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)**: embedded DSL
  - **shallow embedding**: implicit, through constructors of semantic domain + functions
  - **deep embedding**: explicit, through a separate data type
DSLs & Denotational Semantics

Semantic Language

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

(Turing-complete computation)

(General-Purpose) PL

DSL

\[ \text{sem} :: S \rightarrow T \]

Non-computation

\[ \text{sem} :: \text{Shape} \rightarrow \text{Image} \]

\[ \text{sem} :: \text{Move} \rightarrow \text{Pos} \]

\[ \text{Move} \rightarrow \text{Trip} \]

Other computation

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
```
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 \rightarrow Int \rightarrow Int \rightarrow 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
```
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

Add key-word parameters

type KW = String
[from,to,at,width,height] = [
"from","to","at","width","height"]

“Compile” to semantic values

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!"
Syntax Checking Through Pattern Matching

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

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!"

> 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

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

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

Abstract Syntax

Map syntactic operators to semantic functions

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'

Semantic Function
Deep DSLs

```haskell
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 d d') = sem d :+: sem d'

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)
```

Why “go” deep?

- trivial (or: boring)
- identical
- not much to be learned from the definition

DSLs in Haskell
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

Indirect manipulation of the domain

Data type for semantic domain

Additional data type for abstract syntax
"Shallowing" a DSL

(1) Split \texttt{sem} function into separate functions
(2) Eliminate syntax data type

Abstract Syntax

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

type Image = [(Int,Int)]

Semantic Domain

\begin{verbatim}
sem :: Shape \rightarrow 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
\end{verbatim}
“Shallowing” a DSL

**DSLs in Haskell**

```haskell
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
```
Exercises

(1) Turn the `LR` constructor into a function `lr`

(2) “Shallow” the `Move` language

```haskell
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

td :: Image -> Image -> Image
td i1 i2 = adjustY (maxY i2) i1 ++ i2
```
"Deepening" a DSL

1. Introduce syntax data type
2. Combine separate functions into \texttt{sem} functions

```
triangle :: \texttt{<ArgType>} \rightarrow \texttt{Pic}
```

Diagram:
- Abstract Syntax
- Semantic Domain
- Semantic Function
- Syntactic Sugar
- \texttt{data Cmd}
- \texttt{data Pic}
- \texttt{sem :: Cmd \rightarrow Pic}
- \texttt{triangle :: <ArgType> \rightarrow Pic}
“Deepening” a DSL

Abstract Syntax

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

(1) Duplicate constructor
(≈ creating syntax)

(2) Add identity `sem` mapping
(≈ creating semantics)

```haskell
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 ...
```

Semantic Domain

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

(3) Derive constructor from function type & LHS
(≈ creating syntax)

```haskell
triangle :: Tri 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)
```

Syntactic Sugar

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

(4) Transfer function RHS to `sem` mapping
(≈ creating semantics)

```haskell
triangle p@(x,y) w h = Line p (x,y+h) :+:
                       Line p (x+w,h) :+:
                       Line (x,y+h) (x+w,y)
```

DSLs in Haskell
Exercises

1. Extend Pic by a function rectangle

2. “Deepen” the rectangle function
### Summary

#### DSLs in Haskell

Then we extend the function definition for `line` to take additional `KW` arguments and check, using pattern matching, that the correct keywords have been used in a call of `line`.

```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!"
```

As illustrated in the function definition, it is very easy in this approach to extend the syntax on the fly, for example, with alternative orderings of arguments. (It is also easy to extend this definition to produce more elaborate error messages that report the incorrect keywords.) If we write a command for drawing a line we have to use the keywords `from` and `to`. If we do, a semantic `Line` value is produced correctly, if we don’t, the function reports a syntax error.

```haskell
> line from (1,1) to ctr
Line (1,1) (3,2)
> line to (1,1) to ctr
*** Exception: Incorrect keyword!
```

In addition to the constructor names of the semantic domain (and potentially added syntactic sugar), we also introduce function definitions for those operations of the DSL that are not directly represented by constructors of the semantic domain. The operation for drawing triangles is such an example. The corresponding function definition looks as follows.

```haskell
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)
```

As with the command for drawing lines, we could extend the above function definition by arguments representing keywords to enrich the concrete syntax.

The important observation here is that these function definitions are comprised of two parts that combine the definition of DSEL syntax and semantics. First, the function head, that is, the left-hand sides of the equations, with the name of the function and its argument patterns, defines the DSEL syntax. Second, the expressions on the right-hand sides of the equations define the semantics of that particular syntactic construct. Since we obtain different function definitions for different operations of the DSEL, the valuation function from syntactic elements to values in the semantic domain is spread across several function definitions.

A summary of the preceding discussion is presented in Figure 1.

<table>
<thead>
<tr>
<th>Language Aspect</th>
<th>Representation in Metalanguage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Deep Embedding</strong></td>
</tr>
<tr>
<td>syntax</td>
<td>data type</td>
</tr>
<tr>
<td>program</td>
<td>value</td>
</tr>
<tr>
<td>program</td>
<td>p ∈ L</td>
</tr>
<tr>
<td>semantic domain</td>
<td>D</td>
</tr>
<tr>
<td>semantic value</td>
<td>v ∈ D</td>
</tr>
<tr>
<td>valuation</td>
<td>[[·]] : L → D</td>
</tr>
<tr>
<td>syntax + semantics</td>
<td>(L, [[·]])</td>
</tr>
<tr>
<td>data type</td>
<td>f pat :: T</td>
</tr>
<tr>
<td>function LHS</td>
<td>f pat :: T</td>
</tr>
<tr>
<td>semantic value</td>
<td>(L, [[·]])</td>
</tr>
<tr>
<td>expression</td>
<td>p :: D</td>
</tr>
<tr>
<td>semantic domain</td>
<td>D</td>
</tr>
<tr>
<td>function RHS</td>
<td>v :: D</td>
</tr>
</tbody>
</table>

---

### Figure 1

Summary of the language-based view of DSLs and the representation of internal DSLs within a typed, functional metalanguage.