(Domain-specific) language design
(and evolution)

Course "Software Language Engineering"

University of Koblenz-Landau
Department of Computer Science
Ralf Lämmel
Software Languages Team
Semantics-Driven DSL Design
by Martin Erwig and Eric Walkingshaw.

http://web.engr.oregonstate.edu/~erwig/papers/index.html
“Syntax-driven” DSL development: an illustration
Syntax of a Calendar DSL

\[ Op ::= \text{add Appt at Time} \\
| \text{move entry at Time to Time} \\
| \text{delete Time entry} \]

\[ Prog ::= Op^* \]

We just collected some features we thought of...
Semantic domain

Cal = Map Time Appt

Time is a discrete domain (e.g., date/time every minute). A map ("dynamic array") maps each time value to an appointment or an "error" value. (Think of option/maybe types.)

Let’s hope for the Eureka effect: suddenly we knew that this is the right domain.
The semantics of an operation is thus simply written as error (or undefined) element, say defining a valuation function that represents this mapping. Looking at our syntax, we can observe that an appointment is scheduled for. One advantage of the syntax

Syntax

\[
\begin{align*}
\text{update} : & \text{Time} \times \text{Appt} \times \text{Cal} \to \text{Cal} \\
\text{update}(t, a, c) &= c - \{(t, c(t))\} \cup \{(t, a)\} \\
\end{align*}
\]

[[add a at t]] = update(t, a, c)
[[move entry at t to t']] = update(t', c(t), update(t, ε, c))
[[delete t entry]] = update(t, ε, c)

[[o₁, o₂, ..., oₙ]] = [[oₙ]] (... [[o₂]] ([[o₁]] \{(t, ε) | t \in \text{Time}\}) ...)

Op ::= add Appt at Time
    | move entry at Time to Time
    | delete Time entry

Prog ::= Op*
An extension:

**Appointments with length**

\[ Op ::= \text{add} \text{ Appt at } \text{Time with } \text{length } \text{Int} \]
\[
| \quad \ldots \\
[[\text{add } a \text{ at } t \text{ with length } n]] \quad c = \text{update} (t + n - 1, a, \ldots \text{update} (t + 1, a, \text{update} (t, a, c)) \ldots )
\]

Ouch!

Semantics of move and delete must be **rewritten**.
Another extension:

Overlapping appointments

Cal = Map Time [Appt]

Ouch!
Semantics must be rewritten.
Questions

What’s first: syntax or semantics?

How do we decompose design into manageable parts?

How do we find a good semantic domain?

How do we lower chances of running into full rewrites?

How do we achieve modular extensions?
Approaching DSL definition like programming language definition
The PL approach to DSLs

Use a *metalanguage* for grammar and semantics.

Syntax: a context-free grammar (possibly encoded)

Semantics:
- (Data) types for semantic domains
- Functions for an operational or a denotational semantics

(Pragmatics: “documentation”)
Implementation styles

External DSL
- Parse and interpret DSL with the metalanguage

Internal DSL (Domain-specific embedded languages)
- The language exists within the metalanguage.
  - Deep embedding: represent syntax as data type
  - Shallow embedding: represent syntax as functions

Of much interest today
An even simpler example

A DSL for constructing pictures

Semantic concepts: lines, overlaps

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

Syntactic constructs: commands for picture elements
An even simpler example

A DSL for constructing pictures

- **Semantic** concepts: lines, overlaps
- **Syntactic** constructs: commands for picture elements

```haskell
data Cmd = Line' Point Point | Tri Point Int Int | Seq Cmd Cmd
```
Nice concrete syntax

\[
Cmd \ ::= \ line \ from \ Point \ to \ Point \\
| \ triangle \ at \ Point \ width \ Num \ height \ Num \\
| \ Cmd; \ Cmd \\
| \ ...
\]

as opposed to ...

-data \ Cmd = Line' \ Point \ Point \\
| \ Tri \ Point \ Int \ Int \\
| \ Seq \ Cmd \ Cmd
Deep embedding: map syntax to semantics

Trivial correspondence between syntax and semantics.

Mapping invokes several elements of “semantic algebra”.

```
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 (Seq d d') = sem d :+: sem d'
```
Shallow embedding: represent syntax as functions

line :: Point -> Point -> Pic
line = Line

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)

seq :: Pic -> Pic -> Pic
seq = (:+:+)
Improved syntax in shallow embedding

```haskell

type KW = String
from = "from"
to = "to"

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 (1,1) to ctr
*** Exception: Incorrect keyword!
```

Source: “Semantics-Driven DSL Design” by Martin Erwig and Eric Walkingshaw
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>
<th>Deep Embedding</th>
<th>Shallow Embedding</th>
</tr>
</thead>
<tbody>
<tr>
<td>syntax</td>
<td>$L$</td>
<td>$L$</td>
<td>function LHS</td>
</tr>
<tr>
<td>program</td>
<td>$p \in L$</td>
<td>$p :: L$</td>
<td>$f \ pat :: T$</td>
</tr>
<tr>
<td>semantic domain</td>
<td>$D$</td>
<td>$D$</td>
<td>$p :: D$</td>
</tr>
<tr>
<td>semantic value</td>
<td>$v \in D$</td>
<td>$v :: D$</td>
<td>$v :: D$</td>
</tr>
<tr>
<td>valuation</td>
<td>$[[\cdot]] : L \rightarrow D$</td>
<td>$sem :: L \rightarrow D$</td>
<td>rhs :: D</td>
</tr>
<tr>
<td>syntax + semantics</td>
<td>$(L, [[\cdot]])$</td>
<td>$(L, sem)$</td>
<td>f :: T \rightarrow D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>f pat = rhs</td>
</tr>
</tbody>
</table>
Extensibility of embedding styles

Deep embedding
- easy: Addition of new interpretations
- harder: Addition of new constructs

Shallow embedding
- easy: Addition of new constructs
- harder: Addition of new interpretations

Remember the Expression problem?
Semantics-driven design

Source: “Semantics-Driven DSL Design” by Martin Erwig and Eric Walkingshaw

© 2012-13, Ralf Lämmel, Software Languages Team, University of Koblenz-Landau, and collaborators (see 1st slide)
Recap ‘syntax-driven’ design

- Capture syntax (constructs) of DSL.
- Use samples. Chase for features.
- Find a semantic domain for the constructs.
- Define semantic functions.
- Face extensibility/revision scenarios:
  - New constructs
  - Alternative semantic details
Erwig et al.: “The semantics-driven approach begins instead with the **identification of a small, compositional semantics core**, then systematically equips it with syntax. We argue that considering the semantic domain of a language first leads to a more principled language design, in part, because it forces language designers to **begin by carefully considering the essence of what their language represents.** With the proper semantics basis, the language can be systematically extended with new syntax as new features are added.” [Emphasis added.]
Semantics-driven design

- Consider semantic domain at the start.
- Use shallow embedding.
- (Extract syntax and deep embedding possibly later.)
Semantics-driven design process

1. Semantics design
   1. Domain identification
   2. Domain decomposition
   3. Domain modeling

2. Syntax design
   1. Inter-DSL syntax design
   2. Intra-DSL syntax design

Erwig et al.: “One of the biggest advantages of the semantics-driven process is that it decomposes the difficult task of designing a DSL into several smaller and more manageable subproblems.”
Semantics-driven design process

Semantics-driven design leads from a problem domain to a domain-specific language that is described, or implemented, by a metalanguage. The process consists of three major steps, which are illustrated in Figure 2.

The first step decomposes the problem domain into smaller subdomains and identifies the relationships between them. In Figure 2 we find two subdomains $D_1$ and $D_2$ and a relationship $R$ between them. This decomposition determines the semantic domain of our DSL. This step happens completely within the problem realm. No metalanguage concepts are invoked yet.

The second step concerns the modeling of the decomposed semantic domain in the metalanguage. Each subdomain forms the basis of (that is, is the semantic domain for) a little language called a micro-DSL $L_i$. The identified relationships between subdomains are modeled as language schemas $S$. In Figure 2 we observe that each domain $D_i$ is modeled as a micro-DSL $L_i$ and that the relationship $R$ is modeled as a language schema $S$. This step takes the DSL design from the problem realm into the metalanguage realm.

In terms of Haskell, domain modeling means to define types to represent the semantic domains of languages and type constructors to represent the semantic domains of language schemas.

These first two steps taken together comprise the semantic design part of the DSL design process. All decisions regarding the semantics of the DSL happen in this part, which will be illustrated with the help of an example in the next subsection on "Semantic Design".

The third step in the design process is the design of the syntax of the DSL. This step can also be broken down into two parts. Specifically, we can distinguish between the syntactic design of each micro-DSL and the design of syntax that spans several of these micro-DSLs, leading to constructs that build relationships between these elementary objects. We have not specifically illustrated the separate parts of
Domain identification

Perhaps, compare with OOA/OOD.
We need these domains:

- calendars
- time
- appointments

These domains are related in some way.

```plaintext
type Cal = Map Time Appt
```

We identify a relationship “Map”.

Compare this with OOA.
Domain decomposition

Perhaps, compare with the use of design pattern.
Domain decomposition by parametrization

Erwig et al.: “[...] we can also leave some aspects of the domain parameterized, producing a class of related DSLs that can be instantiated for different subdomains; that is, we can define our DSL itself as a language schema.”

```
type CalT a = Map Time a
type Cal = CalT Appt
```
Decomposition continued

“Time” may be reasonably decomposed into a date part and the time (during the day), thereby leading to a more evocative domain also enabling extra operations.

```
type CalDT a = Map (Date, Time) a
```
Composition of language schemas

We use a relationship for composition.

def type CalT a = Map Time a
def type CalD a = Map Date a
def type Cal a = CalD (CalT a)
Extensibility based on schema instantiation

Thus, such extra parametricity solves one of the earlier extension scenarios.

Calendars with entries with “lengths”

```haskell
type CalT a = Map Time a
type CalD a = Map Date a
type Cal a = CalD (CalT a)
type CalL a = Cal (a, Time)
```
A higher-order language operator

```haskell
type \text{CalT} \ a = \text{Map} \ \text{Time} \ \ a
\text{type CalD} \ a = \text{Map} \ \text{Date} \ \ a
\text{type Cal} \ a = \text{Compose} \ \text{CalD} \ \text{CalT} \ a
\text{type Compose} \ s \ t \ a = s (t \ a)
```

Composition of type-level functions
Domain modeling
Domain for maps

```haskell
data Map k v
  = Empty            -- empty calendar
  | k :-> v          -- a single entry
  | Map k v & Map k v -- composed calendar
```

We use infix constructors for what it matters.
We could also reuse Data.Map.Map.
Domain for the **Date** micro DSL

```haskell
data Month = Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec
  deriving (Eq, Show, Enum)

type Day = Int

data Date = D Month Day
  deriving (Eq, Show)
```

© 2012-13, Ralf Lämmel, Software Languages Team, University of Koblenz-Landau, and collaborators (see 1st slide)
Domain for the *Date* micro DSL

```haskell
instance Enum Date
    where
        -- Successors of dates
        -- Simplified implementation:
        -- * no handling of "end-of-year"
        -- * no coverage of leap years
        -- * no implementation of other Enum methods
        succ (D m d) = D m' d'
            where
                d' = d + 1 - (if d+1>days then days else 0)
                m' = if d+1>days then succ m else m
                days | m==Feb = 28
                    | elem m [Jan,Mar,May,Jul,Aug,Oct,Dec] = 31
                    | otherwise = 30
```

This way, we can speak of the next date (day).
Domain for the *Time* micro DSL

```haskell
type Hour = Int
type Minute = Int
data Time = T Hour Minute
  deriving (Show, Eq)
```
Domain for the *Time* micro DSL

instance Num Time
  where
    -- Addition of times
    -- Simplified implementation:
    -- * no coverage of modulo 24h
    -- * no implementation of other Num methods
    (T h m) + (T h' m') = T h'' m''
    where
      h'' = h + h' + if (m+m'>59) then 1 else 0
      m'' = m + m' - if (m+m'>60) then 60 else 0

This way, we can add some “length” to a time.
A DSL “program”

Notationally inconvenient: too many parentheses. Fix priorities and associativity!

Notationally inconvenient: too many semantic constructors. Define smart constructors!

```haskell
week52 :: Cal String
week52 = ((D Dec 30) :-> (T 8 0 :-> "Work"))
  &: ((D Dec 31) :-> (T 22 0 :-> "Party"))
```
Syntax design
Erwig et al.:

“Syntax can facilitate the expression of recurring patterns or templates, and be used to impose constraints on what can be expressed to help avoid erroneous programs. [...] Adding new operations is an inherently ad hoc process since it is impossible to foresee all desired features and use cases.” [Emphasis added.]
Smart constructors for the micro DSLs for time and date

\[
\begin{align*}
\text{[jan,feb,mar,apr,mar, jun, jul, aug, sep, oct, nov, dec]} &= \text{map D [Jan .. Dec]} \\
\text{hours h} &= T h 0 \\
\text{am h} &= \text{hours h} \\
\text{pm h} &= \text{hours (h+12)} \\
\text{mins m} &= T 0 m \\
\text{after t t'} &= t' + t
\end{align*}
\]
We distinguish schedules and calendars.
Some operations

-- Schedule shower and breakfast for a day

\[
\text{wakeAt} :: \text{Time} \rightarrow \text{Sched} \rightarrow \text{String}
\]
\[
\text{wakeAt} \ t = (t :\rightarrow "\text{Shower}"
\quad :\&: (\text{mins} \ 20 \ \text{`after`} \ t :\rightarrow "\text{Breakfast}"))
\]

-- Schedule shower and breakfast for all days

\[
\text{everyDay} :: \text{Date} \rightarrow \text{Sched} \rightarrow \text{Cal} \rightarrow \text{a}
\]
\[
\text{everyDay} \ d \ s = \text{foldr} \ (\d \ c :\rightarrow \ c :\&: (d :\rightarrow s)) \ \text{Empty} \ [d..]
\]
Wrap-up
Comparison

- **Semantics-driven**
  - Incremental, compositional
  - Domain-focused

- **Syntax-driven**
  - Monolithic, idiosyncratic
  - Feature-focused, use-case-oriented
Summary

- Focus on semantics first.
- Think of syntax as use cases.
- Use Haskell et al. for shallow embedding.
- Attempt abstraction and composition.