRegister` *(Type, Addr)* $\longrightarrow$ *InputRegister(Type)* Prozedur
`DeclareOutputRegister` *(Type, Addr)* $\longrightarrow$ *OutputRegister(Type)* Prozedur

Ein `Register` ist eine I/O-Variable, d.h. ein Input-Register ist eine Speicherstelle mit sich änderndem Inhalt, ein Output-Register etwas, wo hineingeschrieben werden kann. Register sind getypt. Typischerweise sind Register über `Cardinal`-*N*s oder `Fix`*NM*-s definiert.

`writeToRegister` *(OutputRegister,Object)* $\longrightarrow$ I/O-Prozedur
`readFromRegister` *(InputRegister)* $\longrightarrow$ *Object* I/O-Prozedur

Dies sind die Prozeduren zum Auslesen bzw. Beschreiben eines Registers. Diese Prozeduren sind zeitsensitiv – der Prozeß, der sie aufruft, muß einen definierten Zeitzustand haben, um sie rufen zu können. Beispiel:

```
proc SampleRegister(F : Duration; R: InputRegister ) =
  readFromRegister(R) ::
  suspend SampleRegister(F, R)
    until true
   within F,F
```

liefert Samples eines Registers. Ein Audio-Abtaster könnte so aussehen:

```
InReg = DeclareInputRegister(Integer16,[!Address]16#ffff0020)
SampleRate = (1/44100)sec
InPipe= SampleRegister(SampleRate, InReg)
```

Die entsprechende Prozedur, die in regelmäßigen Abständen in einen Port schreibt, ist:

```
proc WriteSamplesToRegister(FrameDuration : Duration;
                            R : Outputregister;
                            D : List) =
  seq
    suspend WriteSamplesToRegister(FrameDuration, R, tail(L))
      until true
     within FrameDuration,FrameDuration ;
    writeToRegister(R,head(L))
  endseq
```

Aus den Beispielen ist schnell ein lauffähiges Programm gemacht, z.B. ein einfacher unparametrisierter Lowpass-Filter:

```
func LowPass(in) =
 let
  out = 0 :: delay ((in+out)/2)
 in
  out
```

```
 endlet

net
 WriteSamplesToRegister(SampleRate,
   LowPass(SampleRegister(SampleRate, InReg)))
endnet
```

`Port` $\longrightarrow$ *Type*	Typ
`InputPort` *(Type)* $\longrightarrow$ *Type*	Typ
`OutputPort` *(Type)* $\longrightarrow$ *Type*	Typ
`DeclareInputPort` *(Type, Address)* $\longrightarrow$ *InputPort(Type)*	Prozedur
`DeclareOutputPort` *(Type, Address)* $\longrightarrow$ *OutputPort(Type)*	Prozedur
`writeToPort` *(OutputPort, Object)* $\longrightarrow$	I/O-Prozedur
`readFromPort` *(InputPort)* $\longrightarrow$ *Object*	I/O-Prozedur
`isPortReadable` *(InputPort)* $\longrightarrow$ *Bool*	I/O-Prozedur
`isPortWritable` *(OutputPort)* $\longrightarrow$ *Bool*	I/O-Prozedur

Ein Port ist ein 'asynchrones Register'. Ports repräsentieren zeitlich geordnete Sequenzen von Objekten. Input-Ports sind potentiell unendlich große Pufferspeicher. In der Deklaration, beim Lesen und Schreiben unterscheiden sich Ports in der Benutzung nicht von Registern; jedoch können Ports *blockieren*: Ein Lesezugriff auf einen Port, 'hinter' dem kein Objekt 'steht', blockiert den zugreifenden Prozeß, bis ein Objekt eintrifft (wie und wann dies geschieht, kann hier nicht erklärt werden – es ist I/O, also zeitlich und semantisch unspezifiziert). Gleichermaßen kann ein Schreibzugriff auf einen Port den schreibenden Prozeß blockieren. Die Blockierung vollzieht sich über den Suspension-Mechanismus. Man kann sich vorstellen, daß `readFromPort` und `writeToPort` folgendermaßen definiert sind:

```
proc readFromPort(P) =
 suspend <nicht-blockierendes Read>(P)
   until isReadable(P)
  within 0,0

proc writeToPort(P) =
 suspend <nicht-blockierendes Write>(P)
   until isWritable(P)
  within 0,0
```

`isReadable` und `isWritable` stellen fest, ob ein nicht-blockierendes Lesen bzw. Schreiben möglich ist. Ein offenkundiges Problem liegt vor, wenn mehrere Prozesse simultan auf einen Port zugreifen: ALDiSP kennt keine 'critical sections', also ist das Ergebnis eines derartigen Zugriffs unspezifiziert – Es liegt eine Fehlersituation vor, wenn mehr als ein Prozeß zur selben virtuellen Zeit existieren und I/O betreiben.

`File` $\longrightarrow$ *Type*	Typ
`InputFile` *(Type)* $\longrightarrow$ *Type*	Typ
`OutputFile` *(Type)* $\longrightarrow$ *Type*	Typ
`AccessFile` *(Type)* $\longrightarrow$ *Type*	Typ
`OpenInputFile` *(Type, Name)* $\longrightarrow$ *InputFile(Type)*	Prozedur
`OpenOutputFile` *(Type, Name)* $\longrightarrow$ *OutputFile(Type)*	Prozedur
`OpenAccessFile` *(Type, Name)* $\longrightarrow$ *AccessFile(Type)*	Prozedur
`CloseFile` *(File)* $\longrightarrow$	Prozedur
`writeToFile` *(File,Object)* $\longrightarrow$	I/O-Prozedur
`readFromFile` *(File)* $\longrightarrow$ *Object*	I/O-Prozedur
`setAccessPointer` *(File, Cardinal)* $\longrightarrow$ *Object*	I/O-Prozedur
`getAccessPointer` *(File)* $\longrightarrow$ *Cardinal*	I/O-Prozedur
`getFileSize` *(File)* $\longrightarrow$ *Cardinal*	I/O-Prozedur
`isEndOfFile` *(File)* $\longrightarrow$ *Boolean*	I/O-Prozedur
`isFileReadable` *(InputFile)* $\longrightarrow$ *Bool*	I/O-Prozedur
`isFileWritable` *(OutputFile)* $\longrightarrow$ *Bool*	I/O-Prozedur

Files werden als spezielle Ports behandelt. Es gibt drei Arten von Files:

- reine Lese-Files. Ihre Größe ist durch `getFileSize` feststellbar, das File-Ende ist durch `isEndOfFile` feststellbar.
- reine Schreib-Files
- Files, auf die man lesend und schreibend zugreifen kann.

  Diese Files haben eine feststellbare Größe und einen aktuellen Access-Pointer, der beliebig gesetzt werden kann. Der Access-Pointer gibt an, welches der Index das nächsten Objekts ist, das gelesen oder geschrieben wird. Für jedes *File* kann der Access-Pointer Werte zwischen 0 und `getFilesize(`*File*`)` annehmen.

Files sind deswegen Ports, weil sie ein blockierendes Verhalten zeigen können. `AccessFiles` können als Tripel, bestehend aus einem Input- einem Output- und einem Steuer-Port, modelliert werden.

`Interrupt` $\longrightarrow$ *Type*	Typ
`createInterrupt` *(Address)* $\longrightarrow$ *Interrupt*	Prozedur

Ein Interrupt ist ein Objekt, dessen einzige Funktion darin besteht, daß man darauf warten kann. Interrupts bringen keine Werte mit sich. Die typische Verwendungsform sieht so aus:

```
\"UberdruckMelder = createInterrupt(16#00fe4712)
 ...
suspend \"OffneVentil until \"Uberdruckmelder within 0,0
```

Ein Suspension, die von einem Interrupt abhängt, definiert einen *Handler*. Dieses Programmfragment installiert einen 'Einmal-Interrupt-Handler'. Ist kein solcher Handler installiert, wird der 'Null-Handler' aufgerufen, der nichts tut:

```
proc NullHandler(I : Interrupt) =
 supend NullHandler(I)
  until I within 0,0
```

Es darf immer nur einen Prozeß geben, der auf einen Interrupt wartet. Installiert sich ein neuer Handler, wird der alte gelöscht, d.h. nie aktiviert.

Interprozeß-Kommunikation

`Channel` $\longrightarrow$ *Type*	Typ
`ChannelOf` *(Type)* $\longrightarrow$ *Type*	Typ
`writeToChannel` *(Channel, Obj)* $\longrightarrow$	Prozedur
`readFromChannel` *(Channel)* $\longrightarrow$ *Obj*	Prozedur
`isChannelReadable` *(Channel)* $\longrightarrow$ *Bool*	Prozedur
`isChannelWritable` *(Channel)* $\longrightarrow$ *Bool*	Prozedur
`?` *Channel* $\longrightarrow$ *Obj*	Prefix-Prozedur
Obj `->` *Channel* $\longrightarrow$	Infix-Prozedur

Ein `Channel` ist eine Queue der Länge 1. Jeder Prozeß kann in jeden Channel hineinschreiben und aus jedem Channel lesen. Die Zugriffsprozeduren blockieren, d.h. ein schreibender Prozeß blockiert, bis ein lesender Prozeß auf den Channel zugreift; ein lesender Prozeß blockiert, bis etwas in den Channel geschrieben wird (Rendezvous).

Channel sind das einzige 'echte' IPC-Konstrukt. ALDiSP–Channel entsprechen dem gleichnamigen Konstrukt in *Occam*[9][22].

`?` und `->` sind die Abkürzungen für `ReadFromChannel` und `writeToChannel`. Wenn eine Implementierung einen 'echten Pfeil' (`'`$\rightarrow$`'`) anstelle von `'->'` anbieten kann, sollte sie dies tun.

Sonstiges

`Error` *(Obj .... Obj)* $\longrightarrow$ `bottom`	Prozedur

`Error` ist eine Prozedur, die Fehlerausgaben vornimmt. Das Ergebnis ist `bottom`. Es wird nicht garantiert, daß wirklich eine Fehlermeldung erscheint – in vielen Umgebungen wird es hierzu keine Möglichkeit geben.

Object `is` *Constructor* $\longrightarrow$ *Bool*	essentielles Prädikat

`is` ist kein 'echtes' Prädikat, sondern Teil des Mechanismus der abstrakten Datentypen. Es stellt fest, ob *Object* vom *Constructor* erzeugt worden ist.

[22] Allerdings ohne die elaborierten 'Protokoll'-Strukturen: Da ALDiSP von sich aus polymorph ist, sind diese nicht notwendig.

2.4 Die Arbeitsumgebung

Der einzige genormte Teil der Arbeitsumgebung, d.h. des Interpreters/Compilers, ist die Beschreibung der File-System-Schnittstelle.

Man kann davon ausgehen, daß ein ALDiSP-Programm in mehrere Files aufgeteilt ist, von denen jedes eins oder mehrere Module oder das "Hauptprogramm" enthält. Innerhalb der Sprache Modulnamen irgendwie in Dateinamen um) scheint mir sehr umständlich und fehlerträchtig – was, wenn das File-System bestimmte Zeichen nicht kennt?. ALDiSP sieht daher ein *Link-File* vor, das die Namen der Module mit ihren Aufenthaltsorten assoziiert. Das Format ist einfach:

LinkFile	$\longmapsto$	LinkLine +
LinkLine	$\longmapsto$	ModuleName <*Zwischenraum*> FileName <*Zeilen- oder File-Ende*> | Kommentar
Kommentar	$\longmapsto$	"\" <*Kommentar bis Zeilen- oder File-Ende*>

Die Modul-Namen entsprechen genau den ALDiSP-Namen, nur das Hauptmodul hat den (ansonsten nicht reservierten) Namen `main`, der daher nicht als Modulname auftauchen darf. Ein Link-File unter UNIX könnte so aussehen:

```
\ Demo-Linkdatei
main      \usr\pdv\mfx\test\main.ald
int16     \usr\pdv\mfx\test\stdints.ald
int24     \usr\pdv\mfx\test\ints.ald
int32     \usr\pdv\mfx\test\stdints.ald
i/o       \usr\pdv\mfx\test\iolib.ald
```

Das `main`-File enthält die Netzdefinitionen und eine Reihe von Importen, die anderen Files enthalten (ein oder mehrere) Module. Aus jedem File werden selektiv die Moduldefinitionen gelesen, die in der Linkdatei angegeben sind. Im Kontext des o.a. Beispiels heißt dies: enthält die Datei ...`\stdints.ald` ein Modul `int24`, so wird dies ignoriert; es kommt nicht zum Konflikt mit dem aus ...`\ints.ald` gelesenen Modul.

Kapitel 3

Die ALDiSP-Entwicklungsumgebung

3.1 Interpreter vs. Compiler

ALDiSP soll zuerst in der CADiSP-Umgebung als Zwischensprache ("unterhalb" einer interaktiven objekt-orientierten graphischen Benutzeroberfläche, aber "über" der Assemblersprache der realen Maschine) eingesetzt werden. In dieser Umgebung sollen primär DSP-Algorithmen für (mehr oder weniger) beliebige Hardware-Konfigurationen erstellt werden, d.h. für

- Ein- oder Mehrprozessorsysteme
- Verschiedene DSP-Prozessoren
- verschiedenste Hardware-Unterstützungen (Timer, I/O, Speicher).

DSP-Anwendungen zeichnen sich durch folgenden Eigenschaften aus:

- Oft muß der Code im ROM stehen (eingebettete Applikationen z.B. in der Telekommunikation).
- Der Code muß oft extrem klein sein.
- Der Code muß fast immer extrem schnell sein.

Gerade der letzte Punkt ist wichtig: der Code, der vom ALDiSP–Compiler erzeugt wird, sollte genauso schnell sein wie von Hand programmierter Assembler-Code[1] .

Es werden also verschiedenste Anforderungen an ALDiSP–Programme gestellt – Anforderungen, die kein einzelner Compiler sinnvoll erfüllen kann. Es ist daher notwendig (und auch ansonsten sinnvoll), mindestens zwei Ausführungsmechanismen für ALDiSP–Programme zur Verfügung zu stellen: einen Interpreter und (mindestens) einen Compiler.

[1] Eine performance degradation von bis zu 50% ist wahrscheinlich meist in Ordnung, aber viel mehr nicht.

3.2 Der Interpreter

Ein ALDiSP–Interpreter hat die Funktion, Programme sofort, semantisch korrekt und mit einem maximalen Grad von interaktiver Einsichtnahme auszuführen, solange sie noch noch in der Entwicklung sind. Die Ausführungszeit eines Interpreters ist dabei ziemlich irrelevant[2]. Ziel ist es, mit dem Interpreter ein fehlerfreies Programm zu entwickeln, das dann von einem hochoptimierenden Compiler übersetzt wird.

3.2.1 Simulation von Echtzeit

DSP–Programme laufen, wenn sie einmal auf die Zielmaschine gebracht sind, in Echtzeit, d.h. müssen harten Zeitvorgaben (*timing constraints*) genügen. Der Interpreter muß in der Lage sein, diese Zeitvorgaben modellieren zu können. Dies kann er tun, indem sämtliche Ein- und Ausgaben mit Zeitangaben (*time tags*) versehen werden. Der Interpreter weiß zwar noch nicht, ob der Compiler in der Lage sein wird, Code der gewünschten Geschwindigkeit zu erzeugen, er kann aber helfen, die Zeiteigenschaften der Programme korrekt zu formulieren, d.h., festzustellen, ob die durch die Suspensions vorgegebenen virtuellen Zeiten die gewünschte Funktionalität erbringen.

3.2.2 Fehlermeldungen und Debugging

Die eigentlich zentrale Eigenschaft von Interpretern, die sie von (normalen oder inkrementellen) Compilern unterscheidet, ist das Erhaltenbleiben der Quellprogramm–Struktur bei der Programmausführung. Wenn ein Fehler auftaucht, kann er in der Terminologie des Quellprogramms beschrieben werden; es muß keine umständliche "Rück-Übersetzung" aus dem Assembler (oder der virtuellen Maschine) in die Quellsprache erfolgen. Insbesondere sind Debugging-Techniken wie die Überwachung von Variablenwerten, das Warten auf bestimmte Programmzustände etc., die normalerweise schwer zu realisieren sind, in einem Interpreter trivial. Die Fehlerbehandlungsmöglichkeiten in Smalltalk-80[17] und Scheme[5] sind hierbei vorbildlich: Tritt ein Fehler auf, stoppt die Auswertung, die gesamte Aufrufgeschichte des fehlgegangenen Ausdrucks ist als Datenobjekt vorhanden, also sichtbar und modifizierbar; Korrekturen vor Ort ermöglichen ein schnelles Austesten.

Ein ALDiSP–Interpreter muß die Eigenschaft haben, alle Fehler (gleich wel-

[2]Nun ja, in der Praxis vielleicht nicht: wenn mit dem Interpreter die Verarbeitung nichttrivialer Datenmengen simuliert werden soll, sollte er schon eine Geschwindigkeit von, sagen wir, 1% der avisierten Endgeschwindigkeit, erreichen. Nächtelange Simulationsläufe gefallen niemandem – sind aber immer noch besser als nächtelange Übersetzungszeiten von Programmen, die dann doch falsch sind und in einer nicht-kooperativen Umgebung abstürzen: Debugging am Logik-Analysator ist auch kein Vergnügen.

cher Art) die in der Semantik eines Programms begründet liegen[3], zu finden und korrekt und mit maximaler Ausführlichkeit zu melden. Dies ist beim Compiler ganz anders.

3.3 Der Compiler

Ein ALDiSP–Compiler wird ein langsames, großes, tabellengetriebenes Programm sein: er muß sich relativ leicht an neue Prozessoren, andere Konfigurationen und verschiedene Grade der Parallelität anpassen lassen. Zugleich muß stets eine (im Vergleich zu von Hand codiertem Assembler) ziemlich optimale Übersetzung gewährleistet sein; diese beiden Forderungen implizieren die Existenz einer extensiven Optimierungsphase, womöglich auch in und nach der Code-Erzeugung, die essentiell heuristisch–kombinatorischen Charakters sein wird.

Man muß damit rechnen, daß ein mittelgroßes Programm einige Stunden zur Übersetzung benötigen wird.

3.3.1 Statisches und Dynamisches Scheduling

Neben der Übersetzungsarbeit am Programm muß der Compiler eine Laufzeitumgebung zur Verfügung stellen, die vor allem eine Funktion haben wird: Interrupt-Steuerung und Scheduling der Prozesse.

Im idealen Fall einer total synchronen Anwendung kann man ohne einen Scheduler auskommen, es gibt einen *statischen Schedule*, d.h. einen unveränderbaren Fahrplan, den das Programm nie verläßt. Anstelle von Interrupts besorgen in einem solchen Programm oft *Warteschleifen* die Zeitkontrolle. Wo immer dies möglich ist, sollte der Compiler die Möglichkeit zu einem statischen Schedule erkennen und dementsprechend Code erzeugen.

Im Normalfall jedoch ist dies nicht möglich: je nach Art der eingehenden Daten (und dem Zeitpunkt ihrer Ankunft) wird sich das Zeitverhalten des Programms ändern. Als Konsequenz hiervon muß es einen *Scheduler* geben, d.h. eine Überwachungsroutine, die (z.B. durch einen Zeitscheibenmechanismus) den Prozessen Laufzeit zuteilt, sie schlafenlegt und wieder aufweckt. Je genereller der Einsatzbereich eines solchen Schedulers, desto größer der mit ihm verbundene Overhead: daher sollte der Compiler eine Palette von Schedulern variabler Komplexität erzeugen können: Scheduler für zwei, drei, k oder beliebig viele Prozesse; für einfache und komplexe Trigger-Expressions (Suspension-Erweckungs-Bedingungen); für eine oder mehrere Interruptquellen, usw.

[3] Vom Interpreter kann nicht erwartet werden, daß er Ressourcenfehler oder Zeitüberschreitungen findet kann.

3.3.2 Fehlerverhalten

Von einem compilierten ALDiSP–Programm kann kein "korrektes" Fehlerverhalten erwartet werden. Das Auftreten eines Fehlers[4] führt zu "undefiniertem" Verhalten[5] , insbesondere zu allen Arten von "Abstürzen". Dies ist völlig in Ordnung, da ja alle Programme auf dem Interpreter getestet und daher korrekt sind. Wären da nicht Ressourcen- und Zeitfehler: Während alle "semantischen" Fehler (Typfehler, nondeterministische Ausdrücke und nichtterminierende Berechnungen) im Interpreter gefunden werden können, fällt dies bei "Laufzeitfehlern" schwer.

Ressourcen- und Zeitfehler

Ein Ressourcen-Fehler tritt auf, wenn eine endliche Ressource (meist Speicher oder File-System-Platz, aber auch Bandbreite) erschöpft ist. Offensichtlich sind diese Fehler von der realen Maschine abhängig und nicht durch Testläufe auf einer 'idealen' Maschine, wie sie ein Interpreter nun einmal darstellt, im Voraus feststellbar.

Zeit ist eine ganz speziele Ressource: man kann Zeit nicht einfach wie Speicher kaufen; DSP-Prozessoren befinden sich meist schon an der Grenze des technisch Machbaren, was Geschwindigkeit angeht. Zeitfehler treten auf, wenn (in ALDiSP– Terminologie) die realen Zeitpunkte, an denen I/O stattfindet, mit den virtuellen Zeitpunkten nicht genügend übereinstimmen – in der Praxis heißt dies, "weil das Programm zu langsam läuft". Zeitfehler können in starkem Maße datenabhängig sein und sind oft komplexer als andere Ressourcenfehler.[6]

3.3.3 Eine pragmatische Hilfestellung

Es gibt eine ganze Reihe von Programmen, die sich als 'statisch' im Speicherbedarf und im Zeitverhalten (oder in der Bandbreite oder worin auch immer) erweisen und so, zumindest theoretisch, zur Übersetzungszeit als frei von Laufzeit-Ressourcen-Fehlern bewiesen werden können. Es muß auch Aufgabe eines Compilers sein, eine möglichst große Menge dieser Programme zu erkennen. Ein Hilfmittel hierfür ist der Einsatz von *Funktionalen*, wie er schon in 1.2.4 beschrieben

[4] Mit 'Fehlern' sind keine numerischen Exceptions gemeint – Man kann davon ausgehen, daß zu Laufzeit nur mit saturierter oder wrap-around-Arithmetik gearbeitet wird, da es zusätzlichen Aufwand erfordern würde, arithmetische Fehler überhaupt zu erkennen. 'Fehler' im Sinne dieses Abschnitts sind z.B. ein 'illegaler Array-Index' und 'Stack Overflow'.

[5] Dies geht mit der Interpretation von $\perp$ als dem am schwächsten definierten Wert einher, der durch jedes Berechnungsverhalten, auch durch das nichtterminierende, korrekt berechnet wird.

[6] Interrupt-Überschneidungen, fehlende kritische Sektionen etc. gehören in die Reihe der semantischen Fehler, da sie zu undeterminiertem Verhalten führen: es gibt für einen Ausdruck mehrere Auswertungsreihenfolgen, die zu verschiedenen Ergebnissen führen. Derartige Fehler sollte man mit Hilfe des Interpreters finden können.

wird. Ein Compiler, der Probleme hätte, allgemeine Rekursionstrukturen aufzudecken, kann eine feste Anzahl von Funktionalen als 'build-in' erkennen und über sie argumentieren. Er kann über ihre Laufzeit- und Terminationseigenschaften informiert sein und (da er auch die maximale Laufzeit aller Primitiven kennt) die maximale Laufzeit ganzer Programme einschätzen. Es ist auch möglich, einfache Rekursionsstrukturen durch Analyse des Programms zu gewinnen; dies ist jedoch überaus aufwendig, schwer zu verallgemeinern und fehlerträchtig.

Kapitel 4

Rationals

4.1 Zur Modellierung von Maschinenobjekten

Bei der Definition der Maschinentypen wurde versucht, ein möglichst breites Anwendungsspektrum abzudecken. Die meisten synchronen und asynchronen I/O-Geräte sollten sich durch einfach `Port`s realisieren lassen; die meisten kontinuierlich arbeitenden Geräte (z.B. D/A-Wandler) sollten durch `Register` modellierbar sein.

Die Idee der Maschinentypen ist *nicht*, Objekte zu konstruieren, mit denen man 'rechnen' kann – Maschinentypen sind keine Datentypen –, sondern dem Implementierer die Möglichkeit zu geben, in der Hardwareumgebung vorgegebene Objekte standardisiert zu modellieren. Voraussetzung dazu ist, das es eine feste Hardware-Konstellation gibt, deren Eigenschaften sich nicht zu schnell ändert (oder nur endlich wenigen Variationen kennt: Ein Beispiel ist ein kombinierter I/O-Port, der zu einem Zeitpunkt nur Input oder Ouput liefern kann und mit einem gezielten Kontrollregister manipuliert wird.) Der Programmierer wird also im Normalfall keinen Bedarf haben, Maschinenobjekte selbst zu deklarieren; alle relevanten Objekte werden von der Umgebung vorgegeben.

Eine Erleichterung für den Implementierer einer spezifischen Umgebung ist die Möglichkeit, Hardware-Bausteine durch Funktionspakete zu modellieren. Als bekanntes Beispiel soll hier der ACIA 6850 beschrieben werden, ein beliebtes Teil für die serielle I/O. Um Maschinenobjekte zu deklarieren, werden hier ein Datentyp (Address) und einige Funktionen (`declareInputPort` etc.) benutzt, die in einer Vorversion als Primitive deklariert waren, jetzt jedoch nicht mehr für den 'normalen' Programmierer zur Verfügung stehen.

```
Module Hardware
Exports ACIA6850   : (Address -> Control6850,
                                 Status6850,
                                 Send6850,
                                 Receive6850),
        Control6850: InputRegister(Byte),
```

```
        Status6850 : OutputRegister(Byte),
        Send6850   : InputRegister(Byte),
        Receive6850: OutputRegister(Byte)

proc ACIA6850(Base : Address) =
  let
        Control = declareInputRegister(Byte,Base+4)
        Transmit= declareInputRegister(Byte,Base+5)
        Status  = declareOutputRegister(Byte,Base+6)
        Receive = declareOutputRegister(Byte,Base+7)
   in
        (Control,Status,Send,Receive)
  endlet

 \ Es k\"onnen noch viele Spezialfunktionen folgen, z.B.

proc Reset6850(C : Control6850) =
  WriteToRegister(C,2#00000011)

EndModule Hardware
```

Ein Problem tritt auf, wenn eine Interruptquelle eingebunden werden muß. Auf vielen Prozessoren sind alle Interrupts distinkt; für n Interruptquellen gibt es dann ein n Elemente großes Array von Sprungadressen. Wenn sich jedoch mehrere Quellen einen Interrupt teilen, muß Code erzeugt werden, der alle möglichen Interruptquellen abfragt. Dies muß von Hand geschehen, d.h. kann dem ALDiSP-Compiler nicht allein überlassen werden (immer mit der Einschränkung 'bei den jetzt definierten Typen' – sicher ist es möglich, diese Abfragen explizit in ALDiSP zu schreiben; aber dies ist wahrscheinlich nicht effizient).

Zusätzliche Maschinentypen ermöglichen es, um "Compiler-Direktive" herumzukommen, z.B. könnte es zur Modellierung asynchroner Interrupts, die mit einer bekannten Häufigkeit auftreten, einen Typ `AverageInterrupt(`*Duration*`)` geben.

4.2 Zur Implementierung von Suspensions

Die Suspensions stellen das am schwierigsten zu implementierende Element von ALDiSP da. Bei näherer Betrachtung vereinfacht sich die Implementierung jedoch, wenn man keine zu allgemeinen Abhängigkeiten aufbaut. Es soll im Folgenden eine Strategie zur effizienzen Implementierung der meisten real vorkommenden Suspensions erläutert werden.

Einfache Datenabhängigkeiten

Die meisten Suspensions modellieren Datenabhängigkeiten. Das Programm

```
proc a(X) =
  suspend F(head(X)) :: a(tail(X))
    until isAvailable(X)
   within 0,0

proc b() =
 suspend readFromPort(A) :: b()
   until isReadable(A)
  within 0,0

net in
 a(b())
endnet
```

ist eine explizit notierte Variante[1] des Programms

```
func a(X) = F(head(X)) :: a(tail(X))

proc b() = readFromPort(A) :: b()

net in
 a(b())
endnet
```

Zu jedem Zeitpunkt der Programmausführung wird es theoretisch zwei Suspensions geben – die eine wartet auf einen Input am Port '`A`', die andere auf ein Element des Streams '`X`'. Für alle praktischen Belange handelt es sich dabei jedoch nur um *eine* Suspension, da eine *Datenkopplung* vorliegt, die die von den beiden Suspensions erzeugten Prozesse *synchronisiert*. In dem Beispiel ist dies trivial zu sehen: man kann das Programm derart umschreiben, daß nur mehr eine Suspension sichtbar ist:

```
proc c() =
 suspend F(readPort(A)) :: c()
   until isReadable(A)
  within 0,0

net in
 c()
endnet
```

[1] Man beachte, das "a" einmal eine Prozedur, das andere Mal eine Funktion ist. Das liegt daran, daß im ersten Fall eine explizite Erwähnung der Suspensions auftritt; im zweiten Fall jedoch nicht. Das erste "a" ist somit spezialisierter als das zweite.

Will man dieses Datenabhängigkeitsprinzip verallgemeinern und abschätzen, wieviele Suspensions derart reduziert werden können, so muß man sich vor Augen halten, daß der für die Analyse relevante Teil die Bedingung der Suspension ist. (Es soll hier von normalen, 'asynchronen' Suspensions ohne Verzögerungsfaktor ausgegangen werden). Diese Bedingungen können nur an *Zustandsänderungen* festgemacht werden, das heißt an I/O-Objekten und dem Zustand anderer Suspensions. Typische DSP-Anwendungen sind Input-getrieben bzw. laufen synchron, so daß der Zeitpunkt einer ganzen Abhängigkeitskette von Prozessen vom Zeitpunkt der Eingabe abhängt. Implementierungstechnisch heißt dies, daß an jeder möglichen Eingabequelle ein Interrupt angebracht wird; die Suspensions stellen den Interrupthandler dar. Im Fall einfacher Datenabhängigkeiten braucht es dabei keinen weiteren Scheduler (solange sich die Interrupts nicht überschneiden); etwas problematischer wird es, wenn komplexe Abhängigkeiten bestehen.

Komplexe Datenabhängigkeiten

Die einfachste komplexe Abhängigkeit ist die Abhängigkeit von zwei Zuständen:

```
proc merge(X,Y)
 suspend
     if isAvailable(X) head(X)::merge(Y,tail(X))
      | isAvailable(Y) head(Y)::merge(X,tail(Y)) endif
 until
        isAvailable(X) or isAvailable(Y)
 within 0,0
```

Natürlich kann eine solche Abhängigkeit noch von vielen logischen Verknüpfungen begleitet sein:

```
proc PrinterDaemon()
\ Aufwachen, wenn ein Problem auftaucht ODER
\ wenn ein Printer nichts zu tun hat und ein job anliegt
\ (Es gibt 3 Printer und 2 Job-Kan\"ale)
  suspend
    ...
  until
       OutOfPaper(Printer1)
    or OutOfPaper(Printer2)
    or OutOfPaper(Printer3)
    or ((   NewJobOn(Channel1)
         or NewJobOn(Channel2))
        and
        (   PrinterAvailable(Printer1)
         or PrinterAvailable(Printer2)
```

```
        or PrinterAvailable(Printer3))
      )
```

Es gibt mehrere Möglichkeiten, diese komplexe Abhängigkeit zu implementieren. Die einfachste ist die Umwandlung des Bedingungsausdrucks in eine Routine, die jedesmal aufgerufen wird, wenn sich einer der Zustände ändert (es wird nur endlich viele davon geben). Diese Möglichkeit steht immer offen, ist jedoch nicht von maximaler Effektivität. Wenn die Bedingung wie im obigen Beispiel durch *or*–Ketten zusammengehalten wird, kann es eleganter sein, aus einer Suspension mehrere zu machen. Im o.a. Code kann man mindestens 4 Prozesse bilden: drei für 'out of Paper', einer für 'Job verteilen'.

Abstruse Bedingungen

Unter dem Begriff 'Abstrus' möchte ich die Bedingungen verstehen, die zu aufwendig sind, um in Echtzeit abgetestet werden zu können[2]. 'Abstrus' heißt nicht 'sinnlos', sondern nur 'wahrscheinlich nicht durch einen Compiler hinreichend effizient übersetzbar'. Das 'Printer'-Beispiel zeigt eine Bedingung am Rande der Unimplementierbarkeit. Ein schöneres Exemplar ist dies hier:

```
func FFT(X: Vector) = ....

proc HochfrequenzWarnung
 suspend <Irgendetwas>
   until
         FFT(ListToMatrix
             (256,readPipeFromRegister(1 sec/44100, DA_Wandler)
             )
            )(255)>0.5
   within 0,100ms
```

Die Intention ist es hierbei, 256 Samples einzulesen, die FFT darauf anzuwenden und festzustellen, ob ein bestimmtes Frequenzband (das höchste in diesem Fall) eine Amplitude von 0.5 überschreitet. Tritt dies ein, wacht die Suspension auf.

Hier wurde das Suspension-Konstrukt 'falsch' eingesetzt, denn:

- Die Bedingung einer Suspension soll sich, zumindest theoretisch, in Nullzeit berechnen lassen – d.h. in 'virtueller Nullzeit'. Das Einlesen von 256 Daten erfordert jedoch die Konstruktion von 256 Suspensions – in diesem Beispiel kann die Bedingung nur alle 1/172 Sekunden berechnet werden! (Dies ist der einzige auch 'formal falsche' Aspekt des Beispiels)
- Der Rechenaufwand ist viel zu hoch – eine FFT ist eine größere Operation, die leicht in die Millisekunden gehen kann!

[2]Grober Anhaltspunkt: Eine Bedingung sollte, bei intelligenter Implementierung, in weniger als, sagen wir, 20 Instruktionen abtestbar sein.

- Die Bedingung läßt sich nicht durch daß Interrupthandler-Modell implementieren.
- Der Compiler wird kaum in der Lage zu sein, die 100 Millisekunden Reaktionszeit garantieren zu können. (Auch wenn das Programm dies nachher mit Leichtigkeit bewältigt – es sollte doch statisch beweisbar sein!)

Eine korrektere Implementierung des Problems könnte so aussehen:

```
func FFT(X: Vector) = ....

proc HochfrequenzWarnung()
 if
   FFT(ListToMatrix(256,
        readPipeFromRegister(1 sec/44100, DA_Wandler))
      )(255)>0.5
 then <Irgendetwas>
 else HochfrequenzWarnung()
```

Wird diese Prozedur gestartet, so liest sie 256 Daten ein, berechnet die FFT, und testet die Frequenz. Wie man sieht, ist eine explizit Suspension oft gar nicht nötig! Implizit, d.h. durch das `readPipe...`, wird dafür gesorgt, daß die Prozedur nur alle 1/172 Sekunden aufwacht (wenn alle Daten gesammelt sind) und rechnet. Es wird kein Versprechen hinsichtlich der Ausführungszeit gemacht, also hat der Compiler auch keine Probleme mit der Verifikation. (Natürlich ist auch nicht gesichert, daß überprüft wird, ob das Programm "hinterherkommt".)

4.3 Diverses

Integers sind Fixpunkttypen

Integers können als der Spezialfall von Fixpunktzahlen, der keine Nachkommastellen hat, angesehen werden. Dies wird in der ALDiSP–Beschreibung nicht betont, da

- es etliche Funktionen gibt, die nur auf Integers und Cardinals definiert sind,
- Integers implementierungstechnisch anders behandelt werden und
- Integers und Cardinals stark nicht-numerischen Einsatz finden (z.B. in der Array-Indizierung).

`Integer` ist somit neben `Bool` ein de facto im Sprachkern eingebundener Datentyp, während die Fixpunkt-Typen z.B. auf einem Floating-Point-DSP unter Umständen gar nicht implementiert sind.

Zu den Regeln zur Typisierung numerischer Resultate

Diese Regeln haben reinen Vorschlagscharakter. Im Prinzip können die Resultate numerischer Funktionen beliebig aussehen, solange garantiert ist, daß es nicht zu einem Überlauf kommen kann. Die Idee hinter den Typisierungsregeln ist die, daß es für einen möglichst großen Teil der numerischen Funktionsaufrufe *statisch* feststellbar ist, daß es zu keinem Überlauf kommt. Ist dies garantierbar, folgt daraus, daß es zu keinem Exception-Aufruf kommt, d.h. daß Funktionen ohne den für Exceptions nötigen Overhead übersetzt werden können. Es ist jedem Compiler erlaubt, beliebig detaillierte Aussagen über einen Resultattypen zu treffen, d.h. Wertmengenabschätzungen, die weit über das, was in den vorgeschlagenen Typregeln erwähnt wird, hinausgehen. Ebenso ist es jeder Implementierung erlaubt, nur einen Integer-Typen zu unterstützen und bei jedem Funktionsaufruf den Exception-Overhead zu ertragen. Zum Beispiel für einen Interpreter geht dies völlig in Ordnung.

Funktionen variabler Stelligkeit

In ALDiSP kann man keine Funktion variabler Stelligkeit notieren: so ist es z.B. nicht möglich, eine allgemeine Kompositionsfunktion zu notieren. Es ist ziemlich klar, wie eine Erweiterung in diese Richtung aussähe: Man läßt einfach die Klammern bei der Applikation weg; eine Variable (bzw. eine Expression) steht dann für die ganze formale (bzw. aktuelle) Parameterliste: `func comp(f,g) x = f g x` Das syntaktische Problem dabei ist die Unterscheidung zwischen dem 1-Tupel und seinem Inhalt. `f(g x)` impliziert, das `g` genau einen Wert als Resultat liefert, kann aber auch als einfache Klammerung gesehen werden! Um diesen Problemen zu entkommen (und weil die Struktur eines Programms nicht gerade lesbarer wird, wenn mit variabelstelligen Funktionen gearbeitet wird), ist diese Erweiterung *nicht* übernommen worden.

"Rein funktionale" I/O

Jegliche I/O wird in ALDiSP als Nebeneffekt aufgefaßt. Hier soll kurz erläutert werden, wie sich eine derartige I/O funktional "sauber" interpretieren läßt. Die Idee dahinter ist, die "Umgebung" des Programms (den "Weltzustand") und alle Änderungen dieser Umgebung durch zustandsändernde Prozeduraufrufe zu formalisieren. Im einfachsten Falle geschieht dies, indem jede derartige Prozedur $\mathcal{F}$ mit der Funktionalität $x \rightarrow y$ zu einer Funktion $\mathcal{F}'$ mit der Funktionalität $x, s \rightarrow y, s'$ umgewandelt wird, wobei s den alten, s' den veränderten Zustand modelliert. Es ist möglich (recht einfach, aber sehr schreibaufwendig), jedes ALDiSP–Programm in dieser Weise umzuwandeln (Jede Prozedur, die Nebeneffekte erzeugt, muß in dieser Weise erweitert werden). Diese Transformation ist nicht "fein" genug für viele Bedarfsfälle, da sie zu "vorsichtig" ist. Ein Beispiel:

```
WriteToPort(x,readFromPort(y)+readFromPort(z))
```

wird entweder zu

```
seq
  tmp1,state1 = readFromPort(x,state0);
  tmp2,state2 = readFromPort(y,state1);
  state3=WriteToPort(tmp1+tmp2,state2);
  state3
endseq
```

oder zu

```
seq
  tmp1,state1 = readFromPort(y,state0);
  tmp2,state2 = readFromPort(x,state1);
  state3=WriteToPort(tmp1+tmp2,state2);
  state3
endseq
```

gewandelt – es wird eine Sequentialisierung vorgenommen. (Nebenbei wird eine Fehlermeldung erzeugt, weil der Ausgangs-Ausdruck nicht voll deterministisch war.) Wenn Port `y` und Port `x` jedoch, was wahrscheinlich ist, voneinander unabhängig sind, d.h. nicht zwei Namen für dasselbe Objekt sind oder irgendwie eine gegenseitige Beeinflussung ausüben, und das "`+`" kommutativ ist, ist die Reihenfolge irrelevant; die Sequentialisierung (und die Fehlermeldung) ist unnötig. Unnötige Sequentialisierungen sind nicht nur "aus Prinzip" zu vermeiden (wie alle anderen unnötigen Operationen), sondern, weil sie die Möglichkeiten der Optimierungs- und Parallelisierungsphase des Compilers einschränken.

Ein Nachteil der Zustandsparametrisierung liegt darin, daß sie realistischerweise den aktuellen Zustand nur *modelliert*, nicht aber *implementiert*. Es gibt daher Einschränkungen beim Gebrauch der Zustandswerte: man darf sie nicht "zweimal benutzen": in einem Programm darf es zu jedem Zeitpunkt nur *einen* Zustand geben; würden Zustände in irgendeiner Art gespeichert, würde dies die Implementierung eines ewigen Gedächtnisses in der Art der "Demonic Memories" [15] implizieren – eine recht aufwendige Angelegenheit.

19-Bit Integers

Die Regeln zur Festlegung der Ergebnistypen arithmetischer Funktionen werfen die Frage nach der Implementierbarkeit 'krummer' Typen auf. Der Ausdruck

```
([3 * [Integer16] 10 + [Cardinal8] 1) + 2
```

wertet sich folgendermaßen aus: Zuerst muß der Typ von `3` bestimmt werden. Konstanten sind vom Typ `Integer`. In einer typischen Implementierung könnte Integer die Breite 16 haben. Es ist eine erlaubte Optimierung, einen 'feineren'

Typen zu suchen, für `3` wäre dies zum Beispiel `Cardinal(2)`. Je nachdem ergibt die Multiplikation `[Integer31]30` oder `[Integer18]30`. Nach der ersten Addition gäbe es dann `[Integer32]31` oder `[Integer19]31`, nach der zweiten `[Integer33]33` oder `[Integer20]33`.

Die Implementierung von 18-, 19- und 20-Bit-Integern ist *kein* Problem, da durch die Resultattypisierungsregeln garantiert wird, daß kein Unter- oder Überlauf auftreten kann. Wenn es aber keine derartigen Fehler zu beachten gibt, kann man genausogut die ganze Rechnung z.B. mit 24-Bit- oder 32-Bit-Integern implementieren.

Genau das ist die Motivation hinter den 'krummen' Resultat-Typen: jede Maschine wird nur wenige Wortbreiten effizient verarbeiten können; aber eine n-Bit Wortbreite implementiert alle '<n'-Wortbreiten gleich mit. Die Typisierungsregeln ermöglichen es, zur Übersetzungszeit festzustellen, welche Berechnungszweige keinen Überlauf erzeugen können – implementierungstechnisch haben sie sonst keine Bedeutung, d.h. auf der realen Maschine werden sie (außer vielleicht in I/O-Objekten) nicht mehr zu finden sein.

Keywords

Warum sind Keywords case-insensitive? Schlüsselwörter können groß oder kleingeschrieben werden, weil viele Programmiersprachen sich nicht dahingehend einigen können und viele Programmierer groß- oder kleingeschriebene Keywords erwarten. Ich selbst tendiere zur Kleinschreibung, aber bei Schlüsselworten ist die Notwendigkeit der herausragenden Syntax gegeben, die als Motiv hinter der Großschreibung steht. Also wird beides zugelassen. Zugleich aber sollen bei 'normalen' Namen Groß- und Kleinschreibung verschieden sein, damit gilt, daß keine zwei unterschiedlich geschriebenen Namen dasselbe Ding bezeichnen – gleiches Objekt, gleiche Schreibung.

Warum heißt "=<" nicht "<=" ? "=>" und "<=" sind als Namen für Vergleichsoperationen nicht benutzt worden, um diese Zeichenkombinationen für "Pfeil"-Operationen freizuhalten. Wo immer dies möglich ist, sollten die Symbole "≤" und "≥" benutzt werden, so daß die ungewöhnliche Erscheinung von "=<" nicht weiter stört.

Ursprünglich... sollten `let` und `endlet` sowie `seq` und `endseq` einfache geschweifte Klammern sein; aber dies sah dem Betreuer zu sehr nach 'C' und zu wenig nach Blockstruktur aus. Die Syntax wird aber nicht uneindeutiger, wenn diese Änderung rückgängig gemacht wird; daher empfehle ich, die geschweiften Klammern als Alternativkonstrukt beizubehalten.

Uneindeutigkeiten in der Grammatik Es gibt mindestens eine bekannte Stelle in der Grammatik von ALDiSP, die uneindeutig ist. Dies ist die Verwen-

dung des Gleichheitszeichens sowohl in `let` und den sequentiellen Definitionen als auch als Vergleichsoperator. Ein Ausdruck der Form

```
a = b(c) = d(e)(f)=e
```

hat folgende Interpretationsmöglichkeiten:

```
a = b ; (c) = d(e) ; (f) = e
a = b(c) = d(e) ; (f)=e
a = b ; (c) = ( d(e)(f) = e )
a = ( b(c) = ( d(e)(f) = e ))
```

Um dies zu verbieten, wurde die Klammerung von Namen (auch Namenstupeln) auf der linken Seite von Definitionen verboten. Eine einfachere Lösung wäre es, ein anderes als das Gleichheitszeichen für Definitionen zu wählen; aber $\leftarrow$ ist im ASCII-Zeichensatz nicht vorhanden und `:=`, `<=` und `<-` sehen zu sehr wie von-Neumann-Assignments aus und sind zudem zu lang.

Pipes vs. Streams

Manchmal scheint es so, als wären Pipes und Streams eigentlich dasselbe und die Unterscheidung rein semantisch, während die Syntax für beide die gleiche sein könnte, so daß es nur ein Konstrukt gäbe. Dies ist falsch. Das Problem taucht auf, weil man meist 'synchrone' Anwendungen von Pipes untersucht, bei denen dies in der Tat der Fall ist, die 'asynchronen' Fälle jedoch außer Acht läßt. (Ein Ausdruck der Form `foo(`$pipe_1$`,` $pipe_2$`)` soll *synchron* genannt werden, wenn er, um ausgewertet zu werden, je ein Datum[3] aus $pipe_1$ und $pipe_2$ benötigt. *Asynchron* soll der Ausdruck heißen, wenn ein Datum in einer der beiden Pipes ausreicht, d.h. ein (zeitlich deterministischer) Non-Determinismus vorliegt (man kanns auch als Orakel sehen). Alle einfachen Funktionen des allgemeinen Typs `(pipe -> pipe)` sind zwangsläufig synchron.)

Warum gibt es soviele Integer-Typen und sowenig Reals?

Grundidee ist: es sollte eine breite Palette von `Integer`s geben, da

- Arithmetik mit ungewohnten Integer-Breiten recht einfach zu implementieren ist
- Ungewohnte Breiten im I/O-Bereich häufig auftauchen: sowohl in Hardware-Ports als auch in Dateiformaten sind z.B. 7– oder 14–Bit-Daten nichts Ungewöhnliches.

[3] Eigentlich: je n Daten, aber im Rahmen dieser Diskussion ist der Spezialfall mit n=1 O.B.d.A. ausreichend.

Im Gegensatz hierzu ist das Rechnen mit Floating-Point eine aufwendig Sache, die von fixen Libraries oder mit spezieller Hardware ausgeführt wird. In beiden Fällen sind feste Datenformate (vor allem die IEEE-754 Formate[18]) zu erwarten. Es wäre ziemlich aufwendig, allgemeine Floating-Point-Routinen mit n Bits Mantisse und m Bits Exponenten zu implementieren, und niemand würde sie benötigen. Also gibt es drei vorgegebene `Real`–Typen, für die sich 32, 64 und 80 Bit geradezu anbieten, aber nicht festgeschrieben sind – wer weiß, vielleicht bringt Motorola übermorgen einen 96-Bit DSP-Chip heraus, mit 48-Bit 'Shorts' und 144-Bit 'Longs'.

Fehler und bottom

`bottom` hat zwei Semantiken: im Rahmen einer typischen Interpreterausführung wird erwartet, daß alle Fehlersituationen, auch nur dynamisch auffindbare, zu einer Fehlermeldung und `bottom` als Resultat führen. Tritt zur Laufzeit eines compilierten Programmes eine Fehlersituation auf, bedeutet `bottom` ein "undefiniertes Verhalten", d.h. einen möglichen Absturz, fehlerhafte Resultate, auf jeden Fall aber keine Garantie einer Fehlermeldung gleich welcher Art. Diese zweite Interpretation von `bottom` ist rein effizienzbedingt.

`And` und `Or`

sind in ALDiSP, im Gegensatz zu *Scheme* oder *Modula-2*, echte Funktionen. Die 'von-links-nach-rechts'-Auswertung wurde fallengelassen, weil sie

- Fehleranfällig ist – sie wird zu leicht übersehen.
- nicht funktional ist – es gibt keinen *logischen* Grund, `and` und `or` getrennt zu behandeln.
- nicht so leicht parallelisierbar ist – man weiß nicht, ob es auf die Sequenz ankommt oder nicht.
- nicht notwendig ist – wenn es darauf ankommt, kann man die Sequentialität auch explizit (durch `Ifs`) notieren.

4.4 Features, die ausgelassen wurden

Im folgenden werden einige "Auslassungen", also Konstrukte und Typen, die man vielleicht in ALDiSP erwarten würde, erläutert.

4.4.1 Assignment

Eigentlich sollte es zum Assignment ein eigenes Kapitel geben – so problematisch ist alles, was damit zu tun hat.

Dinge, die ohne Assignment nicht möglich sind

Das wichtigste Konstrukt, das es ohne Assignment nicht gibt, ist die Prozedur mit Zustand, die einen mutablen Datentypen implementiert. (Gibt es von vornherein mutable Datentypen, so kann man das Assignment damit implementieren.) Ohne mutable Datentypen sind diverse Algorithmen und Datenstrukturen schwer formulierbar. So ist ALDiSP z.B. kaum in der Lage, das objekt-orientierte Paradigma zu unterstützen.

4.4.2 Continuations

Von der Einbeziehung von Continuations wurde abgesehen, da

- durch expliziten Continuation-Passing-Style so ziemlich alles gemacht werden kann, was durch vorgegebene Continuations machbar ist,
- Continuations schwer effizient compilierbar sind und
- mit Continuations Obfuszierungen möglich sind, die nicht einmal mit dem GOTO machbar sind. (Immerhin war die Continuation ursprünglich als funktionale Verallgemeinerung des GOTO, d.h. als allgemeinste mögliche Kontrollstruktur überhaupt, entwickelt worden.)

Ich bin der Meinung, daß es kaum gute Gründe für den Benutzer einer Programmiersprache gibt, neue Kontrollstrukturen von solcher Mächtigkeit einzuführen, daß sie nicht mit dem Mittel der allgemeinen Funktionsabstraktion beschreibbar sind. Continuations werden gerne benutzt, um Dinge wie Coroutinen und Streams zu beschreiben - Konzepte, die eine Sprache selbst vorgeben sollte.[4]

4.5 Ungewohnte Sprachkonzepte

Im folgenden werden einige nicht-standard Elemente von ALDiSP erläutert.

4.5.1 Makros

Das (eingeschränkte) Makro-Feature wurde eingeführt, um Spezial-Syntax zu sparen. In ALDiSP werden Makros als Funktionen aufgefaßt, deren Aufruf call-by-name statt call-by-value erfolgt und die keine Rekursion erlauben.

Die Makros bringen also keinen syntaktische Aufwand mit sich, da die Funktionssyntax unverändert übernommen wird. Der Aufwand im Compiler ist minimal (ungefährliche textuelle Substitution, voll zur Übersetzungszeit möglich). Der pragmatische Vorteil eines solches Konstrukts wiegt dagegen schwer: Von

[4]Zum Thema 'Kontrollstrukturen' empfehle ich dem geneigten Leser: "A linguistic contribution to *GOTO*-less programmin" by R. Lawrence Clark, CACM 27,4, April 1984

häufigen Spezialfällen, die nicht-Funktionen enthalten (z.B. `delay`- und `guard`-Ausdrücke), kann beliebig abstrahiert werden.[5]

4.5.2 Das allgemeine Typkonzept

Die Abkehr vom tradierten termbasierten Typkonzept (Aufbau von Termen durch Konstruktoren, Zerlegung durch Pattern Matching), das in den meisten funktionalen Sprachen (exemplarisch in ML) Anwendung findet, hat mehrere Gründe:

- Prädikattypen sind allgemeiner als Termtypen (allgemeiner: *Partition Types* – Typen, die die Menge der Objekte in disjunkte Teilmengen aufteilen) und damit "theoretisch erstrebenswerter".
- Prädikattypen lassen es zu, exaktere Aussagen über Funktionsparameter und Expressions zu machen, als dies bisher möglich ist. Ein Prädikattypisierung kann als *Assertion* bei der Verifikation eingesetzt werden. Beispiel: Statt

```
func f(A: Array(m,n) i: Nat, j: Nat) =
 \ Pre: 0<=i< m, 0<=j< m, 0<=i< n, 0<=j< n
 A[i,j]+A[j,i]
```

kann man direkt

```
type CardRange(n)(x) =
 if isCardinal x then x<n
                 else false
 endif

func f(A: Array(m,n), i:     CardRange(m)
                        and CardRange(n),
                     j:     CardRange(m)
                        and CardRange(n))
 A[i,j]+A[j,i]
```

schreiben. Eine derartige Typisierung hat mindestens 2 Funktionen:

  - Sie wird während der Entwicklungsphase als Laufzeittest durchgeführt und kann somit zum Test und zur Verifikation eingesetzt werden.
  - Der Compiler kann sie als Optimierungshilfe einsetzen.

[5] Man kann die Argumentation führen, daß Makros eine Konsequenz der Orthogonalität sind, da es ohne sie nicht möglich ist, von beliebigen Ausdrücken zu abstrahieren.

- Prädikattypen lassen sich einfach auf dynamisch getypte Sprachen (wie Scheme und APL) aufsetzen. Typ-Deklarationen haben eine ausführbare Semantik (Test auf Wertebereiche von Parametern.)
- Wie das `abstype`–Konstrukt zeigt, bereitet es keine Probleme, 'echte' termbasierte Typen im Kontext einem prädikatbasierten Typsystems zu definieren.

Eine ausführliche Auslassung zu diesem Thema findet sich in [16].

4.5.3 Parser-Deklarationen

Ursprünglich sollten ALDiSP–Ausdrücke ungefähr die Form von *M-Expressions* annehmen[6], also in ziemlich reiner Präfix-Form notiert werden... als einige Beispielprogramme damit geschrieben wurden, war klar, das diese Form zwar völlig ausreicht, aber unübersichtlich ist. Es mußten also Infix-Operatoren her. Normalerweise sind alle Infix-Operatoren links-assoziativ. `cons` aber ist rechts-assoziativ – und es gehört sich einfach nicht, eine funktionale Sprache zu schreiben, in der

`a.b.c.()` $\Longrightarrow$ `(((a.b).c))`

ist. Also gibt es beide Assoziativitäten. Dann kamen die Zeitangaben: man soll in der Lage sein, "`3 ms`" schreiben zu können (statt `ms(3) oder ms 3`). Also gibt es Postfix (außerdem macht es Spaß, das immerwährende Beispiel mal anders zu notieren:

```
func n! =
  if n>0 then 1
         else n*(n-1)!
  endif

).
```

[6] Das *LISP 1.5 Programmer's Manual* [19] kennt sie noch, die "LISP Meta-language". Dort notiert man

```
gcd[x;y]=[x>y→gcd[y;x];
          rem[y;x]=0→x;
          T→gcd[rem[y;x];x]
```

statt (in *Scheme*–Notation)

```
(define (gcd x y)
  (cond ((> x y) (gcd y x))
        ((= (rem y x) 0) x)
        (else (gcd (rem y x) x))
  )
)
```

Das Beispiel ist [19], Seite 7, entnommen.

Und wenn man schon In- und Postfix hat, ist es sinnlos, am Präfix zu sparen, also kann man `sin 42` statt `sin(42)` schreiben. (Natürlich ist letzteres aufgrund der allgemeinen Klammer-Regel weiterhin erlaubt.)

4.5.4 Suspension

Das Suspension-Konstrukt ist das neuartigste Element in ALDiSP. Es stellt eine Fortentwicklung von in diversen LISP-Dialekten eingeführten Konstrukten dar, namentlich dem `delay`, das aus Scheme bekannt ist und direkt übernommen wurde, und dem dem `future`–Konstrukt, das z.B. in Multilisp[10] und MulT[12] benutzt wird, um Prozesse zu erzeugen:

> "The expression `future` X), where X is an arbitrary expression, creates a task to evaluate X and also creates an object known as a *future* to eventually hold the value of X. When created, the future is in an *unresolved*, or *undetermined*, state. When the value of X becomes known, the future *resolves* to that value, effectively mutating into the value of X and losing its identity as a future. Concurrency arises because the expression `future` X) can proceed concurrently with the evaluation of X. [....] *Strict* operations such as addition or comparison, if applied to an unresolved future, are suspended until the future resolves and the proceed, using the value to which the future resolved as though that had been the original operand."[12]

Die Idee hinter der Suspension ist es nun einfach, für die Berechnung der Future einen Zeitrahmen anzugeben.

4.6 Vergleich mit anderen Sprachen

4.6.1 Scheme

ALDiSP war zunächst als Scheme[2]-Erweiterung mit verständlicherer Syntax geplant. Einige Elemente, z.B. das `delay`–Konstrukt, sind auch geblieben. Nach langer Diskussion wurde das imperative `set!` entfernt. Der Typisierungsansatz entspricht, wenn man Typisisierungen in operationalen Code umwandelt, ziemlich dem von Scheme, der Einsatz von `bottom` jedoch nicht. Scheme bietet keine Möglichkeiten, ein automatischem Mapping für nicht-primitive Funktionen einzubauen.

4.6.2 ML, Hope, Miranda etc.

ML, Hope und Miranda stehen hier exemplarisch für den 'Mainstream' moderner funktionaler Sprachen.[7] Das System der abstrakten Datentypen und der

[7] Wer kennt die Namen? Man könnte auch FX-87, FX-89, Haskell, Russell, Poly, Scratchpad II, verschiedenste Lisp-Dialekte, funktional-relationale Sprachen wie POP-11 und Sprachen

Exceptions ist ML[3] entlehnt, aber einfacher, da ALDiSP keine Rücksicht auf eine statische Typdisziplin nehmen muß und alles erlauben kann, was irgendwie Sinn macht. Immerhin ist die Syntax der von ML doch nicht unähnlich.

Anders als z.B. Miranda[4] wird ALDiSP nicht lazy, sondern eager ausgewertet, was vor allem aus Effizienzgründen und wegen der Nebeneffekte geschieht, bei denen die Reihenfolge der Auswertung ja relevant ist, also nicht dem Interpreter überlassen werden darf.

4.6.3 APL

APL hat vor allem die Idee mächtiger Array-Typen und des automatischen Mappings geliefert. ALDiSP kennt keine hochdimensionalen Arrays (APL sieht mindestens 64 Dimensionen vor!).

4.6.4 Modula-2

Modula-2 findet hier nur des Modul-Konzepts wegen Erwähnung. ALDiSP kennt keine 'definition modules', was auch nicht nötig ist, da jene in Modula-2 vor allem für Speicherplatzberechnungen bei der separaten Compilierung und für Typchecks notwendig sind – beides ist so in ALDiSP nicht implementierbar bzw. notwendig.

4.6.5 SILAGE

SILAGE[21] ist eine Spezialsprache zur Spezifikation von DSP-Algorithmen. SILAGE hat die Form einer Sammlung von *single-assignment*-Funktionen, die von einer *main*-Funktion aufgerufen werden. Der elementare Datentyp in SILAGE ist das *Signal*. Ein Signal entspricht einem `List(Number)` (eigentlich: einer Pipe). Signale werden implizit synchron eingelesen (mit einer nicht spezifizerten Abtastrate) und ausgegeben. Man kann mit Signalen verschiedener Abstastraten arbeiten.

Jede SILAGE-Funktion verknüpft die aktuellen Samples der Eingabe-Signale der Funktion zu den aktuellen Samples der Ausgangssignale. Praktisch liegt ein implizites Mapping vor, bei dem man auch auf zeitlich zurückliegende Samples zugreifen kann (der '`@`'-Operator. Das folgende Beispiel zeigt, wie ein SILAGE-Programm in eine ALDiSP–Funktion umgewandelt werden kann:

```
#define word fix<8>
#define coefType int<8>

my_coefs = coefType({5,7,8,9,12,16,27,81,-81,
                     -27,-16,-12,-9,-8,-7,-5})
```

mit funktionalen Anteilen oder abstrakten Datentypen wie CLU oder Mesa nennen.

```
func h_transform(in:word; coefs: coefType[]):word =
begin
  return innerprod(coefs,{i:16::in@(31-2*i)});
end;

func main(in:word):word =
begin
  return=h_transform(in,my_coefs);
end;
```

Das Beispiel entstammt [21], p.1-21, und beschreibt einen FIR-Filter. Die Funktion h_transform multipliziert die letzten 16 an 'ungeraden' Indizies stehenden Abstastwerte mit den Koeffizienten und addiert die Ergebnisse. In ALDiSP könnte man dies, etwas eleganter, so programmieren:

```
myCoefs = [5,7,8,9,12,16,27,81,-81,-27,-16,-12,-9,-8,-7,-5]

func hTransform(in: listOf(Number), coef:VectorOf(Number)) =
 reduce('+^',odds(s)*vectorToList(coef))
   :: hTransform(tail(list),coef)

net
  in = readStdInPipe()
  out= hTransform(in,myCoefs)
in
  writeStdOutPipe(out)
endnet
```

Die `StdIn`-Pipe kommt ins Spiel, weil ALDiSP das Abtasten selbst vornimmt.[8] ALDiSP kennt keine derartigen Standard-I/O-Kanäle.

Man beachte, daß `hTransform` keine explizite Referenz auf die Länge des Filters und die Typen der Streams und der Koeffizienten macht, also 'modularer' als `h_transform` formulierbt ist.

Im direkten Vergleich zeigt sich, daß ALDiSP vielseitiger und mächtiger als SILAGE ist (so konzeptualisiert SILAGE weder I/O noch Zeit und kann keine 'asynchronen' Probleme bewältigen), während SILAGE auf dem Gebiet, für das es entworfen ist, oft kürzere (wenn auch nicht unbedingt verständlichere) Programme ermöglicht.

4.6.6 Occam

ALDiSP hat den Interprozeßkommunikationsmechanismus "Channel" aus der Sprache Occam [occam2] übernommen. Occam verfügt über eine Reihe interresanter Konzepte und Strukturen, die alle in ALDiSP simulierbar sind. Das

[8] Die `read`- und `write`-Funktionen sind keine Primitiven, sondern müssen selbst definiert werden.

grundlegende Konzept von Occam ist, das alles ein Prozeß ist. Die Termination von Prozessen ist sehr wichtig. Durch die `SEQ`- und `PAR`-Konstrukte sind Prozesse sequentiell und parallel ausführbar, wobei eine automatische Synchronisation stattfindet, d.h. erst wenn alle Prozesse eines parallelen Konstrukts terminiert sind, terminiert das parallele Konstrukt.

Die Channel sind das zentrale Modularisierungsmittel von Occam. Sie entsprechen in der Nutzung den Streams und Pipes von ALDiSP.

Das `ALT`-Konstrukt von Occam ermöglicht in eingeschränkter Weise das asynchrone Warten auf eine von mehreren Eingabesituationen. In ALDiSP wird dies durch explizite Suspensions, die von komplexen Bedingungen abhängen, modelliert.

Occam verfügt über call-by-name (optional call-by-value) Prozeduren und Funktionen, die etwas ungewohnt ist. Auch in Occam gilt die Trennung zwischen nebeneffektfreien Funktionen und nebeneffektbehafteten Prozeduren.

4.7 Implementierungs-Restriktionen

Einige Bereiche von ALDiSP sind allgemeiner definiert, als nach heutigem Stand der Technik effizient implementierbar ist. Da ALDiSP–Programme nicht nur korrekt, sondern auch schnell laufen sollen, steht es jedem ALDiSP–Compiler frei, Restriktionen in den nachfolgenden Gebieten aufzustellen:

- Nicht alle (nicht einmal alle essentiellen) numerischen Typen müssen implementiert sein. Insbesondere darf eine Implementierung ohne `Reals` auskommen. `Reals` sind nur deswegen als 'essentiell' markiert, weil Suspensions (genauer gesagt, Zeit-Ausdrücke) mit reellen Zahlen rechnen.

- Es darf starke Restriktionen bei der Bildung von Typtermen geben. Insbesondere können Prädikattypen verboten sein!

- Abstrakte Datentypen können verboten sein.

- Der 'Bedingungs'-Teil von Suspensions kann eingeschränkt werden. Insbesondere können die einzigen erlaubten Abhängigkeiten die booleschen Konstanten, die Abhängigkeit von anderen Suspensions und die Abhängigkeit von asynchroner I/O sein.

Ein zweites Problem sind Speicherplatzbeschränkungen. Es ist prinzipiell unmöglich, vorausszusagen, wieviel (Stack-) Platz eine rekursive Funktion benötigen wird. Es ist daher eine zumutbare Einschränkung, wenn eine Implementierung keine Rekursion erlaubt! Als Ausgleich müssen allerdings hinreichend mächtige Funktionale zur Verfügung gestellt werden.

Anhang A

Einfache Syntax von ALDiSP

Diese Syntax ist vereinfacht und beschreibt ein Superset der eigentlichen Grammatik. Es fehlen die Beschreibungen der Vorrangregeln und der Assoziativität. Weiterhin fehlen die Parser-Anweisungen. Die Grammatik ist in einer Art EBNF beschrieben, die [9] entnommen ist. $\{_{1,}\ x\ \}$ bedeutet "das ein- oder mehrmalige Auftreten von x, bei mehrmaligem Auftreten durch "," getrennt."

program ⟼ $\{_0$ declaration $\}$ net-description

net-description ⟼ NET $\{_1$ declaration $\}$ [IN tupel] [ENDNET]

declaration ⟼ [declaration-qualifier] simple-declaration
| multiple-declaration
| declaration-qualifier abstraction-declaration
| MACRO abstraction-declaration
| module
| imports
| abstype
| parser-declaration

declaration-qualifier ⟼ [OVERLOADED] HEAD

simple-declaration ⟼ ID "=" expr

multiple-declaration ⟼ "(" $\{_{2,}$ ID $\}$ ")" "=" tupel

abstraction-declaration ⟼ ID $\{_{1,}$ "(" $\{_{0,}$ ID [typing] $\}$ ")" $\}$ [typing] "=" tupel
| ID BINOP ID [typing] "=" tupel
| PREOP ID [typing] "=" tupel
| ID POSTOP [typing] "=" tupel

module ⟼ MODULE ID
$\{_1$ exports $\}$

$\{_1$ declaration $\}$
ENDMODULE ID

exports $\longmapsto$ EXPORTS $\{_{1,}$ ID [rename] [typing] $\}$

typing $\longmapsto$ ":" type

rename $\longmapsto$ AS ID

imports $\longmapsto$ [OVERLOADED] IMPORT $\{_{1,}$ [OVERLOADED] ID
[rename] [typing] $\}$
FROM ID

expr $\longmapsto$ detexpr
| suspend
| "(" expr ")"

detexpr $\longmapsto$ CONST
| ID
| conditional
| application
| typdecl
| typcast
| delay
| seq
| local
| guard

suspension $\longmapsto$ SUSPEND tupel
UNTIL detexpr
WITHIN detexpr "," detexpr

tupel $\longmapsto$ expr
| "(" $\{_{2,}$ expr $\}$ ")"

conditional $\longmapsto$ IF expr THEN tupel
$\{_0$ BAR expr THEN tupel $\}$
[ELSE tupel] ENDIF

delay $\longmapsto$ DELAY expr

seq $\longmapsto$ SEQ $\{_0$ stmt ";" $\}$ tupel ENDSEQ

stmt $\longmapsto$ expr

typdecl $\longmapsto$ "[" type "]"

typcast $\longmapsto$ "[" "!" type "]"

type	$\longmapsto$	detexpr | "(" {$_{0,}$ type } TO {$_{0,}$ type } ")"
application	$\longmapsto$	expr BINOP expr | PREOP expr | expr POSTOP | expr "(" {$_{0,}$ expr } ")" | "[" {$_{1,}$ expr } "]"
local	$\longmapsto$	LET {$_{1}$ decl } IN tupel ENDLET | LET tupel WHERE {$_{1}$ decl } ENDLET
abstype	$\longmapsto$	ABSYTPE {$_{1\text{BAR}}$ constructor }
constructor	$\longmapsto$	ID | ID "(" {$_{1,}$ ID [typing] } ")"
parser-declaration	$\longmapsto$	PARSERFIX " ID " [":" CONST]
guard	$\longmapsto$	GUARD expr {$_{1}$ ON declaration } ENDGUARD

Die Terminale entsprechen den String-Mengen
IF $\equiv$ "if" (case insensitive)
THEN $\equiv$ "then" (case insensitive)
BAR $\equiv$ "|"
TO $\equiv$ "->"
ELSE $\equiv$ "else" (case insensitive)
ENDIF $\equiv$ "endif" (case insensitive)
GUARD $\equiv$ "guard" (case insensitive)
ENDGUARD $\equiv$ "endguard" (case insensitive)
ABSTYPE $\equiv$ "abstype" (case insensitive)
EXCEPTION $\equiv$ "exception" (case insensitive)
SUSPEND $\equiv$ "suspend" (case insensitive)
UNTIL $\equiv$ "until" (case insensitive)
WITHIN $\equiv$ "within" (case insensitive)
MODULE $\equiv$ "module" (case insensitive)
IMPORT $\equiv$ "import" (case insensitive)
EXPORT $\equiv$ "export" (case insensitive)
DELAY $\equiv$ "delay" (case insensitive)
WHERE $\equiv$ "where" (case insensitive)
IN $\equiv$ "in" (case insensitive)
AS $\equiv$ "as" (case insensitive)
HEAD $\equiv$ {"func","proc","type", "exception"} (case insensitive)
PARSERFIX $\equiv$ {"postfix","prefix","infix"} (case insensitive)
CONST ist die Menge aller Konstanten, d.h. aller Zahlen- und Stringliterale:

const	$\longmapsto$	number | string

number	$\longmapsto$	int | real
real	$\longmapsto$	decimal-card "." decimal-card["e" decimal-int]
int	$\longmapsto$	["-"]card
decimal-int	$\longmapsto$	["-"]decimal-card
card	$\longmapsto$	decimal-card | based-card
decimal-card	$\longmapsto$	{0123456789}$^+$
based-card	$\longmapsto$	decimal-card "#" (0123456789aAbBcCdDeEfF)$^+$
string	$\longmapsto$	" aChar* "
aChar	$\longmapsto$	*ein sichtbares Zeichen oder ein Leerzeichen*

ID ist die Menge aller Identifier:

ID	$\longmapsto$	standard-ID | special-ID
standard-ID	$\longmapsto$	alpha alphanum*
special-ID	$\longmapsto$	special-char$^+$
alphanum	$\longmapsto$	alpha | numeral
alpha	$\longmapsto$	{a-zA-Z_} *und lokale Umlaute*
numeral	$\longmapsto$	{0123456789}
special-char	$\longmapsto$	{!\$%&/=?‘’^#-.,:;~} *und verfügbare mathematische Sonderzeichen*

BINOP ist die Menge aller Infix-Operatoren. Diese Operatoren sind entweder links- oder rechtsassoziativ und mit einer Bindungsstärke zwischen 1 und 5 versehen (höher bindet schwächer).
PREOP ist die Menge aller Präfix-Operatoren.
POSTOP ist die Menge aller Postfix-Operatoren.

Anhang B

yacc–Syntax von ALDiSP

```
%token IF THEN ELSE ENDIF
%token ID CONST
%token DELAY SUSPEND UNTIL WITHIN
%token MODULE ENDMODULE
%token BINOPL1 BINOPL2 BINOPL3 BINOPL4 BINOPL5
%token BINOPR1 BINOPR2 BINOPR3 BINOPR4 BINOPR5
%token PREOP POSTOP
%token IMPORT FROM
%token EXPORTS AS

%token ARROW
%token LET IN WHERE ENDLET
%token SEQ ENDSEQ
%token NET ENDNET
%token GUARD ON ENDGUARD
%token FUNC PROC TYPE EXCEPTION ABSTYPE OVERLOADED

%%

pgm     : decls NET decls IN expr optendnet
        ;
optendnet: /* emtpy */
        | ENDNET
        ;
expr    : detexpr
        | suspend
        ;
detexpr : expr1l
        | expr1r
        | expr2
        ;
expr1l  : expr2 BINOPL1 expr1l
        | expr2 BINOPL1 expr2
```

```
        ;
expr1r  : expr1r BINOPR1 expr2
        | expr2 BINOPR1 expr2
        ;
expr2   : expr2l
        | expr2r
        | expr3
        ;
expr2l  : expr3 BINOPL2 expr2l
        | expr3 BINOPL2 expr3
        ;
expr2r  : expr2r BINOPR2 expr3
        | expr3 BINOPR2 expr3
        ;
expr3   : expr3l
        | expr3r
        | expr4
        ;
expr3l  : expr4 BINOPL3 expr3l
        | expr4 BINOPL3 expr4
        ;
expr3r  : expr3r BINOPR3 expr4
        | expr4 BINOPR3 expr4
        ;
expr4   : expr4l
        | expr4r
        | expr5
        ;
expr4l  : expr5 BINOPL4 expr4l
        | expr5 BINOPL4 expr5
        ;
expr4r  : expr4r BINOPR4 expr5
        | expr5 BINOPR4 expr5
        ;
expr5   : expr5l
        | expr5r
        | expr6
        ;
expr5l  : expr6 BINOPL5 expr5l
        | expr6 BINOPL5 expr6
        | expr6 ARROW expr6    /* Spezialfall ohne Klammern */
        ;
expr5r  : expr5r BINOPR5 expr6
        | expr6 BINOPR5 expr6
        ;
expr6   : expr6 POSTOP
        | expr7
```

```
        ;
expr7   : PREOP expr7
        | DELAY expr7
        | '[' typ ']' expr7
        | '[' '!' typ ']' expr7
        | expr8
        ;
expr8   : expr8 '(' exprs ')'
        | expr9
        ;
expr9   : ID
        | CONST
        | '(' expr ')'
        | '(' exprs ARROW exprs ')'
        | local
        | seq
        | if
        | guardedexpr
        ;
decls   : decl
        | decl decls
        ;
exprs   : expr
        | expr ',' exprs
        ;
if      : IF expr THEN tupel elsifs else ENDIF
        ;
elsifs  : /* empty */
        | '|' expr THEN tupel elsifs
        ;
else    : /* empty */
        | ELSE tupel
        ;
suspend : SUSPEND expr UNTIL expr WITHIN detexpr ',' detexpr
        ;
local   : LET decls IN tupel ENDLET
        | LET tupel WHERE decls ENDLET
        ;
seq     : SEQ stmts ENDSEQ
        ;
stmts   : stmt ';' stmts
        | tupel
        ;
stmt    : decl
        | expr
        ;
tupel   : expr
```

```
        | '(' expr ',' exprs ')'
        ;
typ     : detexpr
        ;
decl    : paramdecl
        | importdecl
        | moduledecl
        | simpledecl
        | multidecl
        ;
paramdecl : optoverload DECLOP declhead '=' expr
          ;
importdecl: optoverload IMPORT optIDs FROM ID
          ;
moduledecl: MODULE ID exportlists decls ENDMODULE ID
          ;
simpledecl: optoverload optdeclop ID '=' expr
          ;
multidecl : ID ',' IDs '=' tupel
          ;
declhead  : ID paramlists opttyp
          | ID BINOP ID opttyp
          | ID POSTOP opttyp
          | PREOP ID opttyp
          ;
IDs       : ID
          | ID ',' IDs
          ;
optIDs    : optID
          | optID ',' optIDs
          ;
optID     : ID optrename opttyp
          ;
optoverload: /* empty */
          | OVERLOADED
          ;
optdeclop : /* empty */
          | DECLOP
          ;
optrename : /* empty */
          | AS ID
          ;
opttyp    : /* empty */
          | ':' typ
          ;
exportlists: /* empty */
          | exportlist exportlists
```

```
          ;
exportlist: EXPORTS optIDs
          ;

paramlists: paramlist
          | paramlist paramlists
          ;
paramlist : '(' ')'
          | '(' params ')'
          ;
params    : param
          | param ',' params
          ;
param     : ID opttyp
          ;

guardedexpr: GUARD expr ON guards ENDGUARD
          ;
guards   : /* empty */
          | ON declhead '=' expr
          ;
```

Anhang C

Primitive & Typen

Die Typen und Primitiven sind in Form von Modul-Headern zusammengefaßt.

```
Module Standard
Exports
        bottom          : Bottom,
        Obj             : Type,
        Type            : Type,
        ->              : (Type,Type -> Type)       \ spezial-syntax
        Function        : Type,
        Bool            : Type,
        Predicate       : Type,
        Macro           : Type,
        Module          : Type,
        Exception       : Type,
        eq              : (Obj,Obj -> Bool)
        identity        : Obj -> Obj
        "or"            : (Bool, Bool -> Bool)
        "and"           : (Bool, Bool -> Obj)
        "not"           : Bool -> Bool
EndModule Standard

Module Numbers
Exports
        Number          : Type,
        Integers,
        FixInts,
        Reals,
        ComplexNumbers,

        "="             : (Number, Number -> Bool),
        "<>"            : (Number, Number -> Bool),
        ">"             : (Number, Number -> Bool),
        ">="            : (Number, Number -> Bool),
        "<"             : (Number, Number -> Bool),
        "=<"            : (Number, Number -> Bool),
        "+"             : (Number, Number -> Number),
        "-"             : (Number, Number -> Number),
        "*"             : (Number, Number -> Number),
```

```
"/"             : (Number, Number -> Number),
"+^"            : (Number, Number -> Number),
"-^"            : (Number, Number -> Number),

"sqrt"          : Number -> Number,
"sin"           : Number -> Number,
"cos"           : Number -> Number,
"tan"           : Number -> Number,
"asin"          : Number -> Number,
"acos"          : Number -> Number,
overloaded atan : Number -> Number
overloaded atan : (Number,Number -> Number),

log             : (Number,Number -> Number),
"ln"            : Number -> Number,
"ld"            : Number -> Number,
"exp"           : Number -> Number,
"**"            : (Number, Number -> Number),

max             : (Number, Number -> Number),
min             : (Number, Number -> Number)
"abs"           : Number -> Number
"signum"        : Number -> Number

DivisionByZero          : Exception,
DivisionOverflow        : Exception,
MultiplicationOverflow  : Exception,
ExponentiationOverflow  : Exception,
ExponentiationIsComplex : Exception,
AdditionOverflow        : Exception,
SubtractionOverflow     : Exception,
NegativeRoot            : Exception,
UndefinedTangens        : Exception,
TangensOverflow         : Exception,
LogIsComplex            : Exception,
TruncateOverflow        : Exception,
RoundError              : Exception,

SignalingDivision       :(Number,Number -> Number),
RealDivision            :(Number,Number -> Number),
SaturatedDivision       :(Number,Number -> Number),
WrappingDivision        :(Number,Number -> Number),
IgnoringDivision        :(Number,Number -> Number),

SignalingMultiplication :(Number,Number -> Number),
RealMultiplication      :(Number,Number -> Number),
SaturatedMultiplication :(Number,Number -> Number),
WrappingMultiplication  :(Number,Number -> Number),
IgnoringMultiplication  :(Number,Number -> Number),

SignalingExponentiation :(Number,Number -> Number),
RealExponentiation      :(Number,Number -> Number),
ComplexExponentiation   :(Number,Number -> Number),
SaturatedExponentiation :(Number,Number -> Number),
WrappingExponentiation  :(Number,Number -> Number),
```

```
      IgnoringExponentiation  :(Number,Number -> Number),

      SignalingAddition       :(Number,Number -> Number),
      RealAddition            :(Number,Number -> Number),
      SaturatedAddition       :(Number,Number -> Number),
      WrappingAddition        :(Number,Number -> Number),
      IgnoringAddition        :(Number,Number -> Number),

      SignalingSubtraction    :(Number,Number -> Number),
      RealSubtraction         :(Number,Number -> Number),
      SaturatedSubtraction    :(Number,Number -> Number),
      WrappingSubtraction     :(Number,Number -> Number),
      IgnoringSubtraction     :(Number,Number -> Number),

      SignalingRoot           :(Number -> Number),
      ComplexRoot             :(Number -> Number),
      IgnoringRoot            :(Number -> Number),

      SignalingTangens        :(Number -> Number),
      RealTangens             :(Number -> Number),
      SaturatedTangens        :(Number -> Number),
      IgnoringTangens         :(Number -> Number),

      SignalingLog            :(Number,Number -> Number),
      ComplexLog              :(Number,Number -> Number),
      IgnoringLog             :(Number,Number -> Number),

      SignalingTruncate       :(Number -> Number),
      SaturatedTruncate       :(Number -> Number),
      IgnoringTruncate        :(Number -> Number),

      Signaling       : Macro,
      Realizing       : Macro,
      Complexing      : Macro,
      Saturated       : Macro,
      Wrapping        : Macro,
      Ignoring        : Macro,

      RoundToZero     :(Number,Bit -> Number),
      RoundToPlus     :(Number,Bit -> Number),
      RoundToMinus    :(Number,Bit -> Number),
      RoundToEven     :(Number,Bit -> Number),
      RoundToOdd      :(Number,Bit -> Number)


Module Integers
Exports
      Integer         : Type,
      MinInt          : Integer,
      MaxInt          : Integer,
      nBitInteger     : Cardinal -> Type,
      "round"         : Number -> Integer,
      "truncate"      : Number -> Integer,
      "isEven"        : Integer -> Bool,
```

```
      "isOdd"         : Integer -> Bool
      "mod"           : (Integer, Integer -> Integer)
      "rem"           : (Integer, Integer -> Integer)
      BiasedCardinal  : (Type, Cardinal -> Type)

      Cardinal        : Type,
      MaxCard         : Cardinal,
      nBitCardinal    : Cardinal -> Type,

      "bitAnd"        : (Cardinal,Cardinal -> Cardinal),
      "bitOr"         : (Cardinal,Cardinal -> Cardinal),
      "bitXor"        : (Cardinal,Cardinal -> Cardinal),
      "bitInverse"    : Cardinal -> Cardinal,
      "bitReverse"    : Cardinal -> Cardinal,
      "bitLshift"     : (Cardinal,Cardinal -> Cardinal),
      "bitRshift"     : (Cardinal,Cardinal -> Cardinal),
      "bitLrotate"    : (Cardinal,Cardinal -> Cardinal),
      "bitRrotate"    : (Cardinal,Cardinal -> Cardinal),
      "bitCut"        : (Cardinal,Cardinal -> Cardinal),
EndModule Integers

Module FixInts
      ScaledFixpoint  : (Type,Cardinal -> Type),
      FixInt          : (Cardinal,Cardinal -> Type),
      FixCard         : (Cardinal,Cardinal -> Type),
EndModule FixInts

Module Reals
Exports
      Real            : Type,
      SmallestReal    : Real,
      LargestReal     : Real,


      ShortReal        : Type,
      SmallestShortReal: ShortReal,
      LargestShortReal : ShortReal,

      LongReal        : Type,
      SmallestLongReal: LongReal,
      LargestLongReal : LongReal,

      plusInfinity      : Real,
      minusInfinity     : Real,
      plusInfiniteSmall : Real,
      minusInfiniteSmall: Real
EndModule Reals

Module ComplexNumbers
Exports
      Complex         : Type -> Type,
      aComplex        : Type,
      makeComplex     : (Number, Number -> aComplex),
      "realPart"      : Complex -> Number,
      "imagPart"      : Complex -> Number,
```

```
        "phase"          : Complex -> Number,
        pointInfinity    : Complex,
  EndModule ComplexNumbers

  Module Time
  Exports
        Duration         : Type,
        Time             : Type,
        "sec"            : Number -> Duration,
        "ms"             : Number -> Duration,
        MinDuration      : Number,

        Time             : Type,
        SystemClock      : ( -> Time),
        "Year"           : Time -> Cardinal,
        "Month"          : Time -> Cardinal,
        "Day"            : Time -> Cardinal,
        "Hour"           : Time -> Cardinal,
        "Minute"         : Time -> Cardinal,
        "Second"         : Time -> Cardinal,
        makeTime         : (Cardinal,Cardinal,Cardinal,
                            Cardinal,Cardinal,Cardinal -> Time),
        "-"              : (Time, Time -> Duration)

  EndModule Time
EndModule Numbers


Module Strings
Exports
        String           : Type,
        AsciiToString    : Cardinal -> String,
        "++"             : (String, String -> String),
        "length"         : String -> Cardinal,
        ldrop            : (Cardinal, String -> String),
        rdrop            : (Cardinal, String -> String),
        NumberToString   : Number -> String
EndModule Strings


Module Arrays
Exports
        Array            : Type,
        "sizeOf"         : Array -> Cardinal,
        map              : ((Obj -> Obj), Array -> Array),
        dMap             : ((Obj -> Obj), Array -> Array)

  Module Vectors
  Exports
        Vector           : Type,
        VectorOf         : Type -> Type,
        createVector     : (Cardinal, (Cardinal -> Obj) -> Vector),
        "[".,  ... ,."]" : (Obj, ... ,Obj -> Vector),    \ Spezial-Syntax
        changeVector     : (Vector, Cardinal, Obj -> Vector),
        dChangeVector    : (Vector, Cardinal, Obj -> Vector),
```

```
        reduce          : ((Obj, Obj -> Obj), Vector -> Vector),
        rReduce         : ((Obj, Obj -> Obj), Vector -> Vector),
        "rev"           : Vector -> Vector,
        "evens"         : Vector -> Vector,
        "odds"          : Vector -> Vector,
        merge           : (Vector, Vector -> Vector),
        subVector       : (Vector, Cardinal, Cardinal -> Vector),
        compose         : (Vector, Vector -> Vector),
  EndModule Vectors

  Module Matrices
  Exports
        Matrix          : Type,
        MatrixOf        : Type -> Type,
        createMatrix    : (Cardinal, Cardinal, (Cardinal -> Obj) -> Matrix),
        MatrixLiteral   : (Vector, ... ,Vector -> Matrix),
        changeMatrix    : (Matrix, Cardinal, Cardinal, Obj -> Matrix),
        dChangeMatrix   : (Matrix, Cardinal, Cardinal, Obj -> Matrix),
        "hSize"         : Matrix -> Cardinal,
        "vSize"         : Matrix -> Cardinal,
        UnitMatrix      : Cardinal -> Matrix,
        "hRev"          : Matrix ->  Matrix,
        "vRev"          : Matrix ->  Matrix,
        "dRev"          : Matrix ->  Matrix,
        "oRev"          : Matrix ->  Matrix,
        "rot"           : Matrix ->  Matrix,
        "rot2"          : Matrix ->  Matrix,
        "rot3"          : Matrix ->  Matrix,
        "hEvens"        : Matrix ->  Matrix,
        "hOdds"         : Matrix ->  Matrix,
        "vEvens"        : Matrix ->  Matrix,
        "vOdds"         : Matrix ->  Matrix,
        hMerge          : (Matrix, Matrix ->  Matrix),
        vMerge          : (Matrix, Matrix ->  Matrix),
        hSubMatrix      : (Matrix, Cardinal, Cardinal ->  Matrix),
        vSubMatrix      : (Matrix, Cardinal, Cardinal ->  Matrix),
        hCompose        : (Matrix, Matrix ->  Matrix),
        vCompose        : (Matrix, Matrix ->  Matrix),
        Row             : (Cardinal, Matrix -> Vector),
        Col             : (Cardinal, Matrix -> Vector),
        diag            : (Matrix, Cardinal -> Vector),
        ndiag           : (Matrix, Cardinal -> Vector),
        "det"           : Matrix -> Number,
        hReduce         : (Function, Matrix -> Vector),
        vReduce         : (Function, Matrix -> Vector),
  EndModule Matrices
EndModule Arrays

Module  Lists
Exports
        ListOf          : Type -> Type,
        List            : Type,
        null            : List,
        cons            : (Obj, List -> List),
        "::"            : (Obj, List -> List),
```

```
        head            : List -> Obj,
        tail            : List -> List,
        isEmpty         : List -> Bool,
        isSingleton     : List -> Bool,
        select          : (Cardinal, List ->  Object),
        reduce          : (Function, List -> Object),
        ListToVector    : List -> Vector,
        VectorToList    : Vector -> List
        listifyVectors  : (ListOf(Vector) -> List)
        vectorizeList   : (Cardinal, List -> ListOf(Vector))
EndModule Lists

Module MachineTypes
Exports
        Interrupts,
        Registers,
        Ports

  Module Registers
  Exports
        Register        : Type,
        InputRegister   : Type -> Type,
        OutputRegister  : Type -> Type,
        anInputRegister : Type,
        anOutputRegister: Type,
        writeToRegister      : (anOutputRegister, Obj -> ),
        readFromRegister     : anInputRegister -> Obj
        "!"                  : (anOutputRegister, Obj -> ),
        "?"                  : anInputRegister -> Obj
        PipeFromRegister: (Duration, anInputRegister -> List),
        PipeToRegister  : (List -> anInputRegister),
        StreamToRegister: (Duration, List -> anInputRegister)
  EndModule Registers

  Module Ports
  Exports
        Port            : Type,
        InputPort       : Type,
        OutputPort      : Type,
        isPortReadable  : InputPort -> Bool,
        isPortWriteable : OutputPort -> Bool,
        writeToPort     : (OutputPort, Obj ->),
        readFromPort    : InputPort -> Obj
        "!"             : (OutputPort, Obj ->),
        "?"             : InputPort -> Obj
        PipeFromPort    : (anInputRegister -> List),
        PipeToPort      : (List -> anInputRegister)
  EndModule Ports

  Module Files
  Exports
        File            : Type,
        InputFile       : Type -> Type,
        OutputFile      : Type -> Type,
        AccessFile      : Type -> Type,
```

```
        anInputFile     : Type,
        anOutputFile    : Type,
        anAccessFile    : Type,
        openInputFile   : (String, Type) -> anInputFile,
        openOutputFile  : (String, Type) -> anOutputFile,
        openAccessFile  : (String, Type) -> anAccessFile,
        writeToFile     : (File, Obj  -> ),
        readFromFile    : File -> Obj,
        "!"             : (File, Obj  -> ),
        "?"             : File -> Obj,
        setAccessPointer: (File, Cardinal -> ),
        closeFile       : (File -> )
  EndModule Files

  Module Interrupts
  Exports
        Interrupt       : Type,
  EndModule Interrupts

EndModule MachineTypes

Module IPC
Exports
        Channel         : Type,
        ChannelOf       : Type -> Type,
        writeToChannel  : (Channel, Object -> ),
        readFromChannel : Channel -> Object,
        isChannelReadable:Channel -> Bool,
        isChannelWritable:Channel -> Bool,
        "?"             : Channel -> Object,
        "!"             : (Channel, Object -> ),
EndModule IPC

Module Stuff
Exports
        Error           : (Object, ... ,Object -> ),
        isAvailable     : Object -> Bool,
        isSuspended     : Object -> Bool,
        after           : Macro,
        when            : Macro,
        all             : Macro,
        "is"            : (Object, Constructor -> Bool)
EndModule Stuff
```

Anhang D

Minimale Primitive & Typen

Hier wird das Subset der in jeder ALDiSP–Implementierung erforderten Primitiven und Typen vorgestellt.

```
Module Standard
Exports
        bottom          : Bottom,
        Obj             : Type,
        Type            : Type,
        ->              : (Type,Type -> Type)       \ spezial-syntax
        Function        : Type,
        Bool            : Type,
        Predicate       : Type,
        Macro           : Type,
        Module          : Type,
        Exception       : Type,
        eq              : (Obj,Obj -> Bool)
        identity        : Obj -> Obj
        "or"            : (Bool, Bool -> Bool)
        "and"           : (Bool, Bool -> Obj)
        "not"           : Bool -> Bool
EndModule Standard

Module Numbers
Exports
        Number          : Type,
        Integers,
        Reals,

        "="             : (Number, Number -> Bool),
        "<>"            : (Number, Number -> Bool),
        ">"             : (Number, Number -> Bool),
        ">="            : (Number, Number -> Bool),
        "<"             : (Number, Number -> Bool),
        "=<"            : (Number, Number -> Bool),
        "+"             : (Number, Number -> Number),
        "-"             : (Number, Number -> Number),
        "*"             : (Number, Number -> Number),
```

```
        "/"             : (Number, Number -> Number),

        max             : (Number, Number -> Number),
        min             : (Number, Number -> Number)
        "abs"           : Number -> Number
        "signum"        : Number -> Number


  Module Integers
  Exports
        Integer         : Type,
        MinInt          : Integer,
        MaxInt          : Integer,

        "round"         : Number -> Integer,
        "truncate"      : Number -> Integer,
        "isEven"        : Integer -> Bool,
        "isOdd"         : Integer -> Bool

        Cardinal        : Type,
        MaxCard         : Cardinal,

  EndModule Integers


  Module Reals
  Exports
        Real            : Type,
        SmallestReal    : Real,
        LargestReal     : Real,

        plusInfinity      : Real,
        minusInfinity     : Real,
        minusInfiniteSmall: Real
  EndModule Reals


  Module Time
  Exports
        Duration        : Type,
        Time            : Type,
        "sec"           : Number -> Duration,
        MinDuration     : Number,

  EndModule Time
EndModule Numbers


Module Arrays
Exports
        Array           : Type,
        "sizeOf"        : Array -> Cardinal,
        map             : ((Obj -> Obj), Array -> Array),
```

```
  Module Vectors
  Exports
        Vector          : Type,
        createVector    : (Cardinal, (Cardinal -> Obj) -> Vector),
        "[".,  ... ,."]" : (Obj, ... ,Obj -> Vector),    \ Spezial-Syntax
        changeVector    : (Vector, Cardinal, Obj -> Vector),
        reduce          : ((Obj, Obj -> Obj), Vector -> Vector),
        rReduce         : ((Obj, Obj -> Obj), Vector -> Vector),
  EndModule Vectors

  Module Matrices
  Exports
        Matrix          : Type,
        createMatrix    : (Cardinal, Cardinal, (Cardinal -> Obj) -> Matrix),
        MatrixLiteral   : (Vector, ... ,Vector -> Matrix),
        changeMatrix    : (Matrix, Cardinal, Cardinal, Obj -> Matrix),
        "hSize"         : Matrix -> Cardinal,
        "vSize"         : Matrix -> Cardinal,
  EndModule Matrices
EndModule Arrays

Module Stuff
Exports
        isAvailable     : Object -> Bool,
        isSuspended     : Object -> Bool,
        "is"            : (Object, Constructor -> Bool)
EndModule Stuff
```

Anhang E

Vordefinierten Operatoren

Es folgt eine Auflistung aller vordefinierten Infix-Operatoren, geordnet nach ihren Bindungsstärken (1-5). Operatoren der Stärke '1' binden schwächer als Operatoren der Stärke '2', etc. Alle Operatoren bis auf das Infix-Cons (::) sind links-assoziativ.

Bindungsstärke 1

```
and or
```

Bindungsstärke 2

```
= < > >= =< <> ? !
```

Bindungsstärke 3

```
+ - ::
bitAnd bitOr bitXor
bitLshift bitRshift bitLrotate bitRrotate
bitCut
->
```

Bindungsstärke 4

```
* / mod
```

Bindungsstärke 5

```
**
```

Diese Bindungsstärken erlauben die korrekte Parsierung des Ausdrucks

```
y=42 or x=a+b*c**d
```

Es gibt einige vordefinierte Operatoren, die aus dem Konzept fallen, da sie als nicht sinnvoll verallgemeinerbar anzusehen sind. Hierzu gehören:

- Das 'Vektor-Literal' `[`$expr_1$`,...,` $expr_n$`]`
- Der Typkonstruktor `(`$type_1$`,...,` $type_n$ `->`$type_m$`,...,` $type_{m+k}$`)`, der auch ohne Klammern auftreten kann, wenn es nur je einen Argument- und Resultattypen gibt.
- Der Tupel-Konstruktor `(`$expr_1$`,...,` $expr_n$`)` – auch wenn Tupel keine echten Datentypen sind, werden sie doch intuitiv als solche aufgefaßt.

Präfix-Operatoren

```
not      sqrt     sin          cos        tan
asin     acos     ln           ld         exp
signum   bitNot   bitReverse   round      truncate
isEven   isOdd    realPart     imagPart   phase
length   sizeOf   rev          evens      odds
hSize    vSize    hRev         vRev       dRev
oRev     rot      rot2         rot3       hEvens
hOdds    vEvens   vOdds        det        head
tail     isEmpty
```

Postfix-Operatoren

```
sec ms
```

Anhang F

Fallbeispiele

F.1 16-Kanal Mixer

Beschreibung

Ein Mixer vermischt mehrere Signale, indem er sie zusammenaddiert. Die einzelnen Signalquellen können dabei den verschiedensten Modifikationen ausgesetzt sein. Im folgenden Beispiel gehe ich davon aus, daß es 16 analoge Inputs, einen A/D-Wandler (16 Bit) und einen analogen Output (der direkt a einem 16-Bit D/A-Wandler sitzt) gibt. Zudem gibt es 16 Misch-Regler, einen pro Kanal. Die Regler werden über einen billigen 8-Bit A/D-Wandler abgetastet. Es gibt zwei Mutiplex-Steuerregister, die bestimmen, welche Inputs an den beiden A/D-Wandlern anliegen. Die Abtastfrequenz ist theoretisch beliebig, praktisch wird ein Wert von 44KHz vorgegeben.

```
\ Beispielprogramm:
\ 16-Kanal-Mixer

SamplingRate = (1/44000) sec

\ Die Adressen der Hardware-Komponenten seien wie folgt gegeben:

input16bit= [!InputRegister(Word)] [!address]  #x0ef000
output16Bit=[!OutputRegister(Word)][!address]  #x0ef004
input8bit = [!InputRegister(Byte)] [!address]  #x0ef008

selector16bit=[!OutputRegister(Nibble)][!adress]#x0ef010
selector8bit =[!OutputRegister(Nibble)][!adress]#x0ef014

proc readAndMixInput(n:Nibble) =
 {
  writeRegister(selector16bit,n);
```

```
  writeRegister(selector8bit,n)

  Sample = readRegister(selector16bit);
  Aussteuerung = readRegister(selector8bit));

  \
  \ der Aussteuerungswert wird als Faktor zwischen 0 und 1
  \ interpretiert.
  \ die Berechnung muss in der richtigen Reihenfolge
  \ ausgef\"uhrt werden, damit keine Bits verlorengehen.

  tmp=([!nBit(24)]Sample)*Aussteuerung;
  [!Word](tmp/256)
 }

\ wenn es beim Addieren der Samples zu einem Overflow kommt,
\ sollte das Ergebnis der maximale Sample sein.
\ (per Default w\"are es Wrap-Around.)

proc AccumulateInput(n,akku) =
 {
  if n<0 then akku
         else AccumulateInput(n-1,
                              Wrapping(akku+readAndMixInput(n))
                             )
  endif
 }

proc DoMixing() =
 writeRegister(output16Bit,AccumulateInput(15,0))

proc RunMixer() =
 suspend
  {
   RunMixer();
   DoMixing()
  }
 until true
 within SamplingRate,SamplingRate

net
 \ keine Netzbeschreibung
in
RunMixer()        \ dies ist der Programmstart
```

```
endnet
```

F.2 Definition einiger primitiver Funktionen und Prozeduren

Im Folgenden sollen für einige vorgegebene Funktionen beispielhafte ALDiSP–Implementierungen gegeben werden.

```
proc PipeFromRegister(F : Duration; R: InputRegister) =
  readFromRegister(R) ::
   suspend SampleRegister(F, R)
     until true
    within F,F

proc StreamToRegister(FrameDuration : Duration;
                      R : Outputregister;
                      D : List) =
  seq
    writeToRegister(R,head(L));
    suspend PipeToRegister(FrameDuration, R, tail(L))
      until true
     within FrameDuration,FrameDuration
  endseq

proc PipeToRegister(R : Outputregister;
                    D : List) =
  seq
    writeToRegister(R,head(L));
    PipeToRegister(R,tail(L));
  endseq
```

Literaturverzeichnis

[1] Brooks, Frederick P.:
The Mythical Man-Month. Essays on Software-Engineering
Reading, MA: Addison-Wesley, 1982

[2] Rees, Jonathan and William Clinger (eds.):
Revised^{3} Report on the Algorithmic Language Scheme,
SIGPLAN Notices V21 #12, December 1986

[3] Milner, Robin:
A Proposal for Standard ML
MacQueen, David:
Modules for Standard ML
beide in: 1984 ACM Symposion on Lisp and Functional Programming

[4] Turner, D.A.: *Miranda: A non-strict functional language with polymporphic types*
in: Conference on Functional Programming Languages and Computer Architecture, Nancy, Springer, pp.1-16

[5] Dybvig, R. Kent:
The Scheme Programming Language
Englewood Cliffs, NJ: Prentice-Hall, 1987

[6] Anderson, David P., and Ron Kuivila:
Accurately Timed Generation of Musical Events
in: Computer Music Journal 10,3 (Fall 1986), pp. 48-56

[7] O'Toole, James William Jr., and David K. Gifford:
Typ Reconstruction with First-Class Polymorphic Values
in: 1989 ACM Conference on Language Implementation and Design (oder so ähnlich, noch mal genau nachschaun. Da rächt sich die Faulheit!)

[8] Kranz, David, et al.:
ORBIT: An Optimizing Compiler for Scheme
in: Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, pp. 219-233

[9] INMOS, Ltd.:
Occam 2: Das Refernz-Handbuch
München; Wien: Hanser; Hemel
Hempstead: Prentice-Hall International, 1989

[10] Halstead, Robert H.:
Multilisp: A Language for Concurrent Symbolic Computation
in: ACM Transactions on Programming Languages and Systems 7:4, October 1985, pp.501-538

[11] Holt, Richard C., et.al.:
The Turing Programming Language: Design and Definition
Englewood CLiffs, NJ: Prentice-Hall, 1988

[12] Kranz, David A., Robert H. Halstead, Jr. and Eric Mohr:
Mul-T: A High-Performance Parallel Lisp
in: 1989 ACM Conference on Programming Language Design and Implementation, pp. 81-90

[13] Steele, Guy L. Jr.:
Common LISP: The Language
Bedford, MA: Digital Press, 1984

[14] Wirth, Niklaus:
Programming in Modula-2 [Second Edition]
Berlin, Heidelberg, New York: Springer, 1983

[15] Wilson, Paul R. and Thomas G. Moher:
Demonic Memory for Process Histories
in: Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, pp. 330-343

[16] Pratt, Vaughan:
Five Paradigm Shifts in Programming Language Design and Their Realization in Viron, a Dataflow Programming Environment
Department of Computer Science, Stanford University, Report No. STAN-CS-82-951, December 1982

[17] Goldberg, Adele, and David Robson: *Smalltalk-80: The Language and its Implementation* Reading, MA: Addison-Wesley, 1985 o

[18] *IEEE Standard for Binary Floating-Point Arithmetic*
IEEE Standards Board/American Standards Institute: ANSI/IEEE 754-1985

[19] McCarthy, John, et al.:
LISP 1.5 Programmer's Manual [Second Edition]
Cambridge, MA: The M.I.T. Press, 1965

[20] *DSP56000/DSP56001*
Digital Signal Processor User's Manual
(Handbuch zur Motorola 56000-Baureihe)

[21] *EDC/Silage Reference Manual*
European Development Center, 1989

Index

Index