Logic in Computer Software

Yukiyoshi Kameyama

Dept. of Computer Science, Univ. of Tsukuba

Week 4

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.