Shapes

— ett hyfsat funktionellt ritspråk

Henrik Tidefelt

LiTH

20 september 2007

Mål

Med den här UppLYSningen hoppas jag

- Att ni ska få ett hum om vad Shapes är.
- Få höra era invändningar mot designen som den ser ut idag.
- Lyckas hitta någon testpilot.
- Väcka intresse för utvecklingssamarbete.

Plan

De stora inslagen idag är:

- Beskriva hur språkets struktur ser ut idag.
- Visa lite av de funktioner som kärnan erbjuder.
- Diskutera intressanta utmaningar för framtiden.

Introduktion

Historia

- Hösten 2003: Första kontakt med MetaPost.
- Sommaren 2004: Toolbox för plottning i Matlab tar form.
- Hösten 2004: Börjar undersöka möjligheten att ersätta MetaPost.
- September 2005: Shapes, då kallat *MetaPDF*, versionshanteras.
- Januari 2007: Shapes, då kallat *Drool*, har använts till stort antal figurer i en bok.
- April 2007: Kontrollerade tillstånd.
- September 2007: Språket heter *Shapes*, och presenteras för första gången.

Rötter

Shapes har sina rötter i många av de språk jag varit i kontakt med:

- MetaPost (en omarbetning av Knuths MetaFont) — Shapes kom till när jag ledsnade på MetaPost.
- Scheme — syntax och funktions-begreppet.
- Haskell — för sina rena ideal.
- C++ — utmatningssyntaxen.

Alternativ

Några andra ritspråk som finns och/eller används idag:

- MetaPost
- Asymptote
- PGF och TikZ
- Haskell PDF

Varför Shapes?

Givet utbudet av alternativa rit-språk, varför utveckla ett till? Här är några skäl:

- Inte funktionellt orienterade (alla utom Haskell PDF).
- Dålig beräkningskapacitet (MetaPost och PDF/TikZ).
- Saknar domän-specifik syntax (Haskell PDF).
- Inte publicerade när Shapes påbörjades (Asymptote och Haskell PDF).

Hello, shaper!

```
•page << [stroke (0cm,0cm)--(1cm,1cm)]

•page << stroke [] ((0cm,0cm)--(1cm,1cm))

[(\ •dst pth .> { •dst << stroke [] pth } )
  •page  (0cm,0cm)--(1cm,1cm)]
```

Språkets struktur

Exempel på enkla typer

- Flyttal: `14`, `14.5`, `  1`
- Heltal: `'5`, `'~12`, `'0xFF`
- Längd: `7cm`, `~3mm`, `72bp`
- Sträng (mer detaljer senare): `` `Hej!´ ``
- Symbol: `'left`

Lexikala bindningar

Lexikala bindningar fungerar som i Scheme, men kan inte bindas om.

```
a: 42
```

Räckvidden (eng: *scope*) är begränsad till en kod-klammer (eng: *code bracket*):

```
{
  a: 42
  •stdout << a
}
```

Lexikala bindningar — detaljer

- Högerledet evalueras i samma scope som bindningen tillhör. (Jämför `letrec` i Scheme.)

```
odd: \ n .> [if n = '0 false [even n - '1]]
even: \ n .> [if n = '0 true [odd n - '1]]
```

- Skuggade bindningar kan nås:

```
a: ../a + 7
```

Dynamiska bindningar

Dynamisk bindning infördes som ett sätt att undvika den imperativa spagetti-struktur som ett skrivbart *graphics state* lätt kan leda till.

```
@width:4bp | [stroke mypath]
```

- Dynamiska variabler inleds med `@`.
- Den dynamiska variabeln tillsammans med ett värde blir ett nytt värde som representerar en potentiell dynamisk bindning.
- Bindningsvärden kan kombineras:
  `@width:4bp & @dash:[dashpattern 1cm 4mm]`
- Dynamiska bindningar sätts i scope med en “pipe”.
- En dynamisk variabel har ett filter och ett skönsvärde (eng: *default value*).

Dynamiska värden

En dynamisk variabel kan bindas till ett *dynamiskt värde*.

- Ser ut så här:
  ```
  @bigmargin: dynamic 1.3 * @smallmargin
  ```
- Undviker behovet av att binda alla dynamiska variabler till argumentlösa funktioner.

Funktionsdefinitioner

Exempel:

```
\ x y .> x * x + y * y
```

- Argumentens namn är en del av funktionens signatur.
- En slask (eng: *sink*) kan ta hand om ytterligare argument.
  ```
  \ x y <> rest .> x + y + (foo [] <>rest)
  ```
- Vilka argument som helst kan få skönsvärden:
  ```
  \ x:3 y z:2 .> x + y + z
  ```

Enkla funktionsanrop

Ett enkelt funktionsanrop kan ange argument både genom ordning och genom namn.

```
hypot: \ x y .> [sqrt x*x + y*y]
```

- Ordnade argument: se anrop till `sqrt`.
- Namngivna argument: `[hypot y:3 x:4]`.
- Blandat: Ordnade argument måste komma först.
- Endast ett argument: `square [] 3` eller `square [] x:3`

Märk att namngivna argument kan inte ändra betydelsen av ordnade argument!

Snitt

Scheme: *evaluated cuts*

```
[hypot 3 ...]
[hypot y:4 ...]
```

- Ordnade argument blir helt osynliga i den nya funktionen.
- Namngivna argument (er)sätter skönsvärden.
- Endast ett argument: `hypot [...] 3` eller `hypot [...] y:4`

Scenario

Utgångsbudet för att skapa en komplex bild i ett funktionellt språk är att skriva ett stort uttryck som sätter ihop de ingående delarna till en helhet.

- Det skapar lätt en krystad struktur i koden.
- Det stämmer illa med *painter's model* och hur de flesta av oss tänker på att skapa en bild.
- Intuitionen är snarare *imperativ*!
- . . . men variabler kunde ju inte bindas om. . .

Kontrollerade tillstånd

Shapes erbjuder *kontrollerade tillstånd* (eng: *limited states*) för att tillåta en viss grad av imperativ stil.

- Kontrollerade tillstånd binds till variabler som inleds med • eller #: •page
- De kan skickas *by reference* till "funktioner", men kan inte returneras.
- Några finns globalt, andra kan skapas genom avknoppning från speciella *värden*.

Grundläggande operationer

Det finns tre huvudsakliga operationer på ett tillstånd:

- Lägga till (eng: *tack on*): `•dst << pic`
- Frysa (eng: *freeze*), erhålla slutgiltigt värde, och förstöra: `•dst;`
- Tjuvtitta (eng: *peek*), bör ge samma resultat som att frysa, men förstör inte tillståndet: `(•dst)`

Exempel 1

```
mark: \ •dst .>
{
  •dst << [stroke (0cm,0cm)--(1cm,1cm)]
}

[mark •page]
```

Exempel 2

```
•page <<
{
  •dst: newGroup2D
  •dst << house << car
    << dog
  •dst << cow
  •dst;
}

•page << ( newGroup2D << sun << clouds )
```

Mer om tillstånd

- Inbyggda tillstånds-avknoppare:
  `newIgnore`, `newGroup2D`, `newGroup3D`, `newString`, `newTimer`, `newText`, `newFont`, `newZBuf`, `newZSorter`
- Högre-nivå-konstruktorer:
  `newRandom`, `devRandom`
- Globalt definierade (interaktion med omvärlden) tillstånd:
  `•page`, `•catalog`, `•stdout`, `•stderr`, `•randomdevice`, `•time`, `•ignore`

"Funktioner" och tillstånd...

- Tillstånd kan skickas både ordnat och per namn, precis som argument.
- En funktion kommer inte åt tillstånd utanför sin kropp.
- Anrop bör ses som makro-expansion snarare än funktions-anrop.
- En kod-klammer med tillstånd blir ett *uttryck*; utifrån syns det inte att tillstånd används för att konstruera klammerns värde.
- Ett tillstånd kan endast frysas i sin egen kod-klammer, så en funktion kan inte frysa tillstånd som den tar emot.
- Tillstånds-parametrar kan inte ges skönsvärden.
- Ett rent funktionsanrop kan inte påverka några tillstånd!

Procedurer

En *procedur* kan påverka tillstånd utanför sin egen kropp.

- Kan vara praktiskt ibland.
- Svårt att analysera.
- Att användas under kontrollerade former.
- Skapas med egen syntax:
  ```
  proc: \ arg1 arg2 .> ! body
  ```
- Anropas med egen syntax (annars skulle det se ut som ett rent uttryck!):
  ```
  [!proc arg1 arg2]
  ```

Utmaning

Dynamiska tillstånd. . .

Structures

Lat evaluering

Continuation passing style

Funktioner i kärnan

Kurv-konstruktion

Travare (*sliders*)

- Skapa utifrån kurvlängd, kurvtid, eller andra beräkningar
- Punktvisa egenskaper för kurvan
- Del-kurvor

Grundläggande ritning

Travare

2D

3D

PDF

LaTeX och strängar

Utmaningar för framtiden

Trixelering

Kompilera funktioner till PDF

Mer grund-struktur

- Namespaces/packages
- Användar-typer

Sammanfattning

Sammanfattning

- Shapes är...

Slut.