The Multiparadigm language G

„Grundlagen wissenschaftlichen Arbeitens“ aus Programmiersprachen
Wintersemester 2005
Prof. Jens KNOOP

SILVAN YALCIN
Matr.Nr:0227217/ Kennzahl:534
silvanyalcin@hotmail.com

Abstract: *Die Sprache G wurde auf der Oregon State Universität entwickelt. Sie unterstützt die funktionale, objektorientierte, relationale, imperativ-prozedurale, und logische Programmierparadigmen. Diese Paradigmen haben sich als starke und effektive „computationale“ Modelle ergeben. Das Ziel war andere Paradigmen in die Sprache zu integrieren.*

INHALTSVERZEICHNIS

1- EINLEITUNG

Die meisten Programmiersprachen lösen Probleme mit einer einzigen Paradigma aufgrund der Beschränkungen der Art der Lösungen. Pascal verlangt z.B. Problemdomänen in Abschnitten von Kontrollstrukturen, welche den Kontrollfluss pendelt. Es ist schwer diesen verschiedenen Problemen nachzugehen.
Programmiersprachen erleichtern die Problemlösung, lösen Algorithmen auf einem Rechner und sind gleichzeitig auch eine formale Schreibweise und erleichtern somit das Modellieren von Problemdomänen.
Die Sprache G wurde auf der Oregon State Universität entwickelt. Diese Sprache verwendet die Bedeutungslehre von einem einzigen Datentyp, um die ontologischen Elemente von einzelnen Programmierparadigmen zu vereinen. Sie unterstützt die funktionale, objektorientierte, relationale, imperativ-prozedurale, und logische Programmierparadigmen. Diese Paradigmen haben sich als starke und effektive „computationale“ Modelle ergeben. Das Ziel war andere Paradigmen in die Sprache zu integrieren.[1]

2-MULTIPARADIGM FORSCHUNG

Es wurde in Integration von funktionellen und logischen Paradigmen schon viel geleistet. Man versucht diese als verständliche Elemente mit bestimmten Eigenschaften zusammenzufassen und die Eigenschaften von spezifischen Paradigmen zu verstehen.
Flavors und Common Loops kombinieren die applikativen und objektorientierten Paradigmen; SPOOL und Koschman-Events die objektorientierten und logischen, DSM die objektorientierten und relationalen.
Die Sprache **Nial** versucht eine breite multiparadigma Sprache zu produzieren. Sie hat begrenzte vernetzte verschachtelte Wege und unterstützt nur imperative, prozedurale und funktionale Angebote. Bei G sind die Datenwerte begrenzt und unterstützt die relationalen, objektorientierten und logischen Paradigmen. Eine zukünftige Version von **Nial** könnte die logischen und objektorientierten Paradigmen unterstützen.

3- DIE SPRACHE G

G ist eine interaktive, experimentale Sprache und wurde für die Verwendung in der Studie von syntaktischen und semantischen Strukturen, welche die Integration von verschiedenen Programmierparadigmen unterstützen, hergestellt. Die Hierarchie basiert auf einem Datenstream als fundamentale Datenstruktur der Sprache. Ein Datenstream ist ein Datenobjekt, dessen Werte bei der Übersetzung herauskommen. Werte verschiedener Art können als funktionelle Argumente durchgehen, und kommen als Elemente von Datenstreamen zurück. Die Sprache Icon hatte darauf einen großen Einfluss. Die Einfachheit und Generalisierung macht sie zum Grunddatentyp von multiparadigmalen Sprachen. Der G Interpreter ist in der Sprache C durchgeführt.

4- DIE GRUNDLAGEN VON G

Die arithmetischen Operatoren sind für beide **Integertypen Int** und **Real** definiert. Die Vollversion von relationalen Operatoren sind für alle Typen definiert, welche nur mit den Operatoren gleich **(=)** und nicht gleich **(!=)** funktionieren. Die logischen Operatoren **“and“**, **“or“** und **“not“** sind für alle Typen von G nicht definiert. In logischen Ausdrücken ist jeder Datenstreamwert als wahr interpretiert, wenn es nicht ein leerer Datenstream ist; sonst wird es als falsch interpretiert. Jeder relationale Operator wurde hergestellt, um auf die rechten

Operanden zurückzuführen, wenn die Beziehungen falsch oder wahr repräsentieren, ist wahr für beide der Operanden; sonst ergibt sich ein leerer Datenstream .
Nachfolgend sieht man, dass der erste Ausdruck wahr ist und der Integerwert 4 zurückkommt, während die zweite falsch ist und ein leerer Datenstream zurückkommt.

Die logischen **"and"** und die logischen **"or"** Operatoren garantieren links-nach-rechts Evaluationen von ihren Operanden und beide erstellen eine kurze Auswerteschaltung. Der Operator **and** führt das **Integer1** zurück, wenn der Operand kein leerer Datenstream ist. Die primitiven Datenstreamoperatoren **"head"** und **"tail"** erfolgen auch für einzelnen Einbauarten. Head führt den ersten Wert vom Argument **stream** unveränderten **stream**. [1]

5- DIE DREI HAUPTMERKMALE JEDES STREAMS

In der Sprache G ist jeder Wert ein Datenstreamwert. Die Arten I**nt, Real, Type, Char** und **String** werden als skalare Typen betrachtet, welche als einzelne Element-Datenströme betrachtet werden. Stream und Relation sind kombinierte Typen, die irgendwelche Anzahl von Werten in ihrer Wertesequenz haben. Die drei Attribute von jedem Wert in der Sprache G sind: eine Umwelt, eine Wertesequenz, und ein Aufzählungsprotokoll. Diese drei bieten das vereinen von semantischen Kriterien zum Verständnis bei der Datenstrukturen und Kontrollstrukturen.

6- ZUSAMMENGESETZTE STREAM-WERTE

Die einfachste Art kombinierte Datenstreams zu spezifizieren sind die Komponentenwerte von Datenstreams aufzulisten und dessen Werte in eckigen Klammern zu begrenzen; folgender Ausdruck repräsentiert ein Datenstrom, dessen Wertesequenz drei Werte beinhaltet:

[a, 5, 'c'] Variable a, Integer a, Charakter c

Kombinierte Datenströme werden auch durch die Verwendung von Rangspezifikationen repräsentiert. z.b. der Ausdruck **1..5** ist gleich dem Datenstromwert **[1, 2, 3, 4, 5]** und der Ausdruck '**a'..'d'** gleich dem Datenstromwert **['a', 'b', 'c', 'd'].**
Kombinierte Datenströme können verschachtelt sein; diese erhalten das Milieu von Datenströmen, welchen sie systemintegriert sind.

Beispiel:

→[local [x:10],x,[x-1]

[10,[9]]

→[local[x:1],x[local[x:2],x]]
[1[2]]
→

Im ersten Ausdruck hat der Außendatenstream ein lokales Milieu, welche die Variable x beinhaltet, die dem Integ.wert 10 initialisiert sind. Das innere Datenstream [x-1] ist auf dem Milieu vom äußeren Datenstream bewertet. Der vom zweiten Ausdruck produzierte Wert demonstriert, dass die Variable im äußeren Milieu durch die lokale Deklaration in verschachtelte Datenstreams maskiert werden kann.
Kombinierte Datenstreams werden auch durch die Verwendung von einem Verkettungsoperator. Dieser verkettet einfach die Wertesequenzen vom rechten Operand zum Wertesequenz des linken Operanden. Das Semikolon repräsentiert den Verkettungsoperator in G.

Beispiel:

```
→5 ; ``hi`` ;3.
 [5,``hi``,3]
→[1,2];[3,4] ; ``hi`` ; `z`
 [1,2,3,4, ``hi`` ,  'z']
 →local[x:10] ; x ; x; x.
 [10,10,10]
 →
```

Der erste Ausdruck zeigt drei Skalare Werte, welche verkettet sind, um einzelne kombinierte Datenstreams zu formen. Der zweite Ausdruck zeigt, wie Skalare und kombinierte Datenstreams verkettet werden. Der dritte Ausdruck zeigt, dass Verkettungsausdrücke auch Variablen in ihr lokales Milieu einführen können.
Relationen sind Einbauarten, welche kombinierte Datenstreams repräsentieren. Relationen sind mit den zwei Einbauoperatoren verbunden, **insert** und **delete**. Insert wird ein Wert zu einem Datenobjekt vom Typ Relation addieren, wenn dieser Wert noch kein Mitglied dieser Relation ist. Delete wird ein Wert vom Datenobjekt löschen, wenn dieser Wert ein Mitglied dieser Relation ist.

Beispiel:

```
   →rel:= #Int, Int#.
→insert(rel,[1,2]).
 [1,2]
→insert(rel,[3,4]).
 [3,4]
→rel.
 [ [1,2],[3,4] ]
 →
```

*Der erste Ausdruck – eine spezielle Syntax – wird für das Erstellen von werten vom Typ Relation verwendet. Die nächsten zwei Ausdrücke addieren zwei werte um **rel** zu verbinden.*

7- ÄNDERN DES AUFZÄHLUNG-PROTOKOLLS DER STREAMS

Kombinierte Datenströme können auch durch Veränderung von einem Datenwertstream mit einem der imperativen Präfixen **while, foreach** oder **repeat** geformt werden.

Beispiel:

```
→foreach(1..3) do [1,2] end.
  [1,2,1,2,1,2]
  →infinite:=repeat[1,2] end.
    [ ]
     →
```

Im ersten Ausdruck verändert das **foreach (1..3)** Präfixe das Aufzählungsprotokoll vom Argumentdatenstream **[1,2],** sodass es für jedes Element vom Rangwert **1..3** wiederholt wird. Im zweiten Ausdruck erlaubt die Auswertung einen unendlichen Datenstream zu einer Variable **infinite** zuzuweisen.

Beispiel:

```
→foreach(s: 'a' ;``hi`` ;3.3) do s end.
['a' , ``hi`` , 3.3]
→
```

8- UMGEBUNG ÄNDERUNG DES STREAMS

Eine imperative Präfixe kann auch das lokale Milieu vom Argumentdatenstromwert ändern.
Eine Funktion Präfixe wird als Modifikator vom Milieu ihres Datenstreamkörpers interpretiert. Die erste Art ist ein Datenstreamwert zu parametrisieren. Die Parameter bekommen eine Variable, die in das lokale Milieu vom Datenstreamwert eingeführt werden. Zweitens: die Funktion - Präfixe setzt das äußere Milieu in jene um, bei dem die Funktionsdefinition erschienen ist.

Beispiel:

```
 →func myrange (a , b ) a..b.
→myrange ( 'd' , 'f').
  ['d', 'e', 'f']
→myrange ('1' , '3').
  [1,2,3]
→
```

Der Datenstreamausdruck formt den Körper **a..b** den Namen **myrange** und zwei Variablen mit dem Namen **a** und **b** in das lokale Milieu vom Funktionskörper um.
Die Verschachtelung von Präfixen resultiert in der Formung von funktionalen Formen von G. Eine funktionale Form ist normalerweise als eine Funktion definiert, dessen Domäne einen Funktionswert inkludieren und dessen Rang als Funktionswert sein können.

Func apply (f) func (s) foreach (val:s) do f (val) end.

Die äußerste Funktionspräfixe **func apply()** assoziiert den Namen **apply** mit dem Datenstream, der den Funktionskörper darstellt und eine lokale Variable dem lokalen Milieu addiert wird.

Die Veränderung von Wertesequenzen kann durch die Verwendung von Filterausdrücken erreicht werden.

Beispiel

```
→Func apply (f) func (s) foreach (val:s) do f (val) end.
→Increment := func (x) x +1.
→apply(increment) (1..3).
 [2 ,3 ,4]
→
```

9- ÄNDERUNG DER REIHENFOLGEN DER WERTE

Der Datenstrom vom Integer ist von der Variablen a zugeordnet. Ein Filter zum Wert von a wird dann angewendet , wenn alle Werte größer als das Integer 5 sind und nicht gefiltert werden können.

Beispiel:

```
   →b=[[1,2] , [2,4] , [3,6] , [4,5] , [1,6] ].
→b:=[<=2,>=5].
[[1, 6]]
→
```

Die Beziehung von einzelnen Filterkomponenten muss für einzelne entsprechende Komponente des zugrundeliegenden Datenstreamwertes wahr sein, damit diese Werte durchgefiltert werden können.

Beispiel:

```
      →s:= [ [1,2] , [3,4] , [5,6] ].
→{[>2 , ?x]- x}
  [4,6]
→
```

Ein spezieller Präfixoperator **(?)** wird **binding** genannt und hat zwei Funktionen: es fügt eine Argumentvariable ins äußerste lokale Milieu vom Ausdruck; es bindet die Argumentsvariable der Komponenten vom Datenstreamwert .
Ein Filter ändert die Wertsequenzen von seinem Argument und verursacht auch das addieren von Variablen zum lokalen Milieu.
Ein Selektionsausdruck bietet einen Mechanismus, mit dem eine oder mehrere Alternativdatenstreame ausgewählt werden.

10- AUSWAHLSAUSDRUCK

Die Selektion bietet zwei alternative *Datenstreamergebnisse.*

Beispiel:

```
        →a:=10
→if  ( a>5) {[a,a,a,] }
else {``a was less than 5``}.
[10,10,10]
→
```

Welcher ausgewählt wird hängt vom Ergebnis, das mit dem konditionalen Ausdruck a>5 zurückgeführt wird. Jeder einzelner G Ausdruck kann als ein alternativer Datenstreamwert verwendet werden.[1]

11- VERBRAUCHERBESTIMMTE ARTEN

Benutzerdefinierte Typen kann man überall in der Hierarchie von G unter dem Effektivwert Stream definieren. Der folgende Ausdruck definiert eine neue Art mit dem Namen **Stack** und führt es in eine neue Hierarchieart als Subart vom Typ Stream ein.

```
addtype ( Stack , Stream ,Stack, local[stack:0] ).
```

Ein Beispiel von einem benutzerdefinierten Typ muss als erstes kreiert werden bevor es verwendet wird. Eine neue Instanz vom Typ **Stack** ist kreiert und der Variable **mystk** erteilt. Im zweiten Ausdruck sieht man, dass, wenn man **mystk** in den G Interpreter eingibt, der Wert vom Schnittstellenausdruck, das Integerwert 0, zurückkommt.

```
      →mystk := make (Stack).
→mystk.
 0→
```

Der erste Ausdruck addiert die Nachricht **push** und die entsprechende Methode zum Art Stack. Das zweite Element der Verkettung, also das Argument, der die Methode enthält, kehrt zurück, wenn die Methode ausgeführt wird.
Daraus wird ersichtlich, dass eine Instanz von benutzerdefinierten Arten einfach ein Datenstreamwert ist und auf dieselbe Art wie jeder andere Datenstreamwert analysiert werden kann.

12- VERSCHIEDENEN PARADIGMEN IN DER SPRACHE G

Innerhalb jeder gegebenen Ontologie existiert eine sichere Art von Beziehungen. Es ist ein Modell, das bestimmte Wege des Modellierens von Problemdomänen unterstützt.

13- IMPERATIVE-PROZEDURALE PARADIGMAS IN G

Die imperative Paradigma ist mit einer fundamentalen Verwendung von Eingaben wie Abtretung und Flusskontrolle – Strukturen gekennzeichnet. Die sequenzielle Eingaben als imperative Programmkombination darzustellen , erfordern die Produzierung der „computationalen“ Effekte.

Beispiel:

```
   func  monteCarlo ()
  local [winA:0 , winB:0 , sum , numberRolls:50 , total];
 (total := numberRolls);
while (numberRolls > 0) do
 (sum := random(6) + random(6));
if ( (sum>=7 and sum<=10) or sum =12)
{winA := winA + 1}

 else
{winB := winB +1  };
(numberRolls := numberRolls-1)
end;

write(``When the dice were rolled``  ``,numberRolls,```` times:\n``);
write (``Side A won`` , ``winA,`` ``times.\n`` );
write (Side B won`` , ``winB,`` ``times.\n`).
 montecarlo().
```

Die Funktion **monteCarlo** initialisiert Variablen durch lokale Variablendeklarationen. Die Hauptarbeit von Funktionen **monteCarlo** ist als eine **While-Schleife** ausgeführt, dass das Rollen von Würfeln in Einbaufunktionen simuliert und dann die Werte von einer lokalen Variable aktualisiert, um die Ergebnisse vom Würfel zu notieren.

14- LOGIK-PARADIGMEN IN G

Die Eigenschaften des Attributs der logischen Paradigma beinhaltet die Verarbeitung mit Relationen, Existenz von zugrundeliegenden Suchmechanismen, fortschaltender rollenbasierenden Programmstrukturen und einer logischen Variable. Die logische Variable erlaubt einen beitragenden und ausgebenden Spezifikationen von relationalen Ausdrücken. Sie unterstützt auch die Existenz von partiellen Datenstrukturen und Verbindung von Variablen in Kreuzungsbeschränkungen.

Beispiel:

```
aveTaxpayer := {
grossInc[?x , ?inc],
inc>2000 and inc <20000,
not(foreigner[x]),
not(spouse[x , ?p] and grossinc [p ,>3000]) ->}.

grossInc := {
grossSalary[?x , ?y],
investInc[x ,?z],
not(recPension[x,<5000])->[x,y+z]
}.
```

Die Steuerzahlerregel und die Bruttoeinkommenregel hängen auch von verschiedenen Sachen ab: **foreigner, spouse, recPension** und **investInc.[1]**

15- FUNKTIONALE PARADIGMEN IN G

Die funktionale Paradigma in G erlaubt Ausdrucksabhängige Semantik und zielt referentielle Transparenz. Es gibt zwei Grundmöglichkeiten: alle Unterprogramme müssen wahre Funktionen sein oder Werte einer Variable können in einem Bereich nicht verändert werden.

```
func min(s) if (not(tail(s))) {(s)}
     elif (head(s) < tail (s)) {min (head(s ); tail(s)))}
      else {min (tail(s))}.
```

Als nächstes wird die Funktion **min** untersucht, welche mit der Funktion **ham** verwendet wird. Die Funktion verändert einfach den minimalen Wert von seinem Argumentdatenstrom. Min ist eine rekursive Funktion mit einem einzigen Selektionsausdruck für den Körper.
Das relationale Paradigma basiert auf einer Welt von Relationen.

16- BEZIEHUNGS PARADIGMEN IN G

Die relationale Datenbasisprogramm ist in G leicht ausgedrückt, und zwar, es ist gegeben als Einbauart Relation, der Funktionalität von Filterausdrücken und Verbindungsoperatoren. Folgende Ausdrücke kreieren Datenwerte von der Art Relationen.

```
   s:= #Int, String, String, Int.
p:=#Int, String, String#.
sp:= #Int, Int, Int#.
```

17- OO PARADIGMEN IN G

Die objektorientierte paradigmischen Datenwerte existieren als unabhängige Objekte und sind fähig zu kommunizieren, die durch das Senden von Nachrichten erfolgen. Die Vorstellung von einer Welt von Objekten ist in einer Einbeziehungshierarchie zentral organisiert. Das Verhalten von Objekten ist durch Methoden festgelegt und mit der Klasse von Objekten verbunden. Methoden und lokale Information können von Vorgängerklassen gegeben sein. Smalltalk ist ein Beispiel für eine objektorientierte Sprache.
Unabhängige Datenobjekte sind leicht kreiert und manipulieren in G durch die Kreation benutzerdefinierter Arten. Eine benutzerdefinierte Art ist wirklich nur eine abstrakte Datenart, dass in die G Hierarchieart eingefügt ist.
Simulation ist als Subart von Stream definiert. Drei Instanzen von Variablen sind definiert als **currentTime, nextEvent** und **nextEventTime**. Hier verwenden wir die Bezeichnung Instanz Variable auf dieselbe Art, wie auch in der Sprache Smalltalk. **CurrentTime** repräsentiert die Simulation Zeit und behauptet die gegenwärtige Zeit der Simualtion. Die Variable **nextEvent** notiert die nächsten geplanten Ereignisse. Die Variable **nextEventTime,** stellt die Zeit in welcher das nächste Ereignis geplant ist, dar.

Beispiel:

```
func {Simulation} time ( ) currentTime.
func {Simulation} addEvent (event,eventTime)
 ( nextEvent:=event);
 (nextEventTime:=eventTime).

func {Simulation} proceed ( )
(currentTime:=nextEventTime );
me::processEvent(nextEvent).
```

Die Nachricht **time** geht über in gegenwärtige Zeit und wird von der Simulation der Nachricht **"addEvent",** als das nächste Ereignis und die geplante Zeit notiert. Die Nachricht **"proceed"** wird gesendet, wenn die nächste Aktion sich folgt .

Die Nachricht **init** muss gesendet werden, um ein Objekt der Art **IceCreamStore** zu initialisieren. Die Methode, die mit **init** verbunden ist, sendet die Nachricht **scheduleArrival** zum selben Empfänger.

Beispiel:

```
 func {IceCreamStore} init ( )  me:: scheduleArrival ( ).
 func {IceCreamStore} scheduleArrival
 me::addEvent(random(3) ,me::time( ) +random (5)).
 func {IceCreamStore} processEvent(event)
 write("\ncustomer received at" ,me::time( ),"\n");
(profit:=profit+event::*(0.17));
me::scheduleArrival( ).
func {IceCreamStore} reportProfits( )
write("profits are");
profit.
```

18- MISCHENDE PARADIGMEN IN DER SPRACHE IN G

Angenommen wir müssen eine Datenbasis, um Informationen über Arbeitnehmer und Manager von einer Abteilung repräsentieren zu können, kreieren. Um Informationen über die Arbeitnehmer zu repräsentieren, brauchen wir eine Relation, die **workRel** genannt wird**.** Jede Aufzeichnung von **workRel** muss den Namen des Arbeitnehmers und den Namen der Abteilung beinhalten. Um Informationen über Manager darzustellen, brauchen wir eine zweite Relation, die **managersrel** genannt wird **.** Jede Aufzeichnung muss den Namen des Managers und den Namen der Abteilung beinhalten. Die zwei Relationen **workRel** und **managersRel** müssen die einzigen aktuellen Relationen sein, damit sie in unserer Datenbasis existieren. Wenn wir eine Sicht in unserer Datenbasis brauchen, muss sie die jeweilige Informationen darüber haben, welcher Manager für welchen Arbeitnehmer zuständig ist. Jede Aufzeichnung muss den Namen des Arbeitnehmers und den Namen des Managers beinhalten. Dann müssen wir eine Operation definieren, die **add** genannt wird**.**

Die **add** Operation muss folgende Schritte durchführen:

als Beginn werden wir zwei Lösungen mit den Namen **workRel** und **managersRel** kreieren.

```
workRel := #String, String#.          --workRel[employee,department]
managesRel :=#String, String#.        –managesRel[manager,deparment]
```

Die Kommentare nach jedem Ausdruck notiert die logischen Bedeutungen von jedem Wert der Relationen. Die erforderliche Relation ist somit erstellt. Wir wälzen Modelle und beschäftigen das objektorientierte Paradigma, um die erforderliche Sicht in unsere Datenbasis einzufügen. D.h. die Sicht ist dann Benutzerdefiniert mit dem Namen **WorksForView** eingefügt.

VORTEILE

- andere Paradigmen integrierbar
- Einfachheit
- diese Sprache unterstützt mehrere Paradigmen

NACHTEILE

- Es ist schwer Problemdomänen nachzugehen

ZUSAMMENFASSUNG

Zusammenfassend kann man sagen, dass die Sprache G eine experimentelle Sprache ist, dass zur Erforschung der syntaktischen und semantischen und Durchführung der Aufgaben verwendet wird, wenn man versucht, verschiedene Programmierparadigmen zu integrieren. Es müssen noch mehr Paradigmen erstellt und durchgeführt werden.
Hier wurden Paradigmen nicht als atomare Bestandteile angesehen, wir haben sie zerlegt in ihre hauptsächlichen Eigenschaften und behandelten diese als ob sie unabhängige Bestandteile wären.

LITERATURLISTE:

1-The Multiparadigm Language G
Placer, J.R

2-http://fie.engrng.pitt.edu/fie99/papers/1350.pdf