Datenstrukturen unter Forth:
Eine alte Lösung für ein altes Problem.

Dr.-Ing. Egmont Woitzel
FORTecH Software GmbH
Joachim-Jungius-Straße 9
O-2500 Rostock
Tel. (03 81) 44 21 85
Fax (03 81) 4 65 92 00

Datenstrukturen werden von Forth nur sehr stiefmütterlich unterstützt. In jüngster Zeit gab die Programmierung unter Windows™ erneut Anlaß für eine Portierung einer alten Lösung des Problems. Das vorgestellte Paket zur Beschreibung strukturierter Daten erlaubt die Definition beliebiger sequentieller Datenfelder sowie einen effizienten, symbolischen Zugriff auf die Datenelemente. Die Programmierung mit diesem Paket wird der üblichen Programmierpraxis gegenübergestellt, abschließend werden kurz einige Implementierungsvarianten diskutiert.

1. Das alte Problem: Viele Möglichkeiten sind noch keine Lösung.

1.1 Ein Balanceakt: Forth zwischen Algorithmen und Datenstrukturen.

Die Diskussion um Erweiterungen von Forth bzw. des Forth-Standards um Worte zur Beschreibung bzw. Definition von Datenstrukturen ist wahrscheinlich so alt wie Forth selbst. In der Literatur und auf Tagungen wurden diverse Vorschläge zu diesem Thema diskutiert, am ausführlichsten wahrscheinlich in [1]. Die Argumente der Befürworter einer Erweiterung wie die der Gegner sind bekannt und nachvollziehbar.

Das meistzitierte Argument gegen die Standardisierung von Forthworten für die Definition von Datenstrukturen ist die Aussage, man brauche solche Worte überhaupt nicht, da die `CREATE-DOES>`-Konstruktion die Definition beliebiger Datenstrukturen auf die jeweils effektivste Weise erlaube. Als Beispiel wird dann in der Regel angeführt, daß die unterschiedliche Implementierung von Arrays für unterschiedliche Anwendungen zu den jeweils effizienteren Programmen führt.

Für die Standardisierung spricht allerdings, daß der weitaus größte Teil aller Datenstrukturen ohne Effizienzeinbußen mit Hilfe standardisierter Worte definiert werden kann. Einer speziellen Implementierung für ein spezielles Problem steht ja trotz allem nichts im Wege. Der wirkliche Vorteil bestünde in der einfachen, schnellen und übersichtlichen Definition der immer wieder auftauchenden Datenstrukturen und damit in einer deutlichen Entlastung des Programmierers in seiner Alltagsarbeit.

Die Hartnäckigkeit, mit der die Diskussion zum Thema geführt wird, läßt nach den Ursachen fragen. Der Schlüssel für das Problem liegt vermutlich im streng funktionsorientierten Ansatz von Forth selbst. Eines der wichtigsten Konzepte von Forth ist die Vereinheitlichung von Datenmoduln und Funktionsmoduln in Worten. Dies geschieht durch die "Verpackung" von Datenmoduln in Funktionsmoduln, die - übersetzt gesprochen - Zeiger auf die Datenmoduln liefern. Die dadurch ohnehin verursachtete Betonung der Algorithmen wurde dann allerdings durch den Verzicht auf eine Konstruktion zur Strukturierung von Datenmoduln - im Gegensatz zu den hervorragenden Strukturierungsmöglichkeiten für Funktionsmoduln - zusätzlich verstärkt. Die fehlende Balance zwischen den Ausdrucksmöglichkeiten für Datenmoduln und Funktionsmoduln, notwendig für eine ausgewogene Programmentwicklung, liefert seitdem den Diskussionsstoff.

Jüngste Anregung, sich mit diesem "alten Problem" zu befassen, war die Bekanntschaft mit dem Windows-API (siehe auch [2]). Eigentlich als C-Interface ausgelegt, werden durch dieses Interface auch Zeiger auf relativ komplexe Datenstrukturen, vorzugsweise Records, hin und her getragen, die entweder vor der Übergabe oder nach der Rückkehr aus Windows vom Programmierer auszufüllen bzw. auszuwerten sind. Handhabbar ist solch ein Interface nur, wenn die Adressierung symbolisch erfolgt. Da das eingesetzte WinForth™ keine geeigneten Konstrukte enthielt, löste erst die Portierung der "alten Lösung", des comFORTH-Record-Pakets [3], die Probleme.

1.2 Entscheidung: Datenstrukturen oder Datentypen?

Vor der Implementierung eines Datenstrukturpakets sollte man zuerst die zu realisierende Funktionalität definieren. Es lassen sich dabei zwei prinzipielle Varianten unterscheiden. Die einfachere Zielstellung besteht in der Beschreibung von *Datenstrukturen* als Beschreibung ihres Speicherabbildes. Die komplexere wäre die Beschreibung von *Datentypen*, d. h. der Beschreibung der Datenstruktur zusammen mit den zulässigen Operationen über dieser Datenstruktur.

Ausschlaggebend für die Entscheidung für die eine oder die andere Variante ist neben dem wesentlich höheren Aufwand für die Datentypvariante letzendlich das zu erreichende Ziel. Primär wird ein Verfahren gesucht, mit dem man auf strukturierte Daten effektiv symbolisch zugreifen kann. Dies ist bereits mit der einfachen Datenstrukturvariante erreichbar. Erst wenn zusätzliche Sicherungsmechanismen gefordert werden, die die Zulässigkeit von Operationen über den Daten testen sollen, ist die Datentypvariante erforderlich. Allerdings ist solch ein Vorgehen unter Forth eher untypisch, kein standardisiertes Wort überprüft die Korrektheit seiner Eingangsparameter. Die Datenstrukturvariante erscheint daher eine Forth angemessene Behandlung des Problems zu sein.

Die Implementierung einer Datentypvariante erscheint beim heutigen Stand der Softwaretechnik auch nicht mehr sinnvoll. Objektorientierte Systeme gehen in ihren Eigenschaften und Möglichkeiten weit über die nur typisierter Systeme hinaus. In [4] wird eine derartige Implementierung vorgestellt.

1.3 Werkzeugkasten: Was braucht man wirklich?

Wie oben dargestellt, sind die Beschreibung des Speicheraufbaus sowie die Zugriffsorganisation auf die einzelnen Elemente die wichtigsten Aufgaben eines Datenstrukturpakets. Ohne die Verwendbarkeit des Pakets zu beschneiden, ist hier die Eingrenzung auf sequentielle gespeicherte Datenstrukturen zulässig. Glücklicherweise ist Forth aufgrund seiner Freizügigkeit im Umgang mit Speicheradressen weitaus besser als die meisten anderen Sprachen auf die Behandlung von Pointern eingerichtet.

Zur Beschreibung sequentieller, strukturierter Daten genügen drei Elemente:

- ***Atome*** als unstrukturierte Datenfelder, gekennzeichnet durch ihren Speicherbedarf in Byte
- ***Arrays*** als mehrfache Wiederholung derselben Struktur, gekennzeichnet durch die Wiederholungsanzahl und die Art der Elemente
- ***Records*** als Verbund mehrerer Felder unterschiedlicher Art, gekennzeichnet durch Reihenfolge und Art der Felder

Aus diesen Elementen kann man jede sequentielle Struktur mit Ausnahme varianter Records konstruieren, die jedoch durch die Vereinbarung mehrerer Strukturen ersetzen kann.

Um den Zugriff auf die einzelnen Elemente zu ermöglichen, genügt die Bereitstellung ihrer Adressen. Im Fall von Atomen ist das einfach die Anfangsadresse des zugeordneten Speicherbereichs. Bei Arrays ist eine Berechnung der Elementeadresse auf Basis eines *Index* notwendig. Die Berechnung von Elementeadressen in Records erfolgt über die Felder, die eine *Offsetaddition* vornehmen.

Alle genannten Verfahren sind elementar und werden von allen gängigen Hochsprachen und mittlerweile auch von vielen Assemblern durch syntaktische Konstruktionen, also als Bestandteil der Programmiersprache angeboten. In der Regel empfinden deshalb selbst wohlwollende "fremdsprachige" Programmierer das Fehlen entsprechender Sprachbestandteile unter Forth berechtigt als größere Zumutung als umgekehrt polnisch programmieren zu müssen.

Ein Blick auf die in vielen Forth-Programmen auch heute noch benutzten Programmiertechniken zeugt von Problemen, die bei Benutzung anderer Sprachen gar nicht auftreten.

2. Die ältere Lösung: Handverlesene Datenfelder.

2.1 Ein Team: `CREATE` und `ALLOT`.

Das Anlegen von Atomen gestaltet sich unter Forth-83 noch recht einfach und anschaulich mit Hilfe von `CREATE` und `ALLOT`:

```
CREATE PUFFER  128 ALLOT
```

Für "Fremdsprachler" ungewohnt ist nur die Trennung zwischen Definition des Wortes (Moduls) und der Bereitstellung von Speicher. Verständlich wird diese Definition erst, wenn man die Verwaltung des Wörterbuchs kennt. Durch das Laufzeitverhalten von CREATE wird beim Aufruf von PUFFER die korrekte Adresse geliefert.

2.2 Paradebeispiel für DOES>: Arrays.

Zur Definition von Arrays wird meist die CREATE-DOES>-Konstruktion verwendet. Gleichermaßen wird gern die Funktion der genannten Konstruktion anhand von Arrays erläutert. Typische Definitionsworte sehen folgendermaßen aus:

```
: ARRAY: ( ps: size #e ===> )( ib: name )( definiert Array )
         ( name mit #e Elementen je size Byte )
         CREATE OVER , * ALLOT DOES>
         ( ps: i ===> addr )( Adresse von Element i )
         DUP 2+ SWAP @ ROT * + ;
```

Bei der Definition eines konkreten Arrays sind dann die Anzahl der Elemente und deren Größe anzugeben, beim Aufruf des Arrays wird aus dem Index die Elementadresse berechnet:

```
2 10 ARRAY: INVALUE
1 3 INVALUE !
```

Dieser Ansatz birgt mehrere Probleme. Zunächst wird durch die Verbindung von Modulaufruf und Elementzugriff die Definition von Worten, die Operationen über dem gesamten Array ausführen (z. B. den Betrag des Vektors berechnen), erschwert. Dieses Problem kann am einfachsten durch die Auslösung des Elementzugriffs aus dem DOES>-Part gelöst werden. Überraschenderweise erübrigt sich damit die Notwendigkeit einer DOES>-Konstruktion überhaupt.

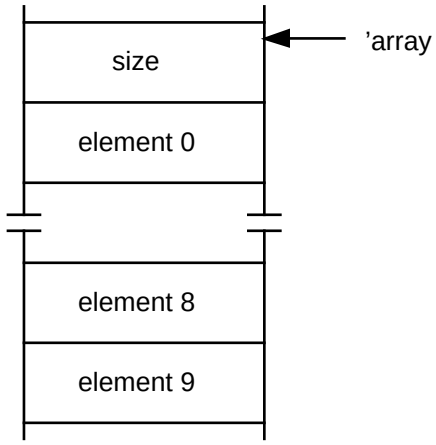

Bild 1: Speicheraufbau des Arrays

```
: ARRAY:  ( ps: size #e ===> )( ib: name )
          ( #e Elememente  je size Byte )
          CREATE OVER , * ALLOT ;
: ELEMENT ( ps: i 'array ===> addr )( Adresse von Element i )
          DUP 2+ SWAP @ ROT * + ;

2 10 ARRAY: INVALUE    \ definiert Array mit 10×2 Byte
1 INVALUE 3 ELEMENT ! \ setzt 4. Element gleich 1
```

Es verbleibt aber als zweites Problem, daß gemeinsam mit den Daten Strukturinformationen gespeichert werden (nämlich die Größe der Elemente). Dies verhindert letztendlich die rekursive Definition zusammengesetzter Datenstrukturen. So ist z. B. die Rückführung einer zweidimensionalen Matrix auf einen eindimensionalen Vektor von eindimensionalen Vektoren nicht mehr möglich, da jeder Teilvektor wieder das size-Feld enthalten müßte.

2.3 Ein weites Feld: Records.

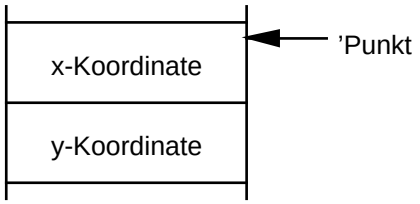

Bild 2: Speicheraufbau des Punktes

Bei der Definition von Records reicht die Programmierpraxis von "überhaupt nicht beachten" über "Einzeldefinitionen" bis zu speziellen "Definitionsworten für Recordfelder". Diese drei Ansätze sollen kurz anhand eines ganz typischen Beispiel illustriert werden. Geometrische Punkte werden im kartesischen Koordinatensystem anhand ihrer x- und y-Koordinate beschrieben. Bei der Speicherung eines Punkts müssen logischerweise beide Koordinaten als Felder eines Records gespeichert werden. Als Genauigkeit sollen hier einfachgenaue Integer genügen.

Datenstrukturen wie der Punkt sind noch so übersichtlich, daß sie von vielen Programmierern ***nicht beachtet*** werden (man hat die Sache eben voll im Griff). Um einen Punkt zu definieren, verwendet man daher einfach 2VARIABLE, und wenn man auf die y-Koordinate zugreifen will, sagt man einfach 2+. Das sieht dann etwa so aus:

```
: POINT+! ( ps: x y 'point ===> )( verschiebt 'point um xy )
          SWAP OVER 2+ +! +! ;

2VARIABLE URSPRUNG
4 5 URSPRUNG POINT+!
```

Das Problem beginnt eigentlich erst, wenn aus irgendeinem Grund die Reihenfolge von x und y in der Struktur getauscht werden muß (z. B. wegen der Ankopplung eines fremden Programmpakets). Die Suche nach fehlenden und überschüssigen 2+ kann dann zum Horrortrip werden. Einmal durch ein derartiges Erlebnis geläutert, verwendet man fortan doch lieber eine symbolische Notation für die Beschreibung des Records, wobei man herkömmliche ***Einzeldefinitionen*** verwendet.

```
: X ( ps: 'point ===> 'x ) ;
: Y ( ps: 'point ===> 'y ) 2+ ;

: POINT+! ( ps: x y 'point ===> )( verschiebt 'point um xy )
          SWAP OVER Y +!  X +! ;
```

Bei einer Änderung der Datenstruktur sind nun nur noch die beschreibenden Worte X und Y betroffen. Lästig ist nur die Definition der Feldnamen, besonders bei etwas größeren Datenstrukturen. Daher findet man häufig ein ***Definitionswort*** für solche Offsetaddierer:

```
: OFFSET: ( ps: offset ===> )( ib: name )
          CREATE , DOES>
          ( ps: addr ===> addr+offset )
          @ + ;

0 OFFSET: X
2 OFFSET: Y
```

Noch nicht behoben ist das Problem, daß der Programmierer sowohl für die Reservierung der richtigen Speichergröße als auch für die Berechnung der korrekten Offsets selbst zuständig ist. Brodie diskutiert verschiedene Verbesserungsmethoden unter dem Stichwort "Compilationszeit-Faktorisierung" ausführlich in [5]. In den meisten Forth-Programmen findet man abgesehen von einigen möglichen Optimierungen kaum Code, der über diese Techniken hinausgeht. Besonders in zeitkritschen Anwendungen scheut man häufig auch den Laufzeit-Overhead, den die beiden letztgenannten Varianten ohne höheren Optimierungsaufwand verursachen.

3. Die alte Lösung: Das comFORTH Recordpaket.

3.1 Überblick: Leistungen und Verfügbarkeit.

Das comFORTH-Recordpaket stellt für alle oben angesprochenen Probleme effiziente Lösungen zur Verfügung. Es werden frei miteinander verschachtelbare Beschreibungsworte für Atome, Arrays und Records sowie die dazugehörigen

Worte für die Definition von Datenobjekten bzw. den Zugriff auf einzelne Elemente angeboten. Der Programmierer braucht sich dabei weder um die physikalische Größe der Datenstrukturen selbst zu kümmern noch Angst vor Laufzeit-Overhead zu haben.

Der Schlüssel für die Erzielung dieser Eigenschaften war die strikte Trennung der Strukturbeschreibung, der Definition von Datenobjekten sowie des Zugriffs auf Elemente dieser Objekte. Durch das Recordpaket werden, wie oben diskutiert, keine Typprüfungen vorgenommen. Seine einzige Aufgabe besteht in der Realisierung eines einheitlichen, effizienten und symbolischen Adreßrechnungsverfahrens für strukturierte Daten.

Das Recordpaket ist optionaler Bestandteil der Basissysteme comFORTH 2.x und comFORTH 3.x, es kann dort von der ELECTIVE.CFx nachgeladen werden. Das Recordpaket ist außerdem auch unter dem CrossCompiler in Form einer Macrobibliothek verfügbar und wird dort z. B. während der Übersetzung des comFORTH-Kernels benutzt. Die jüngste Portierung erfolgte nun, wie bereits oben erwähnt, für das WinForth™ von LMI. Das komplette Listing ist im Anhang 1 zu finden. Die Definitionen der wenigen über den Umfang von Forth-83 hinausreichenden benutzten Worte sind im Anhang 2 aufgeführt, so daß einer Portierung auf ein beliebiges anderes Forth-83-System nichts im Wege stehen dürfte.

3.2 Klartext: Strukturen und ihre Beschreibung.

3.2.1 Strukturbeschreiber

Alle Definitionen im Record-Paket basieren auf dem Konzept eines aktuellen Strukturbeschreibers, der im weiteren als ***aktueller Typ*** (Achtung: Es handelt sich hierbei nicht um einen Typ im oben benutzten strengen Sinn!) bezeichnet werden soll. Dieser aktuelle Strukturbeschreiber wird von der Zeigervariablen `KIND^` verwaltet. Jeder Strukturbeschreiber trägt bei seinem Aufruf die eigene Adresse in diese Variable ein.

Die Strukturbeschreiber für die verschiedenen Arten sind untereinander kompatibel aufgebaut. Das erste Feld enthält immer einen Verweis auf einen Untertyp, das zweite den Speicherbedarf für ein Datenobjekt. Der Untertyp verweist für Atome und Records immer auf sich selbst, nur bei Arrays auf den Typ der Elemente. Im Fall von Arrays enthält der Beschreiber noch ein weiteres Feld für den größtmöglichen Index. Dieses Feld wird in der vorliegenden Implementierung nicht verwendet. Optional wäre aber eine Laufzeit-Überprüfung von Indexgrenzen damit möglich.

Felder von Records benötigen ebenfalls einen Beschreiber. Seinen Aufbau zeigt Bild 4. Das erste Feld verweist auf den Strukturbeschreiber des Records, zu dem das Feld gehört, das zweite auf den Strukturbeschreiber, der zum Typ des Feldes gehört. Im dritten Eintrag ist der Offset des Feldes bezogen auf den Start des Records zu finden. Diese Informationen erlauben zur Compilationszeit eine Überprüfung der Feldzugriffe. Optional wäre auch eine Laufzeitkontrolle denkbar, die aber nicht implementiert wurde.

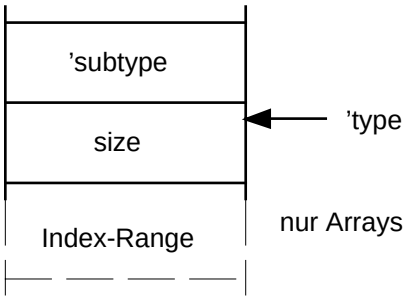

Bild 3: Aufbau von Typbeschreibern

3.2.2 Atome

Atome sind die Basisbausteine für die Strukturbeschreibung. Ihre Definition ist sehr einfach, es genügt die Angabe des Speicherbedarfs in Byte, also z. B.:

```
  1 ATOM: BYTE
  2 ATOM: CELL
128 ATOM: BUFFERTYPE
```

Der Aufruf eines der definierten Worte aktiviert den jeweiligen Typ.

3.2.3 Arrays

Ähnlich gestaltet sich die Definition von Array's, die als eindimensionale Wiederholungen von Elementen desselben Typs aufgefaßt werden. So definiert

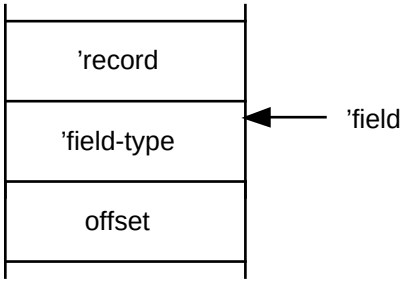

Bild 4: Aufbau von Feldbeschreibern

```
CELL 10 ARRAY: IVALARRAY
```

einen Typ, der ein Array aus 10 Elementen beschreibt, die vom Untertyp `CELL` sind und damit jeweils zwei Byte benötigen.

3.2.4 Records

Ein wenig komplizierter sieht die Definition von Records aus, da hier jedes Element einen anderen Typ besitzen darf. Der Punkt ließe sich damit folgendermaßen beschreiben:

```
RECORD: POINT
        CELL FIELD: X
        CELL FIELD: Y  ;RECORD
```

Unter dem Namen `POINT` steht nun die Strukturbeschreibung eines Punktes bereit.

3.2.5 Etwas verfeinert: Verschachtelt und Namenlos.

Die in den vorangegangenen Abschnitten wurde vorgeführt, wie man mit dem comFORTH-Record-Paket die unter Punkt 2. definierten Datenstrukturen beschreibt. An dieser Stelle sollen in Kurzform die wichtigsten darüber hinaus gehenden Möglichkeiten des Pakets erläutert werden.

Die zweifellos wichtigste Neuerung ist die problemlose Verschachtelbarkeit der Strukturbeschreibung. Arrays und Felder akzeptieren nicht nur Atome, sondern auch Arrays und Records. Eine Datenstruktur Rechteck kann man daher auch mit Hilfe von `POINT` beschreiben:

```
RECORD: RECTANGLE
        POINT FIELD: TOPLEFT
        POINT FIELD: BOTTOMRIGHT ;RECORD
```

Beide Felder beanspruchen jeweils 4 Byte für die x- bzw. y-Koordinate. Eine zweite nützliche Ergänzung ist die Möglichkeit, namenlose Typen zu definieren, vorzugsweise unmittelbar vor der Definition von Feldern oder Arrays. Die entsprechenden Typdefinitionsworte unterscheiden sich nur durch das Fehlen des `:` von den aufgeführten. Die namenlose Beschreibung von Datenstrukturen ist vor allem dann sinnvoll, wenn der Typ nur einmal benötigt wird und kein Datenobjekt dieses Typs in der Applikation benötigt wird. So beschreibt

```
CELL 10 ARRAY 20 ARRAY: IVALMATRIX
```

ein Array aus 20 Zeilen, die selbst aus jeweils 10 Zellen bestehen. Beide Möglichkeiten - Verschachtelung und namenlose Definition - sind jeweils in allen Situationen und in beliebiger Tiefe anwendbar, auch die Definition eines namenloser Records vor der Definition eines Feldes ist möglich.

3.3 Plan und Tat: Datenobjekte bauen.

Die unter Punkt 3.2 behandelten Konstruktionen erfassen zunächst nur die Beschreibung der Struktur eines Datenfeldes. Von keinem der Elemente wird bisher Speicher für ein konkretes Datenobjekt reserviert. Diese Aufgabe wird von dem Wort `OBJECT:` übernommen, das vom jeweiligen aktuellen Typ ausgehend Speicher im Wörterbuch reserviert.

```
 IVALARRAY OBJECT: INVALUE
IVALMATRIX OBJECT: INMATRIX
     POINT OBJECT: URSPRUNG
 RECTANGLE OBJECT: RAHMEN
```

Der Aufruf eines der definierte Worte liefert die Anfangsadresse des zugehörigen Datenobjekts. Eine Typaktivierung findet nicht statt, die Gründe hierfür werden in Abschnitt 4. diskutiert. Zusätzlich ist auch die Reservierung von Speicher ohne die Definition eines Wortes mit Hilfe von `OBJECT` möglich. Dieses Wort liefert die Anfangsadresse des

Datenobjekts auf dem Stapel. Dadurch kann man es z. B. in Definitionsworten verwenden, die das neue Objekt sofort initialisieren oder ihm einen eigenen DOES>-Part mitgeben.

3.4 Arbeit für den Compiler: Zugriff auf Datenelemente.

3.4.1 Adressierung von Arrayelementen

Die Adressierung von Arrayelementen erfolgt mit Hilfe des Worts ELEMENT, das in Abhängigkeit vom aktuellen Typ die Indexberechnung vornimmt. Da das Datenobjekt selbst keine Typaktivierung vornimmt, muß man vor Element noch einmal den Typnamen aufrufen.

```
1 INVALUE IVALARRAY 3 ELEMENT !
```

ELEMENT aktiviert nach der Indexberechnung den Typ der Arrayelemente, so daß bei verschachtelten Strukturen kein weiterer Aufruf von Typnamen erforderlich ist. Mit

```
0 INMATRIX IVALMATRIX  18 ELEMENT 7 ELEMENT !
```

wird die 8. Zelle der 19. Zeile der Matrix INMATRIX Null gesetzt.

3.4.2 Adressierung von Recordfeldern

Analog wird verfahren, wenn man Recordfelder adressiert. Die Offsetberechnung wird dabei von den Feldnamen übernommen. Ebenso wie der Elementzugriff bei Arrays aktiviert die Anwahl eines Feldes den Typ des Feldelements. Die folgenden Zeilen dokumentieren sich selbst.

```
0 URSPRUNG POINT X !
20 RAHMEN RECTANGLE TOPLEFT Y !
```

3.4.3 Compilertricks: Besser als der Profi.

Oben wurde die Forderung bzw. Behauptung aufgestellt, daß auch bei Anwendung des Recordpakets effizienter Code erzeugt wird. In diesem Abschnitt sollen die wesentlichen Verfahren angesprochen werden, mit denen dafür gesorgt wird, daß bei Anwendung des Recordpakets sogar in der Regel schnellerer Code produziert wird als bei der klassischen "Nichtbeachten"-Variante.

Um möglichst viele Berechnungen bereits zur Übersetzungszeit vornehmen zu können, werden alle Worte zur Typbeschreibung automatisch als IMMEDIATE deklariert. Typaktivierungen finden daher zur Übersetzungszeit statt. Dies trifft auch für die Aktivierung von Elementetypen bei Array- und Feldzugriffen zu. Compiliert werden nur die wirklich zur Laufzeit absolut notwendigen Berechnungen. Bei Arrayzugriffen ist dies die Berechnung des Offsets zum jeweiligen Element und die Addition zur Anfangsadresse nach der folgenden Formel

$$'element = 'array + (index \times size)$$

Dabei werden der *index* und meist auch die Arrayadresse *'array* erst zur Laufzeit bekannt. Die Größe der Datenelemente *size* läßt sich jedoch bereits zur Compilationszeit aus dem Typbeschreiber entnehmen. Um einen möglichst effizienten Zugriff zum Element zu erreichen, wird durch ELEMENT ein spezieller Literal-Handler mit dem Namen (*+) compiliert, der die oben aufgeführte Formel berechnet. Der Parameter size wird im Fadencode übergeben.

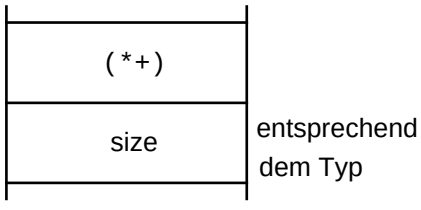

Bild 5: Fadencode für den Zugriff auf Arrayelemente

Analog compilieren auch Feldzugriffe einen speziellen Literal-Handler mit dem Namen `(+)`, der die Formel

$$'field = 'record + offset$$

berechnet. Während die Recordadresse meist erst zur Laufzeit bekannt wird, ist der Offset bereits zur Übersetzungszeit dem Feldbeschreiber zu entnehmen. Der Offset wird analog der Größe der Arrayelemente als Konstante im Fadencode übergeben. Allerdings geht die Optimierung auch hier noch etwas weiter, indem Offsets der Größe Null vollständig unterdrückt werden und Offsets einer Größe von 1 bzw. 2 Byte auf die schnellen Operationen 1+ und 2+ zurückgeführt werden. Außerdem werden nacheinander folgende Feldzugriffe bei verschachtelten Records in ein und demselben Literal untergebracht. Als Beispiel soll die Definition von `POINT+!` dienen:

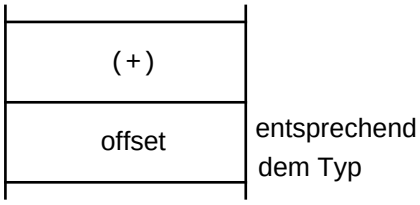

Bild 6: Fadencode für den Zugriff auf Felder

```
: POINT+! ( ps: x y 'point ===> )( verschiebt 'point um x|y )
          SWAP OVER  POINT Y +!  POINT X +! ;
```

Compiliert wird exakt derselbe Code, der auch unter Abschnitt 2.3 per Hand geschrieben wurde. Weiter unterstützt wird die Optimierung durch das Verhalten von `OBJECT:`en. Diese reagieren nicht wie einfache `CREATE`-Worte, sondern compilieren ein Literal mit ihrer eigenen Adresse. Feldzugriffe können daher auch in solch ein Literal hineinaddieren. Von der Sequenz:

```
: name .... RAHMEN RECTANGLE BOTTOMRIGHT Y .... ;
```

bleibt somit nach der Übersetzung nur noch ein Literal mit dem Wert 'pf(`RAHMEN`) + 3 übrig. Wollte man eine derartige Optimierung direkt erreichen, wäre der entstehende Quelltext kaum noch lesbar:

```
: name .... [ RAHMEN 3 + ] LITERAL .... ;
```

Im Anhang 2 sind High-Level-Definitionen für die speziellen Literal-Handler zu finden. Es versteht sich von selbst, daß diese Worte in comFORTH als Primitive implementiert wurden.

4. Pro und Contra: Strukturinformationen zur Laufzeit.

Wie bereits unter Punkt 3.3 erwähnt, wäre es besonders für die interaktive Arbeit wünschenswert, wenn Datenobjekte bei ihrem Aufruf den eigenen Typ aktivieren würden, man könnte redundante Schreibarbeit einsparen. In der aktuellen Implementation wurde jedoch darauf verzichtet.

In der vorgestellten Variante werden weder bei der Definition von Datenobjekten noch im Fadencode Typinformationen gespeichert. Das ermöglicht, daß alle Strukturbeschreiber nur temporär, d. h. während der Compilation benötigt werden. Vergleichbar einem Assembler, der ebenfalls nur während der Übersetzung von Codepassagen im Wörterbuch benötigt wird, können das Record-Paket und alle damit vereinbarten Datentypen unter comFORTH transient geladen werden. Entsprechend werden auch alle Typinformationen nur während der Übersetzung mit dem CrossCompiler benötigt. Ein damit erzeugtes Targetsystem ist frei von diesem Overhead.

Diese Eigenschaften hätten nicht erzielt werden können, wenn im Speicherbereich von Datenobjekten zusätzlich ein Verweis auf den zugehörigen Typ stehen würde, bzw. Objekte einen `DOES>`-Part aus dem Recordpaket besitzen würden.

Zweifelsohne gibt es auch Applikationen, in denen gewichtigere Gründe für die Verfügbarkeit von Typinformationen zur Laufzeit sprechen. Dazu gehören z. B. komfortable Pakete für Vektor- und Matrixalgebra (z. B. im comFORTH-Fließpunktpaket enthalten), bei der Operationen wie die Berechnung des Skalarprodukts selbst auf die Strukturinformationen zugreifen müssen.

Die dort auftretenden Probleme sprengen allerdings den hier behandelten Rahmen und benötigen eine spezielle Implementation. Programmpakete, die derartige Aufgaben lösen, lassen sich mit Hilfe der hier vorgestellten Methoden sicherlich effizient programmieren.

Literatur

[1] Pountain, Dick: Object-Oriented Forth. Academic Press, London 1987

[2] Schütz, Udo: Forth-Programmierung unter Windows™ - Ein Beispiel. Proceedings zur Tagung Forth '93, Nürnberg 1993

[3] Woitzel, Egmont: comFORTH - Das Programmierwerkzeug Forth unter SCPX. Berlin: edv-aspekte 6 (1986) Heft 4.

[4] Köller, Malte: Objektorientierte Programmierung in der Automatisierungstechnik mit optionaler Hardwareunterstützung. Proceedings zur Tagung Forth '93, Nürnberg 1993

[5] Brodie, Leo: In Forth denken. Carl Hanser Verlag 1987

Anhang 1 · comFORTH-Records für WinForth™

```
\ ================================================
\ comFORTH-Datenstrukturen für UR/FORTH für Windows
\ kompatibel zu comFORTH 2.3/3.0
\ ------------------------------------------------
\
\ Bearbeiter: E. Woitzel
\      Stand: 1. Dezember 1992
\
\ (c) FORTecH Software GmbH 1992
\ ================================================

\ --- Typzugriffsoperationen

VARIABLE KIND^  \ Zeiger auf aktiven Typ

: !KIND ( ps: 'typ ===> ) KIND^ ! ;
: @KIND ( ps: ===> 'typ ) KIND^ @ ;
: ,KIND ( ps: ===>      ) @KIND , ;

: KIND   ( ps: ===> )( ersetzt aktuellen Typ durch Untertyp )
         @KIND 2- @ !KIND ;
: @BYTES ( ps: ===> size )( Speicherbedarf des aktuellen Typs )
         @KIND @ ;


\ --- Atome
\     Aufbau:   Record: {Atom}  2 Atom Field: 'Typ
\                                     Field: Size  ;Record

: ATOM  ( ps: size ===> )( definiert und aktiviert unbenanntes Atom )
        HERE 2+ DUP , !KIND , ;

: ATOM: ( ps: size ===> )( ib: atomname )( definiert/aktiviert benanntes Atom )
        CREATE IMMEDIATE ATOM DOES>
        ( C&E: ps: ===> )( aktiviert Typ atomname )
        2+ !KIND ;


\ --- Arrays
\     Aufbau:   Record: {Array}  2 Atom Field: 'Typ
\                                       Field: Size
\                                       Field: Index  ;Record

: ARRAY   ( ps: index ===> )( definiert und aktiviert unbenanntes Array )
          ,KIND @BYTES HERE !KIND OVER * , , ;

: ARRAY:  ( ps: index ===> )( ib: arrayname )( definiert benanntes Array )
          CREATE IMMEDIATE ARRAY DOES>
          ( C&E: ps: ===> )( aktiviert Typ arrayname )
          2+ !KIND ;
```

```
: ELEMENT ( E: ps: addr i ===> addr+[i×size] )( Elementzugriff )
          \ C: dic: (*+) size ; allgemeiner Arrayzugriff
          \ C: dic: +         ; Zugriff auf Bytearrays
          KIND @BYTES STATE @
          IF    DUP 1 = IF    DROP COMPILE +
                        ELSE  COMPILE (*+) ,
                        THEN
          ELSE  * +
          THEN  ;  IMMEDIATE


\ --- Records
\     Aufbau:   Record: {Record}  2 Atom Field: 'Typ
\                                        Field: Size  ;Record

VARIABLE RECORD^  \ Zeiger auf den zuletzt definierten Record
VARIABLE RMODE    \ Zähler für die Verschachtelung der Recorddefinitionen

: RECORD  ( ps: ===> 'rec rmode )( eröffnet unbenannte Recorddefinition )
          HERE 2+ , HERE RECORD^ DUP @ -ROT !
          1 RMODE DUP @ -ROT +! 0 , ;

: RECORD: ( ps: ===> 'rec rmode ; ib: recordname)
          ( eröffnet benannte Recorddefinition )
          CREATE IMMEDIATE RECORD DOES>
          ( C&E: ps: ===> )( aktiviert Typ recordname )
          2+ !KIND ;

: ;RECORD ( ps: 'rec rmode ===> )( beendet Definition, aktiviert Vorgänger )
          -1 RMODE +!  RMODE @ ?PAIRS
          RECORD^ DUP @  -ROT ! !KIND ;


\ --- Recordfelder
\     Aufbau:   Record: {Field}  2 Atom Field: 'Record
\                                       Field: 'Typ
\                                       Field: Offset  ;Record

VARIABLE OFFSET^  \ Zeiger auf letzte Offsetfeldadresse

: ,+    ( ps: offset ===> )( compiliert Offsetaddition )
        \ dic:            ; offset=0 oder bereits vorher Literal compiliert
        \ dic: 1+         ; offset=1
        \ dic: 2+         ; offset=2
        \ dic: (+) offset ; offset>2
        HERE 2- OFFSET^ @ -
          IF    \ kein Literal unmittelbar vorher compiliert
                ?DUP
            IF    DUP 1 =
                      IF    COMPILE 1+ DROP
                      ELSE  DUP 2 =
                            IF    COMPILE 2+ DROP
                            ELSE  COMPILE (+)  HERE OFFSET^ !  ,
                            THEN
                      THEN
                THEN
          ELSE  \ unmittelbar davor Literal compiliert
                OFFSET^ @ +!
          THEN  ;
```

```
: FIELD   ( ps: ===> )( reserviert Platz für ein unbenanntes Feld )
          @BYTES RECORD^ @ +! ;

: FIELD:  ( ps: ===> ; ib: fieldname )( definiert benanntes Feld )
          CREATE IMMEDIATE RECORD^ @ DUP ,  ,KIND @ , FIELD DOES>
        ( C: ps: ===> ; dic: siehe ,+ )
        ( E: ps: addr ===> addr+offset )
        DUP @ @KIND ?PAIRS 2+ DUP @ !KIND 2+ @ ?DUP
          IF    STATE @
                IF    ,+
                ELSE  +
                THEN
          THEN  ;  IMMEDIATE


\ --- Datenobjekte

: OBJECT  ( ps: ===> 'obj )( definiert unbenanntes Datenobjekt )
          \ C: dic: (LITERAL) size DP @+!
          \ E: dic: size ALLOT
          @BYTES STATE @
          IF  [COMPILE] LITERAL COMPILE  THEN  DP
          STATE @
          IF                    COMPILE  THEN  @+!  ;  IMMEDIATE

: OBJECT: ( ps: ===> ; ib: objectname )( definiert benanntes Datenobjekt )
          STATE @ DUP
          IF                     COMPILE  THEN  CREATE
          @BYTES SWAP
          IF  [COMPILE] LITERAL  COMPILE  THEN  ALLOT  ;  IMMEDIATE


\ --- vordefinierte atomare Datentypen

1 ATOM: BYTE
2 ATOM: CELL
4 ATOM: DOUBLE

8 ATOM: FP

2 ATOM: HANDLE
4 ATOM: DHANDLE

2 ATOM: POINTER
4 ATOM: LPOINTER
```

Anhang 2 · Definition der nötigen Ergänzungen zu Forth-83

```
\ ================================================
\ notwendige Erweiterungen zu Forth-83 für die
\ Implementierung der comFORTH-Datenstrukturen
\ ------------------------------------------------
\
\ Bearbeiter: E. Woitzel
\
\ (c) FORTecH Software GmbH 1993
\ ================================================

\ --- spezielle Literal-Handler

: (+)  ( ps: addr ===> addr+off ; ip^: off )( Feldzugriff )
       R> DUP 2+ >R @ + ;

: (*+) ( ps: addr i ===> addr+[i×width] ; ip^: width )( Elementzugriff )
       R> DUP 2+ >R @ * + ;


\ --- Stapelbefehle

: -ROT ( ps: 16b1 16b2 16b3 ===> 16b3 16b1 16b2 )( linkes ROT )
       ROT ROT ;


\ --- Speicherzugriff

: @+! ( ps: w addr ===> 16b )( liest addr aus und addiert anschließend w rein )
      DUP @ -ROT +! ;


\ --- Compilersicherung

: ?PAIRS ( ps: 16b1 16b2 ===> )( Fehler, wenn 16b1<>16b2 )
         - ABORT" falsche Struktur" ;
```