

CONCURRENT EUCLID

A SHORT INTRODUCTION TO CONCURRENT EUCLID

BY R.C. HOLT

von
Nese Ates
0227289
e534

Kurze Einführung in die Programmiersprache CE

Allgemein:

Concurrent Euclid (*kurz*: CE) ist eine Programmiersprache basierend auf Pascal. Der Unterschied zu Euclid besteht darin, dass einige komplizierte Eigenschaften durch neue ersetzt wurden. Dies sollte die damalige Systemprogrammierung vereinfachen. Concurrent Euclid wurde als eine Programmiersprache für Systemsoftware mit hoher Leistung entwickelt, unter anderem für Betriebssysteme, Compiler und Embedded Mikroprozessor Systeme. Concurrent Euclid ist zu vergleichen mit Pascal, es besitzt ein Sprachkonstrukt zur Systemprogrammierung bestehend aus: „getrennte Kompilation, Variablen an absoluten Adressen, Typ Konverter, lange ganze Zahlen (Integers)“, etc.

Geschichtlicher Hintergrund:

Im Jahr 1976 wurde Euclid als Programmiersprache für die Entwicklung nachweisbarer Software entworfen.

1977 begann man die zwei Jahre dauernde Entwicklung von Toronto Euclid, welche von der Universität von Toronto (daher auch der Name) und der SHARP Association durchgeführt wurde. Diese zwei Institutionen wurden von den Verteidigungsministerien der USA und Kanada beauftragt.

Um es für die damaligen UNIX Rechner (einige davon wären zum Beispiel: PDP-11, VAX, Motorola 68000 und Motorola 6809) kompatibel zu machen wurde das „TORONTO Euclid“ experimentell von der Universität Toronto für implementierte Kompilierung bearbeitet, kurz „TUNIS“ genannt.

Dies war ausschlaggebend für die Entwicklung von „Concurrent Euclid“, welches im Jahr 1981 fertiggestellt wurde. Somit war die neue Programmiersprache CE auch in der Lage unter Unix zu laufen, da nun auch für damalige Verhältnisse ein schneller, beweglicher Compiler vorhanden war. „High Quality“ Code Generatoren waren verantwortlich für die schnellere und bessere Übersetzung von Codes Concurrent Euclid Programme, welche „concurrency“ verwenden, können auch unter sogenannten „bare machines“ laufen, die wiederum durch einen kleinen Sprachkernel unterstützt werden. Es gibt auch die Möglichkeit, dass sie im simulierten Modus als gewöhnlich auftretender „Job“ unter einem Betriebssystem laufen.

Ziele von Concurrent Euclid:

Concurrent Euclid übernimmt jene Konstrukte von Euclid, welche für die verständliche Überprüfung zuständig sind. Im allgemeinen helfen diese Konstrukte nicht nur der Überprüfung sondern sie erhöhen auch die Verständlichkeit und die Zuverlässigkeit. Manchmal machen sie es einem Programmierer aber auch schwerer ein Programm zu schreiben, da er sein Programm ausführlicher dokumentieren muss. Ein Beispiel dafür wäre, dass für jede Prozedur eine Liste von

Variablen angeführt werden muss ,welche der Prozedur zugänglich gemacht wird.

Es ist oft nicht leicht den Concurrent Euclid Compiler dazu zu bringen ein Programm zu akzeptieren da dem Concurrent Euclid viele Beschränkungen erliegen wie zum Beispiel die starke „Typen Überprüfung“.

Die Philosophie von Concurrent Euclid besteht darin so viele Fehler wie möglich in einem Programm zu finden . Dies geschieht durch die Ablehnung gefährlicher oder unwahrscheinlicher Konstrukte. Der Compiler hilft so während der Kompilierzeit, bei der Lokalisierung von „bugs“, die Zuverlässigkeit zu erhöhen und gleichzeitig die Wartung zu verringern . Dies ist auf jedenfall besser als die „bugs“ durch den im Verhältnis dazu teuren und aufwendigeren Prüfungsprozess zu finden . Seitdem Concurrent Euclid verwendet wurde um System Software herzustellen , wurden die sogenannten „escape“ Eigenschaften eingeführt ,um die Kompilierzeit Überprüfungen zu übergehen , wobei der Programmierer diese Eigenschaften auf eigene Gefahr nützte und vermutlich nur dann wenn diese wirklich gebraucht werden. Vor allem wurde Concurrent Euclid entwickelt , um effizient generierte Codes und kleine schnelle , im hohen Grad bewegliche Compiler zu gewähren . Ein gutes Beispiel dafür war der damalige PDP-11 Model 50 ,welcher einen Concurrent Euclid Compiler beinhaltete. Dieses Model konnte einen „6000“ Zeilen Durchlauf in ungefähr 8 Minuten kompilieren ,was für die damalige Zeit sehr schnell war.Da auch die Codes gut bzw. mit damals existierenden Codes gleich auf waren ,wurde Concurrent Euclid zur Implementierung von zuverlässiger hochleistungs Software wie Betriebssystemen ,Compilern und eingebetteten Mikroprozessor Software verwendet.

Vergleich mit Pascal :

Wie schon zuvor erwähnt haben wir festgestellt, dass Concurrent Euclid auf der Programmiersprache „Pascal“ basiert und borgt sich Pascal's elegante Datenstrukturen .

Unter Concurrent Euclid werden verschiedenste Eigenschaften von Pascal „gereinigt“,um leichtere Überprüfungen zu erlauben . Zum Beispiel dafür werden in Concurrent Euclid Funktionen die Seiteneffekte aufweisen mißbilligt . Man kann CE aber auch als eine Art gereinigte Version von Pascal sehen, welche Eigenschaften hinzufügt ,die von Seiteneffekten frei sind und zur Systemprogrammierung verwendet wird.

Die Haupteigenschaften ,welche Concurrent Euclid Pascal hinzufügt, sind :

1) Getrennte Kompilation;

Prozeduren Funktionen und Module können separat kompiliert und später zusammengelinkt werden. Unter Unix nützen sie den standart Linker „ld“ und können einfach mit Programmen, welche z.B.: in C geschrieben wurden, zusammengeslossen werden.

2) Module;

Ein Modul ist eine syntaktische Vereinigung von Daten mit Prozeduren/ Funktionen, welche auf diese Daten zugreifen.

3) Concurrency/ Parallelität;

Monitore und Prozesse werden unterstützt. Es gibt ein „Signal“ Statement und ein „wait“ Statement. Das „busy“ Statement erlaubt Concurrent Euclid wie eine Simulationssprache verwendet zu werden.

4) Kontrollbereich

Namen von Variablen, Typen etc. sind nicht automatisch von Bereichen übernommen worden. Import und Export Listen werden verwendet um Namensbereich zu definieren

5) Systemprogrammierende Konstrukte

Diese beinhalten Variablen an absoluten Adressen. Solchen Variablen können Vorrichtungsregister in Computern mit Gedächtnisspeicher für „Input/Output“(Ein/Ausgabe)sein.

Es gibt jedoch einige Pascal Eigenschaften ,die nicht von Concurrent Euclid unterstützt werden. Diese wären die sogenannten „echten“(Gleitkomma) und die „aufgezählten“ Typen. Concurrent Euclid erlaubt Prozeduren nicht sich in andere Prozeduren und Funktionen einzunisten.

Grunddatentypen:

Concurrent Euclid besitzt die traditionelle Grundeigenschaften von Pascal, ausser die zuvor erwähnten Grundtypen. Es gibt mehrer Wege von ganzen Zahlen ,um Hardware Daten wiederzugeben.

Diese Grundtypen sind :

Name	Value	Allocation
ShortInt	0...255	(byte)
SignedInt	-32768...32767	(16-bit)
UnsignedInt	0...65535	(16-bit)
LongInt	signed integer	(32-bit)
Boolean	false...true	(byte)
Char	a character	(byte)
AddressType	integer	(address size)
Pointer	address	(address size)

Neben diesen gibt es aber noch weitere Unterwerte wie zum Beispiel die ganzen Zahlen von 1....10

Strukturierte Datentypen

Concurrent Euclid übernimmt auch strukturierte Typen von Pascal.

Diese sind :

1) Arrays (Das sind Vektoren von Elementen)

Beispiel Array :

```
var a : array 1...1 of SignedInt
var str : packed array 1....5 of Char := 'Hello'
var matrix : array 1....5 of array 1...5 of LongInt
```

'var a' ist eine Reihe von 10 SignedInt Elementen;

'var str' ist ein String;

Concurrent Euclid übernimmt von Pascal die Definition , dass ein sogenannter „packed array“ von Charakteren mit der unteren Schranke von „1“ als String betrachtet wird.

Ein veranschlagter Wert, in diesem Fall das Wort „Hello“ wird als String betrachtet .

Concurrent Euclid stellt zwar keine „multidimensionalen arrays“ zur Verfügung erlaubt aber arrays von arrays , welche gleichwertig sind.

2) Records (Diese sind in CE gleichwertig wie in Pascal)

Beispiel Records:

- Var r :
- record
- var status : boolean
- var count : SignedInt
- end record

Dieses Beispiel erklärt r zu einem record mit den Feldern status und count .
Records in CE sind gleichwertig mit denen in Pascal.

3) set

Beispiel Set :

- var s : set of 0..2

Sets sind wesentliche bit strings

Die set Variable s ist implementiert in einen PDP-11 als bit Nummern 0,1 und 2 in ein byte. Diese bits können individuell geändert und inspiziert werden

Wissenswertes:

Alle Prozeduren und Funktionen in Concurrent Euclid sind „re-entrant“ sprich „einspringend“ ,was wiederum bedeutet ,dass sie simultan von mehr als einem Prozess ausgeführt werden können.

Concurrent Euclid definiert „input /output“ Operationen nicht, kann aber vom Programmierer implementiert werden . Ausserdem wurde ein Standart IO Packet für Concurrent Euclid programmiert ,welches dann immer per Hand ins Programm eingefügt wird.

Concurrent Euclid verwendet „Monitors“. „Monitors“ sind Sprachkonstrukte ähnlich wie Module , jedoch garantieren sie,dass nur ein Prozess zu einer Zeit in ihnen aktiv ist. Wenn ein Prozess also versuchen sollte in einen Monitor,in welchem schon ein anderer Prozess aktiv ist, einzudringen, wird dieser solange geblockt bis der „Monitor“ wieder im Leerlauf ist.

Unter Unix ist es einfach Concurrent Euclid mit anderen Programmiersprachen zu verlinken ,wie C und assembler, da die Standart „linker“ und „loader“ verwendet wurden.

Ein komplettes Programm :

```

1   var Example:
2     module
3       include '%IO1'
4     { Print characters up to a period }
5     initially
6       imports( var IO )
7       begin
8         var ch : Char
9         IO.PutString( 'Test starts$N$E' )
10        loop
11          IO.GetChar( ch )
12          IO.PutChar( ch )
13          exit when ch = $.
14        end loop
15      end { of initially }
16    end module

```

Die Zeile 1 bezeichnet den Programmnamen “ Example”

Die Zeilen 1,2 und 16 sind analog zu den „Programm“ Köpfen in Pascal

verwendungslos; Jedoch in CE werden sie gebraucht damit das Programm als „Modul“ anerkannt wird;
Zeile 3 erlaubt die Verwendung der IO Pakete;
Zeile 4 ist ein Kommentar , die Pascal Variante mit (* Print.....*) ist als alternative in CE nicht gestattet;
Zeilen 5 bis 7 und bis 15 beinhalten die Logik unseres Programms ;
Zeile 6 zeigt dass die „Initially“ Prozedur die IO Module verwendet;
Zeile 8 erklärt die charakter Variable ch, welche in Zeile 11 gelesen und in Zeile 12 geschrieben wird.
Zeile 9 druckt den Text „Test starts“ und beginnt dann wieder eine neue Zeile.
Zeilen 10 und 14 sind die Schleifen Konstrukte welche die beiliegenden Statements wiederholt ausführt; Die Schleife geht solange weiter bis ein sogenanntes „exit“ oder „return“ Statement auftritt; Wie es hier in Zeile 13 der Fall ist .

Quellenverzeichnis:

SIGPLAN Notices Volume 17 Issue 5 (May 1982)