TDDA69 Data and Program Structure

Declarative Programming Techniques

Cyrille Berger

LINKÖPING UNIVERSITY

Lectures

Lecture content

- Recursion
- Function composition
- Verification

Recursion

What is recursion?

A function is called recursive if the body of that function calls itself, either directly or indirectly.

Factorial: the classical example (1/2)

- Factorial in Haskel:

```
factorial :: Integral -> Integral
factorial 0 = 1
factorial n = n * factorial (n-1)
```

- Factorial in Common LISP:

```
(define (factorial n)
            (cond ((= n 0) 1)
                       (t (* n (factorial
(- n 1))))))
```


Factorial: the classical example (2/2)

- With a loop:

```
function factorial(n)
{
  var r = 1;
  for(var i = 2; i <= n; ++i)
  {
    r *= i;
  }
  return r;
}
```

- With a recursive call:

```
define factorial = function (n) -> r
{
  r = cond(n == 0, 1, n * factorial(n-1))
}
```


Recursion vs loops

- ```
  while(expression)
  {
    do_something();
  }
  ```
- ```
  define loop_something = function (args...) -> ret
  {
    cond(expression, { do_something(); ret =
  loop_something(args...); }, ret = null);
   }
  }
  ```
- ```
  (define (loop_something args...)
            (cond (expression) value)
                      (t (do_something) (loop_something
  args...))
  ```
  ```

Function calls

- Calling a function is usually more expensive than a loop
- In many programming language, the number of function call is limited by the size of the stack
  - `factorial(1000)`

    RecursionError: maximum recursion depth exceeded in comparison
  - `sys.getrecursionlimit()`

    1000
  - `sys.setrecursionlimit(1003)`
  - `factorial(1000)`

    4023872600770937735437024339.....00000
- Tail-call optimisation

Tail-call

- A *tail-call* is a call to an other function performed as the last statement in a function
- Are those tail-call?

```
function foo0(data) {
     a(data);
     return b(data);
}
function foo1(data) {
     return a(data) + 1;
}
```

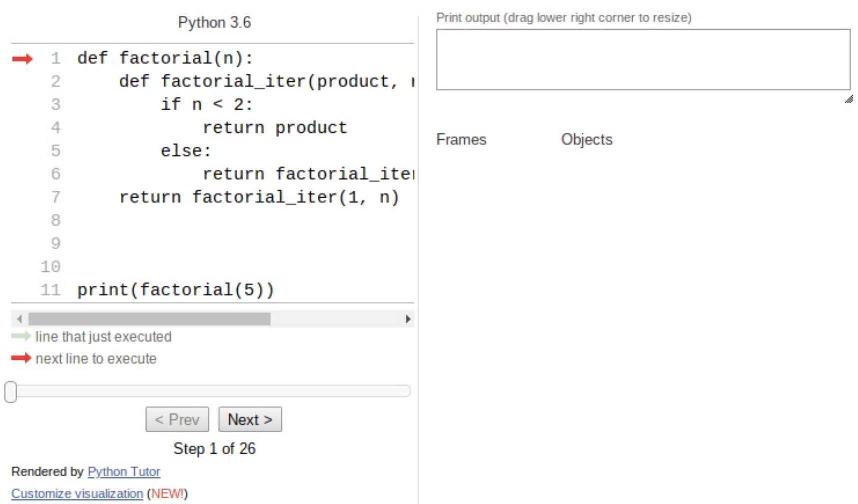

Tail-call optimisation

- In case of a *tail-call*, the execution does not need to return to the function, there is no need to save the function call on the stack
- Recursion without tail-call

```
define factorial = function (n) -> r
{
  r = cond(n == 0, 1, n * factorial(n-1))
}
```

- Recursion with tail-call

```
define factorial = function (n) -> r
{
  define factorial_iter = function (product, n) -> r
  {
    r = cond(n < 2, product, factorial_iter(product * n, n-1))
  }
  r = factorial_iter(1, n))
}
```

State

- Can we have state when we cannot change a value in the store?
- Implicit state, consider

```
function f(S)
{
  f(S+1);
}
```


Function composition

Function composition

- Many operations on lists (or iterables) are very similar
  - modification, filtering, accumulation...

For each elements (1/2)

```
function lower_case(text) -> r
{
  r = lower_case_iter(text, 0);
}
function lower_case_iter(text, idx) -> r
{
  r = cond(r < length(text), lower(text[idx]) +
lower_case_iter(text, idx + 1), "")
}
```

For each elements (2/2)

```
function lower_case(text) -> r
{
  r = for_each(text, lower);
}
function for_each(val, func) -> r
{
  r = for_each_iter(val, func, 0);
}
function for_each_iter(val, func, idx) -> r
{
  r = cond(r < length(val), func(text[idx]) +
lower_case_iter(text, idx + 1), "")
}
```


When closure matters

- Filter a list:

```
function filter_small(list,
value) -> r
{
  r = filter(list, function(x) ->
r {
    r = x < value;
  });
}
```


Verification

Correctness

- How can we tell a program is correct?
  - Test a few selected values, ie, unit
- In general, we need:
  - a mathematical
  - a specification of the
  - to reason using the model and

Verification and proving

- To prove a program correct, we must consider everything a program depends on
- In pure functional programs, dependence on any data structure is explicit
- The program can be correct but still give wrong results!
  - We need to verify compiler, run-time system, operating system, hardware!

Proving properties in functional programming

```
define power = function(b, n) -> r
{
  r = cond(n == 0, 1, b * power(b, n-1));
}
```

- Claim: for any integer $n \geq 0$ and any number b, power(b, n) = b^n
- Proof:
  - 1) Verify the base case: power(b,0)
  - 2) Assume that power (b, n - 1)) is correct
  - 3) Verify that power(b, n) is correct assuming that power(b, n - 1)) is correct

Proving properties in imperative programming

```
function power(b, n) {
  int result = 1;
  for(int i = 0; i < n; ++i)
  {
    result *= b;
  }
  return result;
}
```

- Devise a *loop invariant*:
  - $(n \geq i) \wedge (result = b^i)$
  - Prove that it is true for the first loop iteration
  - Prove that each loop iteration preserves it

    Assume that $(n \geq i) \wedge (result = b^i)$

    Prove that $(n \geq j) \wedge (result = b^j)$ with $j = i + 1$

Declarative Components (1/2)

- Declarative components are written using only pure functions
  - A declarative component can be written, tested, and proved correct independent of other components and of its own past history.
  - Programs written in the declarative model are much easier to reason about than programs written in more expressive models (e.g., an object-oriented model).

Declarative Components (2/2)

- Since declarative components are mathematical functions, algebraic reasoning is possible i.e. substituting equals for equals
  - Given f(a)=a^2, we can replace f(a) in other equations, b=7f(x)^2 becomes b=7x^4
- The declarative model of chapter 4 guarantees that all programs written are declarative
- Declarative components can be written in programming models that allow stateful data types, but there is no guarantee

```
int f(int x) { return x * x; }
```

  - **constexpr** in C++ allows to offer the guarantee

Conclusion

- Performance issues
- Verification is easier in functional programming
- Declarative components