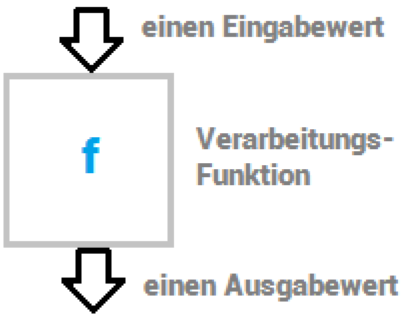
25.01.2018

Die Show der "Backus Kombinatoren"

Ich möchte über die Function-Level Programmierung schreiben, die wie ich finde eine einfache Technik ist, mit der man aus einfachen Funktionen mittels PFO's (Program-Forming-Operatoren) größere Funktionen erstellt. Die PFO's sind eigentlich nichts geringeres als Kombinatoren. - Die Definition ist: "Ein Kombinator kann nur das verwenden, was lokal gegeben ist, auf mehr hat er einfach keinen Zugriff."
In diesem Skript verwende ich Infixkombinatoren, also die aktiven Kombinatoren kommen an der zweiten Stelle vor. Für die Syntax der Kombinatoren habe ich die gebräuchlichen Formen aus dem FP-System-Dialekt "twinte" genommen.
Am Ende des Skripts ist ein Link zum Download eines Interpreters angegeben - zum Austesten der Beispiele, und um eigene Versuche bis hin zu Programmskripten zu machen.

Die Funktion

Eine Funktion hat nach den Vorschlägen von Backus genau einen Eingangswert und einen Ausgangswert.

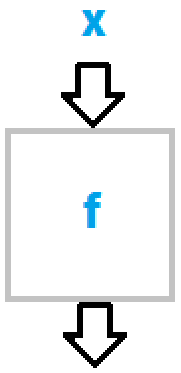


Die Applikation

Die Applikation ist eine Anwendung einer Funktion auf ein Argument.

```
f : x
```

Ergäbe folgende Versinnbildlichung.

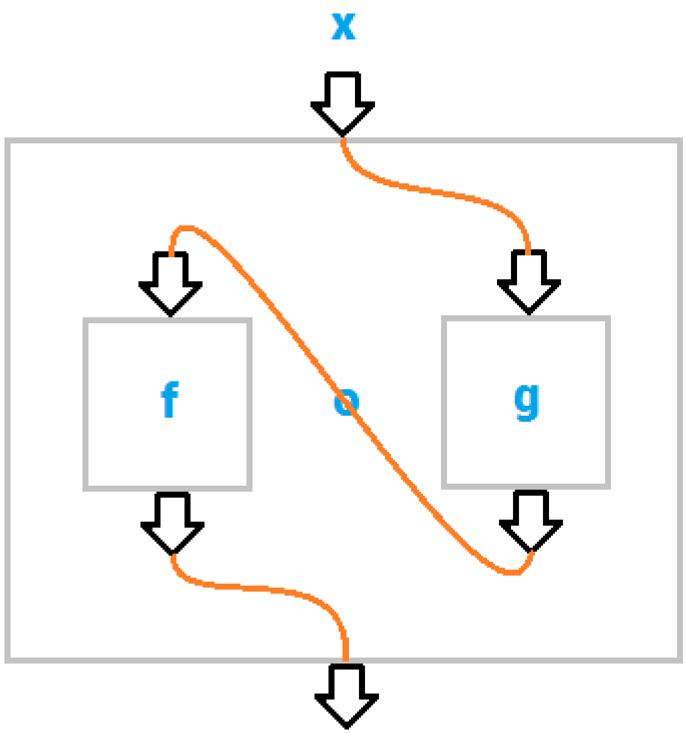


Die Komposition

Die Komposition ist eine Verkettung - also die Hintereinaderschaltung - von Funktionen; das ist sehr einfach, da jede Funktion genau einen Ausgabewert und Eingabewert hat. Durch die Komposition ist auch die variablenfreie Programmierung möglich, da die Resultate einfach durchgegeben werden. (Man kann sich das vorstellen, wie durch das o)

`(f o g) : x` *bzw.* `(f ° g) : x`

Man erhält eine seqenzielle Verarbeitung des ursprünglichen Arguments.

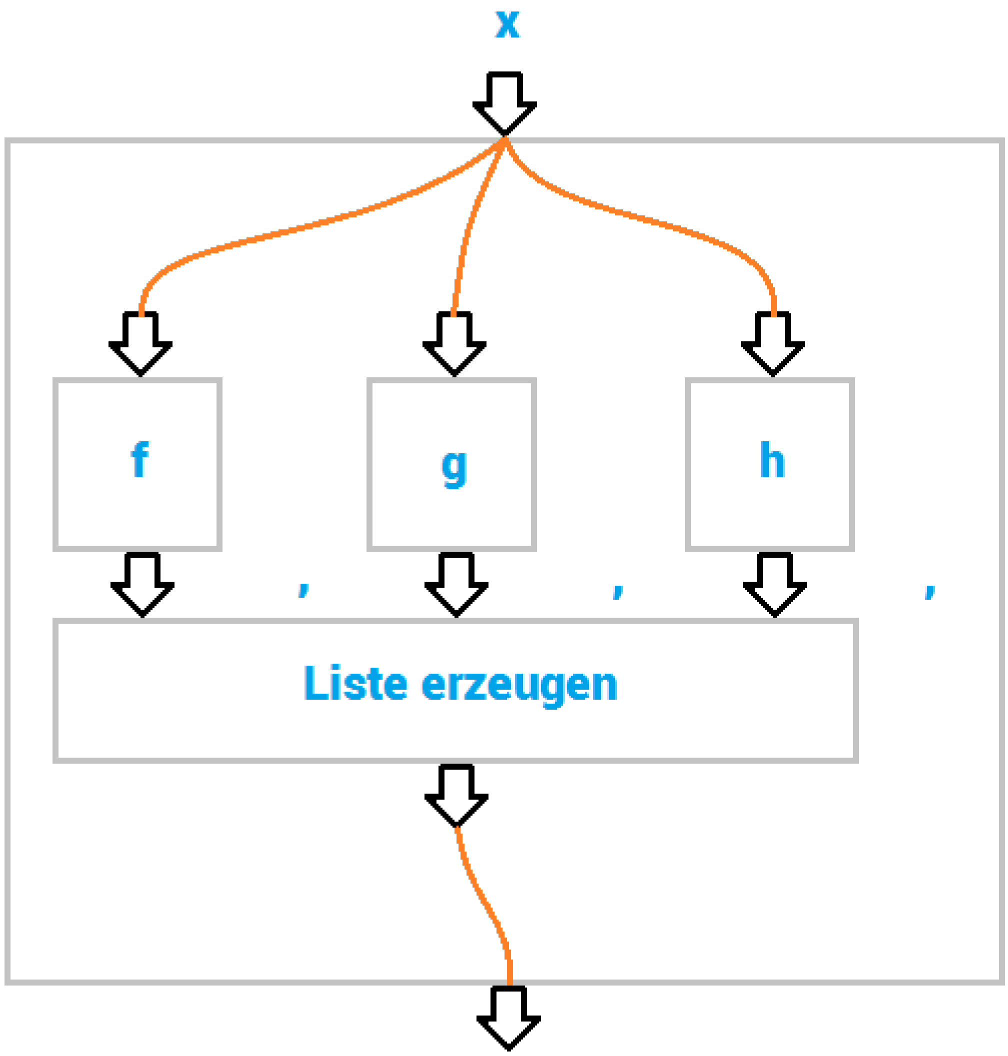


Die Konstruktion

Die Konstruktion dient zur Erzeugung einer Liste aus der parallelen Anwendung der angegebenen Funktionen auf das Argument. (Hinter dem letzten Komma kommt eigentlich ein ( ) - kann aber weggelassen werden.)

```
(f , g , h ,) : x
```

Sähe im Diagramm dann so aus.

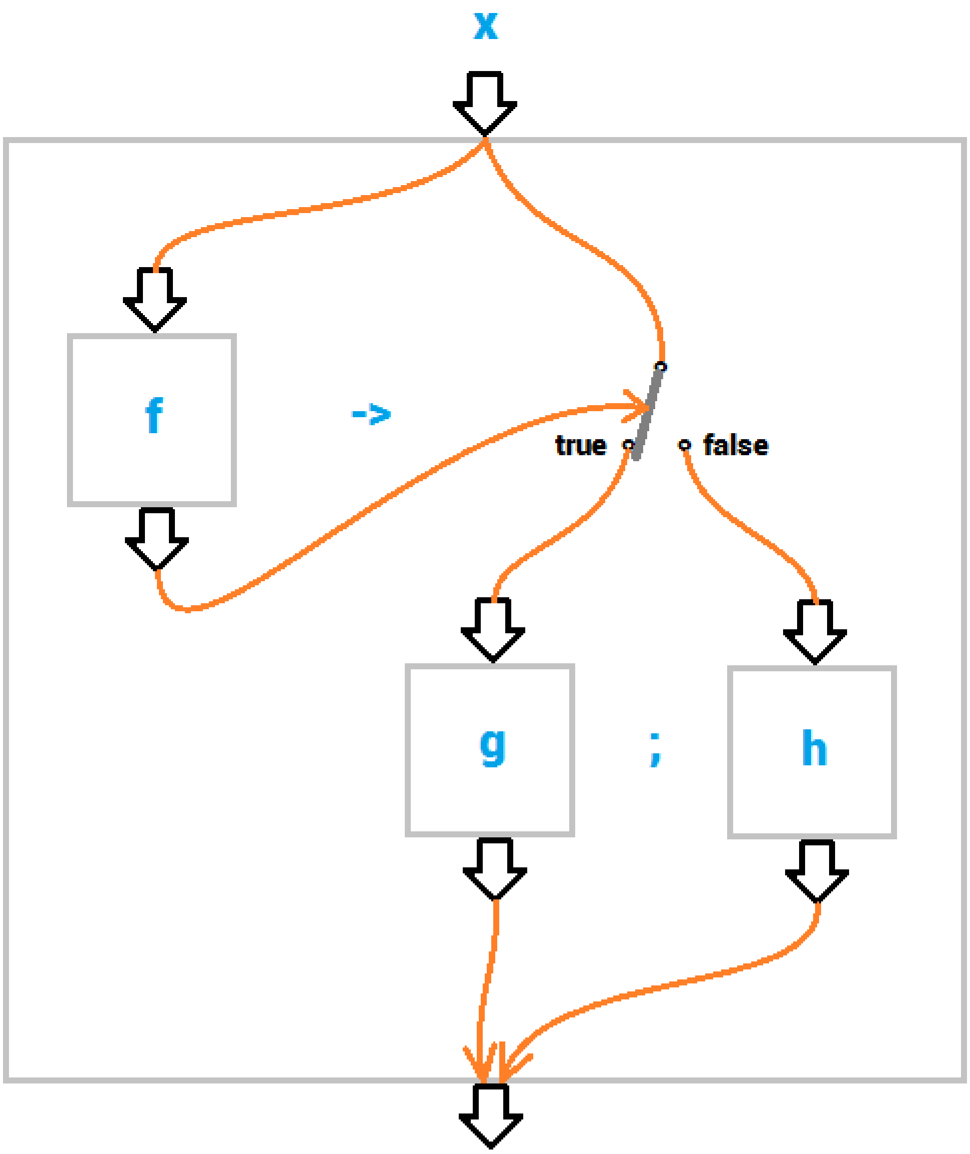


Die Kondition

Die Kondition ist eine Fallunterscheidung. Ergibt die Auswertung von f den Wert true wird die Funktion g auf das Argument angewendet sonst die Funktion h.

```
(f -> g ; h) : x
```

Ich habe das hier mal über eine Weiche realisiert.



Man kann sehr leicht damit eine "Massenabfertigung" erstellen:

```
(p0 -> e0 ; p1 -> e1 ; p2 -> e2 ; p3 -> e3 ; ... ; en) : x
```

Die Konstante

Man möchte auch mal einen bestimmten Wert haben. Da liefert die Konstantenoperation:

```
(y &) : x
```

hier den y Wert.

Die Liste

Zur Aufzählung von mehreren Werten gibt es den dynamischen Datentyp der Liste.

```
(x ; y ; z ; ... ;)
```

Der Selektor

Aus der Liste kann man einen Wert mittels eines Selektors herauspicken;
der Selektor hat die Form: [*i*]
Die Zählweise beginnt bei [*0*], was dem ersten Element aus der Liste entspricht.

```
[2] : (x ; y ; z ;)
```

ist somit das z.

Apply-to-All

Der Apply-to-All-Kombinator wendet die Funktion f auf jedes Element der Liste an.

```
(f aa) : (x ; y ; z ; ... ;)
```

Würde dann so übersetzt.

```
((f : x) , (f : y) , (f : z) , ... ,)
```

Insert-right

Das Insert-right ist eine Art "rechtsassoziative Queranwendung" der Funktion auf die Listenelemente.

```
(f \) : (x ; y ; z ; ... ;)
```

kann man sich so vorstellen:

```
(f o (x &) , (f o (y &) , (f o (z &) , (f o ...) ,) ,) ,)
```

...

Es gibt noch viele andere Kombination, zB die Whileschleife, den ee-Operator oder der Apply-Operator zur Ausführung von Funktionalergebnissen.
Was noch wichtig zu erwähnen ist, ist die globale Definition von Bezeichnern...

Die Definition

Möchte man einer neugestalteten Funktionen einen Namen geben, so gibt es die Möglichkeit einer einfachen globalen Definition:

```
bezeichner  ==  term aus funktionen und infixkombinatoren
```

Der Term offenbart den simplen Aufbau aus Funktionen und Kombinatoren/Operatoren, der nach der Regel "Rechts-vor-Links" ausgewertet wird (es gibt Ausnahmen):

```
func op func op func op func op ... ...
```

In gleicher Weise sind ja auch die Listen aufgebaut. Die Listen sind aber nicht die einzige dynamische Datenstruktur.

Dict

Eine wichtige Datenstruktur ist die Value-Key-Struktur - das Dict mit dem Aufbau:

```
(super ~ value1 key1 value2 key2 value3 key3 ... ...)
```

Es bietet eine Möglichkeit so etwas wie lokale Variablen zu verwenden.

Beispiele für einen Funktionalen Programmierstil

Prädikat zur Prüfung eines Fließkommazahlenvektors

```
isfloatvector == (([0] and [1]) \) o ((type = _float &) aa)
```

Entworfen nach dem Beispiel von John Backus aus der Turing-Award-Lecture, 5.2

```
innerproduct == (([0] + [1]) \) o (([0] * [1]) aa) o trans
```

Rekursion

Beispiel einer (einfachen) Rekursion

```
fact == (id = 0 &) -> (1 &) ; id * fact o id - 1 &
```

Dieses Beispiel kann aber auch anders dargestellt werden

```
fact == (([0] * [1]) \) o (1 &) , iota
```

Lizenz & Autor

Links & Referenzen

Download "twinte" Interpreter
https://www.heise.de/download/product/twinte

Can programming be liberated from the von Neumann style?
A functional style and its algebra of programs.
https://dl.acm.org/citation.cfm?doid=359576.359579