Adv. Course in Programming Languages

Yukiyoshi Kameyama

Department of Computer Science, University of Tsukuba

Program Generation: Performance vs Abstraction

Papers for reports

Basic/General:

- **A Gentle Introduction to Malt-Stage Programming**, Taha, Dagstuhl Seminar, 2003.
- **GoMeta! A Case for Generative Programming and DSLs in Performance Critical Systems**, Rompf et al., SNAPL'15.

Application/Specific

- **Terra: A Multi-Stage Language for High-Performance Computing**, DeVito et al., PLDI'13.
- **Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines**, Ragan-Kelly et al., PLDI'13.
- **Functional Pearl: A SQL to C Compiler in 500 Lines of Code**, Rompf et al., ICFP'15.

Language Choice

- Quasiquotation in Scheme
- C++ template
- Template Haskell
- MetaOCaml
- Scala LMS (Lightweight Modular Staging)

Today

GoMeta! A Case for Generative Programming and DSLs in Performance Critical Systems, Rompf et al., SNAPL'15.

Survey Paper on Program Generation

- Performance-critical software
- Abstraction without regret
- Generative performance programming
- Case Study 1: Compiling queries in database systems
- Case Study 2: Parser combinators
- Case Study 3: DSL compiler framework for heterogeneous hardware
- Case Study 4: Synthesis of high-performance numeric kernels

X-critical system

X=Mission (Mission-critical system)

X-critical system

X=Safety (Safety-critical system)

X-critical system

X=Performance (Performance-critical system/software)

Abstraction vs Performance

Low-level code: C, assembly etc.

- Good: High-performance
- Bad: Unsafe (security vulnerability), less agile and less productive
- Bad: Not portable for different targets (multi-core, cluster, NUMA, GPU)

High-level code:

- Good: Abstraction (types, modules, classes ...)
- Bad: Tend to be inefficient, Abstraction overhead

Solution:

- Generative performance programming

Generative performance programming

Generative programming

- Another name for staged computation, or program generation

Generative 'performance' programming

- New phrase in this paper
- Program generation for high-performance (or performance-critical) code

Abstraction/Productivity vs Performance

Abstraction is great for productivity

- data, type, procedure, function, module, class abstraction.

Abstraction overhead:

```
let rec search a t =
if a.(t) = 0 then
  do_nothing
else begin
  search a (t*2);
  print_node a.(t);
  search a (t*2+1);
end
```

```
let rec search t =
match t with
| Leaf -> do_nothing
| Node(n,t1,t2) ->
begin
  search t1;
  print_node n;
  search t2
end
```


Removing Abstraction Overhead

Optimizing compilers remove abstraction overhead by inlining etc., but ...

- Yet, it is often sub-optimal (not optimal)
- Compilers do not know each domain/architecture.
- Often we don't have time to write optimizing compiler for DSLs.

DSLs and generative programming help the situation.

Generative Performance Programming

- Different hardware: parallel, heterogeneous, distributed
- Applications which need high efficiency
- High-level programming languages provide more generality and abstraction

Case Study 1: database queries

Scala with metaprogramming feature (Lightweight modular staging):

```
processCSV("data.txt") { record =>
  if (record("Flag") == "yes")
    println(record("Name"))
}
```

Example:

```
Name, Value, Flag
 A     7     no
 B     2     yes
==>  "B" is printed
```


Case Study 1: database queries

```
processCSV("data.txt") { record =>
  if (record("Flag") == "yes")
    println(record("Name"))
}

class Record(fields: Array[String], schema:
   Array[String]) {
   def apply(key: String) = fields(schema
   indexOf key)
}
```

This is VERY slow than the following hand-written code:

```
while (lines.hasNext) {
  val fields = lines.next().split(",")
  if (fields(2)) == yes) println(fields(0))
}
```


Staged Interpreter is a compiler

In Scala LMS, we only have to specify dynamic/static by types.

```
(before staging)
class Record(fields: Array[String], schema:
   Array[String]) {
   def apply(key: String) = fields(schema
   indexOf key)
}

(after staging)
class Record(fields: Rep[Array[String]],
   schema: Array[String]) {
   def apply(key: String) = fields(schema
   indexOf key)
}
```

This modification means that the field argument is static and the schema argument is dynamic.

Case Study 1: database queries

By specifying 'fields' is static, we automatically get a staged version of processCSV:

```
(before staging)
processCSV("data.txt") { record =>
  if (record("Flag") == "yes")
    println(record("Name"))
}

(generator for staging)
processCSV(file: String) (yld: Record => Rep[
   Unit]) = {
  val lines = FileReader(file); val schema =
   lines.next.split(",")
  run (while (lines.hasNext) {
        val fields = lines.next().split(",")
        yld (new Record(fields.schema))
  })
}
```



```
(generated code fragment)
while (lines.hasNext) {
```

Summary of Case Study 1

Generative programming with Scala LMS:

- can generate the best code (hand-written code),
- by only specifying the static/dynamic information through types

The authors' group has succeeded in

- writing a highly efficient SQL compiler, with only 500 lines
- ...and got the best paper award in VLDB (top conference on database)

Other Case Studies

Case Study 2: Parser combinators

- cf. Hand-optimized HTTP parsers for Apache etc. (2000 lines of C code)
- Staged parser combinators for HTTP and JSON data, which have comparative (0.75 or 1.2 times faster/slower) performance with hand-written parsers.

Case Study 3: DSL compiler framework for heterogeneous hardware

- Delite: a compiler framework for embedded DSLs.

Case Study 4: Synthesis of high-performance numeric kernels

- Kernels for linear algebras, FFT (Fast Fourier Transform), filters etc.
- Re-implemented Spiral's DSLs by Scala LMS
- Uses type classes and generic programming

Summary of this paper

- High-level vs low-level programming; abstraction vs high-performance
- By eliminating abstraction overhead is the key to resolve this tension
- It can be done by **generative performance programming**
- Many success stories using the authors' Scala LMS
- The core idea of Scala LMS is 'staging by types'

Summary of my lectures

Staged computation or Program generation (or Generative Programming)

- is a key to achieve 'Abstraction without Guilt/Regret/Tears'
- can be done using types
- has a big potential to achieve high performance with high reliability

Study on programming languages

- has solid foundation by logical/mathematical theories,
- is useful in understanding programming and designing new languages and new way of computing

Report (for the first 5 weeks)

Report:

- Choose one (or more) paper from the five papers about program generation
- Write three (or more) pages of reports about the paper you choose
- Submit the report through the Manabe system by May 23rd (Tue.), 2017.

Recommended organization of the report:

- Summary of the paper
- Your evaluation/impression/thoughts on the paper
- Argue the relation between your topics and the paper (your 'topics' can be your research topics for master thesis, or other topics for part-time jobs, hobbies etc.)