Adv. Course in Programming Languages

Yukiyoshi Kameyama

Department of Computer Science, University of Tsukuba

No.1: Program Generation

プログラム生成 (Program Generation)

Power function in C:

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

If we use the function for fixed n (e.g. 12), for various values of x, we had better use:

```
int power12 (int x) {
  int y = x * x * x;
  int z = y * y;
  return (z * z);
}
```


Program Generation

Quite a few applications need such specialization:

- image processing (Halide)
- linear algebra kernel (Spiral, Terra etc.)
- database query (Quel, Scala LMS etc.)
- DSL in general

We want to write a program which generates such a specialized program (code).

- Program Generator
- Generated Code

Topic of this course: How can we write a program generator in a safe, easy, extensible way ?

Strings as code (1)

Terminology: we say **programs** for **generators**, and **code** for **generated** programs.

First question: how to represent code as data ?

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

Strings as code (1)

Power function (べき乗を求める関数) in the C language:

```
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 code (2)

A generator for power, assuming n is known, x is unknown.

```
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"), ");}"); }
```

gen_power(5) returns
"int power (int x) { return(x*(sqr(sqr(x))));}".

Strings as code (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 code (4)

Generator: (n is known, a and b are not known.)

```
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 code (5)

Generating more specialized code: (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 code (summary)

The "string as code" approach:

- (+) Can be done in any programming languages.
- (-) Is error prone; risk of erroneously bound/unbound variables and type errors.
- (-) Is not composable; we cannot combine two generators both of which use "x" as an internal variable.

Trees as code (1)

Lisp/Scheme has S-expressions (trees) as primitive data.

```
(+ 1 2)   returns 3
'(+ 1 2)   returns (+ 1 2)
(list (+ 1 2) (* 2 3)) returns (3 6)
(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.)

Trees as code (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)))))
```

Trees as code (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)))
```

Better than the "strings as code" approach. Splicing is still problematic.

Trees as code (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)))
```

Quasi-quotation allows splicing.

Trees as code (5)

Evaluation of "trees ad code" approach:

- (+) Better syntax. Ease of writing and understanding. Fewer errors.
- (-) Still not composable; we cannot combine one generator with internal variables "x" and "y", and another generator with internal variables "x" and "z".
- (-) Risk of run-time type errors or unbound/erroneously bound variables.

Data types as code

We can use user-defined data type instead of S-expressions:

```
type code =
 | Var of string
 | Fun of string * code
 | App of code * code
 | Plus of code * code
 | Times of code * code
```

We still make mistakes in mixing up variables.

Language support for quasi-quotation (1)

Power function 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 for quasi-quotation (2)

Generator for Power (OCaml plus quasi-quotation):

```
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))
```

Generator for Power (in MetaOCaml):

```
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)>
```


Language support for quasi-quotation (3)

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>)>
```

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


Program generation: overview

We have (at least) two stages:

- First stage: generating code using static data
- Second stage: executing the generated code using dynamic data

Assumption: our program has two kinds of input data:

- Static input: their values are known at the first stage.
- Dynamic input: their values are not known the first stage, but known at the second stage.

It is very essential for generators to know which data is static and which is not.

Language support for quasi-quotation (4)

But is anything better than Lisp's S-expression approach ? Support for types.

- Types give a certain reliability of generator.
- Types give a certain reliability of generated code,
- AND it ensures "no free variables" in generated code.

Errors:

```
x + 1, <x + 1>, <fun x -> 3.0 + 1> <fun x -> ~x + 1>
```

Ok:

```
<fun x -> x + 1>, fun x -> <~x + 1>,
fun x -> <fun y-> ~x + y + 1>,
```


Staged Programming with MetaOCaml

MetaOCaml is a multi-stage extension of the programming language OCaml.

Creating code:

```
let x =    3 + 5 ;;
==> 8
let x = .< 3 + 5 >.;;
==> .< 3 + 5 >.;;
let x = .< 3 + 5 * y>.;;
==> (error)
```


Staged Programming with MetaOCaml

Composing code:

```
let x = .< 3 + 5 >.  ;;
==> .< 3 + 5 >.;;
let y = .< 7 * .~x >. ;;
==> .< 7 * (3 + 5) >.
let z = .< .~x / .~y >. ;;
==> .< (3 + 5) / (7 * (3 + 5)) >.
```


Staged Programming with MetaOCaml

Executing code:

```
let x = .< 3 + 5 >. ;;
==> .< 3 + 5 >.;;
let y = run x ;;
==> 8
let z = run .< .~x * .~x >. ;;
==> 64
```

Staged Programming with MetaOCaml

We can write a code 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 code = gen_power1 3 .<5>. ;;
==> .< 5 * .~(gen_power1 2 .<5>.) >.
==> .< 5 * .~(.<sqr .~(gen_power1 1 .<5>.)>.) >.
==> .< 5 * .~(.<sqr .~(.<5>.)>.) >.
==> .< 5 * .~(.<sqr 5>.) >.
==> .< 5 * (sqr 5) >.
```


What's the difference ?

Types of code are checked (and inferred).

```
.<3 + 5>.        : int code
.<3 + "abc">.    type error
let x=.<10>. in .<.~x + 1>. : int code
fun x -> .<.~x + 1>. : (int code) -> (int code)
.<fun x -> x + 1>. : (int -> int) code
.<fun x -> .~x + 1>. type error
.<fun x -> y + 1>. error
fun x -> .<fun y -> .~x + y>.
               : int code -> (int -> int) code
```

The programming languages Scala also has an advanced support for staged programming.

Today's Summary

- "Code as strings" are available in most languages, but no support for program generation.
- "Code as trees (or datatypes)" are available in several languages, but no support for program generation.
- Staged computation: Language support for code generation (type system).

Next week(s)

- Basic techniques of code generation. (2 weeks)
- Case studies (2 weeks): Code Generators for Image Processing, Linear algebra, GPGPU, Domain-Specific Languages etc.
- Report on a paper; See the web page.

`http://www.cs.tsukuba.ac.jp/~kam/acpl/`