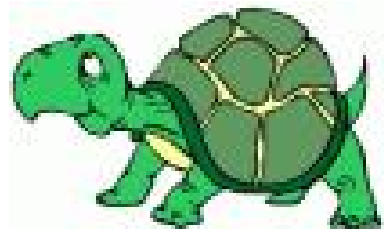


Funktionale Programmierung

– der direkte Weg vom Modell zum Programm

Peter Padawitz, TU Dortmund

19. Juni 2010

Inhalt

- Modelle
- Bestandteile eines Modells
- Konstruktor- versus destruktorbasierte Modelle
- Links

Modelle

sind **mathematische Strukturen**,

vor allem **mengentheoretische**

und, darauf aufbauend, **algebraische**,

manchmal auch mit **Ordnungsrelationen**

oder **Wahrscheinlichkeitsmaßen** versehene.

Bestandteile eines Modells

- **primitive Daten** (Zahlen, Wahrheitswerte, etc.), das sind Objekte, die im Modell vorkommen, aber dort selbst nicht modelliert werden,
- eine **Trägermenge** der Objekte, die das Modell beschreibt,
- **Konstruktoren**, das sind Funktionen, mit deren Hilfe die Objekte des Modells **induktiv** aus primitiven Daten zusammengesetzt werden,
- **Destruktoren**, das sind Funktionen, die ein Objekt in seine Bestandteile zerlegen, in einen neuen **Zustand** überführen (OO-Sprech: **Methoden**) oder primitive Daten berechnen, die Eigenschaften des Objekts (Farbe, Größe, etc.) wiedergeben (OO-Sprech: **Attribute**).

Konstruktor- versus destruktorbasierte Modelle

Die Trägermenge eines

maximalen konstruktorbasierten Modells

besteht aus den Ausdrücken (**Termen**), die man aus den Konstruktoren und den primitiven Daten des Modells formen kann.

Weitere konstruktorbasierte Modelle entstehen durch **Abstraktion**, mathematisch: **Quotientenbildung**,

d.h. Gleichsetzung von Termen, die dasselbe Objekt repräsentieren, z.B. von 5+4, 3*3 und 9.

In der funktionalen Programmiersprache **Haskell** können konstruktorbasierte Modelle mit Hilfe des `data`-Konstrukts implementiert werden.

Beispiel

Ein Teil des Datentyps von *Expander2* zur Beschreibung zweidimensionaler geometrischer Figuren:

```
data Widget = Arc State ArcStyleType Float Float |
              Circ State Float |
              File String |
              Oval State Float Float |
              Path0 Color Int [Point] |
              Path State [Point] |
              Poly State [Float] Float |
              Rect State Float Float |
              Text State [String] |
              Tria State Float |
              Turtle State [TurtleAct]
```

Hinter = werden die Konstruktoren mit den Typen ihrer jeweiligen Argumente – getrennt durch das oder-Symbol | – aufgelistet.

`[a]` bezeichnet den Typ der Folgen von Elementen des Typs `a`.

Weitere Hilfstypen von `Widget`:

```
type Point = (Float,Float)
type State = (Point,Float,Color,Int)
```

Auch diese Typen sind mit Konstruktoren aufgebaut, z.B. `Point` mit der Bildung von Paaren reeller Zahlen.

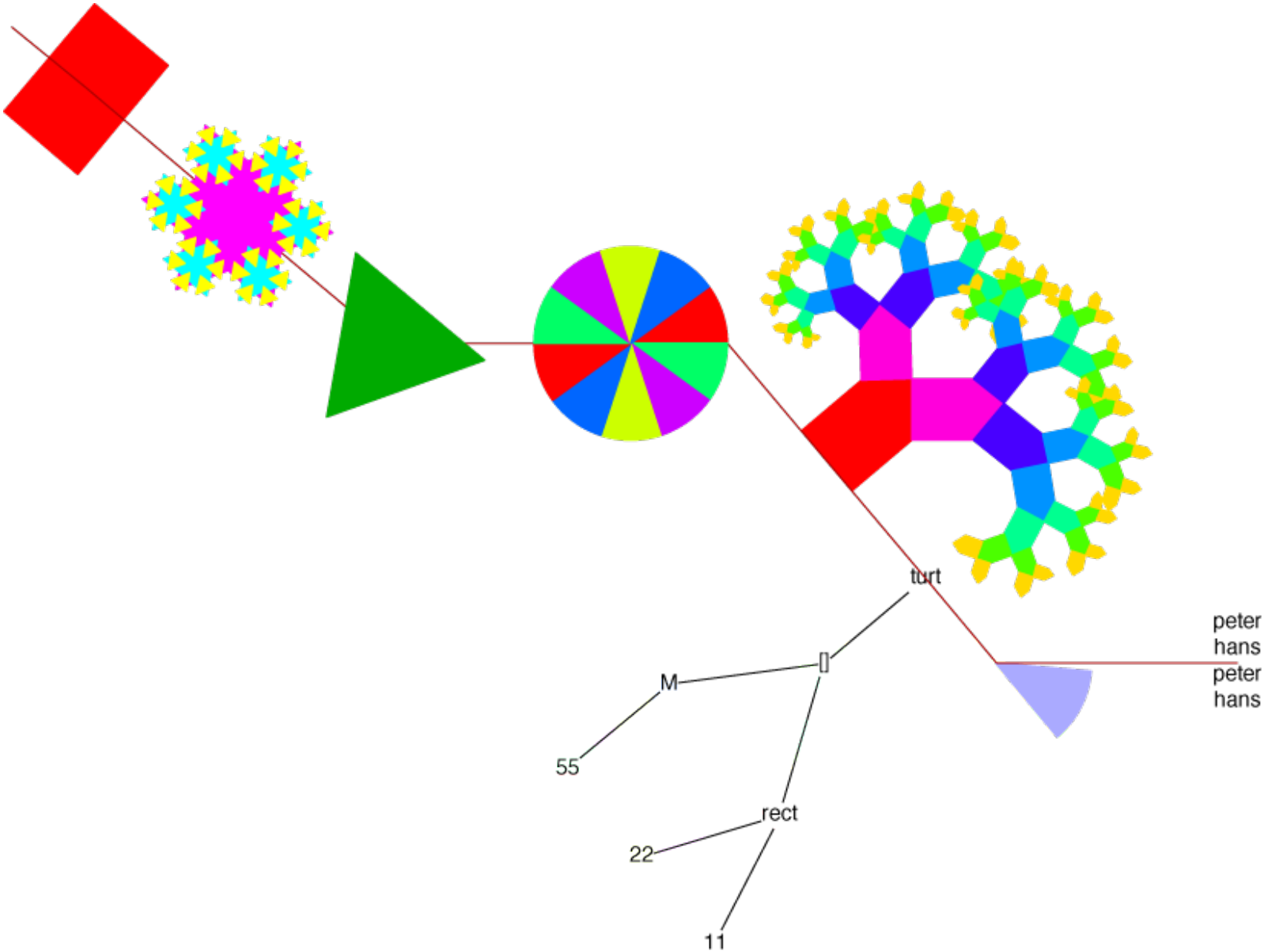
Die vier Komponenten eines **Zustands** vom Typ `State` sind der Mittelpunkt, die Orientierung (in Grad), der reine Farbwert und die (zum Farbwert hinzuaddierte) Helligkeit der jeweiligen Figur.

`Widget` ist **wechselseitig rekursiv** mit `[TurtleAct]` definiert:

```
data TurtleAct = Move Float | Jump Float |
                 Turn Float | Widg Widget |
                 Open Color | Close
```

Ein Objekt des Typs `[TurtleAct]` ist also eine Folge von Anweisungen (an eine Schildkröte), sich auf der Fläche weiter zu bewegen (`Move`), weiter zu springen (`Jump`), ihre Orientierung zu ändern (`Turn`) oder ein Widget zu zeichnen (`Widg`).

Trifft die Schildkröte auf den Befehl `Open color`, dann merkt sie sich den Punkt p, an dem sie gerade steht. Trifft sie später auf `Close`, dann zeichnet sie den Weg, den sie von p aus – unter Auslassung der Sprünge – zurückgelegt hat, in der Farbe `color` und springt anschließend zu p zurück.


turt
[]
M
55
rect
22
11
peter
hans
peter
hans

Der **Pythagoreische Baum** wurde **rekursiv** erzeugt:

```
pytree :: Int -> Color -> [TurtleAct]
pytree n = f n
 where f 0 c = []
       f i c = growActs (bud 3 c)
                        [[],acts,acts,[]]
               where acts = f (i-1) (nextColor n c)
```

n ist die Höhe des Baumes.

`bud 3 c` ist der c-farbige Stamm des Baumes.

`growActs` lässt zwei Zweige aus den Schrägen des Stammes herauswachsen.

`nextColor n c` berechnet die in einem Farbkreis von n äquidistanten Farben auf c folgende Farbe.

Die durch ein Widget der Form

```
Turtle (p,a,c,i) acts
```

beschriebene Figur besteht aus den Wegen und Widgets, die entstehen, wenn die Schildkröte die Anweisungsfolge `acts` im Zustand `(p,a,c,i)` ausführt.

Die Anweisungsfolge beschreibt die Figur zwar nur indirekt, aber dennoch eindeutig.

Eine solche Beschreibung ist ein

destruktorbasiertes oder **zustandsorientiertes Modell**.

Die Trägermenge eines

minimalen destruktorbasierten Modells

besteht aus `Verhaltensbeschreibungen`, die man durch – wenn möglich, wiederholte – Anwendung der Destruktoren erhält.

Aus Destruktoren bestehende Terme beschreiben nicht die Objekte selbst, sondern bilden "Messinstrumente" und liefern Ergebnisse in primitiven "sichtbaren" Datenbereichen. Aus den Ergebnissen wird auf bestimmte Eigenschaften des Objekts geschlossen.

Weitere destruktorbasierte Modelle entstehen durch **Restriktion**, mathematisch: **Unterstrukturbildung**,

d.h. durch Beschränkung auf Objekte, deren Verhalten eine bestimmte, gegenüber der Anwendung von Destruktoren **invariante** Eigenschaft aufweist.

Z.B. liefert die Menge der Bilder, die eine Schildkröte erstellen kann, wenn sie in einem festen Anfangszustand startet, eine Invariante.

In **Haskell** können destruktorbasierte Modelle ebenfalls mit Hilfe des `data`-Konstrukts implementiert werden, allerdings angereichert um Destruktoren.

Beispiel

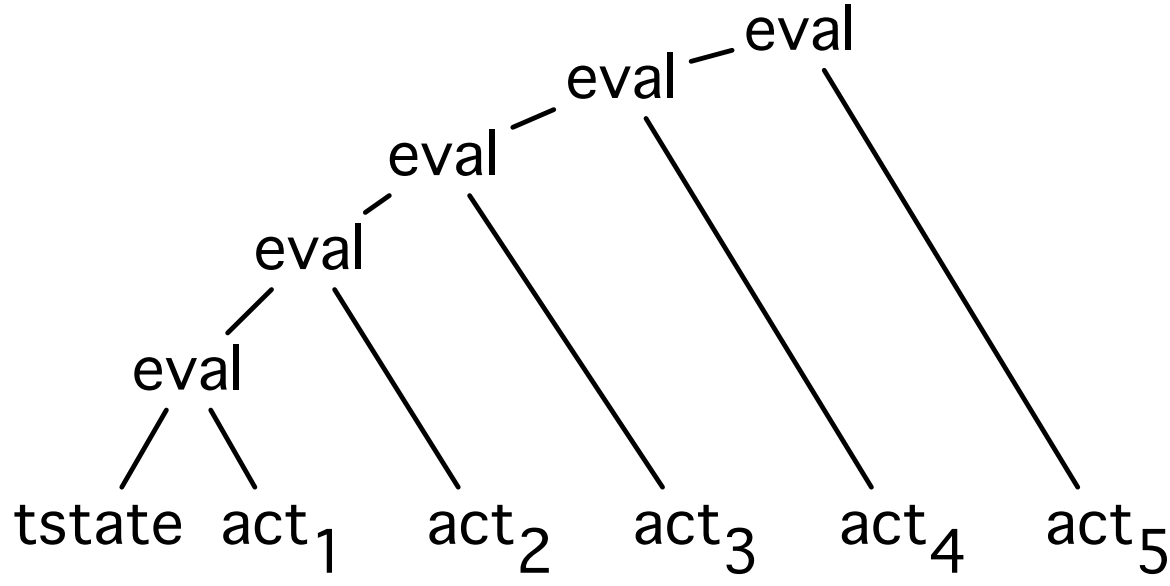
Der folgende **Automat** simuliert die Ausführung von Anweisungen an die Schildkröte:

```
data TurtleAut state =
     C {move,jump,turn :: Float -> state -> state,
        widg  :: Widget -> state -> state,
        open  :: Color -> state -> state,
        close :: state -> state}

type TState = ([Widget],[State,[Point])])

turtleAut :: State -> TurtleAut TState
```

```
turtleAut (p,a,c,i) =
 C {move d (pict,(p,a,c,ps):s) =
           (pict,(q,a,c,ps++[q]):s)
           where q = successor p d a
    jump d (pict,(p,a,c,ps):s) =
           (pict++[Path0 c i ps],(q,a,c,ps):s)
           where q = successor p d a
    turn b (pict,(p,a,c,ps):s) =
           (pict,(p,a+b,c,ps):s)
    widg w (pict,(p,a,c,ps):s) =
           (pict++[moveTurn p a w],(p,a,c,ps):s),
    open d (pict,(p,a,c,ps):s) =
           (pict,(p,a,d,[p]):(p,a,c,ps):s),
    close (pict,(p,a,c,ps):s) =
          (pict++[Path0 c i ps],s)}
```

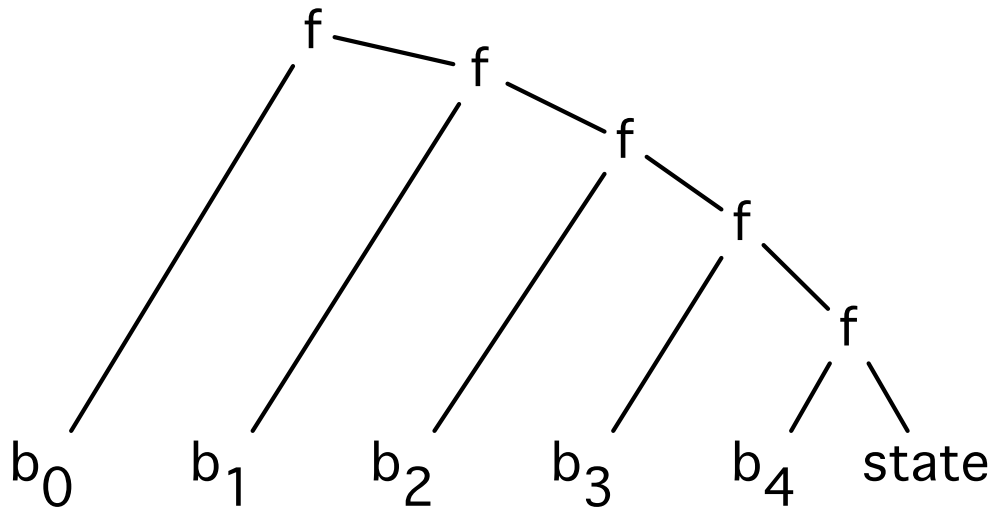
Der Automat wird in die Funktion `makePict` eingebunden, die alle Widgets in `Oval`-, `Path0`- und `Text`-Widgets übersetzt:

```
makePict :: Widget -> [Widget]
makePict (Turtle state acts) =
                          fst (foldl eval tstate acts)
 where tstate = ([],[(state,[p])])
       aut = turtleAut state
       eval :: TState -> TurtleAct -> TState
       eval state act =
            (case act of Move d -> move aut d
                         Jump d -> jump aut d
                         Turn a -> turn aut a
                         Widg w -> widg aut w
                         Open c -> open aut c
                         Close -> close aut) state
```



Faltung der Aktionsfolge $[act_1, \ldots, act_5]$ von links her

Faltung einer Folge $[b_0, b_1, \ldots, b_n]$ von rechts her




```
foldr :: (b -> state -> state) ->
          state -> [b] -> state
foldr f state []     = state
foldr f state (b:bs) = f b (foldr f state bs)

sum     = foldr (+) 0       product = foldr (*) 1
```

Beispiel

Horner-Schema (effiziente Berechnung von Polynomwerten)

Anstelle des Terms

$$b_0 + b_1 * x + b_2 * x^2 + \cdots + b_{n-1} * x^{n-1} + b_n * x^n$$

wird der Term

$$b_0 + (b_1 + (b_2 + \cdots + (b_{n-1} + b_n * x) * x \ldots) * x) * x$$

ausgewertet.

```
horner :: [Float] -> Float -> Float
horner bs x = foldr f (last bs) (init bs)
                where f b state = b+state*x
```

Sowohl die Zustände als auch die Folgenelemente sind hier reelle Zahlen vom Typ `Float`.

Links

http://haskell.org

http://tryhaskell.org

http://ls1-www.cs.tu-dortmund.de/~padawitz