Concept of Programming Languages (CS320) Lecture 2

By Zhiqiang Ren (Alex)
aren@cs.bu.edu

Content

- ATS Syntax Rephrase
- Tail recursive v.s. non-tail recursive
- Translation from “while loop” to “recursive function”
- List Operations

ATS Syntax Rephrase (0)

- Expression: something leading to a value
- Name binding: give a name to an expression

ATS Syntax Rephrase (1)

- Function Declaration
  - `extern fun foo (x: int, y: int): mylist`
- Function Implementation
  - `implement foo (x, y) = exp`
  - `fun foo (x: int, y: int): mylist = exp`
- **`main`** is special
  - `implement main () = exp`
  - `implement main0 (argc, argv) = exp`

ATS Syntax Rephrase (2)

- Expression: simple, compound, control flow expression
- Simple expression: constant, function call, object construction

`3`	`“abc”`	`foo (exp, exp)`	`foo`	`list0_cons ()`

- Compound expression:

```
// all exp except the last one must be of type void
begin exp1; exp2; …… ; expn end
```

```
// all exp except the last one must be of type void
(exp1; exp2; …… ; expn)
```

ATS Syntax Rephrase (3)

- Control flow expression

```
let
  val x = exp1
  val y = exp2
  …
in
  exp
end
```

```
if exp then
  exp
else
  exp
```

```
case exp of
| pattern => exp
| pattern => exp
……
| pattern => exp
```

```
exp where {
  val x = exp1
  val y = exp2
  …
}
```

Tail recursive v.s. non-tail recursive (1)

- sum (x) = 1 + 2 + … + x, for x > 0;
- sum (x, accu) = 1 + 2 + … + x + accu, for x > 0.

file:///G|/Boston%20University/Teaching/sum1_c_sum2_c.html

Left file: sum1.c

Right file: sum2.c

int sum(int x)	<>	int sum(int x, int accu)

{
  if (1 >= x)
	=	{
		  if (1 >= x)

return 1;	<>	return 1 + accu;
else	=	else
return x + sum(x-1);	<>	return sum(x-1, accu + x);
}	=	}

Tail recursive v.s. non-tail recursive (2)

- gcc -S sum1.c ➔ sum1.s V.S. gcc -S -O2 -o sum2_opt.s sum2.c ➔ sum2_opt.s

```c
int sum(int x)
{
    if (1 >= x)
        return 1;
    else
        return x + sum(x-1);
}
```

```c
int sum(int x, int accu)
{
    if (1 >= x)
        return 1 + accu;
    else
        return sum(x-1, accu + x);
}
```

1	.file "sum1.c"	<>	1	.file "sum2.c"
2	.text	=	2	.text
		-+	3	.p2align 4,,15
3	.globl sum	=	4	.globl sum
4	.type sum, @function		5	.type sum, @function
5	sum:		6	sum:
6	pushl %ebp		7	pushl %ebp
7	movl %esp, %ebp		8	movl %esp, %ebp
8	subl $8, %esp	+-		
9	cmpl $1, 8(%ebp)			
10	jg .L2			
11	movl $1, -4(%ebp)			
12	jmp .L4			
13	.L2:			
14	movl 8(%ebp), %eax			
15	subl $1, %eax			
16	movl %eax, (%esp)			
17	call sum			
18	movl 8(%ebp), %edx	=	9	movl 8(%ebp), %edx
		<>	10	movl 12(%ebp), %eax
			11	cmpl $1, %edx
			12	jle .L4
			13	.p2align 4,,7
			14	.L6:
19	addl %eax, %edx		15	addl %edx, %eax
			16	subl $1, %edx
20	movl %edx, -4(%ebp)		17	cmpl $1, %edx
			18	jne .L6
21	.L4:	=	19	.L4:
22	movl -4(%ebp), %eax	<>	20	popl %ebp
23	leave		21	addl $1, %eax
24	ret	=	22	ret
25	.size sum, .-sum		23	.size sum, .-sum
26	.ident "GCC: (GNU) 4.1.2 20080704 (Red Hat 4.1.2-46)"		24	.ident "GCC: (GNU) 4.1.2 20080704 (Red Hat 4.1.2-46)"
27	.section .note.GNU-stack,"",@progbits		25	.section .note.GNU-stack,"",@progbits

From while to recursive function (1)

- transform ***while loop*** into ***tail recursive function***

```
int foo(int x) {
  int index = x;
  int accu = 0;
  while (index > 0) {
    accu += index;
    index = index – 1;
  }
  int output = accu;
  return output;
}
```

```
fun foo(x:int):int = let
  // loop(index, accu) =
  //    (0, 1 + 2 + … + index + accu)
  fun loop (index: int, accu: int): (int, int) =
    if index > 0 then let
      val accu’ = accu + index
      val index’ = index – 1;
    in
      loop (index’, accu’)
    end else
      (index, accu)

  val ret = loop(x, 0)
  val output = ret.1
in
  output
end
```

From while to recursive function (2)

$$x^y \bmod z$$

$$y = a_n 2^n + a_{n-1} 2^{n-1} + \ldots + a_1 2^1 + a_0 = \sum_{k=0}^{n} a_k 2^k$$

$$x^y = (x^{2^n})^{a_n} \cdot (x^{2^{n-1}})^{a_{n-1}} \ldots (x^{2^1})^{a_1} \cdot (x)^{a_0} = \prod_{k=0}^{n} (x^{2^k})^{a_k}$$

$$(x^{2^n}) = (x^{2^{n-1}})^2$$

$$y_0 = y = a_n 2^n + a_{n-1} 2^{n-1} + \ldots + a_1 2^1 + a_0, a_0 = y_0 \% 2$$

$$y_1 = y_0 / 2 = a_n 2^{n-1} + a_{n-1} 2^{n-2} + \ldots + a_1, a_1 = y_1 \% 2$$

$$\ldots$$

$$y_n = y_{n-1} / 2 = a_n, a_n = y_n \% 2$$

From while to recursive function (3)

$$y = a_n 2^n + a_{n-1} 2^{n-1} + \ldots + a_1 2^1 + a_0 = \sum_{k=0}^{n} a_k 2^k$$

$$x^y = (x^{2^n})^{a_n} \cdot (x^{2^{n-1}})^{a_{n-1}} \ldots (x^{2^1})^{a_1} \cdot (x)^{a_0} = \prod_{k=0}^{n} (x^{2^k})^{a_k}$$

$$(x^{2^n}) = (x^{2^{n-1}})^2$$

```
int expx(int x, int y) {
  int xk = x;
  int yk = y;
  int accu = 1;
  while (yk > 0) {
    if (1 == (yk % 2)) {
      accu = accu * xk;
    }
    yk = yk / 2;
    xk = xk * xk;
  }
  int output = accu;
  return output;
}
```

```
fun expx(x:int, y:int):int = let
  fun expx_log (xk: int, yk: int, accu: int):
    (int, int, int) =
    if yk > 0 then let
      val accu' = if (yk mod 2) = 1 then accu * xk
                  else accu
      val yk' = yk / 2
      val xk' = xk * xk
    in
      expx_log(xk', yk', accu')
    end else  // [if vk > 0]
      (xk, yk, accu)

  val ret = expx_log(x, y, 1)
  val output = ret.2
in
  output
end
```

From while to recursive function (3)

- while loop <-> tail recursive function: easy
- recursive function -> tail recursive function (while loop): hard but possible
- Is “while loop” equal to “recursive function”?
  - Yes and No

From while to recursive function (4)

- Mutually recursive functions

```
extern fun isOdd (x: int): bool
extern fun isEven (x: int): bool

implement isOdd (x) =
  if x = 1 then true
  else if x = 0 then false
  else isEven (x - 1)

implement isEven (x) =
  if x = 0 then true
  else if x = 1 then false
  else isOdd (x - 1)

implement main () = print_bool
(isOdd 42)
```

```
fun isOdd (x: int): bool =
  if x = 1 then true
  else if x = 0 then false
  else isEven (x - 1)

and isEven (x: int): bool =
  if x = 0 then true
  else if x = 1 then false
  else isOdd (x - 1)

implement main () = print_bool
(isOdd 42)
```

Operations of List

- Think of list as an abstraction / interface.
- Operations ($PATSHOME/libats/ML/SATS/listo.sats)

```
fun{a:t@ype} list0_head_exn (xs: list0 a): a
fun{a:t@ype} list0_length (xs: list0 a):<> int
fun{a:t@ype} list0_nth_exn (xs: list0 a, i: int): a
fun{a:t@ype} list0_reverse (xs: list0 a): list0 a
fun{a:t@ype} list0_reverse_append(xs: list0 a, ys: list0 a): list0 a
fun{a:t@ype} list0_tail_exn (xs: list0 a): list0 a
// take the first n
fun{a:t@ype} list0_take_exn (xs: list0 a, n: int): list0 a
// drop the first n
fun{a:t@ype} list0_drop_exn (xs: list0 a, n: int): list0 a
```

Operations of List

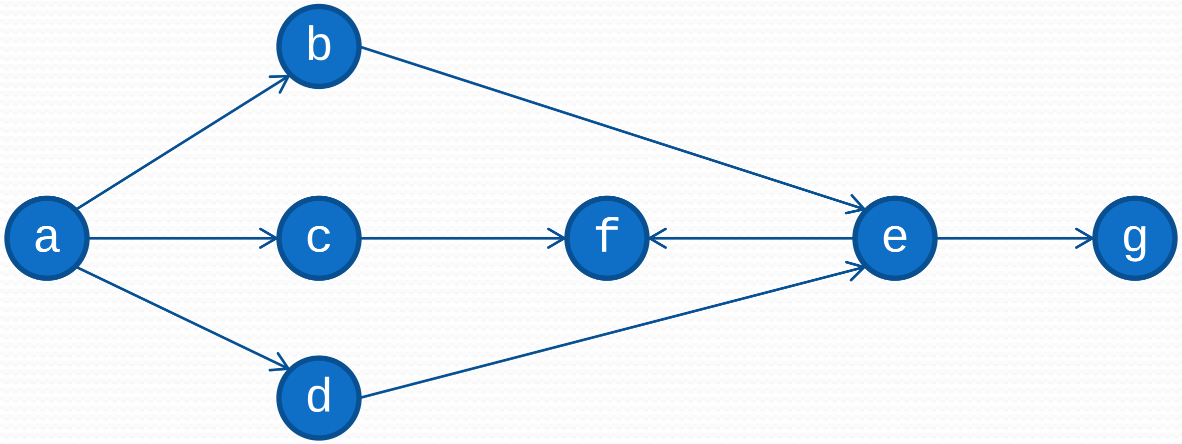
- Load library files

```
staload "libats/ML/SATS/basis.sats" // type of list0
staload "libats/ML/SATS/list0.sats" // operation of list0

staload _ = "libats/ML/DATS/list0.dats" // template definition
```

Graph algorithm (list implementation)

- Representation of graph by list of pairs
- ("a", "b") :: ("a", "c") :: ("a", "d") :: ("b", "e") :: ("c", "f") :: ("d", "e") :: ("e", "f") :: ("e", "g") :: nil



Graph algorithm (list implementation)

- Depth First Search
- To remember the visited nodes
  - Mark the node (not feasible in functional programming)
  - Extra booking
    - record the node
    - check whether a node has been recorded

Graph algorithm (list implementation)

```
// Don’t forget standard headers

#define :: list0_cons
#define nil list0_nil

typedef node = string
typedef edge = (node, node)
typedef graph = list0 edge

abstype set
extern fun set_new (): set
extern fun set_contains (
  s: set, n: node): bool
extern fun set_add (
  s: set, n: node): set
```

```
extern fun depth (
  n: node, g: graph): void

implement main () = let
  val g =  ("a", "b") ::
  ("a", "c") :: ("a", "d") ::
  ("b", "e") :: ("c", "f") ::
  ("d", "e") :: ("e", "f") ::
  ("e", "g") :: nil
in
  depth ("a", g)
end
```

Quiz

- Divide r^2 into $x^2 + y^2$
- Find all the possible pairs
- `fun factor (r: int): list0 (int, int)`
- Algorithm (Dijkstra 1976)
- (x, y) x goes down from r, y goes up from 0
  - $x^2 + y^2 < r^2$ then increment y by 1, and move on
  - $x^2 + y^2 = r^2$ then record it, and move on (change x and y)
  - $x^2 + y^2 > r^2$ then decrement x by 1, and move on
  - $x < y$ then stop