Domain Specific Languages

YSC3208: Programming Language Design & Implementation

Răzvan Voicu

Week 10, March 20 – 24, 2017

Outline

DSL definition

Definition

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

- It is not general purpose.
- Captures precisely the semantics of the domain.
- No more and no less.

DSL definition

Definition

The ultimate abstraction of an application domain.

- A language that you can teach to an intended user within a day
- A user inmersed in the domain already knows the domain semantics
- All we need to do is to provide a notation to express that semantics

Why build an DSL?

- Simplify the life of a domain specialist
  - There is no need to learn/master a general purpose language (GPL)
- Simplify the life of regular software developers
  - Task automation: generate code for complex tasks
  - Product line: restrict the programming patterns, enforce conventions
  - Data structure representation and traversal
  - “System” front-end: easy and reliable system configuration
  - GUI construction: eliminate the tedious work
- Enable highly specialized optimizations

DSL examples

- HTML
- Verilog, VHDL
- MATLAB and GNU Octave
- Mathematica and Maxima
- SQL
- FLEX/YACC
- gnuPLOT
- Generic Eclipse Modeling System for creating diagramming languages
- Csound for sound and music synthesis

Advantages of a DSL

Programs in target domain are

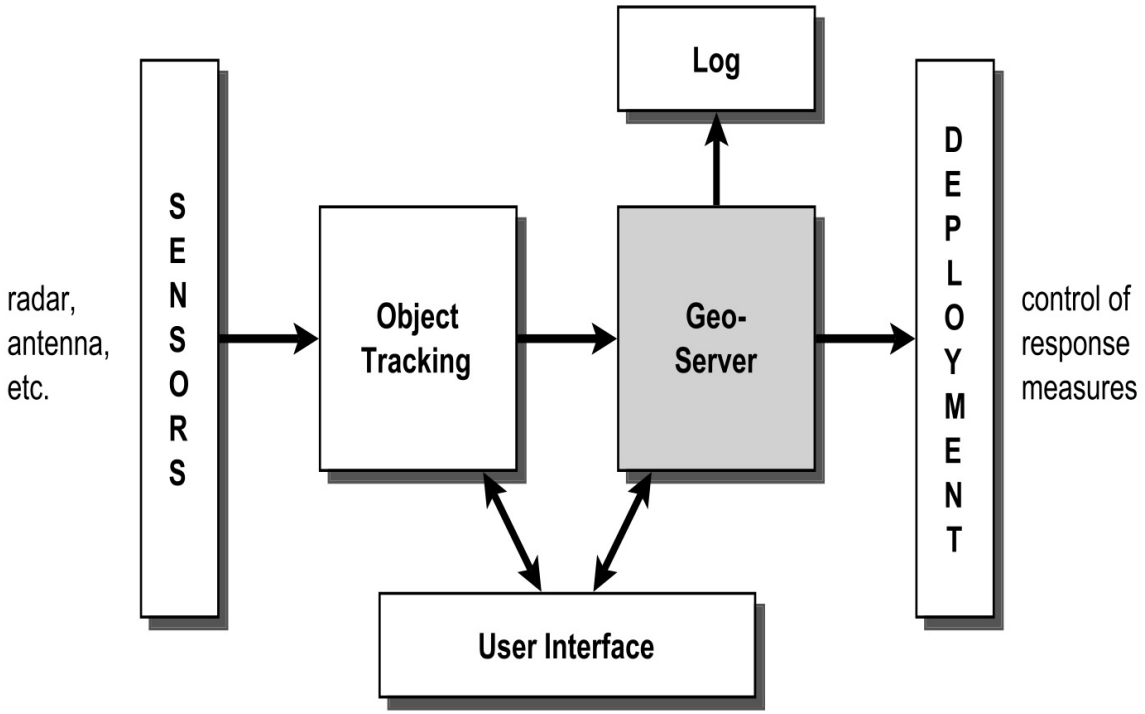
1. more concise
2. quicker to write
3. easier to maintain
4. easier to reason about

1,2	contribute to programmer productivity
3	dominant cost in SW systems
4	formal verification, program transformation, compiler optimization

DSL build process

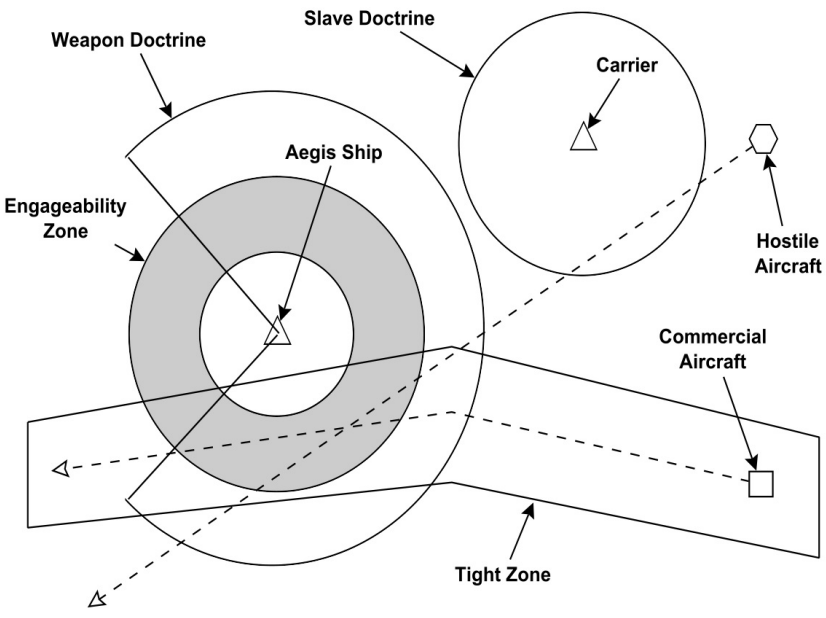
- Pick the target domain
- Design a DSL that effectively captures the domain:
  - Syntax:
    - concrete language vocabulary and abstract language structure
    - abstract syntax: names the core concepts and their links (AST)
    - concrete syntax: describes the lexical glue: identifiers, keywords, delimiters, operators, precedence
  - Semantics: description of domain concepts
  - Runtime
- Build software tools to support the DSL
  - at least a compiler or translator
  - IDE integration, debugger, documentation generator
- Develop applications using the DSL infrastructure

The geometric region server



- Part of a larger system
- **Input:** object positioning data
- **Output:** Geometric relationships between objects

The geometric server input



- Can be seen as a map
- The server should monitor the position/status of friendly ships (triangles)
- There are zones of interest around each ship
- There are tight zones of arbitrary form
- There are hostile aircrafts. Geo-server has to determine presence of these in zones.

The geometric server output

Time 0.0:
commercial aircraft: (38.0,25.0)
– In tight zone
hostile craft: (258.0,183.0)
Time 20.0:
commercial aircraft: (58.0,30.0)
– In tight zone
hostile craft: (239.0,164.0)
Time 40.0:
commercial aircraft: (100.0,43.0)
– In engageability zone
– In tight zone
hostile craft: (210.0,136.0)
– In carrier slave doctrine
Time 60.0:
commercial aircraft: (159.0,36.0)
– In engageability zone
– In tight zone
hostile craft: (148.0,73.0)
– In carrier slave doctrine
Time 80.0:
commercial aircraft: (198.0,27.0)
– In engageability zone
– In carrier slave doctrine
– In tight zone
hostile craft: (110.0,37.0)
– In tight zone

- Output could be extended to extrapolate positions and estimate times for future events
- The server is given a sequence of maps representing a temporal sequence of movements

Language aspects

- Concrete Syntax:
  e.g. Region S = square of size 20 at 10,10 intersected with union of circle of size 10 and square of size 2 at 3,3
- Abstract syntax:
  - key concepts are Geometric Regions
  - a DSL program describes Geometric Regions:
    - circle, square,
    - region union, intersection, scaling, translation
- Semantics: what interpretation can be given to Geometric Regions

General problems in DSL design and implementation

- Language design is difficult!
- Programmers want more features!
- Performance might be poor!
- New languages for every domain!

Implementation options

- start fresh (seldom used in DSLs)
- extend/customize an existing language, often by using "metaprogramming"

Metaprogramming

- the writing of computer programs that write or manipulate other programs (or themselves) as their data
- allows the definition and manipulation of a new language within an existing one

One extreme: Start from scratch

Develop a fresh language:

- new syntax
- new semantics
- new toolchain: custom parsers/analysers/backends/intepreters

Unsatisfactory

Full customization at the cost of complicated development.

Another extreme: Full reuse

GPLs as DSLs

For a given domain, in combination with an application library, a general purpose programming language (GPL) can act as a DSL.

That is:

- The library API constitutes the DSL vocabulary
- The DSL reuses the syntax, semantics, analyses (typechecking) and tools from the GPL
  (e.g. DirectX: for multimedia concepts)

Unsatisfactory

It is good for prototyping. However the reused elements (e.g. typechecking) remain too general, they are independent of the DSL.

Best option: find the middle ground

Reuse **and customize** components of an existing language.

Middleground Solution

Embed the DSL into a GPL

- focus on semantics
- reuse all features of the language
- use metaprogramming tools to make it practical and have a uniform look-and-feel

Definition

DSLs designed within an existing higher-order and typed programming language (e.g. Haskell, Scala, OCaml) are called domain specific embedded languages (DSEL)

Facilitate reuse *AND* customization of: syntax, semantics, implementation code (toolchain/software tools)

Embedding a DSL in OCaml

- OCaml is a multi-paradigm language suitable as DSL backend
- the DSL developer has only to write a conversion from the DSL into a regular OCaml program
- Supports several mechanisms for easy construction of such converters
  - extendable preprocessors (e.g. camlp4, ppx)
  - OCaml libraries (e.g. fan)
  - OCaml extensions/dialects (e.g. MetaOCaml)
- The DSL is defined by a OCaml module, which is passed to one of the above along with the program to process

Embedding design choices

- If the DSL syntax/semantic:
  - matches the target language $\rightarrow$ sufficient to provide a specialized runtime $\approx$ specialized library
  - differs from the target language $\rightarrow$ a translator is required
- If the semantic differs, embeddings can be either:
  - **Shallow** : define a type (usually a function type) that directly captures the DSL semantics
  - **Deep** : do a full translation guided by the DSL abstract syntax.

Shallow embedding

- Will capture directly the denotational semantics of the DSL.

Definition

Denotational semantics construct denotations (mathematical objects) that describe the meanings of language expressions.

- Semantics should be compositional
- Shallow embedding a language typically consists of modeling DSL core concepts as functions
- Such functions capture the meaning, making it very easy to interpret!

Shallow embedding example

- A geometric region is a function indicating whether points are in the region or not (the characteristic function for the set of points!)

```
type point = float * float
type sRegion = point → bool
```

- The meaning is captured in the type, thus it is very easy to interpret!

```
runS : sRegion → point → bool
runS r p = r p
```

- The language is given by defining the regions:

$$\mathit{empty}\ p = \mathit{false} \qquad \mathit{scale}\ (s_x, s_y)\ r\ (x, y) = r\ (x/s_x, y/s_y)$$
$$\mathit{circle}\ p = \mathit{distance}\ p\ (0,0) \leq 1 \qquad (\mathit{inters}\ r_1\ r_2)\ p = r_1\ p\ \&\ r_2\ p$$
$$\mathit{square}\ (x, y) = \mathit{abs}\ x \leq 1\ \&\ \mathit{abs}\ y \leq 1$$

Deep embedding example

- A deep embedding will use algebraic types to directly model the core concepts
- the meaning is given by a separate component, an interpreter
- For a deep embedding, the language is already encoded in the resulting type
- To interpret a program we have to cater to each component in the type

Deep embedding example

- A deep embedding will enumerate the ways of building regions

A geometric region is one of the following

```
type point = float * float
type dRegion =
| Empty
| Circle
| Square
| Scale of (point * dRegion)
| Translate of(point * dRegion)
| Outside of dRegion
| Intersect of (dRegion * dRegion)
| Union of (dRegion * dRegion)
```

Deep embedding example

The language is already there, to interpret it we have to take care of all the cases:

```
runD : dRegion -> point -> bool
let runD reg (x, y) = match reg with
| Empty -> false
| Circle -> distance (x,y) (0,0) <= 1
| Square -> abs x <= 1 & abs y <= 1
| Scale ((sx, sy), reg) -> runD reg (x/sx, y/sy)
| Translate ((sx, sy), reg) -> runD reg (x - sx, y - sy)
| Outside reg -> not (runD reg (x, y))
| Intersect (r1, r2) -> runD r1 p & runD r2 p
| Union (r1, r2) -> runD r1 p | runD r2 p
```

Deep vs Shallow

- Shallow
  - Works directly with the (denotational) semantics.
  - Concise and elegant.
  - Allows trivial creation of an interpreter.
  - Difficult to debug and/or analyze, limited to a single interpretation.
  - Easy to extend the DSL, and easy to add new constructs.
- Deep
  - Full control over the AST, many different interpretations possible.
  - Allows on-the-fly runtime transformations, optimizations and conversions. They are easily defined by pattern matching.
  - We can visualize and debug the AST.
  - Allow better leverage of the type-checker.
  - Easier extension of the language.

Advantages of embedding in OCaml

- Higher order functions provide for very powerful shallow embeddings!
- When the semantics of a particular domain is given by some kind of functions, operations will naturally take functions as arguments and produce functions as results!
- Infix operators provide practical notation

Quantified Boolean Formulae (QBF) DSL

- Allows the description of formulae in first-order.
- Basic atoms: True, False, variables.
- Operators: conjunction, disjunction, negation, implication.
- Quantifiers: Universal.
- E.g.: $\forall p.\mathit{True} \Rightarrow (p \vee \neg p)$

QBF shallow embedding

A formula is a function from variable assignments to boolean values.

```
type varEnv = string -> boolean
let extEnv env x v = fun y -> if y=x then v else env y
type sQBF = varEnv -> boolean
let fTrue env = true                    let fFalse env = false
let fNot f env = not (f env)            let fVar s env = env s
let fAnd f1 f2 env = (f1 env) && (f2 env)
let fOr f1 f2 env = (f1 env) || (f2 env)
let fImplies f1 f2 env= (not (f1 env)) || (f2 env)
let fForall q f env = (f (extEnv env q true)) && (f (extEnv env q false))
```

Deep embedding

A QBF formula is an instance of the dQBF type

```
type dQBF=
| True
| False
| And of (dQBF*dQBF)
| Or of (dQBF*dQBF)
| Not of dQBF
| Implies of (dQBF*dQBF)
| Forall of (string*dQBF)
| Var of string
```

QBF deep embedding interpreter

```
exception VarNotFound;;
let env0 x = raise VarNotFound
let extEnv env x v = fun y -> if y=x then v else env y
let rec eval b env = match b with
| True -> true
| False -> false
| Var x -> env x
| Not b1 -> not (eval b1 env)
| And (b1, b2) -> (eval b1 env) && (eval b2 env)
| Or (b1, b2) -> (eval b1 env) || (eval b2 env)
| Implies (b1, b2) -> eval (Or (b2, And( Not b2, Not b1))) env
| Forall (x, b) -> ((eval b (ext env x true)) && ((eval b (ext env x false))
```

Metaprogramming Detour

- metaprogramming conventions:
  - metalanguage: the language in which the metaprogram is written. For DSELs, this is the language in which the DSL is embedded (e.g. OCaml)
  - object language: the language of the programs that are manipulated. In our case the DSL
  - reflection or reflexivity: the ability of a programming language to be its own metalanguage

The metalanguage allows direct manipulation of DSL programs within the metalanguage program (e.g. the DSL to OCaml translator)

metaprogramming for DSLs

Essential operations

quotations/antiquotations: operations that allow the construction and evaluation of metalanguage objects that encapsulate object language programs.

Camlp4 Quotations

Quotations are expressions or patterns enclosed by special parentheses <:id< and >> (:id is a quotation identifier)

Can appear in OCaml programs that are fed as input to camlp4.

examples of quotations

<:expr< let a = b in c >> ← might contain OCaml code

<< [x](x y) >> ← code snippet in another language

<:myquot< quotations can be any text >>

Camlp4 quotation handling

E.g. <:id<quoted_text>>

The camlp4 preprocessor replaces the quotation with the result of executing the “quotation expander", an OCaml component, denoted by “id" with the “quoted_text" input.

Typically “id” denotes a customizable parser for the “quoted_text” which generates the OCaml objects that encode the AST corresponding to “quoted_text”.

Example of Camlp4 Quotations

- the OCaml object

$$\begin{aligned} &MLast.ExLet(loc, false, \\ &\qquad [MLast.PaLid(loc, "a"), MLast.ExLid(loc, "b")], \\ &\qquad MLast.ExLid(loc, "c")) \end{aligned} \tag{1}$$

- Is the AST for "let a = b in c"
- If “expr” denotes a camlp4 “quotation expander” for OCaml AST the camlp4 preprocessor could replace:
  <:expr< let a = b in c>> with (1)

Example of Camlp4 Quotations

- A custom quotation expander “cq” might accept a more C like syntax and replace:
  <:cq< {a=b; c}>> with the same (1)

The quotation expander took a string and generated an OCaml object. Can be customized to cater for a custom language syntax.

Quotations for the region server

Ideally, a quotation expander “rs” would take a DSL region description and generate the OCaml embedding.

Quotations for the region server

For example:

region description

<:rs< square of size 20 at 10,10 intersected with union of circle of size 10 and square of size 2 at 3,3>>

Depending on the “rs” implementation, camlp4 might produce:

shallow embedding (Function of type sRegion)

(translate (10,10) (scale (20,20) square))/\((scale (10,10) circle)\/(translate (3,3) (scale (2,2) square)))

Quotations for the region server

For example:

region description

<:rs< square of size 20 at 10,10 intersected with union of circle of size 10 and square of size 2 at 3,3>>

Depending on the “rs” implementation, camlp4 might produce:

deep embedding (Instance of type dRegion)

Intersect (Translate (10,10)(Scale (20,20) Square)) (Union (Scale (10,10) Circle) (Translate (3,3) (Scale (2,2) Square)))

Typically metaprogramming

- Relies on staged interpretation
- Is based on quotes
- Handled at runtime (of the meta-interpreter)
- Reuses the target language compiler
- Guarantes well-typedness
- No code inspection
- Able to access IO during quote processing
- Homogeneous, generator and generated languages are the same
- CSP, cross-stage persistance, possible to embed values that are outside quotations directly into quoted terms (e.g. antiquotations)