Embedded Domain-Specific Languages

Dima Szamozvancev
University of Cambridge
ds709@cl.cam.ac.uk

CS141 – Functional Programming
University of Warwick
11 March 2019
Pop quiz

Guess the domain!
main :: IO ()
main = hspec $ do
  describe "Prelude.head" $ do
    it "returns the first element of a list" $ do
      head [23 ..] `shouldBe` (23 :: Int)
menu :: Css
menu = header △ nav ?
do background white
    color "#04a"
    fontSize (px 24)
    padding 20 0 20 0
textTransform uppercase
hilbert :: Int → Trail
hilbert 0 = mempty
hilbert n = hilbert' (n-1) # reflectY ◇ vrule 1
◇ hilbert (n-1) ◇ hrule 1
◇ hilbert (n-1) ◇ vrule (-1)
◇ hilbert' (n-1) # reflectX

where
    hilbert' m = hilbert m # rotateBy (1/4)

diagram :: Diagram B
diagram = strokeT (hilbert 6) # lc silver
            # opacity 0.3
hilbert :: Int -> Trail
hilbert 0 = mempty
hilbert n = hilbert' (n - 1) # reflectY # vrule 1

where
  hilbert' m = hilbert m # rotateBy (1/4)

diagram :: Diagram B
diagram = strokeT (hilbert 6) # lc silver # opacity 0.3
tricycle :: Behaviour Shape
tricycle u =
  buttonMonitor u `over`
  withColor (cycle3 green yellow red u)
  (stretch (wiggleRange 0.5 1) circle)
  where
cycle3 c1 c2 c3 u =
  c1 `untilB` nextUser_ lbp u ➞
  cycle3 c2 c3 c1
tricycle :: Behaviour Shape
tricycle u =
  buttonMonitor u `over``
  withColor (cycle3 green yellow red u)
  (stretch (wiggleRange 0.5 1) circle)
where
  cycle3 c1 c2 c3 u =
    c1 `untilB` nextUser_ lbp u ⟷
    cycle3 c2 c3 c1
m1 = c' en :|: tripletE g fs g :|: 
    start (melody :< a :| g :~| r :| b :| c')

m2 = c_ majD ec :|: pad3 (r hr) :|:
    g__ dom7 inv inv ec :|: c_ majD ec

\textbf{comp} :: Score

\texttt{comp = score section "The end"}
\texttt{setKeySig c\_maj}
\texttt{setTempo 100}
\texttt{withMusic $ m1 \ `hom` m2}
main :: IO ()
main = do
  scotty 3000 $ do
  get "/hello/:name" $ do
    name ← param "name"
    text ("Hello " ◦ name ◦ "!")
  get "/users/:id" $ do
    id ← param "id"
    json (filter (matchesId id) allUsers)
Why was this so easy?
Domain-Specific Languages
Domain-Specific Languages

If in doubt, quote Wikipedia

A **domain-specific language** *(DSL)* is a computer language specialised to a particular application domain.  

This is in contrast to a **general-purpose language** *(GPL)*, which is broadly applicable across domains.

GPL ~ Jack of all trades  

DSL ~ Master of one
Examples of DSLs
Examples of DSLs

Markup languages
HTML, Markdown, LaTeX

```html
<html>
  <body>
    <p>Normal text.</p>
    <p><strong>Bold</strong> text.</p>
  </body>
</html>

# Heading

+ List with _italic_ text
  - **Bold** text
  - [Link](https://commonmark.org)

> Block quote
Examples of DSLs

Markup languages
- HTML, Markdown, LaTeX

Modelling languages
- UML, Z

\[
\begin{align*}
\text{Update} & : \text{CheckSys} \\
\Delta & : \text{ADDR} \\
p? & : \text{PAGE} \\
\text{working}' & = \text{working} \oplus \{a? \mapsto p?\} \\
\text{backup}' & = \text{backup}
\end{align*}
\]
Examples of DSLs

**Markup languages**
- HTML, Markdown, LaTeX

**Modelling languages**
- UML, Z

**Description languages**
- Verilog, PostScript

```systemverilog
module Sign (A, B, Y1, Y2, Y3);
  input [2:0] A, B;
  output [3:0] Y1, Y2, Y3;
  reg [3:0] Y1, Y2, Y3;
  always @(A or B)
  begin
    Y1 = +A/-B;
    Y2 = -A+-B;
    Y3 = A*-B;
  end
endmodule
```

```postscript
newpath 100 200 moveto
200 250 lineto
100 300 lineto
closepath
gsave
0.5 setgray
fill
grestore
4 setlinewidth
0.75 setgray
stroke
```

Adobe PostScript® 3®
Examples of DSLs

Markup languages
   HTML, Markdown, LaTeX

Modelling languages
   UML, Z

Description languages
   Verilog, PostScript

Special-purpose languages
   SQL, Yacc, MATLAB, Sonic Pi

```
with_fx :reverb, mix: 0.2 do
  loop do
    play scale(:Eb2, :major_pentatonic, num_octaves: 3).choose,
      release: 0.1, amp: rand
    sleep 0.1
  end
end
```

```
SELECT Name FROM Customers WHERE EXISTS
  (SELECT Item FROM Orders
   WHERE Customers.ID = Orders.ID
   AND Price < 50)
```

PostgreSQL
Examples of DSLs

Markup languages
- HTML, Markdown, LaTeX

Modelling languages
- UML, Z

Description languages
- Verilog, PostScript

Special-purpose languages
- SQL, Yacc, MATLAB, Sonic Pi

Other?
- Automator, Siri, ZORK

> look under the rug
Why use DSLs?

Focus on a particular problem

Higher level of abstraction

Domain-specific expressivity

Optimisation opportunities

Made for domain experts, not programmers
<table>
<thead>
<tr>
<th>Why use DSLs?</th>
<th>Why not use DSLs?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Focus on a particular problem</td>
<td>Need to learn another language</td>
</tr>
<tr>
<td>Higher level of abstraction</td>
<td>Need compiler, tooling, support</td>
</tr>
<tr>
<td>Domain-specific expressivity</td>
<td>Lose general expressivity</td>
</tr>
<tr>
<td>Optimisation opportunities</td>
<td></td>
</tr>
<tr>
<td>Made for domain experts, not programmers</td>
<td></td>
</tr>
</tbody>
</table>
Cutting out the middleman

CL compiles HL compiles DSL
Cutting out the middleman

CL compiles HL compiles DSL

CL compiles HL DSL
Domain-Specific Languages
Embedded Domain-Specific Languages
Embedded Domain-Specific Languages

A domain-specific language implemented *inside* some host language

Usually built as a library or a package, so distinction is not always clear

**My rules of thumb:**

1. *Is the domain recognisable from the syntax?*
2. *Does the syntax hide the complexities of the host language?*
EDSLs vs. DSLs

+
Inherit compiler, tooling, and other features of the host language
Combine with host language programs and other EDSLs
Easy to extend
No need to learn another language
Usable without familiarity with the host language

−
Constrained by the host language syntax and features
Possibly less efficient
The cost argument

(John Hughes)
Examples of EDSLs

The term appears more frequently in the context of functional programming

Closest notion in object-oriented languages: fluent programming via method chaining
Fluent interfaces

Simulate “English prose” within the syntactic constraints of the language

Often used with the Builder pattern, and testing and mocking frameworks

```java
public Person getPerson() {
    return Person.builder()
        .name("John")
        .age(27)
        .occupation("Lawyer")
        .build();
}
```
Fluent interfaces

Simulate “English prose” within the syntactic constraints of the language

Often used with the Builder pattern, and testing and mocking frameworks

```java
List<Integer> transactionsIds =
    transactions.stream()
    .filter(t -> t.getType() == Transaction.GROCERY)
    .sorted(comparing(Transaction::getValue).reversed())
    .map(Transaction::getId)
    .collect(toList());
```
Fluent interfaces

Simulate “English prose” within the syntactic constraints of the language

Often used with the Builder pattern, and testing and mocking frameworks

```csharp
IEnumerable<string> query = translations
    .Where (t => t.Key.Contains("a"))
    .OrderBy (t => t.Value.Length)
    .Select (t => t.Value.ToUpper());
```
Fluent interfaces

Simulate “English prose” within the syntactic constraints of the language

Often used with the Builder pattern, and testing and mocking frameworks

```javascript
var foo = 'bar'
var beverages = { tea: [ 'chai', 'matcha', 'oolong' ] };

foo.should.be.a('string');
foo.should.equal('bar');
foo.should.have.lengthOf(3);
beverages.should.have.property('tea').with.lengthOf(3);
```
Embedded DSLs
Functional Embedded DSLs
Functional Embedded DSLs

Abstractions of functional languages allow for a more systematic way of embedding DSLs
Functional Embedded DSLs

Abstractions of functional languages allow for a more systematic way of embedding DSLs

Express domain as an *abstract type*

type Diagram
Functional Embedded DSLs

Abstractions of functional languages allow for a more systematic way of embedding DSLs

Express domain as an *abstract type* and associated operations:

```haskell
type Diagram
```
Functional Embedded DSLs

Abstractions of functional languages allow for a more systematic way of embedding DSLs.

Express domain as an abstract type and associated operations: embedding

```
type Diagram
shape :: Shape → Diagram
```
Functional Embedded DSLs

Abstractions of functional languages allow for a more systematic way of embedding DSLs

Express domain as an *abstract type* and associated operations: *embedding, combinators*

type Diagram
shape :: Shape → Diagram
onTop :: Diagram → Diagram → Diagram
nextTo :: Diagram → Diagram → Diagram
Functional Embedded DSLs

Abstractions of functional languages allow for a more systematic way of embedding DSLs

Express domain as an abstract type and associated operations: embedding, combinators and evaluators

```haskell
type Diagram
shape :: Shape → Diagram
onTop :: Diagram → Diagram → Diagram
nextTo :: Diagram → Diagram → Diagram
draw :: Diagram → Svg
```
## Deep and shallow embedding

Dual ways of embedding a domain in the host language

<table>
<thead>
<tr>
<th>Deep</th>
<th>Shallow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intermediate syntactic representation</td>
<td>Interpret as semantics right away</td>
</tr>
<tr>
<td>Algebraic data type</td>
<td></td>
</tr>
<tr>
<td>Embedding: constructor</td>
<td>Type synonym</td>
</tr>
<tr>
<td>Combinators: constructors</td>
<td>Embedding: interpreter</td>
</tr>
<tr>
<td>Evaluator: interpreter</td>
<td>Combinators: domain functions</td>
</tr>
<tr>
<td></td>
<td>Evaluator: identity function</td>
</tr>
</tbody>
</table>
type Region

circle :: Radius → Region
outside :: Region → Region
inter :: Region → Region → Region
inRegion :: Point → Region → Bool

Deep

data Region = Circle Radius
| Outside Region
| Inter Region Region

Shallow

type Region = Point → Bool

circle :: Radius → Region
circle r = \p → magnitude p ≤ r
outside :: Region → Region
outside rg = \p → not (rg p)
inter :: Region → Region → Region
inter rg1 rg2 = \p → rg1 p && rg2 p
Deep

data Region = Circle Radius  
   | Outside Region
   | Inter Region Region

circle :: Radius \rightarrow Region
circle = Circle

outside :: Region \rightarrow Region
outside = Outside

inter :: Region \rightarrow Region \rightarrow Region
inter = Inter

inRegion :: Point \rightarrow Region \rightarrow Bool
inRegion p (Circle r) =
   magnitude p \leq r
inRegion p (Outside rg) =
   not (inRegion p rg)
inRegion p (Inter rg1 rg2) =
   inRegion p rg1 \&\& inRegion p rg2

Shallow

type Region = Point \rightarrow Bool

circle :: Radius \rightarrow Region
circle r = \lambda p . magnitude p \leq r

outside :: Region \rightarrow Region
outside rg = \lambda p . not (rg p)

inter :: Region \rightarrow Region \rightarrow Region \rightarrow Region
inter rg1 rg2 = \lambda p . rg1 p \&\& rg2 p

inRegion :: Point \rightarrow Region \rightarrow Bool
inRegion p rg = rg p
Deep vs. shallow embedding

Two dimensions of extensibility:
adding new operations, and adding new interpretations

- e.g. union of two regions
- e.g. area of a region

<table>
<thead>
<tr>
<th>Deep</th>
<th>Shallow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Difficult to add a new operation</td>
<td>Easy to add a new operation</td>
</tr>
<tr>
<td>Extend the data type</td>
<td>Define new combinator</td>
</tr>
<tr>
<td>Define new combinator</td>
<td></td>
</tr>
<tr>
<td>Add new case to every evaluator</td>
<td></td>
</tr>
<tr>
<td>Easy to add a new interpreter</td>
<td>Difficult to add a new interpreter</td>
</tr>
<tr>
<td>Define new evaluator</td>
<td>Usually need to change the type representation</td>
</tr>
<tr>
<td>Pattern-match on the AST</td>
<td></td>
</tr>
</tbody>
</table>
Deep vs. shallow embedding

This duality is an instance of the expression problem

"The expression problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts)."

Phil Wadler

Still a very active area of research!
Functional EDSLs
Functional EDSLs in Haskell
Functional EDSLs in Haskell

EDSLs are at the intersection of PL research, industrial applications, and pet projects

And so is Haskell!
Functional EDSLs in Haskell

Scholarly articles for **embedded domain specific language**
- Evolving an **embedded domain-specific language** in ...
  - Freeman - Cited by 76
- Building **domain-specific embedded** languages
  - Hudak - Cited by 588

**Embedded domain specific language - HaskellWiki**
[https://wiki.haskell.org/Embedded_domain_specific_language](https://wiki.haskell.org/Embedded_domain_specific_language)
22 Oct 2015 - Embedded Domain Specific Language means that you embed a Domain specific language in a language like Haskell. E.g. using the ...
Functional EDSLs in Haskell

EDSLs are at the intersection of PL research, industrial applications, and pet projects.

And so is Haskell!

Designing EDSLs is an interesting programming challenge, and Haskell provides a huge playground for experimentation.

Several reasons why Haskell is a great choice for EDSLs.
1. Syntactic flexibility

Very minimalistic syntax
  Little boilerplate
  Type inference
  Application by whitespace

Syntactic sugar
  Monadic do-notation
  Infix operators and sections
  Overloading

Flexible source code layout
  Whitespace-insensitive
1. Syntactic flexibility

Very minimalistic syntax
  Little boilerplate
  Type inference
  Application by whitespace

Syntactic sugar
  Monadic do-notation
  Infix operators and sections
  Overloading

Flexible source code layout
  Whitespace-insensitive

```haskell
menu :: Css
menu = header [nav ?
  do background white
     color "#04a"
     fontSize (px 24)
     padding 20 0 20 0
     textTransform uppercase
```
1. Syntactic flexibility

Very minimalistic syntax
  Little boilerplate
  Type inference
  Application by whitespace

Syntactic sugar
  Monadic do-notation
  Infix operators and sections
  Overloading

Flexible source code layout
  Whitespace-insensitive

m1 = c' en :|: tripletE g fs g :|:
  start (melody < a :| g
  ~| r :| b :| c')

m2 = c_ majD ec :|: pad3 (r hr) :|:
  g__ dom7 inv inv ec :|: c_ majD ec

comp :: Score
comp = score section "The end"
  setKeySig c_maj
  setTempo 100
  withMusic $ m1 `hom` m2
2. Powerful abstractions

**Type classes**
- Exploit the formal structure and properties of the domain
- Overloaded functions that work on all instances of a class
- Syntactic sugar, e.g. do-notation

**Denotational design**
- Think of the domain in terms of its formal semantics
- Implementation follows the laws of the semantic domain
2. Powerful abstractions

Type classes
Exploit the formal structure and properties of the domain
Overloaded functions that work on all instances of a class
Syntactic sugar, e.g. do-notation

Denotational design
Think of the domain in terms of its formal semantics
Implementation follows the laws of the semantic domain

Combination $\rightarrow$ Monoid
Pretty printers, diagrams, music

\[
\text{mconcat [text "foo", space, text "bar"]}
\]
\[
\text{square 1 $\triangle$ circle 2}
\]
2. Powerful abstractions

**Type classes**
Exploit the formal structure and properties of the domain
Overloaded functions that work on all instances of a class
Syntactic sugar, e.g. do-notation

**Denotational design**
Think of the domain in terms of its formal semantics
Implementation follows the laws of the semantic domain

<table>
<thead>
<tr>
<th>Combination</th>
<th>Monoid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pretty printers, diagrams, music</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Choice</th>
<th>Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parser combinators</td>
<td></td>
</tr>
</tbody>
</table>

parseString "CS141" <> many integer
2. Powerful abstractions

**Type classes**
- Exploit the formal structure and properties of the domain
- Overloaded functions that work on all instances of a class
- Syntactic sugar, e.g. do-notation

**Denotational design**
- Think of the domain in terms of its formal semantics
- Implementation follows the laws of the semantic domain

**Combination** \(\leadsto\) **Monoid**
- Pretty printers, diagrams, music

**Choice** \(\leadsto\) **Alternative**
- Parser combinators

**Composition** \(\leadsto\) **Category**
- Lenses

```
("hello",("world","!!!"))^.2._2.to length
```
2. Powerful abstractions

**Type classes**
Exploit the formal structure and properties of the domain
Overloaded functions that work on all instances of a class
Syntactic sugar, e.g. do-notation

**Denotational design**
Think of the domain in terms of its formal semantics
Implementation follows the laws of the semantic domain

<table>
<thead>
<tr>
<th>Combination</th>
<th>~&gt;</th>
<th>Monoid</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Pretty printers, diagrams, music</td>
</tr>
<tr>
<td>Choice</td>
<td>~&gt;</td>
<td>Alternative</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Parser combinators</td>
</tr>
<tr>
<td>Composition</td>
<td>~&gt;</td>
<td>Category</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lenses</td>
</tr>
<tr>
<td>Sequencing</td>
<td>~&gt;</td>
<td>Monad</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Everything</td>
</tr>
</tbody>
</table>

```
sat :: (Char -> Bool) -> Parser Char
sat p = do x <- item
  guard (p x)
  result x
```
2. Powerful abstractions

Type classes
Exploit the formal structure and properties of the domain
Overloaded functions that work on all instances of a class
Syntactic sugar, e.g. do-notation

Denotational design
Think of the domain in terms of its formal semantics
Implementation follows the laws of the semantic domain

hilbert :: Int → Trail
hilbert 0 = mempty
hilbert n = hilbert' (n-1) # reflectY ◦ vrule 1
 ◦ hilbert (n-1) ◦ hrule 1
 ◦ hilbert (n-1) ◦ vrule (-1)
 ◦ hilbert' (n-1) # reflectX

where
hilbert' m = hilbert m # rotateBy (1/4)

diagram :: Diagram B
diagram = strokeT (hilbert 6) # lc silver
 # opacity 0.3

hilbert <=LIG:: Int <=LIG-> Trail
hilbert 0 = mempty
hilbert n = hilbert' (n-1) # reflectY ◦ vrule 1
 ◦ hilbert (n-1) ◦ hrule 1
 ◦ hilbert (n-1) ◦ vrule (-1)
 ◦ hilbert' (n-1) # reflectX

where
hilbert' m = hilbert m # rotateBy (1/4)

diagram :: Diagram B
diagram = strokeT (hilbert 6) # lc silver
 # opacity 0.3
2. Powerful abstractions

Type classes
Exploit the formal structure and properties of the domain
Overloaded functions that work on all instances of a class
Syntactic sugar, e.g. do-notation

Denotational design
Think of the domain in terms of its formal semantics
Implementation follows the laws of the semantic domain

```
main :: IO ()
main = withSQLite "people.sqlite" $ do
  createTable people
  insert_ people [ ... ]

adultsAndTheirPets ← query $ do
  person ← select people
  restrict (person ! #age .≥ 18)
  return (person ! #name :*: person ! #pet)
liftIO $ print adultsAndTheirPets
```
3. Type system

**Strong typing**
- Guide EDSL development and use
- (Sometimes) good documentation
- Error prevention

**Domain-specific type systems**
- Type-level programming features to precisely model the domain
- Custom compiler errors
- “Logic” programming with type classes
- Term- and type-level embedding
3. Type system

**Strong typing**
- Guide EDSL development and use
- (Sometimes) good documentation
- Error prevention

**Domain-specific type systems**
- Type-level programming features to precisely model the domain
- Custom compiler errors
- “Logic” programming with type classes
- Term- and type-level embedding

```haskell
type UserAPI = "user" :> Capture "userid" Integer
  :> Get '[JSON] User
  :<> "list-all" :> "users"
    :> Get '[JSON] [User]
  -- equivalent to 'GET /user/:userid'
  -- or 'GET /list-all/users'

userAPI :: Proxy UserAPI
userAPI = Proxy

userDocs :: String
userDocs = markdown $ docs userAPI

start :: IO ()
start = do
  run 8000 (serve userAPI userServer)
```
3. Type system

Strong typing
Guide EDSL development and use
(Sometimes) good documentation
Error prevention

Domain-specific type systems
Type-level programming features to precisely model the domain
Custom compiler errors
“Logic” programming with type classes
Term- and type-level embedding

score withMusic $ c qn :-: b qn ❌
type error:
• Major sevenths are not permitted in harmony: C and B
• In the expression:
  score withMusic $ c qn :-: b qn

score setRuleSet empty
  withMusic $ c qn :-: b qn ✅
Conclusions
Conclusions

EDSLs are useful, fun to work with and even more fun to work on

   Good exercise in programming, using advanced language features and even user experience design

Don't be afraid to experiment, break (monad) rules and try weird hacks – you might end up inventing something cool
EDSLs are useful, fun to work with and even more fun to work on.

Good exercise in programming, using advanced language features and even user experience design.

Don't be afraid to experiment, break (monad) rules and try weird hacks – you might end up inventing something cool!
Conclusions

EDSLs are useful, fun to work with and even more fun to work on

Good exercise in programming, using advanced language features and even user experience design

Don't be afraid to experiment, break (monad) rules and try weird hacks – you might end up inventing something cool

Also a great for third year projects (ask Michael)
Thank you!
Any questions?

Dima Szamozvancev
University of Cambridge
ds709@cl.cam.ac.uk