A Multi-layered Domain-specific Language for Stencil Computations

Christian Schmitt

Hardware/Software Co-Design, University of Erlangen-Nuremberg

SPPEXA-Kolloquium, Erlangen, Germany; July 09, 2014
Challenges for Software Development in Computational Science and Engineering

Current Situation

- **Hardware**: HPC clusters are massively parallel and heterogeneous
  - Parallel intra-core, intra-node, and inter-node
  - Increasing heterogeneity
- **Applications**: Become more complex with increasing computation power
  - More complex (physical) models
  - Code development in interdisciplinary teams
- **Algorithm**: Solvers are general ideas that need specialised implementation and parameter settings
  - Components and parameters depend on type of problem, grid, hardware, . . .

⇒ Software development has to address these issues!
Challenge: The 3 P’s

Performance
- Portable: high performance on different target hardware
- Competitive: performance comparable to hand-written code

Portability
- Support different target architectures from the same algorithm description
- Support different target languages and technologies from the same algorithm description

Productivity
- Algorithm description at a high level
- Hide low-level details from the user
Idea: Abstraction of Solution Implementation

Separate the solver algorithm from its implementation to allow end user to

- Focus on solving his problem
- Rapidly evaluate different solution approaches
- Easily run solvers on a large scale of hardware platforms with near-optimal performance
Idea: Abstraction of Solution Implementation

Separate the solver algorithm from its implementation to allow end user to

- Focus on solving his problem
- Rapidly evaluate different solution approaches
- Easily run solvers on a large scale of hardware platforms with near-optimal performance
Idea: Abstraction of Solution Implementation

Separate the solver algorithm from its implementation to allow end user to

- Focus on solving his problem
- Rapidly evaluate different solution approaches
- Easily run solvers on a large scale of hardware platforms with near-optimal performance
Outline

Motivation
Current Situation and Challenges
Abstraction of Solution Implementation

Basics of Domain-Specific Languages
Domain-Specific Language (DSL) Implementation Technologies
Language Design Guidelines

The ExaStencils DSL
General Remarks
Multi-layered Structure

ExaStencils Framework
Architecture
Examples

Summary and Outlook
DSL Definitions

“Domain-Specific Languages: An Annotated Bibliography”, Arie van Deursen, Paul Klint, und Joost Visser:

“A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.”

“Domain-Specific Languages“, Martin Fowler:

“Domain-specific language: a computer programming language of limited expressiveness focused on particular domain.”
Goals of a DSL

**Domain-specific**: provide only expressions relevant to the topic

**Orthogonal**: one single way of specifying something

**Expressive and compact**: describe relevant constructs with few statements

**Abstract**: work on a high-level point of view

**Adaptable**: support complex things

**Adaptable**: employ terms and concepts of the domain

**Regular**: all terms should follow the same syntax and ideas

**Well-defined**: non-ambiguous and easy to understand

→ **Talk about what should be computed, not how.**
   (declarative vs. imperative)
Two Approaches to Creating DSLs

Internal / embedded DSLs

- Utilize a general-purpose programming language (host language)
- **Extension** or restriction of the host language (or both at the same time)
- Extensions possible in form of libraries (e.g., data types, objects, methods), annotations, macros, etc.
- Same syntax as host language and usually the same compiler or interpreter

External DSLs

- Completely **new defined** programming language
- More flexible and expressive than internal DSLs
- Syntax and semantics defined freely, but often related to existing languages
- Higher design effort, but supporting tools exist
- Potential to create a powerful semantic model as intermediate representation (IR)
DSL Implementation Technologies

Language Workbenches

- Integrated development for reading input files, matching rules, transformations
- Ability to generate DSL IDE
- Example: Spoofax

Meta-Programming Languages (MPLs)

- Generation of programs via other programs using general-purpose program transformation systems
- Example: Rascal MPL

Custom approaches

- Manual implementation of parsing, transformations, etc.
- Examples: C++ with Flex/Bison, Scala
Methodology of DSL Implementation Technologies Review

General idea

- Take a fixed amount of time to spend on each technology: 3 weeks
- Set goal: Transformation of a simple DSL to C++ code: Generate simple numerical solver for Poisson’s equation
- Weighted rating of technologies according to pre-defined criteria

Work flow

1. Define semantics of DSL program (input)
2. Define expected output: manually write example solver that is to be generated
3. Think about needed transformations to convert input into output code
4. Implement parsing, transformations and output in each technology
Review Results: Total Points per Technology

max. core points: 60; max. environment points: 24

---

Language Design Guidelines

Identify users

Identify uses

Appropriate representation

Language realisation

Use domain concepts

Re-use existing languages

Language content

Focus on essentials

Avoid redundancy

Concrete syntax

Be descriptive

Abstract syntax

Align abstract & concrete syntax

Enable modularity

Stay consistent

Design Guidelines

Language purpose

Focus on essentials

Avoid redundancy

Concrete syntax

Be descriptive

Abstract syntax

Align abstract & concrete syntax

Enable modularity

Stay consistent

Design Guidelines

Language purpose

Focus on essentials

Avoid redundancy

Concrete syntax

Be descriptive

Abstract syntax

Align abstract & concrete syntax

Enable modularity

Stay consistent

Language Design Guidelines

It’s all about simplicity!

The ExaStencils DSL
Goals of the ExaStencils DSL

Three different types of users with individual expectations:

**Natural scientists**
- Care about *effectivity* of solving the problem
- Little knowledge of underlying methods
- No requirements to override compiler-chosen decisions

**Mathematicians**
- Care about *applicability* and *convergence* of mathematical models
- Little interest in implementation specifics
- Want to extend/override compiler-chosen decisions with custom multigrid components

**Computer scientists**
- Care about *performance and software engineering* approach
- Little interest in the mathematical problem
- Have to understand of the compiler and its decision process
ExaStencils DSL

- Multi-layered structure
- Top-down approach: From abstract to concrete
- Very few mandatory specifications on one layer → room for decisions on lower layers
- Decisions may be taken by user via specification in DSL
- Mandatory specifications include
  - Shape and size of computational domain
  - Continuous equation
- External DSL
  - Better reflection of extensive and radical ExaStencils approach
  - Enables greater flexibility of different layers
  - Eases tailoring of DSL layers to users
Multi-layered DSL Structure

Different layers of DSL tailored towards different users and knowledge:

1. Continuous Domain & Continuous Model
2. Discrete Domain & Discrete Model
3. Algorithmic Components & Parameters
4. Complete Program Specification
Multi-layered DSL Structure

Different layers of DSL tailored towards different users and knowledge:

- Continuous Domain & Continuous Model
- Discrete Domain & Discrete Model
- Algorithmic Components & Parameters
- Complete Program Specification

Natural scientists

Hardware Description
Multi-layered DSL Structure

Different layers of DSL tailored towards different users and knowledge:

- Continuous Domain & Continuous Model
- Discrete Domain & Discrete Model
- Algorithmic Components & Parameters
- Complete Program Specification

Mathematicians

Hardware Description
Multi-layered DSL Structure

Different layers of DSL tailored towards different users and knowledge:

- Continuous Domain & Continuous Model
- Discrete Domain & Discrete Model
- Algorithmic Components & Parameters
- Complete Program Specification

Computer scientists

Hardware Description
Multi-layered DSL Structure

Continuous Domain & Continuous Model (Layer 1)

- Size and structure of computational domain
- Variables
- Functions and operators (pre-defined functions and operators also available)
- Mathematical problem

```plaintext
1 Domain d = UnitCube
2
3 Function f = 0
4 Unknown solution = 1
5 Operator Lapl = Laplacian
6
7 PDE pde { Lapl(solution) = f }
8 PDEBC bc { solution = 0 }
9
10 Accuracy = 7
```
Multi-layered DSL Structure

Discrete Domain & Discrete Model (Layer 2)

- Computational domain into fragments (e.g., triangles)
- Variables to fields
  - Specification of data type
  - Selection of discretised location (cell based or node based)

Transformation of energy functional to PDE or weak form

```java
Fragment f1 = UnitCube
Discrete_Domain d { ... }
Field<Double, 1>@nodes f
Stencil<Double, 1, 1, FD, 2>@nodes Lapl

// ....
```
Multi-layered DSL Structure

Algorithmic Components & Parameters (Layer 3)

- Multigrid cycle type
- Multigrid components (e.g., selection of smoother)
- Definition of operations on sets (parts of the computational domain)
- Operations in matrix notation

1. **Iteration** smoother \( u = Mu + Nf \}
2. **Set** \( s1 = \{0, 0\} \ldots \{N, N\} \), \([+=1, +=1]\)
3. \( u(s1) = A(s1, s1) \cdot u(s1) \)
4. **Set** \( s2 = (\{0, 0\} \ldots \{N, N\}) \), \([+=2, +=2]\)
5. **Set** \( s3 = \{0:2, 0\} \ldots \{N:2, N\} \), \([+=2, +=2]\)
6. /// ...
Multi-layered DSL Structure

Complete Program Specification (Layer 4)

- Complete multigrid V-cycle, or
- Custom cycle types
- Operations depending on the multigrid level
- Loops over computational domain
- Communication and data exchange
- Custom output data formats

```
1  def V @(1 to 6) () : Unit { 8  def Application() : Unit {
2    repeat up 3 {  GaussSeidel 9    var res0 : Real = sqrt (
3      @(current) () }   L2Residual @6 () )
4      Residual@(current) () 10     // ...
5      Restrict@(current) () 11     repeat up 10 {
6        // ... 12     V @6 ()
7      } 13     } // ...
8  } // ...
```
Multi-layered DSL Structure

Target Hardware Description

- Availability of compute units (e.g., CPUs, GPUs, FPGAs) and capabilities
- Memory and cache architecture
- Cluster nodes (e.g., number of cores, types of available accelerators, available software)
- Complete cluster (e.g., number of nodes, interconnect topology and characteristics, I/O storage specifications)

```plaintext
Hardware {
  MPI {
    name "my_cluster"
    id HYBRID_CLUSTER
    nodes 32
    components {
      XEON_CPU[2]
      GTX_680_GPU[2]
    }
  } // ...
```
ExaStencils Framework
**ExaStencils Concept**

- **End-user**
- **Domain expert**
- **Mathematician**
- **Software specialist**
- **Hardware expert**

**DSL program**

**Discretisation and algorithm selection**

**Software component selection via SPL**

**Polyhedral optimisation**

**Code generation**

**Tuning towards target hardware**

**ExaStencils Compiler**

**Exascale C++**
ExaStencils Framework

Implementation details

- Approach towards a Scala-based Meta-Programming Language (MPL)
- Specialised data structures for each DSL layer
- Domain-knowledge modeled as compiler-wide accessible module
- A central instance keeps track of program state changes: `StateManager`
- Functionality separated into namespaces, e.g.,
  - `exastencils.core`: Log functionality, `StateManager`, compiler settings
  - `exastencils.core.collectors`
  - `exastencils.datastructures`: Annotations, program state duplication, `trait Strategy`, `trait Transformation`
  - `exastencils.datastructures.{l1, l2, l3, l4, ir}`
  - `exastencils.parsers`
  - `exastencils.prettyprinting`
ExaStencils Framework

Transformations

- Are grouped together in strategies
- Are atomic – either applied completely or not at all
- Are applied to program state in depth search order
- May be applied to a part of the program state
- Carry an identifier

Strategies

- Are applied by the StateManager
- Carry an identifier
- A standard strategy for sequential execution of transformations is provided
- Custom strategies possible
ExaStencils Framework

Transition between layers

- Transition from one layer to another via transformations
- Step-wise from one layer to the next
- Domain knowledge needed for each transition (e.g., which discretisation or which smoother to choose)
- Transitions are atomic – no mix of nodes of different layers

Intermediate Representation (IR) Layer

- Layer in which most optimisations take place
- Progress is made by application of many strategies with a single and narrow purpose
- Polyhedron model to be used for partitioning of computational domain
- Can be prettyprinted
Example transformations (IR)

```javascript
var s = Strategy("example standard strategy")

// replace constant '1' under a certain node with '3'
s += Transformation("t1",
    { case x : Constant if(x.Value == 1)
      => Constant(3)
    }, someProgramStateNode)

// rename all variables to 'j'
s += Transformation("t2",
    { case x : Variable
      => Variable("j", x.Type)
    })

// duplicate all methods
s += Transformation("t3",
    { case x : FunctionStatement
      => List(x, FunctionStatement(
          x.returntype, x.name + "_", x.parameters, x.body))
    })

s.apply // execute transformations sequentially
```
Summary and Outlook
Summary and Outlook

Discussed in this talk
- Idea and Definitions of DSLs
- Language Design Guidelines
- Multi-layered DSL for ExaStencils
- Implementation details

Future work
- Improve transformation framework
- Finalisation of language specifications on different layers
- Improve transformations of more abstract layers to lower layers
- Domain knowledge as a base for automated decision making and optimisation
- Optimise generated code
Thanks for listening. Questions?