Comparative Programming Languages  
CM20318

Russell Bradford

2023/24

### 1. Evaluation

#### Call by Reference

More terminology: expressions like x and a[n+1] that refer to memory locations are called *lvalues*, as you can put them on the left side of an assignment: a[n+1] = 42;

An lvalue is associated with a memory location

Lvalues are occasionally called *locator values*

A non-lvalue is an *rvalue*, as you find them on the right of an assignment

**Exercise** Some languages allow expressions on the left, as long as they evaluate to an lvalue (memory location). Read about this

### 2. Evaluation

#### Call by Reference

An rvalue does not (necessarily) have an associated memory location

So things like 42 = n; and (2\*m)++; do not make sense

Call by reference works on lvalues only

### 3. Evaluation

#### Behind the Scenes

In reality, call by reference is implemented by use of pointers, thus if you wrote code like

void f(int& a) {  
 a = a + 3;  
}

and called it with f(n) this is transformed behind the scenes by the compiler to the equivalent of

void f(int \*a) {  
 \*a = \*a + 3;  
}

and the function call is rewritten to f(&n)

### 4. Evaluation

#### Behind the Scenes

The advantage is that you write the simpler code without the proliferation of \*s and &s, and the compiler does the pointer chasing for you

**Exercise** Work though the code carefully to explain to yourself that it works as an implementation of call by reference

### 5. Evaluation

**Exercise** Fortran is call by reference, but *does* allow things like inc(2\*m). Find out what is happening here

**Advanced Exercise** Read about how lvalues are implicitly coerced/dereferenced/converted to rvalues on the right of an assignment

**Advanced Exercise** Read about Algol 68

**Advanced Exercise** Read about *rvalue references* in C++

### 6. Evaluation

References are a sharp tool and there are roughly three different approaches to sharp tools.

* 1. Don’t give programmers sharp tools. They may make mistakes and cut their fingers off. This is the Java/Python/Perl/Ruby/PHP... approach.
* 2. Give programmers all the sharp tools they want. They are professionals and if they cut their fingers off it’s their own fault. This is the C/C++ approach.
* 3. Give programmers sharp tools, but put guards on them so they can’t accidentally cut their fingers off. This is Rust’s approach.

trentj

### 7. Evaluation

#### Call by Name

*Call by name* takes this a bit further, lifting the restriction that the arguments are lvalues (memory locations)

For example the function in Algol 60:

[-1]integer procedure sumsq(n, m)  
integer n, m;  
begin  
 sumsq := (n + m)\*(n + m);  
end;

that squares the sum of the arguments

### 8. Evaluation

#### Call by Name

Then

sumsq(x+1, y+2)

is effectively evaluated as

begin  
 ((x+1) + (y+2)) \* ((x+1) + (y+2))  
end

i.e., the whole *expressions* in the call are substituted into the function body, which is then evaluated

**Exercise** For hackers. Compare with *inlining* code

### 9. Evaluation

Implementations avoid name clashes so that local variables in the function body will never coincide with variables passed in

integer procedure foo(n)  
integer n;  
begin integer m;  
 m := 1;  
 foo := n + m;  
end;

And then foo(m + 1) is *not* evaluated as

begin integer m;  
 m := 1;  
 foo := (m + 1) + m;  
end;

as here there is inadvertent *capture* of the outer m by the local m

### 10. Evaluation

#### Call by Name

Rather, something more like

begin integer m001;  
 m001 := 1;  
 foo := (m + 1) + m001;  
end;

where the local m is renamed

**Advanced Exercise** Compare with name capture in the Lambda Calculus, and read about *alpha renaming*

**Exercise** Read about Algol 60 and its mechanism for implementing call by name

### 11. Evaluation

#### Call by Name

**Exercise** Read about *Jensen’s Device*

**Exercise** Read about *fexprs* in Lisp

**Exercise** Does call by reference need local variables to be renamed?

### 12. Evaluation

#### Call by Name

CBN is an interesting evaluation strategy that is occasionally more efficient than call by value:

integer procedure k(x, y)  
integer x, y;  
begin  
 k := x;  
end  
...  
n = k(1+1, 1+2+3+4+5+6+7);

Here the second argument is not used in the function body, so will not be substituted in, and therefore not evaluated

### 13. Evaluation

#### Call by Name

The evaluation is essentially

begin  
 k := 1+1;  
end

Thus using CBN is more efficient than CBV in this example

**Exercise** How many functions that you have written have had unused arguments?

### 14. Evaluation

#### Call by Name

More interestingly, call by name can evaluate some expressions that call by value cannot:

n = k(1 + 1, infiniteloop());

This will never terminate in a call by value evaluation, but is fine (though weird) in a call by name evaluation

Thus CBN is a more powerful evaluation mechanism than CBV, in the sense that it can evaluate a larger class of expressions

### 15. Evaluation

#### Call by Name

On the other hand, the call by name substitution mechanism is usually relatively expensive, so we don’t often win overall

Modern architectures are designed for CBV, so CBV is almost always faster in practice

And in the sumsq example above, the x+1 and y+2 are both evaluated in the body *twice*, less efficient than a call by value:  
((x+1) + (y+2)) \* ((x+1) + (y+2))

Algol 60 defaults to CBN, but also allows CBV, for this reason

**Exercise** Compare with *applicative order reduction* and *normal order reduction* in the Lambda Calculus

### 16. Evaluation

#### Call by Need

Next: *call by need*, or *lazy evaluation*

A form of call by name that tries to get closer to the efficiency of call by value, where you only evaluate a given argument at most once, but with the behaviour and benefits of call by name

Now

[-1]sumsq(x+1, y+2)

would evaluate as call by name, but now the x+1 and the y+2 are only evaluated *at most once* each

### 17. Evaluation

#### Call by Need

The argument evaluations are *memoised*, i.e., remembered, so when the same expression is seen again (within the function body), the previously computed valued can simply be reused

The trade-off here is single evaluation of the arguments against a more complicated evaluation mechanism

### 18. Evaluation

#### Call by Need

Memoisation of expressions

Memoisation of expressions

Memoisation of expressions

Memoisation of expressions

The memoisation will even notice the outer addition, and not re-evaluate that

Examples. Haskell and Miranda; also see special forms like delay and similar in some languages, e.g., Scheme

### 19. Evaluation

#### Call by Need

Call by need does need some care with expressions that are *supposed* to produce a different value each time you evaluate them

For example, read() and random():  
is random() + random() always the same even number?

**Exercise** Read about *referential transparency*

### 20. Evaluation

#### Call by Need

Call by need is good, but is an expensive mechanism. You don’t want to pay that cost when you are not using laziness

Proponents of languages like Haskell claim that, when given code that doesn’t need the power of call by need, the compiler can analyse the code and compile it in “the normal way” so avoiding the cost of laziness and memoisation

This is true, *if* the compiler is good enough

There has been a long history of language design predicated on the future existence of a “sufficiently clever compiler”

Mostly, that compiler was never created

### 21. Evaluation

#### Call by Need

Perhaps, in the last few years, such compilers are just about beginning to appear

But we have still a long way to go to as people keep inventing new ideas that need clever compiler support

### 22. Evaluation

#### Laziness

Perhaps surprisingly, laziness is a feature that has been supported in many languages (in a small way) for a long time: called *short-circuit evaluation*

Quite often, the logical operators, like and and or and others (perhaps written && and ||, etc.) are lazy

For example, x == 0.0 || 1.0/x == 2.0 is evaluated lazily

Evaluate the x == 0.0 first. If true, the whole expression is true, and the 1.0/x == 2.0 never gets evaluated

Only if x == 0.0 returns false does the 1.0/x == 2.0 get evaluated

### 23. Evaluation

#### Laziness

So this is lazy evaluation

Of course this is done in many languages as it is a useful thing

If x == 0.0 || 1.0/x == 2.0 is evaluated eagerly, there will be problems when x is 0.0

It means, though, that && and || are not like other operators, such as + and &, which are evaluated normally. And so they have to be treated differently in the compiler

### 24. Evaluation

#### Laziness

**Exercise** Make sure you understand the difference between the & and the && operators

**Exercise** For C++ geeks: you can overload && and so on in C++. What happens to the evaluation?

**Exercise** What happens in foo() || bar() if the functions have side-effects?

**Exercise** Investigate *extended boolean* operators, such as in Python and JavaScript

### 25. Evaluation

More examples to compare call by whatever. Suppose we have

struct Big {  
 int stuff[1000];  
 int things[1000];  
};

This structure might occupy 8000 bytes

### 26. Aside

#### “Big values”

Be careful about saying “a big value”: this is ambiguous and can mean two different things

It can mean the size of the value represented, or the size of the memory used to store the value

If you have int n = 100000000; then the value of n is a big value, but the variable n occupies maybe just 4 bytes

The array int v[1000] is a big value in the sense it occupies a large number of bytes

So always make it clear what you mean

### 27. Evaluation

Then if b is a struct Big we get

**call by value**  
foo(b); slow, as it copies 8000 bytes of b into the function  
 

So many languages (e.g., C) support pointers:  
bar(&b); fast, copies 8 (typically) bytes of pointer into the function, and we use this reference in the function

### 28. Evaluation

**call by reference**  
foo(b); fast, as it copies only (say) 8 bytes of reference to b (e.g., a pointer) into the function

### 29. Evaluation

**call by name**  
foo(b); the expression b is substituted into function; cost likely moderately high without a good optimiser

### 30. Evaluation

**call by need**  
foo(b); as call by name, but with extra complication of the memoisation check

Which is best in real life? It depends

On the language, how it implements stuff, the cleverness of the compiler writers, the data, the computation, and many other things

### 31. Evaluation

CBV is fast on values that occupy few bytes; slow on values that occupy many bytes

CBR is moderately fast on all sizes; but slower than CBR on values that occupy a small number of bytes

And slower as it needs to do pointer chasing to get to the value in the function body

And so on

### 32. Evaluation

**Exercise** Many other evaluation strategies have been thought about. Read about them

**Exercise** What is the difference between call by reference and using references in a call by value language?

**Exercise** Is Java call by value or call by reference? Explain (take care: the Java Language Specification differs in its definitions of some terms)

### 33. Evaluation

**Exercise** What is Python’s calling mechanism?

**Exercise** Consider the following Python code

>>> x = [1,2,3]  
>>> y = x   
>>> x.append(4)  
>>> x  
[1, 2, 3, 4]  
>>> y

What is the value of y? Explain. Then explain the result of doing x.append(x)

### 34. Evaluation

**Exercise** For C++ hackers. C++ has native CBR. Read about how to use lambdas to mimic call by name

**Exercise** Explain if is possible to mimic CBR in Python

**Exercise** Read about *generators* (more generally, *coroutines*) as a way an eager language can mimic lazy behaviour

**Exercise** Read about *thunks* as a way an eager language can mimic lazy behaviour

**Advanced Exercise** How do C++’s rvalue references differ from call by name?

### 35. Evaluation

**Exercise**

func foo(n) {  
 if (n < 2) { return 1; }  
 return n\*foo(n-1);  
}

Trace the evaluation of this function in a call by need language

**Exercise** Read about how lazy evaluation enables you to describe (effectively) *infinite* datastructures

**Exercise** Some people regard lazy evaluation as declarative. Do you agree with this?

**Exercise** Some people regard lazy evaluation as dataflow. Do you agree with this?

### 36. Application

Carrying on looking at general features of languages…

In contrast to the “generic” languages, several applications areas have languages specifically designed for that area

Sometimes called *domain specific languages* (DSL)

Example. HTML for doing web pages

Example. Maple for doing maths. The basic datatypes are numbers, variables, polynomials, matrices, functions (trig, exp, etc.) and the like. The basic operations are arithmetics of all these things, integration, differentiation, and so on

### 37. Application

> diff(ln(x), x);   
 1/x  
  
> int(sin(x), x);  
 -cos(x)

### 38. Application

> expand((x+1)^100);  
 100 99 98 97 96 95  
1 + x + 100 x + 4950 x + 161700 x + 3921225 x + 75287520 x  
  
 94 93 92 91  
 + 1192052400 x + 16007560800 x + 186087894300 x + 1902231808400 x  
  
 90 89 88  
 + 17310309456440 x + 141629804643600 x + 1050421051106700 x  
  
 87 86 85  
 + 7110542499799200 x + 44186942677323600 x + 253338471349988640 x  
  
 84 83  
 + 1345860629046814650 x + 6650134872937201800 x  
  
 82 81  
 + 30664510802988208300 x + 132341572939212267400 x   
  
 80 79  
 + 535983370403809682970 x + 2041841411062132125600 x  
  
 78 77  
 + 7332066885177656269200 x + 24865270306254660391200 x  
  
 76 75  
 + 79776075565900368755100 x + 242519269720337121015504 x  
  
 ...

### 39. Application

Cobol: business. Data on employees, payroll and so on

Fortran: numerical computation. Numbers and almost nothing else

### 40. Application

Visual Basic: interfaces, teaching

Postscript and its compact cousin, PDF: printing and display

Cisco IOS (Internetwork Operating System): Networking hardware

Makefile, Scons, Ant, Ninja, Meson, etc.: languages for project code builds

R (derived from the earlier S): for statistical analysis

And so on

### 41. Application

It is so easy to create new language these days, people rarely stop to consider whether they should: is there an existing language that would suit this application well?

**Exercise** Go, Rust, Zig, Julia and Swift are new languages presently being developed. Look at them and decide what is new and different in each language (if anything)