ソフトウェア論理
Logic in Computer Software

Yukiyoshi Kameyama

Dept. of Computer Science, Univ. of Tsukuba

Week 3

プログラム生成 (Program Generation)

How to represent programs (codes) as data ?

- Strings
- Data types for trees
- Language support for code generation (Built-in data types)

(To distinguish two kinds of programs from each other, we write "programs" for **generating** programs, and "codes" for **generated** programs.)

This choice greatly affects the quality of programs and codes. (ease of writing/understanding, reusability efficiency, reliability, etc.)

Strings as codes (1)

A standard C-program for the power function (べき乗を求める関数):

```
int power (int n, int x) {
  if (n == 1) {
    return x;
  } else if (even(n)) {
    return sqr(power(n/2,x));
  } else {
    return x*power(n-1,x);
  }
)
```


Strings as codes (2)

Suppose `n` is known now, and `x` is not known now. A generator for the power function in C-like notation:

```
string gen_power1 (int n, string xs) {
  if (n == 1) { return xs;
  } else if (even(n)) {
    return concat("sqr(", gen_power1(n/2,xs), ")");
  } else {
    return concat(xs, "*(", gen_power1(n-1,xs), ")");
  }
}
string gen_power (int n) {
  return
   concat("int power (int x) { return(",
          gen_power1(n, "x"), ");}");
}
```

assuming that `concat` does the right job.

Strings as codes (3)

Inner product of vectors in C-like notation:

```
float ip (int n, float a[], float b[]) {
  int i;
  float sum = 0.0;
  for (i = 0; i < n; i++) {
    sum += a[i] * b[i];
  }
  return sum;
}
```


Strings as codes (4)

Suppose n is known, a and b are not known. Generator for inner product:

```
string gen_ip1 (int n, int idx, string as, string bs) {
  if (idx == n) return "0.0";
  else return
     concat(as, "[", int_to_string(idx), "] * ",
            bs, "[", int_to_string(idx), "] + ",
            gen_ip1(n, idx + 1, as, bs));
}
string gen_ip (int n, string as, string bs) {
  return
   concat("float ip (int ", as, "[], int", bs, "[]) {"
          "return ", gen_ip1(n, 0, as, bs), ";", "}");
}
```


Strings as codes (5)

Sometimes, we want to generate more specialized code: Suppose n and a are known, and b is not known.

```
string gen_ip1 (int n, int idx, float a[], string bs) {
  if (idx == n) return "0.0";
  else return
     concat(float_to_string(a[idx]), " * ",
            bs, "[", int_to_string(idx), "] + ",
            gen_ip1(n, idx + 1, a, bs) );
}
string gen_ip (int n, float a[], string bs) {
  return concat("float ip (int ", bs, "[]) {"
         "return ", gen_ip1(n, 0, a, bs), ";", "}");
}
```


Strings as codes (summary)

Evaluation:

- (+) It can be done in almost all programming languages.
- (+) So, we don't have to learn more techniques.
- (-) It needs a certain (boring) rewriting of the non-generating version
- (-) It is error prone, especially when we embed codes into code ("splicing")
- (-) It is not composable; we cannot combine one generator with internal variables "x" and "y", and another generator with internal variables "x" and "z".
- (-) Sometimes (or, often) the generated codes cannot be compiled due to type errors or unbound variables.

Data types for trees as codes (1)

Lisp/Scheme has trees as primitive data ("Symbolic expression" or S-expression for short):

```
(+ 1 2)   returns 3
'(+ 1 2)   returns (+ 1 2)
(list (+ 1 2) (* 2 3)) returns 9
(list '(+ 1 2) '(* 2 3)) returns ((+ 1 2) (* 2 3))
```

Suitable for symbolic computation (mathematical formulas, logical formulas, programs, XML data, sentences in natural languages etc.)

Data types for trees as codes (2)

Power function in Scheme:

```
(define (power n x)
  (if (= n 1) x
    (if (even n)
       (sqr (power (/ n 2) x))
      (* x (power (- n 1) x)))))
```


Data types for trees as codes (3)

Generator for Power function in Scheme:

```
(define (gen_power1 n xs)
  (if (= n 1) xs
    (if (even n)
        (list 'sqr (gen_power1 (/ n 2) xs))
      (list '* xs (gen_power1 (- n 1) xs)))))

(define (gen_power n)
   (list 'define '(power x)
         (gen_power1 n 'x)))
```

Slightly better than the "strings as codes" approach.
Still splicing is problematic.

Data types for trees as codes (4)

(from the previous slide)

```
(define (gen_power n)
   (list 'define '(power x)
         (gen_power1 n 'x)))
```

Generator for Power function in Scheme using **quasi-quotation**:

```
(define (gen_power n)
    `(define (power x)
         ,(gen_power1 n `x)))
```

Can represent splicing neatly.

Quasi-quotation is like quotation, but allows splicing.

Data types for trees as codes (5)

Evaluation:

- (+) Better syntax. Ease of writing and understanding. Much less error-prone.
- (+) No overhead; runs in exactly the same speed as the one without quasi-quotation (it is just an input-macro).
- (-) Programming language (or its preprocessor) must support it.
- (-) Still not composable; we cannot combine one generator with internal variables “x” and “y”, and another generator with internal variables “x” and “z”.
- (-) Sometimes (or, often) the generated codes cannot be compiled due to unbound variables.

Language support (built-in data types) (1)

Power in OCaml (a dialect of ML):

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


Language support (built-in data types) (2)

Generator for Power:

```
let rec gen_power1 n xs =
  if n=1 then xs
  else if (even n) then
       `(sqr ,(gen_power1 (n / 2) xs))
  else `(,xs * ,(gen_power1 (n - 1) xs))

let gen_power n =
  `(fun x -> ,(gen_power1 n `x))
```


Language support (built-in data types) (2’)

Generator for Power:

```
let rec gen_power1 n xs =
  if n = 1 then xs
  else if (even n) then
       <sqr ~(gen_power1 (n / 2) xs)>
  else <~xs * ~(gen_power1 (n - 1) xs)>

let gen_power n =
  <fun x -> ~(gen_power1 n <x>)>
```

Intuitively: `<a b c>` is `'(a b c)` and `<a ~b c>` is `'(a ,b c)`
Then, we have:

```
 gen_power 3 <x>
-> < ~<x> * ~(gen_power 2 <x>) >
-> < x * ~(<sqr ~(gen_power 1 <x>)>) >
-> < x * ~(<sqr ~(<x>)>)>
-> < x * ~(<sqr x>)>
```

Language support (built-in data types) (3)

But why is it better than Lisp/Scheme ?
Support for types.

- Types give a certain reliability of generator.
- Types give a certain reliability of generated codes,
- AND it ensures “no free variables” in generated codes.

Errors:
`x + 1`, `<x + 1>`, `<3.0 + 1>` `<~x + 1>`
Ok: `<fun x -> x + 1>`, `fun x -> <~x + 1>`,
`fun x -> <fun y-> ~x + y + 1>`,

Language support (built-in data types) (4)

Type for codes

- if e is type `int`, then $< e >$ is of type `int code`.
- In general, if e has type T, then $< e >$ is of type T `code`.
- If e has type T `code`, then ~e is of type T.

Types for `gen_power1`:

```
let rec gen_power1 n xs =
  if n = 1 then xs
  else if (even n) then
       .<sqr .~(gen_power1 (n / 2) xs)>.
  else .<.~xs * .~(gen_power1 (n - 1) xs)>.
```

n is of type int, *xs* is of type int code.
the return type of the generator is int code.
then the generator has type
`int -> (int code) -> (int code)`.

Summary

- “Codes as strings” are available in most languages, but no support.
- Staged computation: Language support for code generation.

Exercises.

Assign types to the following terms (some terms do not have types.)

- `<fun x -> x+10> + 20` or
  `(+ ‘(lambda (x) (+ x 10)) 20)`
- `<fun x -> x+10> 20` or `(‘(lambda (x) (+ x 10)) 20)`
- `<(fun x -> x+10) 20>` or
  `‘((lambda (x) (+ x 10)) 20)`
- `<fun x -> ~x + 10>` or `‘(lambda (x) (+ ,x 10))`
- `fun y -> <fun x -> ~y +10>` or
  `(lambda (y) ‘(lambda (x) (+ ,y 10)))`
- `fun f -> <fun x -> ~(f <x+10>)>` or
  `(lambda (f) ‘(lambda (x) ,(f ‘(+ x 10))))`

Exercises with Answers.

- `<fun x -> x+10> + 20` . (Not typable, since we cannot add 20 to a code, since a code itself is not an integer.)
- `<fun x -> x+10> 20` (Not typable, since we cannot apply a code to a value, since a code is not itself a function.)
- `<(fun x -> x+10) 20>` (Has a type (int code) under an empty typing context.)
- `<fun x -> ~x + 10>` (Has a type `(int code)` under a typing context x :`(int code)`.)
- `fun y -> <fun x -> ~y +10>` or (Has a type `int -> (int code)` under an empty typing context.)
- `fun f -> <fun x -> ~(f <x+10>)>` (Has a type `(int code -> int code) -> (int -> int) code` under an empty typing context.)