An Evaluation of Domain-Specific Language Technologies for Code Generation

Christian Schmitt‡, Sebastian Kuckuk†, Harald Köstler†, Frank Hannig‡, Jürgen Teich‡

‡Hardware/Software Co-Design, †System Simulation, Friedrich-Alexander University Erlangen-Nürnberg (FAU)

ICCSA 2014, Guimaraes, Portugal; July 01, 2014
Challenges for Software Development in Computational Science and Engineering

Current Situation

- **Hardware**: Modern HPC clusters are massively parallel and heterogeneous
- **Applications**: Become more complex with increasing computation power
- **Algorithm**: Solvers are general ideas that need specialised implementation and parameter settings

=> Software development has to address these issues!
Idea: Abstraction of Solution Implementation

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

- Focus on solving their mathematical problems
- 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 users to

- Focus on solving their mathematical problems
- 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 users to

• Focus on solving their mathematical problems
• Rapidly evaluate different solution approaches
• Easily run solvers on a large scale of hardware platforms with near-optimal performance
Domain-Specific Language (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.”

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

Internal / embedded DSLs

- Utilise general-purpose programming languages (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 often same compiler or interpreter

External DSLs

- Completely newly defined programming languages
- More flexible and expressive than internal DSLs
- Syntax and semantics defined freely, but often related to existing languages
- Potential to create a powerful semantic model as intermediate representation (IR)
Basics & Technologies
Basics

Goal

Build a **textual external DSL** for solving PDEs via the multigrid method.

Lexer

- Also called *tokenizer* or *scanner*
- Splits textual input into stream of tokens
- Output is fed into the parser

Parser

- Matches stream of tokens to rules
- Executes code depending on matched rule
- Specification of parser rules in *Grammar*
- Different parser types exist
Spoofax

- *Language Workbench* based on Eclipse platform
- Development started in 2007
- SWERL group at TU Delft, Netherlands
- License: GNU LGPL
- Specification of grammar via Syntax Definition Formalism (SDF) → scannerless parser generated automatically
- Automatic generation of a DSL IDE with syntax highlighting, code folding, bracket matching, etc.
- Transformation specification via *Stratego*:
  - Matching via rules and optional conditions
  - Context-sensitive transformations via strategies and dynamic rewrite rules
Rascal MPL

- Meta-Programming Language (MPL) based on the Java VM
- In development since late 2008
- License: Eclipse Public License (EPL)
- Developer: SWAT group at CWI Amsterdam, Netherlands
- Syntactic features based on SDF
- Generation of scannerless parser
- Transformation features similar to Stratego
- Available as plugin for Eclipse and command-line REPL
Custom Approaches

C++ based

- Utilises popular tools *Flex* for tokenizing and *Bison* for parsing
- Construction of a parse tree that is transformed to an Abstract Syntax Tree (AST)
- `boost::function` and C++ 11 lambdas and function binding for node matching

Scala based

- Object-functional language based on the Java VM
- In development since 2001 and released to the public in 2004
- License: BSD-Style
- Developer: EPF Lausanne, Switzerland
- Different lexer and parser types integrated
Methodology & Comparison
Methodology

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

Workflow

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
Criteria

Sort criteria into two orthogonal groups:

Core & environment criteria
- **Core** criteria concern the technology itself
- **Environment** criteria refer to a technology’s surroundings

Hard & soft criteria
- **Hard** criteria have to be fulfilled
- **Soft** criteria are optional goodies

Weighting is adjusted depending on group combination. Our weights are:

<table>
<thead>
<tr>
<th></th>
<th>core</th>
<th>environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>hard</td>
<td>5</td>
<td>—</td>
</tr>
<tr>
<td>soft</td>
<td>1–3</td>
<td>1–3</td>
</tr>
</tbody>
</table>

For each criterion, points in the range $[-2; 2]$ are assigned.
Some of our Criteria

Explanation: **core**, environment

- **Hard**
  - Development status, Expressivity, License

- **Soft**
  - Code structuring, Components, Concepts, Debugging, Dependencies, Developer IDE, Development activity, Development team, Documentation, **Grammar**, IDE Generation, User-base size
Results: Total Points per Technology

max. core points: 60; max. environment points: 24
## Results: Example discussion

<table>
<thead>
<tr>
<th>Criterion</th>
<th>$\omega$</th>
<th>Spoofax</th>
<th>Rascal</th>
<th>C++</th>
<th>Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core criteria:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Development status</td>
<td>hard</td>
<td>5</td>
<td>-1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

- Development status denotes the maturity of a technology
- We experienced slow-downs and a few crashes with Spoofax
- Rascal relies on hidden assumptions that are hardly/not at all documented and subject to further change by its authors
- The C++ builds on the mature *Flex* and *Bison* tools
- We did not experience any bugs or hidden assumptions in Scala
Conclusions

Technology rating

- Specialised approaches Spoofax and Rascal loose core points for
  - Development phase
  - External dependencies
  - Debugging mechanisms

- Chicken-egg problem in environment criteria:
  Lack of documentation quantity and examples because of small user base size and vice versa

Rating methodology

- Categorisation of criteria based on our requirements
- Weights adjusted to our needs
  - adjust to your own needs!
Summary & Outlook
Summary

Discussed in this talk

• Idea of DSLs
• Internal / external DSLs
• Four approaches to the creation of external textual DSLs
• Methodology for evaluation of DSL implementation technologies
• Subjective comparison of four technologies at one point in time

→ Take-home message: DSLs are fun to work on!

Future work

• Language Definition for our DSL
• Transformation framework and supporting infrastructure
• Optimisation of the generated programs
Thanks for listening. Questions?