Domain-Specific Languages

Klaus Ostermann

University of Marburg, Germany

Outline

Outline

A new chapter in the PLT course

- So far we have mainly studied general-purpose PL mechanisms.
- We have learned a lot about language design.
- For the remainder of the course we will take a look at the related area of domain-specific languages.
- We will start with some hands-on practical stuff.
- Later we will see how several of the techniques you learned are useful for DSL development.

What is a DSL?

- Typical definition: “Languages tailored to a specific application domain”.
- This is quite vague, hard to specify precisely.
- Typical examples: HTML, Make, LaTeX, SQL, BNF.
- Borderline examples: Cobol, Perl, Fortran, ...
- Once a language is sufficiently rich, one can encode everything into it (*somehow*).
- My own (equally vague) take: A language whose words describe concepts in a specific domain.
- Domain-specificity is a matter of degree.
- DSLs can also be seen as software product lines.

Why DSLs?

- Common wisdom holds that DSLs...
  - “are an order of magnitude productivity improvement over GPLs”
  - “can use domain-specific notation, possibly visual”
  - “are easier to learn/read/write by non-programmers”
  - “lead to programs (models) that are more ‘high-level’ than GPL programs”
  - “can be compiled to different ‘platforms’ (MDD)”
- Many of these wisdoms implicitly assume a certain style of DSL development and implementation.
- We will look at DSLs from a broader point of view.

Programming is Language Design

- When we program we define new names and give them a meaning.
- Every defined name enriches the language that is available.
- A programming language is in this sense not a fixed language but a meta-language to design languages.
- A library is a DSL, and a program using that library is a program written in that DSL.
  - The program could use multiple libraries - we'll talk about that later.

"We should think of language design as a pattern for language design, a tool for making more tools of the same kind" - Guy Steele, 'Growing a Language'

What is a language, anyway?

- Language theory says a language is a set of words or sentences over an alphabet.
- In programming, it is typically understood to be a language *and* a corresponding semantics.
- Languages are often infinitely big and contain (are supersets of) other languages.
  - For example, there is an infinite set of different Java programs.
  - The language denoted by the regular expression `[a-zA-Z0-9_\n]*` contains Java, SQL, Make, English and pretty much every other written language.
- The infinity of PLs is the foundation of their ability to grow.
- A PL contains an infinite number of languages.

What is a language, anyway?

- A more general form of containment is if a language can be *represented* in another language.
  - meaning a *structure-preserving* embedding, i.e., a compositional injection function.
  - That's often good enough – the remaining differences are not essential (*to a programmer*).
- Almost any GPL has a universal, tree-like syntax (s-expressions in Lisp/Scheme, method calls in Java/C#/...)
- Any (context-free) DSL can be syntactically embedded into a GPL with universal syntax.
- A GPL even contains itself as a sublanguage.
- A PL is typically not defined by a context-free syntax, however.
  - e.g., static type checking

.

From syntax to semantics

- Why don't we write all DSLs as libraries, then?
  - The question is whether these programs can be given the right meaning!
  - We come back to this question later.

“Typical” DSL scenarios

- Programming *is* (domain-specific) language design, hence there are no more “typical” use cases for DSLs than “typical” use cases for programming.
- The library is the most common and successful form of DSL.
- There are typical scenarios where people develop DSLs with specific (non-library) DSL implementation techniques, though.
- We’ll talk about those in the following slides.

“Typical” DSL scenarios

- Scenario: Predefined Notation
  - Requirement: Support a notation as close to the notation common in a domain.
  - Maybe preexisting code in that notation is available and must be understood by the DSL implementation.
  - Examples: BNF, VHDL, XML
- Scenario: Restricted Expressiveness
  - Requirement: Tight control over what DSL programs can do to, say,
    - make the programs more amendable to (domain-specific) optimization and static analysis, model checking, ...
    - guarantee invariants such as termination, security, side-effect-freeness
    - ease learning and understanding the DSL
  - Examples: Finite automata, regexps, CFGs, SQL, XPath

“Typical” DSL scenarios

- Scenario: Repetitive Code
  - The code contains patterns, but these patterns cannot (easily) be abstracted over within the language.
  - Small-scale examples: Swapping two variables in Java, infrastructure for Visitor pattern.
  - Large-scale examples: Preparing code for persistence, distribution, synchronization, integration with legacy code/apps.

Outline

Basic DSL implementation approaches

- Interpreter
  - Extensible interpreter
- Compiler
  - Stand-alone
  - Preprocessing, Macros
  - Extensible compiler
- Embedding
  - Polymorphic Embedding
- A mixture thereof
- We call approaches that represent DSL programs as abstract syntax trees *syntactic* (Interpreter, Compiler).
- We call approaches that represent DSL programs as semantic objects *semantic* (Embedding).

Talk is silver, code is golden!

Outline

Scalability

- Scalability denotes the degree to which the basic architecture of a system remains stable when the system grows.
- There are many dimensions of architecture, growth, and hence scalability.
  - Deployment architecture, Testing architecture, **Software architecture**
  - “Transactions per second”, **Complexity of the requirements/source code, LoC**
- We will concentrate on the software architecture dimension.

Requirements for Scalable DSLs

- Composability
  - Hierarchical
  - Parallel
  - With GPL
- Regularity
- DSL is a proper abstraction
  - Traceability of errors
    - Static
    - Dynamic
  - Not necessary to understand DSL implementation
  - Compositional reasoning
- Efficiency
  - Little interpretative overhead
  - Domain-specific optimizations
- Polymorphism

Requirements for Scalable DSLs: Composability

- The most basic strategy to deal with complexity is divide&conquer.
- Applied to DSLs, it means that it must be possible to compose and decompose DSLs.
- One man's end-user application is another man's component in a bigger system.
- We need a component-oriented style of DSL development where
  - DSLs can be defined in terms of other DSLs (hierarchical).
  - Independent DSLs can be composed to form new DSLs (parallel).
  - DSLs can be combined with GPL programs.

Evaluation: Composability

- Composability in stand-alone DSL compilers or extensible GPL compilers is effectively nil.
  - Hierarchical composition requires quite complex build process.
  - Parallel composition does not work, because code generators or metaprograms generally don't compose.
- Better composability if compiler architecture forces locality (macros, preprocessing)
  - DSLs can be used together inside GPL as long as DSLs do not need to interact directly.
  - Composability improved by notions such as macro hygiene.
- The same remarks apply to interpreters.
- Composability is where embedding shines!
  - All kinds of composition are trivially possible; we know how to use libraries together.
- All syntactic approaches share a common composability limitation:

Inherent composability limitations of syntactic abstraction

- With syntactic abstraction, composability of DSLs is inherently limited.
- The “curse of syntax”: A composite expression in a DSL must be assembled from syntactic objects in the DSL.
  - Let $\texttt{c}_A$ be a constructor for composite expressions in DSL *A*. Let $\texttt{t}_B$ be an expression in DSL *B*.
  - Then something like $\texttt{c}_A\texttt{(t}_B\texttt{,t'}_B\texttt{)}$ is not possible because $\texttt{c}_A$ expects syntactic *A* objects.
- In contrast, with semantic abstractions, $\texttt{c}_A$ is a constructor that expects semantic objects that might stem from anywhere and are not necessarily constructed by *A* syntax.

Requirements for Scalable DSLs: Regularity

- Regularity denotes the degree to which the “look-and-feel” of a layer architecture is the same on each layer.
  - Example: Unix file/directory systems (regular)
  - Example: Nesting structure in a Java application - methods, classes, packages (irregular)
- A scalable architecture must be regular.
  - Irregular architectures need new technologies on each layer.
  - Problems that have been solved on one layer must be solved again on other layers.
  - Software comes in sizes of so vastly different orders of magnitudes that irregular architectures are too expensive.
- In the DSL context, regularity means regularity of a layer architecture of DSLs.

Evaluation: Regularity

- Interpreters can be stacked, but it makes things quite complicated and slow (at run-time).
- Compilers can be stacked, but it makes things quite complicated and slow (at compile-time).
- Difficult/impossible to achieve with extensible interpreters/compilers.
  - But see more disciplined approaches such as Stratego or ableJ
- Embedding shines: A library using other libraries is a library.
- Regularity becomes more interesting in connection with parallel composition.

Requirements for Scalable DSLs: DSL is proper abstraction

- Two facets:
  - Abstraction in terms of information hiding.
  - Abstraction in terms of DSL entities being first-class.
- Both are important for scalability
  - because information hiding is *the* foundation of components.
  - because otherwise the illusion of being an abstraction will break sooner or later.

Requirements for Scalable DSLs: DSL is proper abstraction

- DSL that requires programmers to understand generated code is not a proper abstraction.
  - e.g. having to fill in code templates destroys all abstraction
- Can the code be understood compositionally? Does a non-trivial notion of substitutability exist?
- Are error messages (both static and dynamic) in terms of the DSL?
  - Or does one have to understand the implementation instead?
  - If no, then the abstraction breaks down again.

Evaluation: Abstraction

- Whenever DSL abstractions are “compiled away”, the abstraction barrier will break at some point.
  - Dynamic error messages will invariably break it, difficult to trace error messages to their cause.
  - The situation is even worse if the DSL compiler may generate code that does not pass the GPL compiler.
- Interpreters shine regarding domain-specific error messages.
- The situation is better with embedding because the error messages will at least be modular.
  - Difficult to generate domain-specific error messages, though
- When generated code must even be extended (such as generation of code templates), the last illusion of abstraction breaks
  - regardless of whether the extension is invasive or not.

Abstraction: The Meta-Indirection

- Scenario 1: You tell your friend Joe how to cook your favorite meal: “First, take the rice. Cook for 20min...”
- Scenario 2: You can’t tell Joe directly. You have to tell Jack, and he’ll tell Joe.
  - However, Jack is stupid. He cannot understand the meaning of recipes.
  - Instead you have to explain to him how he should tell Joe about it. “First you say, ‘First, take the rice’. Then make a little break and lower your voice. Continue by saying ‘Cook for 10min’ ...”
    - Note the nested quotes
    - Maybe Jack is so stupid that you have to give the phonetic spelling ‘First, take the rice’ instead.
    - ...or its wave form...

Abstraction: The Meta-Indirection

- Obviously, scenario 2 makes the situation much more complicated.
  - We want to describe the thing, and not describe a description of the thing!
  - Imagine what happens when you say “mice” instead of “rice”.
    - Jack won’t notice the error, but when he tells Joe you are gone.
- But this is essentially what happens when you express the meaning of things via code to be generated rather than saying directly what they mean.

Requirements for Scalable DSLs: Efficiency

- Obviously the efficiency of a DSL can hinder the scalability of an application.
- In particular, what about interpretative overhead and domain-specific optimizations?

Evaluation: Efficiency

- Compilers shine: No interpretative overhead, domain-specific optimizations possible, direct generation of efficient code.
- Domain-specific optimizations also possible with interpreters, but suffer from interpretation overhead.
- Embedded DSLs do not have interpretation overhead and makes them more amendable for the optimizations of the underlying host language compiler, but compositionality makes domain-specific optimizations difficult.

Requirements for Scalable DSLs: Polymorphism

- A component-oriented approach to DSLs requires flexible binding of DSL programs to their meaning
- ...such that DSL definitions can also be componentized.
- Example: DSLs defined as libraries or macros are typically not polymorphic.
- Example: DSLs approaches that work on ASTs are typically polymorphic.

Evaluation: Polymorphism

- AST-based approaches can easily support polymorphism through multiple interpreters/compilers.
- Usually not supported by macros or other forms of preprocessing.
- Embedded DSLs are typically not polymorphic.
- Polymorphic DSLs are (surprise, surprise) polymorphic.

Scalability Evaluation

	Interpreter	Compiler	Embedding	Polym. Emb.
Composability	O	–/O	+	+
Regularity	–	–	+	+
Abstraction	O	–/O	+	+
Efficiency	–/O	+	O	O
Polymorphism	+	+	–	+

Polymorphic Embedding for the win?

- This looks pretty good for (polymorphic) embedding. Why doesn't everybody use it, then?
- Embedding depends heavily on the expressiveness of the host language.

Reasons why embedding may not work in some host language

- Language entities not being first-class, not (easily) possible to abstract over patterns
  - Example: Classes in Java
- Semantic abstraction mechanisms not powerful enough
  - Example: Cross-cutting concerns such as persistence or logging
- Wrong argument evaluation/transfer regime
  - Example: Measure time it takes to evaluate an expression
- Too verbose
  - Example: Java
- Too inefficient
  - Cf. automata example
- Type system too weak
  - E.g. zipN

Outline

Conclusions

- Programming is language design, hence DSLs are every programmer's 'bread and butter' business.
- DSLs can be realized in many ways, each with its own set of advantages/disadvantages.
- When assessing a technology, think about composability, regularity, polymorphism etc.
- Distrust technologies that require large tool chains because the underlying 'abstractions' are rarely real abstractions.
- My theological agenda: Software development is about the design of good abstractions and not about 'using tools'. The engineering metaphor is wrong!