Program generation for schema-based, typed data access

Ralf Lämmel
Software Engineer
Facebook, London

Program generation —
A use case at Facebook

- Purpose of generation: **typed data access** ("O/R mapping" et al.)
- Meta = Object language = **Hack** (not to be confused with PHP)
- Input for generation: **schemas** (or mapping definitions)

Generate access code from schema

```
class UserSchema extends NodeSchema {
  protected function getFields(): Map<string, INodeField> {
    return Map{
      'Name' => string_field('name')
        ->description('Full name of the user')
        ->example('John Smith'),
      'Birthdate' => timestamp_field('birthdate')->optional(),
      'Gender' => int_enum_field('gender', UserGender::class),
    };
  }
}
```

We illustrate the use case with open-source software.

```
class UserNode extends Node {

  public function getName(): string {
    return $this->data['name'];
  }

  public function getBirthdate(): ?int{
    return $this->data['birthdate'] === 0 ? null : $this->data['birthdate'];
  }
}
```

https://code.facebook.com/posts/1624644147776541/writing-code-that-writes-code-with-hack-codegen/

The underlying codegen library

For example, this:

```
hack_builder()
  ->startForeachLoop('$users', '$id', '$user')
  ->startIfBlock('$id === $search')
  ->addReturn('$id')
  ->endIfBlock()
  ->endForeachLoop();
```

. . . generates this code:

```
foreach ($users as $id => $user) {
  if ($id === $search) {
    return $id;
  }
}
```

We illustrate the use case with open-source software.

https://code.facebook.com/posts/1624644147776541/writing-code-that-writes-code-with-hack-codegen/

Signing of generated code

```
/**
 * This file is generated. Do not modify it manually!
 * Run php ./scripts/generate_code.php to regenerate
 *
 * @generated SignedSource<<d6168d52d82d350d4907c1e835f6f2f5>>
 */
```

Key ideas:

- Generate hash code from generated code.
- Prevent diffs that change the generated code.

We illustrate the use case with open-source software.

https://code.facebook.com/posts/1624644147776541/writing-code-that-writes-code-with-hack-codegen/

Further reading

- Introducing "hack-codegen"
  - https://code.facebook.com/posts/1624644147776541/writing-code-that-writes-code-with-hack-codegen/
- Open source code base for hack-codegen
  - https://github.com/hhvm/hack-codegen
- Typed data access proof of concept (some form of EntSchema)
  - https://github.com/hhvm/hack-codegen/tree/master/examples/dorm
- Docs on hack-codegen
  - http://hhvm.github.io/hack-codegen/
- Information on the "Ent" framework
  - https://github.com/facebook/dataloader

This is publicly available information
which does not perfectly match Facebook's internal approach.

Program generation **challenges**

not just necessarily within the chosen scope at Facebook

- Metaprogramming support?
- Customization of generated code?
- Integration of generation with development?
- **Complexity of generator!**
- **Size of the generated code!**
- ...

Let's look at these challenges, one by one, in terms of the aforementioned use case and its Facebook instance.

Metaprogramming support?

Staging?

- Compile- or runtime generation may be too slow.
- It may disrupt development (e.g., source-code-based tooling).
- Staged programming support was not there to start with.

Type safety?

- The generators are tested a lot.

Concrete object syntax?

- Code construction is served by fluent APIs.
- There is a lot of logic; there is less templates.

Customization of generated code

```
public function getName(): string {
  /* BEGIN MANUAL SECTION User::geName */
  return $this->data['name'];
  /* END MANUAL SECTION */
}
```

- Manual sections are removed from signature for signing.
- Signing helps with avoiding unintended changes.
- Manual sections are kept when re-generating code.

- We may need to manage partially generated code.
- Changes of defaults may be non-obvious.
- Customized code may fail to break when it should.
- Host language may limit factoring of customization.

We illustrate the use case with open-source software.

https://code.facebook.com/posts/1624644147776541/writing-code-that-writes-code-with-hack-codegen/

Integration of program generation with development

+ We generate code to test generated code.

- Generated code may be out of sync with database.
- Generated code may permit too much on database.
- Generated code may swamp version control.
- Generated code is in the face of the developer.

Complexity of generator

- Developers may face many different files per schema.
- The code generator is tied into lifecycle management.
- The codebase may get locked into large API.
- The evolution of the generator may be difficult.

Size of the generated code

- We start to have a problem, when the size of generated code is in the same ballpark as the size of the genuine code. Think of the time to ...
  - re-generate,
  - teach IDE,
  - compile,
  - run tests,
  - commit,
  - ...

Wanted!

- Improvements:
  - Generate less code!
  - Compile less code!
  - Version-control much less code!
  - Don't think so much in terms of code!
  - Better separate generated and non-generated code!
- Preservation:
  - Data access is typed!
  - Provide IDE support such as code completion!
  - Enable intuitive customization of generated code!

Get in touch, if interested.

Towards a relatively systematic literature survey on program generation

Questions of interest — **Broad category**

- Does the approach address
  - a **case study** (on program generation), or
  - **language support** (for program generation), or
  - an **application domain** (of program generation), or
  - something else?

In reality, approaches may combine all of these.

Questions of interest — **Application domain**

- If the approach addresses a **case study**, then
  - what is the underlying application domain?
- If the approach addresses **language support**, then
  - does it point at some application domain?
- If the approach addresses an **application domain**, then
  - which one is it and
  - how well does it relate to data access?

Questions of interest — **Language aspects**

- What is the **object language**?
- What is the **metalanguage**?
- What language support is facilitated?
- What infrastructural support (e.g., a library) is facilitated?

In reality, meta = object language
may be a common case, of course.

Questions of interest — **Generation time**

- Is the code generated at ...
  - (RT) **run time** or
  - (CT) **compile time** or
  - (BT) **build time** (i.e., ahead of compile time)?
- Does the generated code provide **guarantees** ...
  - at run time (e.g., **typed** data access) or
  - at compile time (e.g., **well**-typedness)?

Questions of interest — **Customization**

- Is **customization** of generated code facilitated, and, if so, ...
  - is customization achieved by
    - some form of specialization (e.g., subclassing) or
    - some form of code changes, and, if so, ...
      - is **round-tripping** / synchronization supported, and, if so, ...
        - by what means and to what extent, or
  - by other means and
- is such customization essential, and, if so,
  - because of which specifics (language or domain)?

Questions of interest — **Software engineering**

- How is code generation addressed by the **software building**?
- How is code generation addressed by **software testing**?
- How is code generation addressed within the **IDE**?
- What other software engineering aspects are addressed?

Questions of interest — **System design**

- Does the generated code entail ...
  - a **black/gray box** (e.g., a parser) ...
    - as a service or
    - at the command line or
- a **library** (essentially an interface) or
- a **framework** (typically requiring specialization) or
- a **template** (typically requiring some code changes)?

Venues

- **GPCE https://dblp.uni-trier.de/db/conf/gpce/**
- DSL https://dblp.uni-trier.de/db/conf/dsl/
- SAIG https://dblp.uni-trier.de/db/conf/saig/
- GTTSE https://dblp.uni-trier.de/db/conf/gttse/
- SLE https://dblp.uni-trier.de/db/conf/sle/
- PEPM https://dblp.uni-trier.de/db/conf/pepm/
- ICFP https://dblp.uni-trier.de/db/conf/icfp/
- POPL https://dblp.uni-trier.de/db/conf/popl/
- PLDI https://dblp.uni-trier.de/db/conf/pldi/
- ...

GPCE 2017 papers

[...]	Case study	Language support	Application domain	Software engineering	Time	Customization	Typed data access
HSpiral	◦	◦	◉	Search	CT	◦	
CaVa	●		◉	Multi-lingual & distributed	BT		●
Guix	◦	◦	●		RT		
PAX	●	●	●	Synthesis	BT		
QSR	◦	●	◦	Library optimization	CT	◦	
MetaML		◉		Formal semantics	RT		
Haskino	◦	●	◦	Debugging	BT		
Silverchain	◦	◉		Usability of APIs	BT		
SpiralS	●	◦	●	High performance	BT		
Control		◉		Type soindness	RT		
Tensor	●	●	◉	Reusability	BT		

[HSpiral] A Haskell Compiler for Signal Transforms (GPCE'17)

- *Domain*: signal transforms (FFT)
- *Meta*: Haskell
- *Object*: Haskell, C, deep embedded DSLs
- *Concepts*: type classes, monads, GADTs, type families, index types, rewriting, compilation (to C), deep embedding, quasi-quotation, search (for optimization)

[CaVa] Automatic Generation of Virtual Learning Spaces Driven by CaVaDSL: An Experience Report (GPCE'17)

- *Domain*: virtual learning spaces for cultural heritage
- *Meta*: Java
- *Object*: scripts, SPARQL, HTML, CSS, PHP, ...
- *Concepts*: external DSL, multi-lingual and distributed target

[Guix] Code Staging in GNU Guix (GPCE'17)

- *Domain*: OS configuration and system administration
- *Meta*: Scheme
- *Object*: embedded DSLs for package definitions and build actions
- *Concepts*: staging, macros, S-expressions

[PAX] Parser Generation by Example for Legacy Pattern Languages (GPCE'17)

- *Domain*: parsing for legacy languages
- *Meta*: C# (for metamodel inference and C# generation)
- *Object*: metamodels (for patterns), C#
- *Concepts*: pattern synthesis (grammar inference), finite state machines

[QSR] Quoted Staged Rewriting: A Practical Approach to Library-De ned Optimizations (GPCE'17)

- *Domain*: stream fusion
- *Meta*: Scala
- *Object*: Scala
- *Concepts*: staging, quasi-quotation, rewriting, macros, CPS, inlining

[MetaML] Refining Semantics for Multi-stage Programming (GPCE'17)

- *Domain*: None (run-time code generation by staging)
- *Meta*: MetaML
- *Object*: ML
- *Concepts*: explicit substitutions, structural operational semantics

[Haskino] Rewriting a Shallow DSL using a GHC Compiler Extension (GPCE'17)

- *Domain*: embedded systems / microcontrollers
- *Meta*: Haskell
- *Object*: C
- *Concepts*: shallow and deep embedding, debugging, firmware interpretation, translation, monads, shallow-to-deep transformation, compiler plugin

[Silverchain] Silverchain: A Fluent API Generator (GPCE'17)

- *Domain*: fluent API generation
- *Meta*: not defined (Java?)
- *Object*: grammars (input), class definitions (output)
- *Concepts*: fluent APIs, deterministic push-down automata, BNF

[SpiralS] Staging for Generic Programming in Space and Time (GPCE'17)

- *Domain*: convolution on images and FFT
- *Meta*: Scala
- *Object*: Java
- *Concepts*: lightweight modular staging (LMS), polymorphism, generic programming

[Control] Staging with Control: Type-Safe Multi-stage Programming with Control Operators (GPCE'17)

- *Domain*: None (run-time code generation by staging)
- *Meta*: None (a calculus in MetaML style)
- *Object*: None
- *Concepts*: staging, control operators (shift0, reset0), computational effects, type soundness, type inference

[Tensor] Towards Compositional and Generative Tensor Optimizations (GPCE'17)

- *Domain*: Tensor computations (quantum chemistry and physics, big data analysis, machine learning, computational fluid dynamics)
- *Meta*: Undefined
- *Object*: C, custom intermediate language
- *Concepts*: intermediate language

Thanks!