Logic in Computer Software

Yukiyoshi Kameyama

Dept. of Computer Science, U. of Tsukuba

Summary

"Staging": code generation by programs (a kind of meta-programming)

- Naive approach: strings as codes
- Preproessor approach: quasi-quotation (in Lisp/Scheme), C++ template
- Native support by programming languages: MetaOCaml, Scala/LMS, (template Haskell)
- refers to type-safe meta-programming, implemented in MetaOCaml (only)

Today:

- Safety of codes in staging (in particular, MetaOCaml-style staging)
- Application of staging
- Summary

Example (1)

What is the type of the following program ?

```
<fun x -> x + 10>
```

or

```
`(lambda (x) (+ x 10))
```

```
(int -> int) code

or, ('a, int -> int) code in MetaOcaml
```


Example (2)

What is the type of the following program ?

```
<fun x -> ~x + 10>
```

or

```
`(lambda (x) (+ ,x 10))
```

```
Error: Wrong level: variable bound at level 1 and used at level
```

Why ?

```
`(lambda (x) (+ ,x 10))
```

is exactly the same as (or rewritten at the input time to):

```
(list 'lambda '(x) (list '+ x '10))
```

Namely, `x` is a free variable.

Example (2)-added

But we sometimes want to define a function now (at the code-generation time), and to use it later (when the generated code is used and executed).

```
let sqr x = x * x;;
let rec s_power = <.... sqr ....>;;
```

We define the function sqr now, and use it in future .
MetaOCaml allows it: Cross-stage persistence (CSP).

Example (2)-added (cont'd)

```
# let sqr x = x * x;;
val sqr : int -> int = <fun>

# let rec s_power n x =
  if n = 0 then x
  else if (n mod 2)=0 then .< sqr .~(s_power (n/2) x)>.
  else .< .~x * .~(s_power (n-1) x)>.;;
val s_power : int -> int code -> int code = <fun>
```

(Ignore periods before/after staging constructs).

sqr has type `int -> int`, but is used as having the type `(int -> int) code`.

Example (3)

What is the type of the following program ?

```
   fun y -> <fun x -> ~y + 10>
or
   (lambda (y) '(lambda (x) (+ ,y 10)))

  int -> ((int -> int) code)
```


Example (4)

What is the type of the following program ?

```
   fun f -> <fun x -> ~(f <x + 10>)>
or
   (lambda (f) '(lambda (x) ,(f '(+ x 10))))

  (int code -> int code) -> ((int -> int) code)
```

Example (4)-added

Is there any real application of such a program (having a complicated type) ?
Yes, we often use such patterns in code generation. (cf. Design Pattern):

```
# let eta = fun f -> .<fun x -> .~(f .<x + 0>.) + 0>.;;
eta: (int code -> int code) -> (int -> int) code = <fun>

# let rec s_power n x = ...;;
s_power : int -> int code -> int code = <fun>
(s_power 13 .<x>. would be .<x * ...>.

# let s_power13 = eta (s_power 13);;
val s_power13 : (int -> int) code = <fun>

# let p13 = .! s_power13 in p13 2;; (*compiled*)
_ : int = 8192
```


Safety of generated codes

Static safety guarantee of no syntax error, no type error and no scope error (no free variables) in generated codes:

Approach	no syntax error	no type/scope error
Strings as codes	NG	NG
Lisp Quasiquotation	OK	NG
C++ template	OK	NG
Template Haskell	OK	NG
Scala LMS	OK	NG
MetaOCaml	OK	OK

Note. Safety means "no compile errors for generated codes", and it does NOT imply the generated codes are correct.

Applications of Staging (GE-1)

A generic yet efficient algorithm [Jacques Carette 2005 "Gaussian Elimination: a case study in efficient genericity with MetaOCaml", in SCP].

- Maple: a large **commercial** computer algebra system.
- Gaussian Elimination (GE) is an algorithm on square matrices.
- Maple contains **35** different implementations of GE with parameters such as:
  - Domain of matrix elements: Z, Q, Z_i, Z[x], Q(x), Q(α), and `float`, $\cdots$ (20 different domains).
  - Fraction-free (or remainder-free) division or not.
  - Representation of matrices: array of arrays, one-dimensional array, hash table, and indexing is done in C-style or FORTRAN-style.
  - Length measure for pivoting.
  - Output choices.
  - Normalization and zero-equivalence.
- Parameter choices are not independent and more design choices: 35 different implementations, still share the same algorithm.

Applications of Staging (GE-2)

Question: is it OK to maintain all 35 implementations ?

Probably no.

- algorithm change may affect all implementations.
- some implementation may not utilize optimizations used in other implementations.
- we need abstraction (for reusability, maintainability, and reliability).

Applications of Staging (GE-3)

Question: is it enough to represent all 35 implementations by one generic program ?

No, definitely.

- Maple is a **commercial** system, and efficiency is important.

```
let ge_high findpivot swap zerobelow a m n = ...
  if (domain = int) then ...
  else if ....
  if (fraction_free) then ...
  else ...
  let new_matrix = add_row (i, j, old_matrix) in ...
```

Many places to be improved:

- Conditionals and Case analysis are bad for efficiency.
- Function calls should be eliminated by inlining (if we know which function is called statically).

Applications of Staging (GE-4)

Solution: use staging.

- We write only one (generic) program in MetaOCaml, and maintain this code only.
- By staging, we generate 35 (and possibly more) different codes using specific parameters for each choice.
- The package (commercial product) contains these codes.
- The generic program is easy to read, but bad in efficiency.
- The generated codes are hard to read, but good in efficiency.

Moreover, in the MetaOCaml-style staging, the generator (program) looks like the generated codes.

Applications of Staging (DSL interpreter-1)

Domain-specific language (DSL) is ubiquitous. (vs. General-purpose programming languages):

- database access
- parser generator (e.g., lex/yacc)
- wiki
- (according to wikipedia) DSL for life insurance policies, combat simulation, salary calculation, billing, ..

If we have enough resource (time, man-power etc.), we can develop a compiler for each DSL, but often it's not possible (and not necessary).

We usually write an interpreter of DSL.

Applications of Staging (DSL interpreter-2)

A typical DSL interpreter (written in OCaml):

```
let rec eval exp env =
  match exp with
  | Int i -> i
  | Add e1 e2 -> (eval e1 env) + (eval e2 env)
  | If e1 e2 e3 -> if (eval e1 env) then ...
...
```

interpretive overhead: if we run the power function (or any recursive function), we interpret the same function many times. (power 1000 needs to be interpreted 1000 times, rather than once.)

Applications of Staging (DSL interpreter-3)

Staging will help the situation. We can generate the code (in the general purpose language) for a program in DSL, then run it.

We only have to interpret each program once.

```
let rec eval exp env =
  match exp with
  | Int i -> <i>
  | Add e1 e2 -> <~(eval e1 env) + ~(eval e2 env)>
  | If e1 e2 e3 -> <if ~(eval e1 env) then ...>
...
```

Walid Taha 2005, "A Gentle Introduction to Multi-Stage Programming" reported 5 - 20 times speed up of staged DSL interpreter.

Applications of Staging (High Performance Code)

Aktemur, Kameyama, Kiselyov,Shan [2013], "Shonan Challenge for Generative Programming":

- Based on our technique for generating efficient codes using MetaOCaml-like staging,
- we propose a set of **challenging** programs in high-performance computing (such as computer algebra, fast Fourier transform, hidden Markov model etc.),
- and ask Staging people to provide **good** solutions for them.

Evaluation Criteria of Shonan Challenge

- efficiency
- reusability/maintainability
- safety (and possibly verifiability)
- ease of use
- and more

Still hot and active research area.

Report

これまでの授業を参考にして、以下の各項目について、自分の考えを述べよ。

- 「プログラムによるコード生成」という考え方 (MetaOCaml 風でなくてよい) について，(1) 自分の研究や趣味に関連する分野で，どのように使われているか，あるいは使われていると良いか，(2) また，コード生成のためのどのような言語，ライブラリ，プログラミング環境があれば良いとおもうか，などを，自由に書きなさい．
- 「プログラム言語の型システムの役割」について、自分がこれまで作成したソフトウェア (あるいは作成中のシステム) において、どのように使われているか，役に立ったか立たなかったか，どういう風な型システムであれば嬉しいか，など，自由に書きなさい．