Adv. Course in Programming Languages

Yukiyoshi Kameyama

Department of Computer Science, University of Tsukuba

How to *stage* programs

Today's Topic

How to automatically obtain a specialized code from an ordinary program?

GenPower function

	name	input	output
ordinary function	power	x,n	x^n
specialized code	power13	x	x^{13}
code generator	gen_power	n	power$n$

$$gen\_power\ 5 \rightsquigarrow power5$$
$$power5\ 2 \rightsquigarrow 32$$
$$power5\ 3 \rightsquigarrow 243$$
$$power5\ 4 \rightsquigarrow 1024$$

gen_power is a code geneartor, and power5 is a specialized code

Quasi-quotation

In MetaOCaml, <e> is quotation, and e is anti-quotation.

```
<1+2> —> <1+2>
<1+2*3—4+5> —> <1+2*3—4+5>
< ~<1+2> * 3> —> < (1+2) * 3>
let x = <1+2> in < ~x * 3> —> <(1+2) * 3>
let x = <1+2> in < ~x — ~x> —> <(1+2) — (1+2)
   >
```

Computation rule:

```
~<e> —> e
```

Basics of Types

Types of functions:

```
let incr x   = x + 1  incr: int -> int
let foo(x,y) = x + y foo : (int * int) -> int
let goo x y = x + y  goo : int -> int -> int
```

Arrow ($\rightarrow$) represents the function type.

```
power : int -> int -> int
  power 2 5 = 32
gib : int -> int -> int -> (int*int)
  gib 3 1 1 = (3,5)
```


GenPower Function

GenPower Function (code generator for the power function):

```
let rec genp n x =
  if n = 0 then <1>
  else if (even n) then
       <square ~(genp (n/2) x)>
  else < ~x * ~(genp (n-1) x)>

genp 0 <y> -> <1>
genp 1 <y> -> < ~<y> * ~(genp 0 <x>)>
           -> < ~<y> * ~<1>>
           -> <   y  *   1 >
genp 2 <y> -> <square ~(genp 1 <y>)>
           -> <square (y * 1)>
genp 3 <y> -> <y * (square (y + 1))>
```


How to make GenPower

How to obtain GenPower from Power?

```
let rec power n x =
  if n = 0 then 1
  else if (even n) then
       square (power (n/2) x)
  else x * (power (n-1) x)

let rec genp n x =
  if n = 0 then <1>
  else if (even n) then
       <square ~(genp (n/2) x)>
  else < ~x * ~(genp (n-1) x)>
```

Quite similar: we only have to insert <> and ˜ at some suitable places.

Coloring power function (1)

Idea: Coloring by red for static, black for others

let rec *power n x* =
if n = 0 then 1
*else x * (power (n - 1) x)*

1. 0 and 1 are static. We regard power itself is static.
2. We assume n is static and x is dynamic.
3. Then n=0 and n-1 become static.
4. The conditional (if n=0 then ...) becomes static.
5. Nothing more is static (the remaining things are dynamic.)

By quoting red parts, we get a code generator:

let rec *genp n* <*x*> =
if n = 0 then 1
else < *x* * ~(*genp (n - 1)* <*x*>) >

Coloring power function (2)

Problem 1 in the previous code:

> **let rec** *genp n <x> =*
> *if n = 0 then 1*
> *else < x * ∼(genp (n - 1) <x>) >*

Here, then-branch returns an integer 1, while else-branch returns a code (program). Solution: we replace 1 by <1>, even though we can compute the value of 1 statically.

> **let rec** *genp n <x> =*
> *if n = 0 then <1>*
> *else < x * ∼(genp (n - 1) <x>) >*

Coloring power function (2)

Problem 2 in the previous code:

> **let rec** *genp n <x> =*
> *if n = 0 then <1>*
> *else < x * ∼(genp (n - 1) <x>) >*

Here, <x> cannot be an argument of a function. Solution: we replace it by y. Then <x> becomes ∼y, hence:

> **let rec** *genp n y =*
> *if n = 0 then 1*
> *else < ∼y * ∼(genp (n - 1) y) >*

with some magical code:

> **let** *genp_main n =*
> *< fun x -> ∼(genp n <x>) >*

Coloring better power function

Starting from:

```
let rec power2 n x =
  if n = 0 then 1
  else if (even n) then
       square (power2 (n/2) x)
  else x * (power2 (n−1) x)
```

We obtain the following generator by coloring:

```
let rec genp2 n x =
  if n = 0 then <1>
  else if (even n) then
       <square ~(genp2 (n/2) x)>
  else < ~x * ~(genp2 (n−1) x)>
```


Generalized Fibonacci Function

```
let rec gib n x y =
  if n = 0 then (x, y)
  else let (r1,r2) = gib (n−1) x y in
       (r2, r1+r2)
```

```
gib 0 1 1  −> (1,1)
gib 1 1 1  −> (1,2)
gib 2 1 1  −> (2,3)
gib 3 1 1  −> (3,5)
gib 4 1 1  −> (5,8)
gib 4 3 7  −> (27,44)
```

Generalized Fibonacci Function

Assumption: we know the value of n, but not that of x or y.

let rec gib n x y =
if n = 0 then (x, y)
else let (r1,r2) = gib (n-1) x y in
(r2, r1+r2)

We get a generator for Gibonacci:

```
let rec gib_gen n x y =
  if n = 0 then (x,y)
  else let (r1,r2) = gib_gen (n-1) x y in
    < (~r2, ~r1 + ~r2) >
```


Coloring power function (Summary)

- Starting from an ordinary function (e.g. power), and the static/dynamic information about its arguments,
- We can automatically derive a code genrator for it, by coloring.

Very simple analysis on programs (a modern compiler does a lot more).

But coloring is not sufficient for all the cases! (Bad news)

Slightly Different Power

When the order of arguments is reversed:

```
let rec power3 x n =
  if n = 0 then 1
  else if (even n) then
       square (power3 x (n/2))
  else x * (power3 x (n−1))
```

Impossible to color “power3” and “(n/2)”, but not “x”.

Generalize Fibonacci

Assumption: we know the values of n and x, but not that of y.

let rec gib n x y =
if n = 0 then (x, y)
else let (r1,r2) = gib (n-1) x y in
(r2, r1+r2)

The dynamic expression r2 appears in the first element of the pair, which should be static. (Problematic!)

Higher-order Function

Map is a typical higher-order function:

```
let rec map f lst =
  match lst with
  | [ ]  -> [ ]
  | h :: tl -> (f h) :: (map f tl)
in
  map (fun x -> x + 1) [3; 7; 2; 5]
==>
  [4; 8; 3; 6]
```


Staging with Higher-order Function

Suppose we know the length of *lst*, but don't know *f* and *lst* (dynamic), and we want to get:

```
let rec map_gen2 f lst =
  match lst with
  | [ ] -> [ ]
  | h :: tl -> .<(.~f .~h) :: .~(map_gen2 f tl)>.
in
  map_gen2 .<some_fun>. [.<a1>.; .<a2>.;... ]
==> .< [some_func a1; some_func s2; ...]>.
```

It is rather difficult to distinguish the two, by coloring only.

How can we obtain code generators in general?

Coloring *expressions* is NOT sufficient.

We need a better way than two colors:

- The first Power. Input: n and x, Output.
- The second Power. Input: x and n, Output.
- The first Gib. Input: n, x, y, Output: left and right.
- The second Gib. Input: n, x, y, Output: left and right (?).
- Higher-order case: a variable f may contain two colors.

Solution: coloring *types*.

Coloring Types in GenPower

```
let rec power n x =
  if n = 0 then 1
  else x * (power (n-1) x)
power : int -> int -> int
```

The type of power: `int -> int -> int`

The type of power_gen: ?

- *n* is static (red), so its type is static
- *x* is dynamic (black), so its type is dynamic
- the returned object is dynamic (black), so its type is dynamic

The type of power_gen: `int -> int -> int`

Coloring Types in GenPower

The type of power_gen: `int -> int -> int`

We write it as `int -> (int code) -> (int code)`

```
gen_power : int -> (int code) -> (int code)
   gen_power 3 <5> ==>  <5 * 5 * 5 * 1>
   gen_power 3 <2+3> ==>  <(2+3) * (2+3) * (2+3) * 1>
```

If n is dynamic, and x is static, then we have another type:

```
gen_power2 : (int code) -> int -> (int code)
```


Type Inference does the job

```
let rec gib n x y =
  if n = 0 then (x,y)
  else let (r1,r2) = gib (n-1) x y in
    (r2, r1+r2)
gib : int -> int -> int -> (int * int)
```

Assume n and x are staitc, and y is dynamic.

We want to assign consistent types for all expressions.

- (NG) gib : int → int → (int code) → (int * (int code))
- (OK) gib : int → (int code) → (int code) → ((int code)* (int code))

Type inference tells how we can make a consisten generator.

Type Inference does the job

gib : int → (int code) → (int code) → ((int code)* (int code))

```
let rec gen_gib n x y =
  if n = 0 then (x,y)
  else let (r1,r2) = gen_gib (n-1) x y in
    (r2, < ~r1 + ~r2 >)
gen_gib : int -> int code -> int code
          -> int code * int code
```

To make the argument x static (of type int), we need a wrapper function:

```
let wrapper n x y =
   gen_gib n <x> y
wrapper : int -> int -> int code
          -> int code * int code
```

Using the lifting (of MetaOCaml):

```
let x = 3 in <x + 2> ==> <3 + 2>
```


Quiz

Which one is computed statically?

```
Assumption: a, b, c are static, others are not

let rec foo a b c d e =
  if a > 0 then
   foo (a+b) (b+c) (c+d) (d-e) (e*a)
  else
   foo 3 5 (foo b c d e a) 7 9
```

Summary

"Staging": converting an ordinary function to a code generator:

- Coloring expressions sometimes work, but not always.
- Coloring types does work, and correctly detects errors.

Type inference is quite fundamental in many modern programming languages:

- ML (SML, OCaml and F#) and Haskell have built-in automatic type inference systems.
- Theories and algorithms for type inference are well studied.
- Many object-oriented programming languages (including Java and Scala) have powerful type systems.

Static Safety Guarantee

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