3 DSLs in Haskell

Implementation of DSLs

Shallow DSLs

Deep DSLs
Types of DSLs

- **external DSL**: Stand-alone, including parser
- **internal DSL**: Embedded in a host language
- **domain-specific embedded language (DSEL)**
  - **shallow embedding**: implicit, through constructors of semantic domain + functions
  - **deep embedding**: explicit, through a separate data type

Implementation of embedded DSL

Representation of abstract syntax?
DSLs & Denotational Semantics

Semantic Language

\[ S \oplus [\cdot] : S \rightarrow D \]

(General-Purpose) PL

Turing-complete computation

Other computation

Non-computation

sem :: S → T

sem :: Shape → Image
sem :: Move → Pos
Move → Trip

deep embedding

data types
3 DSLs in Haskell

- Implementation of DSLs
  - Shallow DSELs
  - Deep DSELs
A Shallow DSL

DSL for describing geometric figures on a 2D grid

type Point = (Int,Int)

data Pic = Line Point Point
| Circle Point Int
| Pic :+: Pic

ctr :: Point
ctr = (3,2)

pic :: Pic
pic = Line (1,0) (5,3) :+: Circle ctr 4 :+: Circle ctr 5

Semantic Domain

Constructors provide (basic) syntax

Semantic Value
Adding Syntax

```haskell
type Point = (Int, Int)

data Pic = Line Point Point |
          Circle Point Int |
          Pic :+: Pic
```

Domain is limited.
E.g. no triangles.

```
triangle :: Point → Int → Int → Pic
triangle p@(x,y) w h = Line p (x,y+h) :+: Line p (x+w,y) :+: Line (x,y+h) (x+w,y)
```

Functions extend (basic) syntax

```
pic :+: triangle ctr 2 1
```
Syntactic Sugar

Syntax is too terse

Line (1,0) ctr
triangle ctr 2 1

Adding key words

line from (1,0) to ctr
triangle at ctr width 2 height 1

Type KW = String

[from, to, at, width, height] = ["from", "to", "at", "width", "height"]

Add key-word parameters

line :: KW → Point → KW → Point → Pic
line "from" p "to" q = Line p q
line "to" p "from" q = Line q p
line _ _ _ _ = error "Incorrect keyword!"

"Compile" to semantic values
Syntax Checking Through Pattern Matching

```haskell
line :: KW → Point → KW → Point → Pic
line "from" p "to" q = Line p q
line "to" p "from" q = Line q p
line _ _ _ _ = error "Incorrect keyword!"
```

Functions perform 2 tasks:
1. syntax checking (LHS)
2. semantics/valuation (RHS)

```haskell
> line from (1,1) to ctr
Line (1,1) (3,2)

> line to ctr to (1,1)
*** Exception: Incorrect keyword!
```
Design Guidelines for Semantic Domains

- **Semantic domain needs** basic objects 
  & composition operator(s)

- **Keep semantic domain small**

- **Try to realize extensions through new syntax (i.e. functions)**

```haskell
type Point = (Int, Int)
data Pic = Line Point Point 
     | Circle Point Int 
     | Pic :+: Pic
```
Observations About Shallow DSLs

- Major component & primary focus: **Semantic Domain**
- Constructors of Semantic Domain constitute core syntax
- No separate representation of syntax
- Function definitions extend core syntax
- Function definitions comprise syntax & semantics
- Semantics is spread across several function definitions
3 DSLs in Haskell

Implementation of DSLs
  Shallow DSLs
  Deep DSLs
Deep DSLs

Separate data type for syntax

Abstract Syntax

data Cmd = Line’ Point Point
| Tri Point Int Int
| Circle’ Point Int
| Seq Cmd Cmd

type Point = (Int,Int)
data Pic = Line Point Point
| Circle Point Int
| Pic :+: Pic

Semantic Function

sem :: Cmd → Pic
sem (Line’ p1 p2) = Line p1 p2
sem (Circle’ p r) = Circle p r
sem (Tri p@(x,y) w h) = Line p (x,y+h) :+: Line p (x+w,h) :+: Line (x,y+h) (x+w,y)
sem (Seq c c’) = sem c :+: sem c’

Map syntactic operators to semantic functions

DSLs in Haskell
Deep DSLs

Why “go” deep?

\[
\begin{align*}
\text{sem} & : \text{Cmd} \rightarrow \text{Pic} \\
\text{sem} (\text{Line' } p1 \ p2) & = \text{Line } p1 \ p2 \\
\text{sem} (\text{Circle' } p \ r) & = \text{Circle } p \ r \\
\text{sem} (\text{Tri } p@(x, y) \ w \ h) & = \\
& \text{Line } p \ (x, y+h) :+:
\text{Line } p \ (x+w, h) :+:
\text{Line } (x, y+h) \ (x+w, y) \\
\text{sem} (\text{Seq } d \ d') & = \text{sem } d :+: \text{sem } d'
\end{align*}
\]

\[
\begin{align*}
\text{triangle} & : \text{Point} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Pic} \\
\text{triangle } p@(x, y) \ w \ h & = \\
& \text{Line } p \ (x, y+h) :+:
\text{Line } p \ (x+w, h) :+:
\text{Line } (x, y+h) \ (x+w, y)
\end{align*}
\]
Shallow vs. Deep Embedding

**Advantages of shallow embeddings**
- Simple & fast
- No redundancy in representation
- Easy extensibility

**Advantages of deep embeddings**
- Multiple interpretations
- Enables analyses
- Enables transformations

**Direct manipulation of the domain**
Data type for semantic domain

**Indirect manipulation of the domain**
Additional data type for abstract syntax
“Shallowing” a DSL

(1) Split `sem` function into separate functions

(2) Eliminate syntax data type

Abstract Syntax

```
data Shape = X
  | TD Shape Shape
  | LR Shape Shape
```

Semantic Domain

```
type Image = [(Int,Int)]
```

Need 3 functions

```
sem :: Shape -> Image
sem X = [(1,1)]
sem (TD s1 s2) = adjustY (maxY p2) (sem s1) ++ p2
  where p2 = sem s2
sem (LR s1 s2) = p1 ++ adjustX (maxX p1) (sem s2)
  where p2 = sem s2
```
“Shallowing” a DSL

\begin{verbatim}
data Shape = X  
    | TD Shape Shape  
    | LR Shape Shape

type Image = [(Int,Int)]

sem :: Shape → Image
sem X = [(1,1)]
sem (TD s1 s2) = adjustY (maxY p2) (sem s1) ++ p2
    where p2 = sem s2
sem (LR s1 s2) = p1 ++ adjustX (maxX p1) (sem s2)
    where p2 = sem s2

x :: Image
x = [(1,1)]

td :: Image → Image → Image
td i1 i2 = adjustY (maxY i2) i1 ++ i2
\end{verbatim}
Exercises

(1) Turn the LR constructor into a function \( \text{lr} \)

(2) “Shallow” the Move language
“Deepening” a DSL

1. Introduce syntax data type
2. Combine separate functions into `sem` functions
“Deepening” a DSL

Abstract Syntax

```
data Cmd = Line' Point Point  
     | Tri Point Int Int  
     | ...  
```

Semantic Domain

```
data Pic = Line Point Point  
     | Circle Point Int  
     | Pic :+: Pic  
```

Semantic Function

```
sem :: Cmd -> Pic  
sem (Line' p1 p2)  = Line p1 p2  
sem (Tri p@(x,y) w h)  = Line p (x,y+h) :+:  
                       Line p (x+w,h) :+:  
                       Line (x,y+h) (x+w,y)  
sem ...  
```

Syntactic Sugar

```
triangle :: Point -> Int -> Int -> Pic  
triangle p@(x,y) w h  = Line p (x,y+h) :+:  
                       Line p (x+w,h) :+:  
                       Line (x,y+h) (x+w,y)  
```
Exercises

(1) Extend \textit{Pic} by a function \textit{rectangle}

(2) “Deepen” the \textit{rectangle} function
## Summary

<table>
<thead>
<tr>
<th>Language Aspect</th>
<th>Representation in Metalanguage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Deep Embedding</td>
</tr>
<tr>
<td>syntax</td>
<td>data type</td>
</tr>
<tr>
<td>program</td>
<td>value</td>
</tr>
<tr>
<td>semantic domain</td>
<td>(data) type</td>
</tr>
<tr>
<td>semantic value</td>
<td>value</td>
</tr>
<tr>
<td>valuation</td>
<td>function</td>
</tr>
<tr>
<td>syntax + semantics</td>
<td>data type + function</td>
</tr>
<tr>
<td></td>
<td>((L, [[·]]))</td>
</tr>
</tbody>
</table>

- **Deep Embedding**
  - L (data type)
  - p :: L (value)
  - D (data type)
  - v :: D (value)
  - [[·]] : L → D (function)
  - f pat :: T (function)

- **Shallow Embedding**
  - p :: D
  - D
  - v :: D
  - f :: T → D
  - f pat = rhs