Subroutines, parameters, exceptions

nprenet@bsu.edu

CS431, Spring 2022

Contents

Handling recursion

Tail-recursion as iteration: Scheme

Consider the naïve, recursive implementation of the factorial function in Scheme:

```
(define (factorial n)
        (if (= n 1)
                1
                (* n (factorial (- n 1)))))
```

Tracing the evaluation: observe that each subroutine has a pending multiplication, which cannot complete until the recursive call returns the second operand. In order to keep track of this chain of *deferred operations* the **stack** grows linearly until `factorial 1` is evaluated. From then on, the process shrinks back, as the multiplications are evaluated.

```
> (trace factorial)
> (factorial 10)
|(factorial 10)
| (factorial 9)
| |(factorial 8)
| | (factorial 7)
| | |(factorial 6)
| | | (factorial 5)
| | | |(factorial 4)
| | | | (factorial 3)
| | | | |(factorial 2)
| | | | | (factorial 1)
| | | | | 1
| | | | |2
| | | | 6
| | | |24
| | | 120
| | |720
| | 5040
| |40320
| 362880
|3628800
3628800
```

Compare with the following implementation, which is also recursive:

```
(define (factorial n) i
        (define (iter product counter)
                (if (> counter n)
                        product
                        (iter (* counter product) (+ counter 1))))
        (iter 1 1))
```

Tracing the process results in the surprising picture below:

```
> (trace factorial-iter)
> (factorial-iter 10)
|(factorial-iter 10)
|3628800
3628800
```

The second procedure is recursive, but there is no stack growth. Both *procedures* are recursive, due to the syntactic fact that they are defined in terms of themselves. However, their *process* is different:

- `factorial` is a true recursive process, that builds up a chain of deferred multiplications, that contracts when the operations are actually performed: the stacks grows linearly.
- `factorial-iter` is an iterative process: at each step, the evaluator only needs to keep track of the current values of the variables `product` and `counter`; the number of steps grows linearly, but the process executes in constant space. This is possible because the recursive call is a **tail call**: because there is nothing left to do after it returns, such a call can be treated by the compiler as a jump call, or *goto*.

Functional languages like Scheme and ML typically **optimize tail recursion by compiling it into an iterative process**. Other languages offer no such guarantee. In Javascript, both recursive functions overflow the stack, when dealing with large input values:

```
function factorial( n ){
   if (n == 0){ return 1; }
   else { return n * summation ( n-1 ) }
}
factorial( 10000 ); // Uncaught RangeError: Maximum call stack size exceeded

function factorial_iter( n ){
   function iter( counter, total ){
      if (counter > n){ return total; }
      else { return iter(counter+1, total*counter); }
   }
   return iter( 1, 1);
}
factorial_iter( 10000 ); // Uncaught RangeError: Maximum call stack size exceeded
```

Haskell?

Although Haskell is a direct descendant of ML, it does not seem to handle tail-recursion better than Javascript:

```
summation :: Integer -> Integer
summation 0 = 0
summation n = n + summation (n-1)

*Subroutine> summation 999999
499999500000
*Subroutine> summation 99999999
*** Exception: stack overflow
```

```
summation' :: Integer -> Integer
summation' n = iter 0 0
  where iter counter sum
          | counter > n = sum
          | otherwise = iter (counter + 1) (counter + sum)

*Subroutine> summation' 99999999
*** Exception: stack overflow
```

What is going on there? To get the answer, we need to learn more about the ways programs pass parameters to routines.

Passing parameters

Subroutines takes arguments that

- control aspects of their behaviour
- specify the data on which they operate

Declaration	Call
`void func( Integer i, String s){ ... };` `i` and `s` are the **formal** parameters.	`func( 2, "Hello" );` `2` and `"Hello"` are the **actual** parameters, or **arguments**

The simplified view: value vs. reference

The picture is pretty straightforward for languages where variables store values, and not references. Suppose a global variable `x` that stores an integer value, and that we wish to pass `x` as a parameter to subroutine `p`:

```
void p( int i){ ... }
...
p(x);
```

The language's implementors have two choices:

1. Provide subroutine `p` with a copy of `x`'s value → **call by value**: the formal parameter of the function (`i`) is assigned `x`'s value, and from then (in the body of the function), the two variables are independent from each other.

What can you pass by value? Values or expressions that return a value (e.g. result of an arithmetic computation)

2. Provide subroutine `p` with `x`'s address → **call by reference**: both `x` and the formal parameter `i` are two names for the same object (aka. *aliases*), and changes made through one are visible through the other.

   What can you pass by reference? Any value that can be on the LHS of an assignment (aka. l-value, or named container for a value); that exclude results of arithmetic operations or any value without an address.

In this C program, variable `x` is passed by value: assigning formal parameter `y` does not change `x`'s value:

```
static int x = 0;
...
void foo(int y){
  y = 3;
  printf("%i", x);
}
x = 2;
foo(x);            // 2
printf("%i", x);   // 2
```

Compare with this Pascal program, where `x` is passed by reference instead: `y` is an *alias* for variable `x`: the assignment at line 3 changes its value.

```
program pascalProg;
 var
    x: integer;

function foo(var y: integer): integer;
begin
    y:=3;
    writeln(x);
    foo := 0;
end;

begin
  x := 1;
  writeln(x);   // 1
  foo(x);       // 3
  writeln(x);   // 3
end.
```

In C, parameters are always passed by value; but arrays are passed by their pointers, so that modifications to the array element are visible to the caller. More generally, pointer types allow C programmers to pass *any* parameter by reference, even this much less casual that in Pascal.[1] Historically, this has been justified by

- the obvious need for functions that can modify the objects they operate on;
- the cost of copying large arguments, even if they are not meant to be modified by the subroutine.

The latter justification has lead to a wider use of pointers that called for, and to buggy code, when a subroutine modifies an object that was not meant to be changed by the caller.[2]

[1] A variable is either an `int` or a pointer to an `int` (`int *`), but certainly not both: passing a pointer to a function that expects a value is a programming error, that the compiler flags accordingly.

[2] In C, the `const` keyword ensures that a subroutine will not be able to modify the object that is referred to by a pointer.

The complicated view: parameters mode

In most languages, variables may store not only values, but also references to primitive values and objects. If a variable is already a reference, neither of the two options above (call by value or reference) makes sense. We need to nuance the picture.

Call by sharing

In some languages (SmallTalk, Lisp/Scheme, ML, Ruby), any variable is a not a named container for a value, but a *reference to* a value. Then, it is natural to pass to the subroutine the reference itself:

- the formal and actual parameters refer to the same object
- in contrast with a call by reference, the subroutine can change the value referred to by the actual parameter, but cannot make the *argument* refer to a different object

Hybrid models

Even with a reference model for variables, immutable objects (string literals, numbers) are typically copied to subroutines (i.e. passed by value), instead of being passed by reference. Many languages, that include C#, Java, Javascript, Python follow this hybrid model for variables:

- variables containing **primitive values** and built-in types are passed **by value** and thus cannot be changed by the subroutine;
- variables for **user-defined class types are all references** and are shared with the subroutine. In both Java and C#, actual parameters (primitives or references) **cannot be modified** by the subroutine by default. However, C# provides a way to override this behaviour with the `ref` keyword, such that a variable of `class` (reference) type that is passed to a subroutine may not point at the same object after the routine's execution.

Call-by-name

Call-by-name describes a way of passing an entire expression as a parameter to a function, to be evaluated *by the subroutine* (instead of of being evaluated by the caller): it is known as **normal order evaluation** and amounts to replacing every ocurrence of the actual parameter in the subroutine's body with the expression it represents. Let us illustrate with a Haskell-like example. Consider the program below:

```
times3 x = x * 3
plus2 y = y + 2

(times3 (plus2 5))
```

In most programming languages, include Scheme and Python, the beta-reduction of the expression at line 3 would go through the following steps:

```
(times3 (plus2 5))
=  (\x -> x * 3) ((\y -> y + 2) 5)
=  (\x -> x * 3) (5 + 2)            -- applying plus2 to 5
=  (\x -> x * 3) 7
=  (7 * 3)                          -- applying times3 to 7
=  21
```

What we have just described is known as the **strict** or **applicative** evaluation order. The rules are different under **normal evaluation order**, where the lambda-expression for `plus2` is expanded into `time3`'s body *before* it is applied its own parameters:

```
(times3 (plus2 5))
=  (\x -> x * 3) ((\y -> y + 2) 5)
=  ((\y -> y + 2) 5) * 3
=  (5 + 2) * 3
=  (7 * 3)
=  21
```

Note that number of steps is the same for function above. However, a pure implementation of normal order can be quite inefficient. Consider this:

```
square x = x * x
plus2 y = y + 2

square (plus2 5) = (\x -> x * x) ((\y -> y + 2 ) 5)
                 = ((\y -> y + 2) 5) * ((\y -> y + 2 ) 5)
                 = (5 + 2) * ((\y -> y + 2 ) 5)
                 = 7 * ((\y -> y + 2 ) 5)
                 = 7 * (5 + 2)
                 = 7 * 7
                 = 49
```

The duplicated expression `((\y -> y + 2) 5)` is evaluated twice! Actually, very few languages follow this model. Haskell, as well as R[3] combine normal order with **lazy** evaluation: this implementation is known as **call by need**.

Call-by-neeed

Haskell implements a *memoized* version of normal order evaluation, by creating a temporary variable bound to the expression[4], which is then evaluated only once. The following gives you the intuition of it, by simulating the thunk idea at the programming language-level:[5]

```
square (plus2 5) = (\x -> x * x) ((\y -> y + 2 ) 5)
                 = (thunk * thunk)
                           where thunk = ((\y -> y + 2 ) 5)
                 = (thunk * thunk)
                           where thunk = (5 + 2)
                 = (thunk * thunk)
                           where thunk = 7
                 = (7 * thunk) where thunk = 7
                 = 7 * 7
                 = 49
```

This explains why Haskell is not particularly good at recursion when working with operations that evaluate their arguments strictly (such as arithmetic operations): its lazy evaluation model does not obviate the need for creating long chains of thunks. Let us go back to our introductory `summation` example (see "Handling recursion, p. 7):

```
summation 4
4 + summation 3             -- Note that the `4-1` only got evaluated to 3
4 + (3 + summation 2)        -- because it has to be checked against 0 to see
4 + (3 + (2 + summation 1))  -- which definition of `summation` to apply.
4 + (3 + (2 + (1 + summation 0)))
4 + (3 + (2 + (1 + 0)))
4 + (3 + (2 + 1))
4 + (3 + 3)
4 + 6
10
```

[3]A powerful, open-source statistics package used by scientists.
[4]Also called a *thunk*, a function without argument, that can be called on a as-needed basis.
[5]This is usually the compiler's job.

Note that there is no stack of `summation` function calls hanging around waiting to terminate (compare with the Scheme recursive factorial), but the stack grows anyway after each reduction: because the addition `(+)`, it triggers the evaluations of its second argument, and an activation record is created to keep track of the pending computation.

Now let's look at the tail-recursive `summation' 4`:

```
summation' 4
iter 0 0
iter 1 {0+1}            -- Note that the `0+1` only got evaluated to 1
iter 2 {{0+1}+2}        -- because it has to be checked against 4 to see
iter 3 {{{0+1}+2}+3}    -- which definition of `iter` to apply.
iter 4 {{{{0+1}+2}+3}+4}
{{{{0+1}+2}+3}+4}        -- the thunk "{...}"
({{{0+1}+2}+3}+4)        -- is retraced
(({{0+1}+2}+3)+4)        -- to create
((({0+1}+2)+3)+4)        -- the computation
((((0+1)+2)+3)+4)        -- on the stack
(((1+2)+3)+4)
((3+3)+4)
(6+4)
10
```

The tail recursion by itself has not saved any time or space. This thunk is unraveled by traversing it to the bottom, recreating the computation on the stack. There is also a danger here of causing stack overflow with very long computations, for both versions.

If we want to hand-optimise this, all we need to do is make it strict. We can use the strict application operator `$!` to define

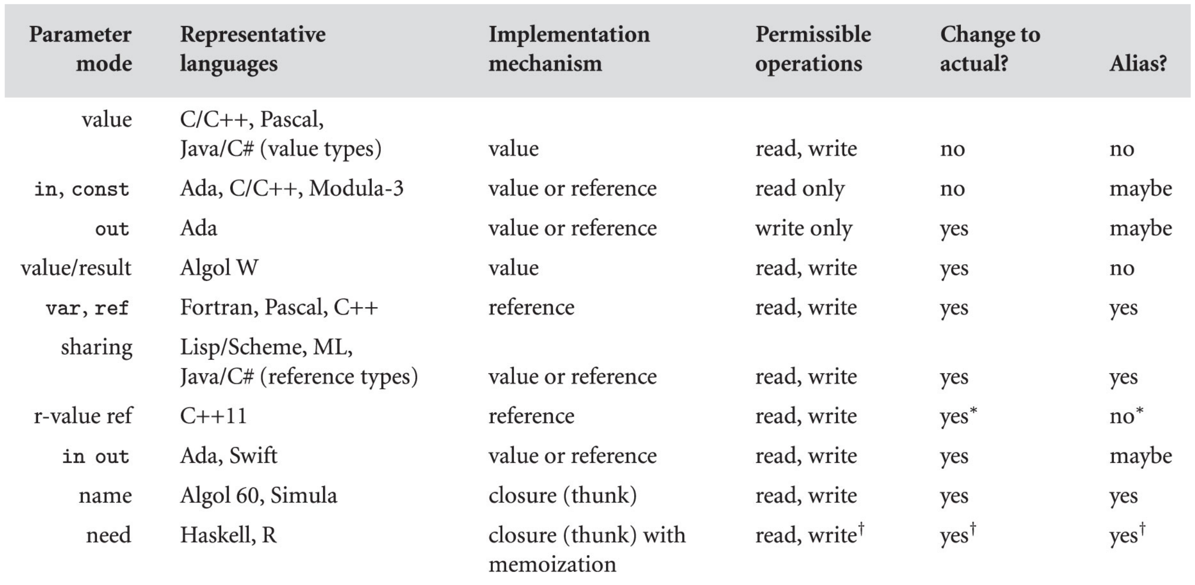
```
strictSummation :: Integer -> Integer
strictSummation n = iter 0 0
  where iter counter total
          | counter > n = total
          | otherwise = iter (counter + 1) $! (total + counter)
```

The `$!` infix operator forces the evaluation of the function's second argument.[6] The strict version does not overflow the stack:

```
*Subroutine> summation' 99999999
4999999950000000
```

[6] It's already strict in its first argument because that has to be evaluated to decide which definition of `iter` to apply.

A summary



Parameter mode	Representative languages	Implementation mechanism	Permissible operations	Change to actual?	Alias?
value	C/C++, Pascal, Java/C# (value types)	value	read, write	no	no
`in`, `const`	Ada, C/C++, Modula-3	value or reference	read only	no	maybe
`out`	Ada	value or reference	write only	yes	maybe
value/result	Algol W	value	read, write	yes	no
`var`, `ref`	Fortran, Pascal, C++	reference	read, write	yes	yes
sharing	Lisp/Scheme, ML, Java/C# (reference types)	value or reference	read, write	yes	yes
r-value ref	C++11	reference	read, write	yes*	no*
`in out`	Ada, Swift	value or reference	read, write	yes	maybe
name	Algol 60, Simula	closure (thunk)	read, write	yes	yes
need	Haskell, R	closure (thunk) with memoization	read, write†	yes†	yes†

Source: Scott, *Programming Languages Pragmatics*, 2016.

Exceptions

An **exception** is an unexpected —or at least unusual— condition that arises durnig program execution, that cannot easily be handled in a subroutine's local context.

- I/O expection
- unsuccessful network request
- ...

Less-than-satisfactory solutions

1. Invent a dummy value (when the expected return value cannot be obtained)
2. Return an error code, to be inspected by the caller
3. The caller passes to the callee a closure[7] that contains an error handling routine, to be called when running in trouble.

(1) works only for certain cases; (2) and (3) tend to clutter the code.[8] **Exception-handling mechanisms** address these issues

- normal case is specified simply
- control branches to **handler** when appropriate

Exception handlers

In most recent languages (Python, Ruby, C++, Java, C#, ML), handlers are lexically bound to a block of code, typically through a `try ... catch ...` syntax:

[7]A callback, that is.
[8]See callback-style for asynchronous Javascript code.

```
try {
  ...
  if (something unexpected)
    throw my_error("oops!");
  ...
} catch (my_error e){              // exceptions transfers control to
  cout << e.explanation << "\n";  // catch bloc == handler code
}
```

If there is no matching handler, the subroutine returns abruptly and the exception is re-raised by the calling code: if the caller itself has no handler, the exception is propagated back up the stack chain.

What do handlers do?

- (if local recovery possible) allow for the program to recover and continue execution
- (if local recovery impossible) clean up resources allocated locally before propagating the exception back up the chain
- (if no recovery possible) terminate the program

Implementing exception handlers

- maintaining a **linked-list stack of handlers**: when control enters a block, the handler for that block is added to the head of the list: straighforward, but costly, because handlers have to be pushed and popped from the stack, whether the exception is raised or not.
- a **table generated at compile time** captures the correspondence between handlers and protected blocks: each entry is made of the address of the block, and the address of the handler: raising an exception is more costly that way, but this occurs rarely.
- simulated exception-handlers: in Ruby and Scheme, the `call-with-current-continuation` routine (aka. `call/cc`) may be used to implement a simple exception handling system: a **continuation** is a way of recording the state of your running program (i.e. the current state of the computation and the referencing environment) and then resuming from that state at some point in the future.[9]

Scheme example (for those interested): non-local exit from a recursion

The function below recurses through a list and computes the product of its elements (source: R. Kent Dybvig, *The Scheme Programming Language*, 3rd edition, 2003). A continuation is used to deal with the special case where 0 is an element of the list: the continuation allows for terminating the computation, while returning the appropriate value (0).

```
(define product
 (lambda (ls)
  (call/cc                  -- when is applied to a function, call/cc passes
   (lambda (break)          -- to it a concrete representation of the continuation
    (let f ((ls ls))        -- (the 'break' argument)
      (cond
        ((null? ls) 1)
        ((= (car ls) 0) (break 0))              -- if a list element is 0, the
        (else (* (car ls) (f (cdr ls))))))))))   -- continuation is called, with return value 0

(product '(1 2 3 4 5)) => 120
(product '(1 2 3 0 5)) => 0
```

When applied to the function `(lambda ... )`, `call/cs` stores in `break` a snapshot of the ongoing computation, that may be called later, in order to resume the computation at the point where `call/cc` was called. This

[9]Continuations have a number of interesting applications when dealing with concurrency and coroutines.

object is then passed to the recursive list product routine. If at any point during the recursion, the sublist passed to subroutine `f` starts with a 0, the continuation is called: the value passed to the continuation (0 in this case) becomes the value of the function.