I'm Not ~~Dead~~ undef Yet

Perl: persistent subroutine memory allocation

(follow up to Cees' Lightning Talk)

Source code available at
https://github.com/dnstandish/TPM_perl_persistent_subroutine_memory_allocation

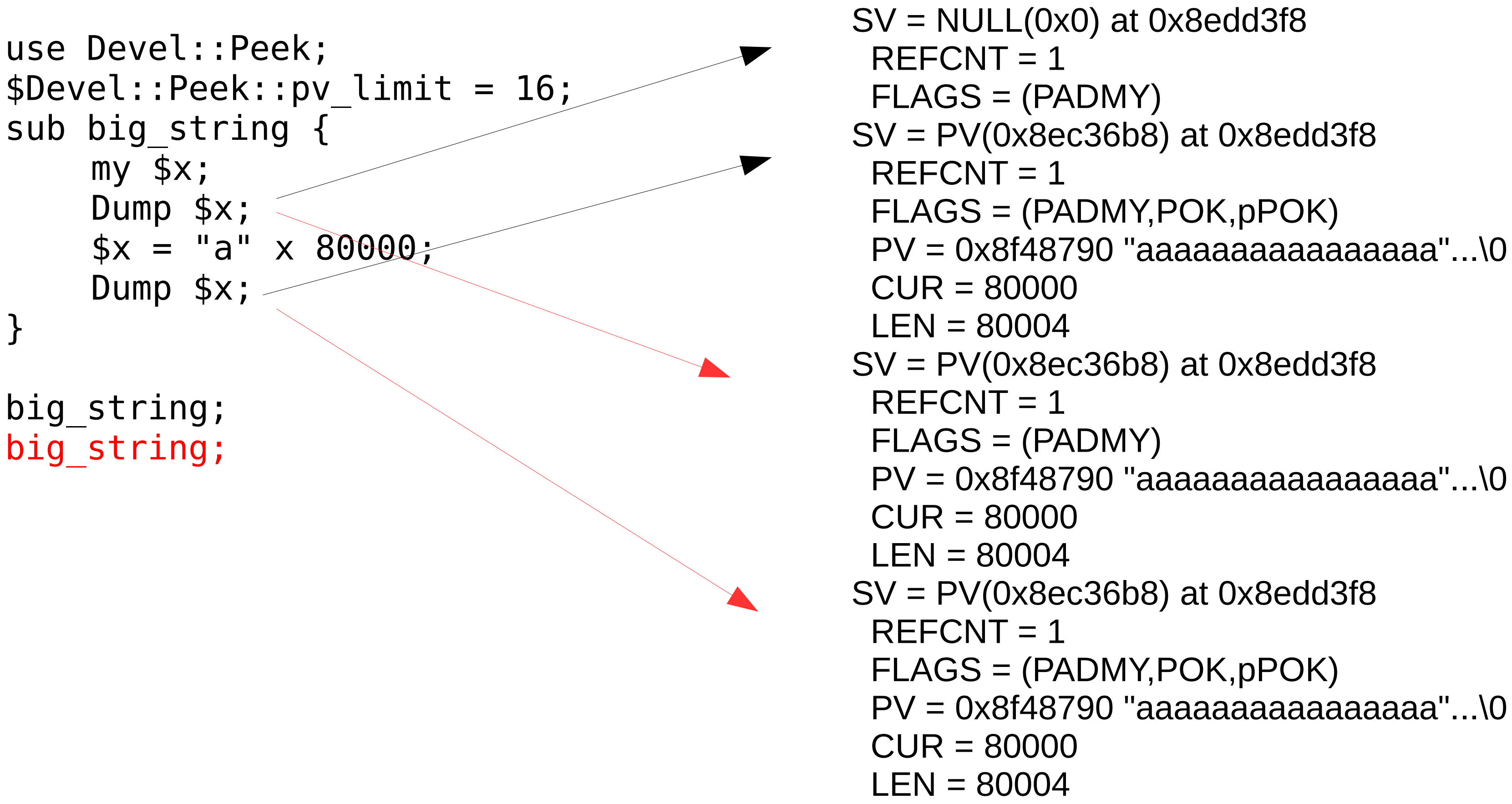
This presentation and actual output available under releases

This exploration was primarily conducted with perl 5.18.2 on Linux

At the end I'll compare to perl 5.28.1

Persistence of memory allocation in functions

- storage allocated to lexical variable persists between calls



```
use Devel::Peek;
$Devel::Peek::pv_limit = 16;
sub big_string {
    my $x;
    Dump $x;
    $x = "a" x 80000;
    Dump $x;
}

big_string;
big_string;
```

```
SV = NULL(0x0) at 0x8edd3f8
  REFCNT = 1
  FLAGS = (PADMY)
SV = PV(0x8ec36b8) at 0x8edd3f8
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x8f48790 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
SV = PV(0x8ec36b8) at 0x8edd3f8
  REFCNT = 1
  FLAGS = (PADMY)
  PV = 0x8f48790 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
SV = PV(0x8ec36b8) at 0x8edd3f8
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x8f48790 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
```

scripts/mem_test1.pl

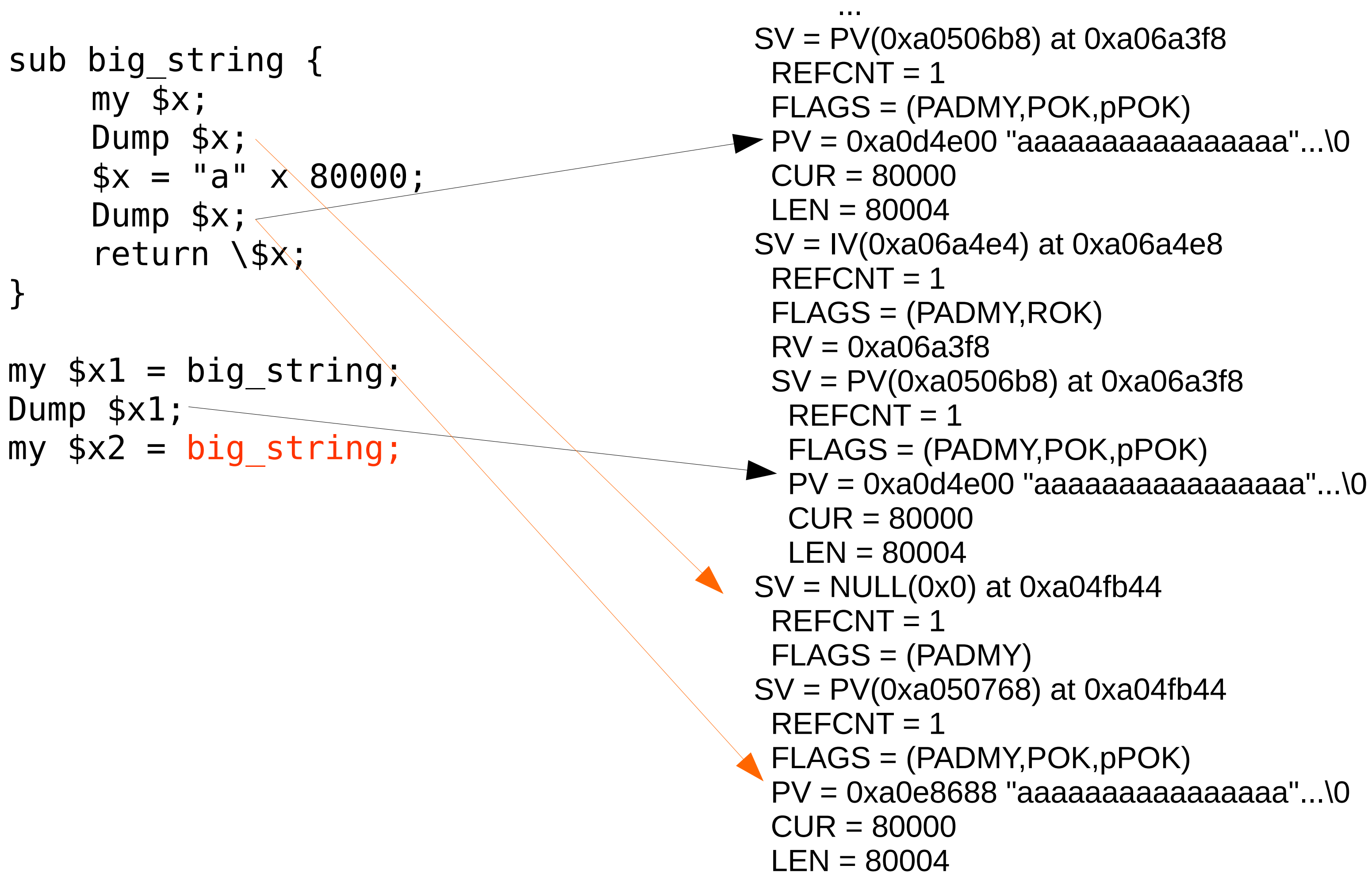
Returning a value does not deallocate

```
sub big_string {
    my $x;
    Dump $x;
    $x = "a" x 80000;
    Dump $x;
    return $x;
}

my $x1 = big_string;
Dump $x1;
my $x2 = big_string;
Dump $x2;
```

```
SV = NULL(0x0) at 0x854b3f8
  REFCNT = 1
  FLAGS = (PADMY)
SV = PV(0x85316b8) at 0x854b3f8
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x85b5df8 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
SV = PV(0x85317a8) at 0x854b4e8
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x85c9680 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
SV = PV(0x85316b8) at 0x854b3f8
  REFCNT = 1
  FLAGS = (PADMY)
  PV = 0x85b5df8 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
SV = PV(0x85316b8) at 0x854b3f8
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x85b5df8 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
SV = PV(0x85317b0) at 0x8566e40
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x85dcf08 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
```

scripts/mem_test2.pl

Returning a reference frees storage from function



```
sub big_string {
    my $x;
    Dump $x;
    $x = "a" x 80000;
    Dump $x;
    return \$x;
}

my $x1 = big_string;
Dump $x1;
my $x2 = big_string;
```

```
   ...
SV = PV(0xa0506b8) at 0xa06a3f8
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0xa0d4e00 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
SV = IV(0xa06a4e4) at 0xa06a4e8
  REFCNT = 1
  FLAGS = (PADMY,ROK)
  RV = 0xa06a3f8
  SV = PV(0xa0506b8) at 0xa06a3f8
    REFCNT = 1
    FLAGS = (PADMY,POK,pPOK)
    PV = 0xa0d4e00 "aaaaaaaaaaaaaaaa"...\0
    CUR = 80000
    LEN = 80004
SV = NULL(0x0) at 0xa04fb44
  REFCNT = 1
  FLAGS = (PADMY)
SV = PV(0xa050768) at 0xa04fb44
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0xa0e8688 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
```

scripts/mem_test3.pl

What about asigning undef?

```
sub big_string {
    my $x;
    Dump $x;
    $x = "a" x 80000;
    Dump $x;
    $x = undef;
    Dump $x;
    return;
}

big_string;
big_string;
```

```
SV = NULL(0x0) at 0x912e3f8
  REFCNT = 1
  FLAGS = (PADMY)
SV = PV(0x91146b8) at 0x912e3f8
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x9198de8 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
SV = PV(0x91146b8) at 0x912e3f8
  REFCNT = 1
  FLAGS = (PADMY)
  PV = 0x9198de8 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
SV = PV(0x91146b8) at 0x912e3f8
  REFCNT = 1
  FLAGS = (PADMY)
  PV = 0x9198de8 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
SV = PV(0x91146b8) at 0x912e3f8
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x9198de8 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
SV = PV(0x91146b8) at 0x912e3f8
  REFCNT = 1
  FLAGS = (PADMY)
  PV = 0x9198de8 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
```

scripts/mem_test4.pl

undef call works!

```
sub big_string {
    my $x;
    Dump $x;
    $x = "a" x 80000;
    Dump $x;
    undef $x;
    Dump $x;
}
big_string;
big_string;
```

```
SV = NULL(0x0) at 0x9c803f8
  REFCNT = 1
  FLAGS = (PADMY)
SV = PV(0x9c666b8) at 0x9c803f8
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x9ceade0 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
SV = PV(0x9c666b8) at 0x9c803f8
  REFCNT = 1
  FLAGS = (PADMY)
  PV = 0
SV = PV(0x9c666b8) at 0x9c803f8
  REFCNT = 1
  FLAGS = (PADMY)
  PV = 0
SV = PV(0x9c666b8) at 0x9c803f8
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x9ceade0 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 100012
SV = PV(0x9c666b8) at 0x9c803f8
  REFCNT = 1
  FLAGS = (PADMY)
  PV = 0
```

scripts/mem_test5.pl

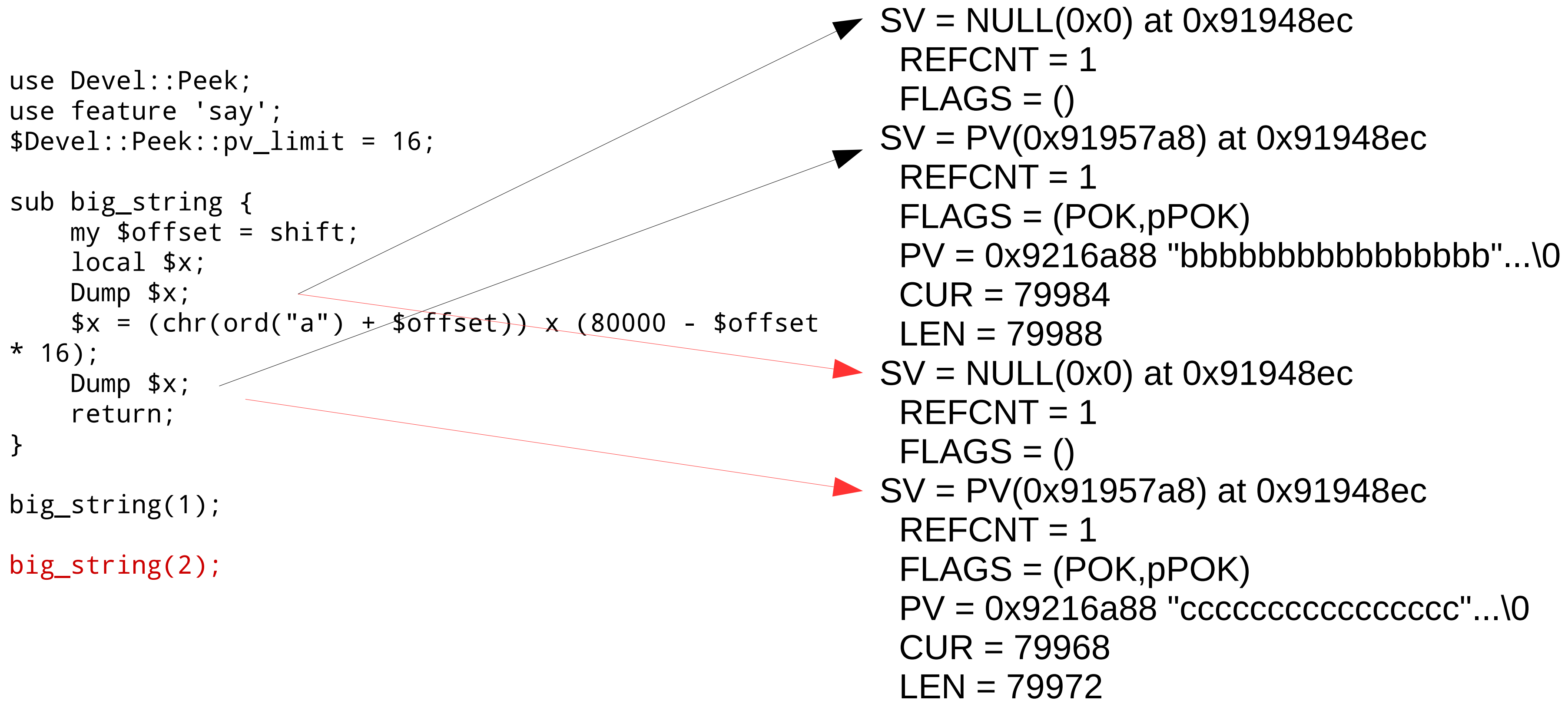
Assigning a value doesn't help

```
sub big_string {
    my $x;
    Dump $x;
    $x = "a" x 80000;
    Dump $x;
    $x = "b";
    Dump $x;
    $x = 1;
    Dump $x;
}
big_string;
```

```
SV = NULL(0x0) at 0x9a113f8
  REFCNT = 1
  FLAGS = (PADMY)
SV = PV(0x99f76b8) at 0x9a113f8
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x9a7bde8 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
SV = PV(0x99f76b8) at 0x9a113f8
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x9a7bde8 "b"\0
  CUR = 1
  LEN = 80004
SV = PVIV(0x9a12ca0) at 0x9a113f8
  REFCNT = 1
  FLAGS = (PADMY,IOK,pIOK)
  IV = 1
  PV = 0x9a7bde8 "b"\0
  CUR = 1
  LEN = 80004
```

scripts/mem_test6.pl

local instead of lexical



```
use Devel::Peek;
use feature 'say';
$Devel::Peek::pv_limit = 16;

sub big_string {
    my $offset = shift;
    local $x;
    Dump $x;
    $x = (chr(ord("a") + $offset)) x (80000 - $offset
* 16);
    Dump $x;
    return;
}

big_string(1);

big_string(2);
```

```
SV = NULL(0x0) at 0x91948ec
  REFCNT = 1
  FLAGS = ()
SV = PV(0x91957a8) at 0x91948ec
  REFCNT = 1
  FLAGS = (POK,pPOK)
  PV = 0x9216a88 "bbbbbbbbbbbbbbbb"...\0
  CUR = 79984
  LEN = 79988
SV = NULL(0x0) at 0x91948ec
  REFCNT = 1
  FLAGS = ()
SV = PV(0x91957a8) at 0x91948ec
  REFCNT = 1
  FLAGS = (POK,pPOK)
  PV = 0x9216a88 "cccccccccccccccc"...\0
  CUR = 79968
  LEN = 79972
```

local seems to free up allocation between calls

scritps/mem_test8.pl

Lots of functions with big variables will consume memory

- How to test?
- Devel::Peek::mstat()
  - not available
    - Devel::Peek::mstat: : perl not compiled with MYMALLOC
- on Linux can use /proc/<pid>/status

```
me@host:$ grep '^Vm'  /proc/2471/status
VmPeak:     9156 kB
VmSize:     9092 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:      4280 kB
VmRSS:      4280 kB
VmData:     2456 kB
VmStk:       132 kB
VmExe:       944 kB
VmLib:      2160 kB
VmPTE:        36 kB
VmSwap:        0 kB
```

```perl
package ProcVM;

use autodie;


sub print_proc_vm {

    open my $fd, '<', "/proc/$$/status";
    while ( <$fd> ) {
        print if /^Vm/;
    }
    close $fd;

}

1;
```

lib/ProcVM.pm

use Template to generate script with multiple functions

```
 ...
[% FOREACH suffix IN suf_list %]
sub f[% suffix %] {
    my $offset = shift;
    print "sub [% suffix %]\n";
[% IF use_devel_peek %]
    Dump($s);
[% END %]
    my $s = (chr(ord("a") + [% suffix %] + $offset)) x
[% size %]
;
[% IF use_devel_peek %]
    Dump($s);
[% END %]
[% IF assign_undef %]
    $s = undef;
[% END %]
[% IF call_undef %]
    undef($s);
[% END %]
[% IF use_devel_peek %]
    Dump($s);
[% END %]
    return "";
}
[% END %]
```

```
[% FOREACH i IN call_list %]
print "cycle [% i %]\n";
ProcVM::print_proc_vm();
[% FOREACH suffix IN suf_list %]
f[% suffix %]([% i %]);
ProcVM::print_proc_vm();
sleep(1);
[% END %]
[% END %]
```

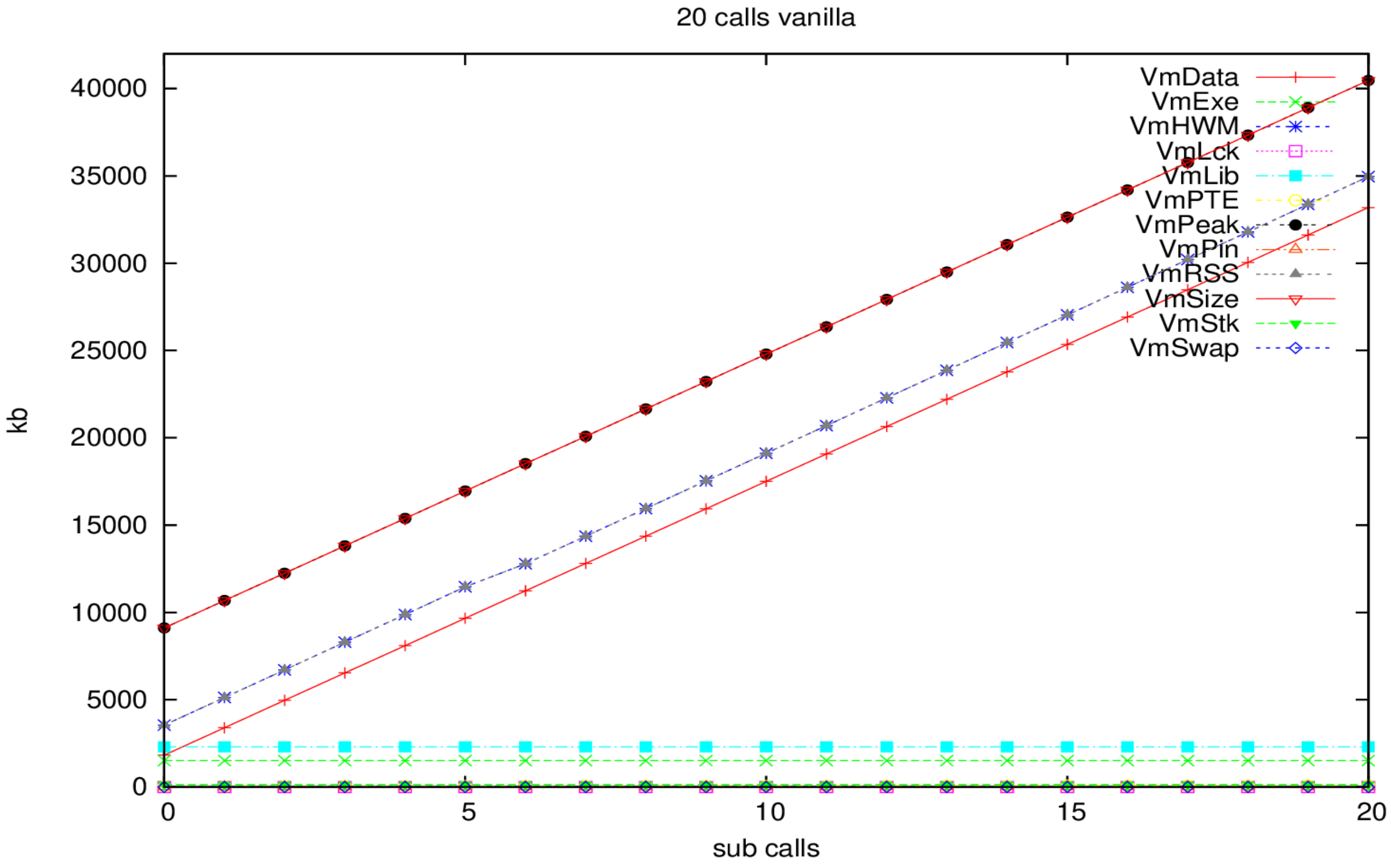
templates/many_func1.template

```
 ...
sub f1 {
    my $offset = shift;
    print "sub 1\n";
    my $s = (chr(ord("a") + 1 + $offset)) x 800000;
    return "";
}
 ...
sub f20 {
    my $offset = shift;
    print "sub 20\n";
    my $s = (chr(ord("a") + 20 + $offset)) x 800000;
    return "";
}
print "cycle 1\n";
ProcVM::print_proc_vm();
f1(1);
ProcVM::print_proc_vm();
 ...
sleep(1);
f20(1);
ProcVM::print_proc_vm();
sleep(1);
```

generated/func1.pl

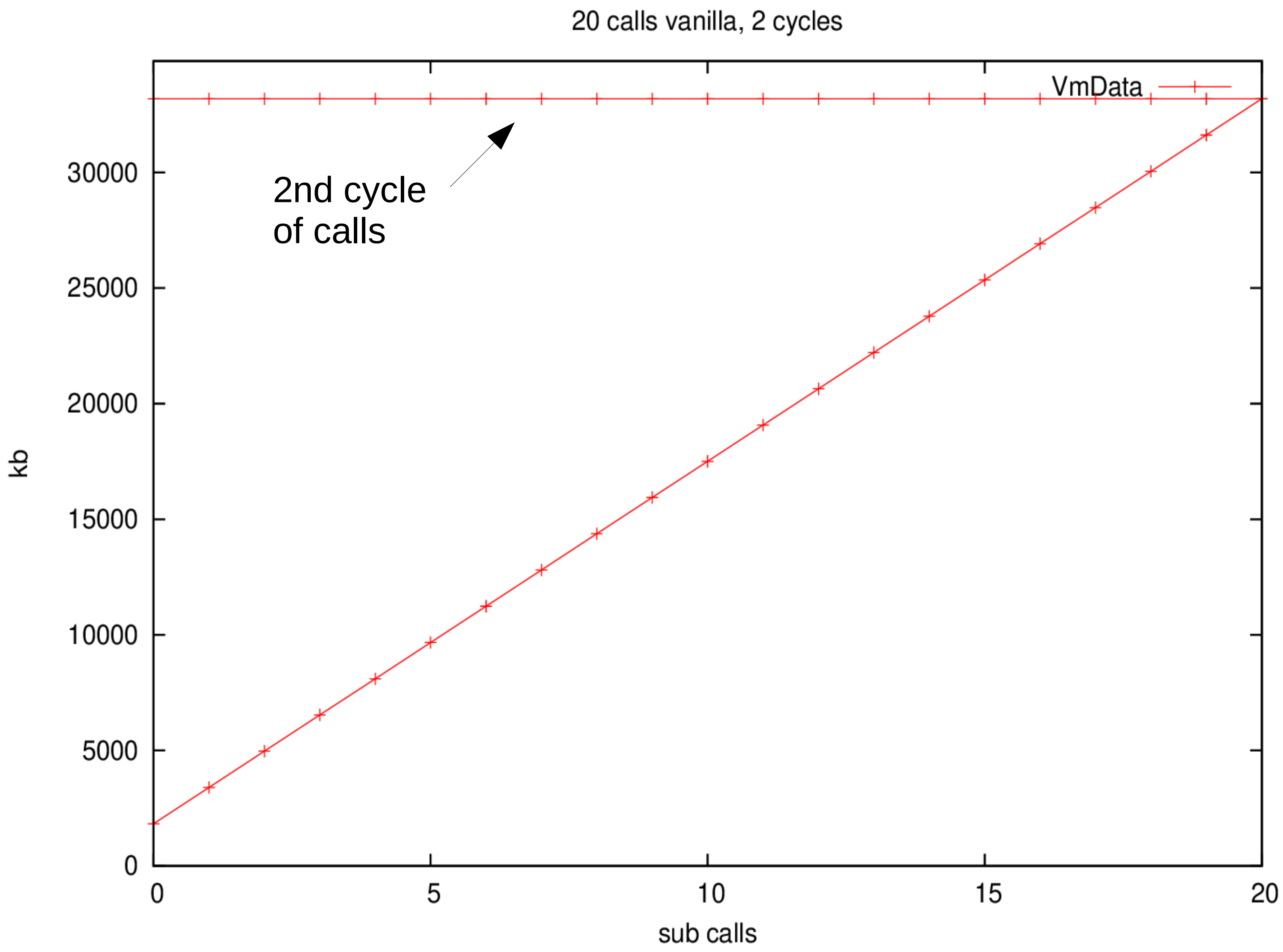
Parse output and use Chart::Gnuplot

per function memory consumption reflected in VmData, VmHWM, VMPeak, VmSize, VmRss



func1

This is not a leak: calling function more than once does not consume additional memory



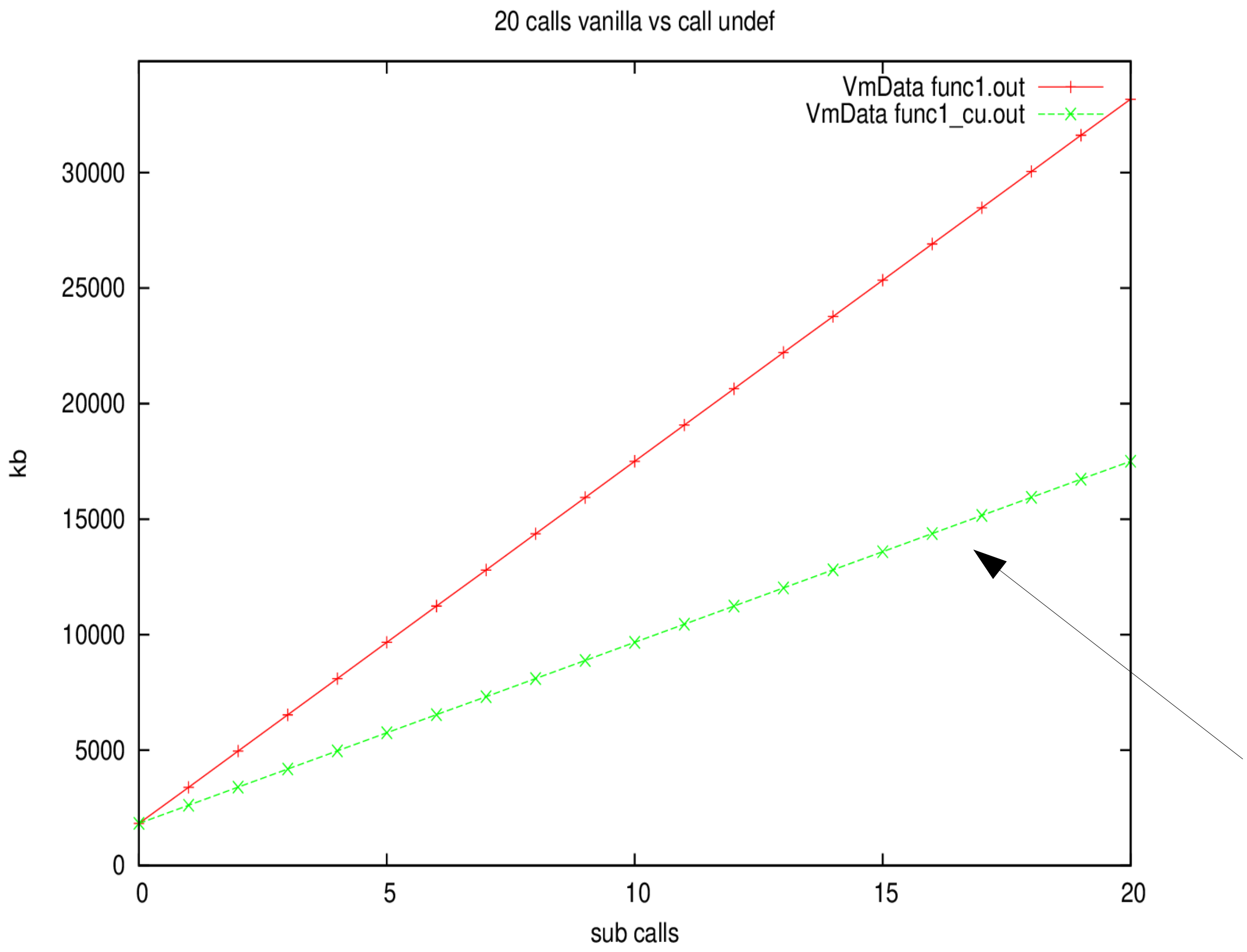
func1_c2

What happens if perl runs into a memory limit?

Does it reuse storage allocated to other functions? No, it dies.

```
./aa.out 16000000 func1.pl
setrlimit ok
execing func1.pl
template_file: many_func1.template
use_devel_peek:
size: 800000
n_func: 20
n_call: 1
assign_undef: 0
call_undef: 0
-----------------------
cycle 1
VmPeak:     9112 kB
VmSize:     9112 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:      3528 kB
VmRSS:      3528 kB
VmData:     1828 kB
VmStk:       136 kB
VmExe:      1508 kB
VmLib:      2292 kB
VmPTE:        28 kB
VmSwap:        0 kB
sub 1
VmPeak:    10680 kB
VmSize:    10680 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:      5112 kB
VmRSS:      5112 kB
VmData:     3396 kB
VmStk:       136 kB
VmExe:      1508 kB
VmLib:      2292 kB
VmPTE:        32 kB
VmSwap:        0 kB
 .
 .
 .
sub 4
VmPeak:    15384 kB
VmSize:    15384 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:      9864 kB
VmRSS:      9864 kB
VmData:     8100 kB
VmStk:       136 kB
VmExe:      1508 kB
VmLib:      2292 kB
VmPTE:        44 kB
VmSwap:        0 kB
sub 5
Out of memory!
```

Effect of calling undef on memory



Still see rise in memory use as each function called

Suspect memory is also allocated for temporary expression

```
sub f1 {
    my $offset = shift;
    print "sub 1\n";
    my $s = (chr(ord("a") + 1 + $offset)) x 800000;
    undef($s);
    return "";
}
```

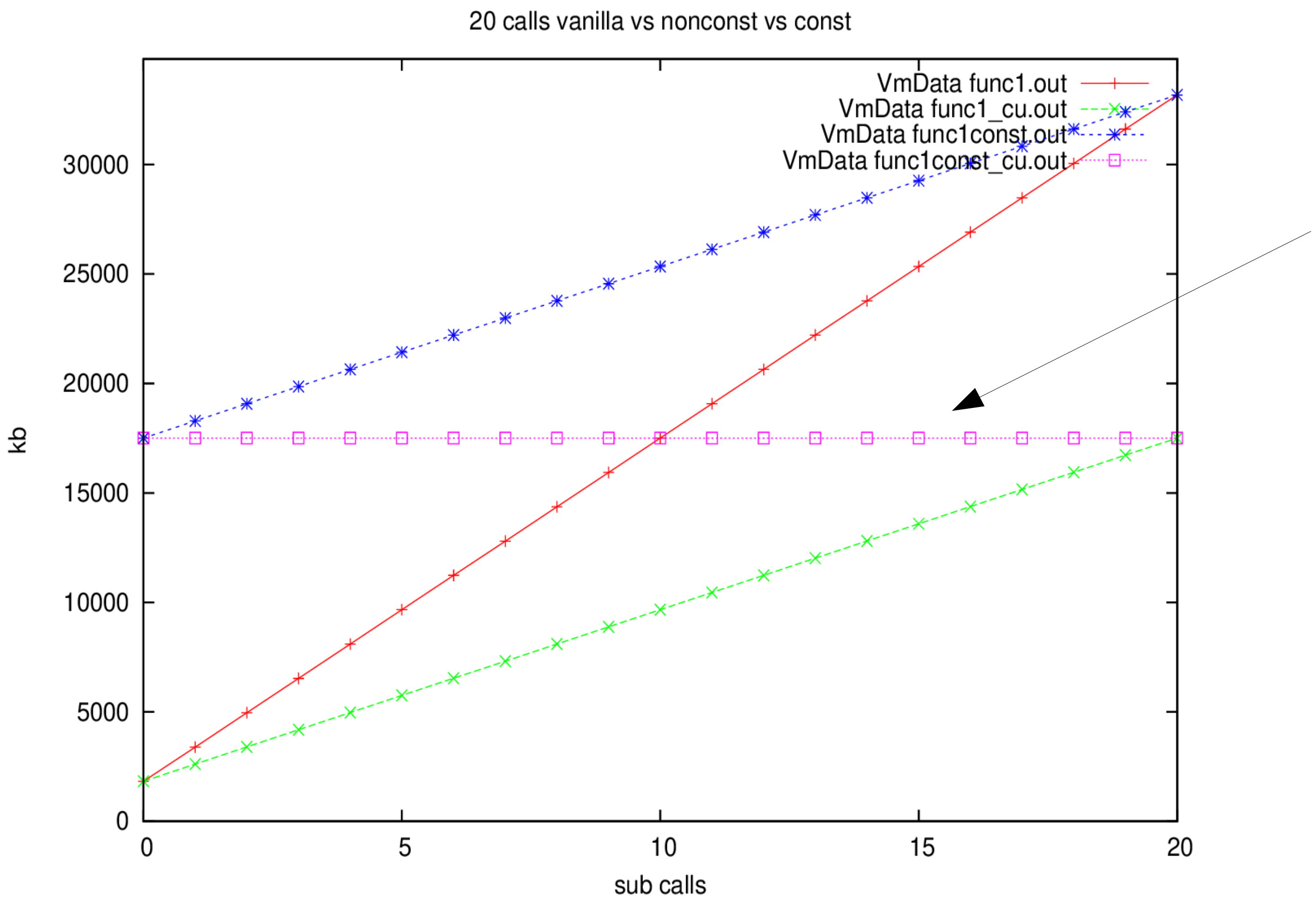
What if expression is constant?

```
sub f1 {
    my $offset = shift;
    print "sub 1\n";
    my $s = "a" x 800000;
    undef($s);
    return "";
}
```

templates/many_func1const.template

generated/func1const_cu.pl

What if temporary is a constant? Is there optimization



- Yes, constant allocated before call
- not shared between functions

func1_effect_const.eps

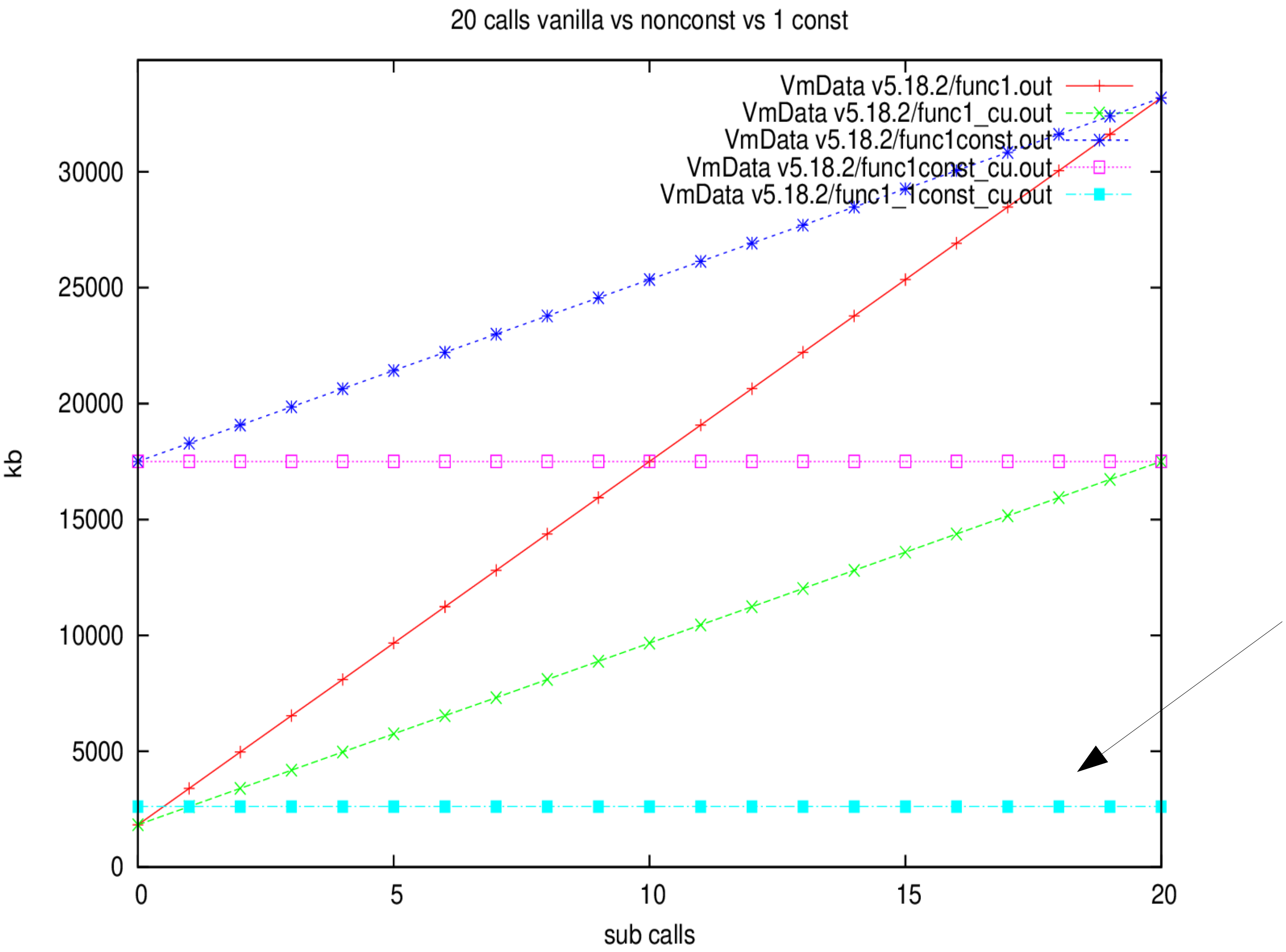
Put constant into a separate function

```
sub constant {
   return "a" x 800000;
}
sub f1 {
    my $offset = shift;
    print "sub 1\n";
    my $s = constant();

    undef($s);
    return "";
}
...
```

templates/many_func1_1const.template

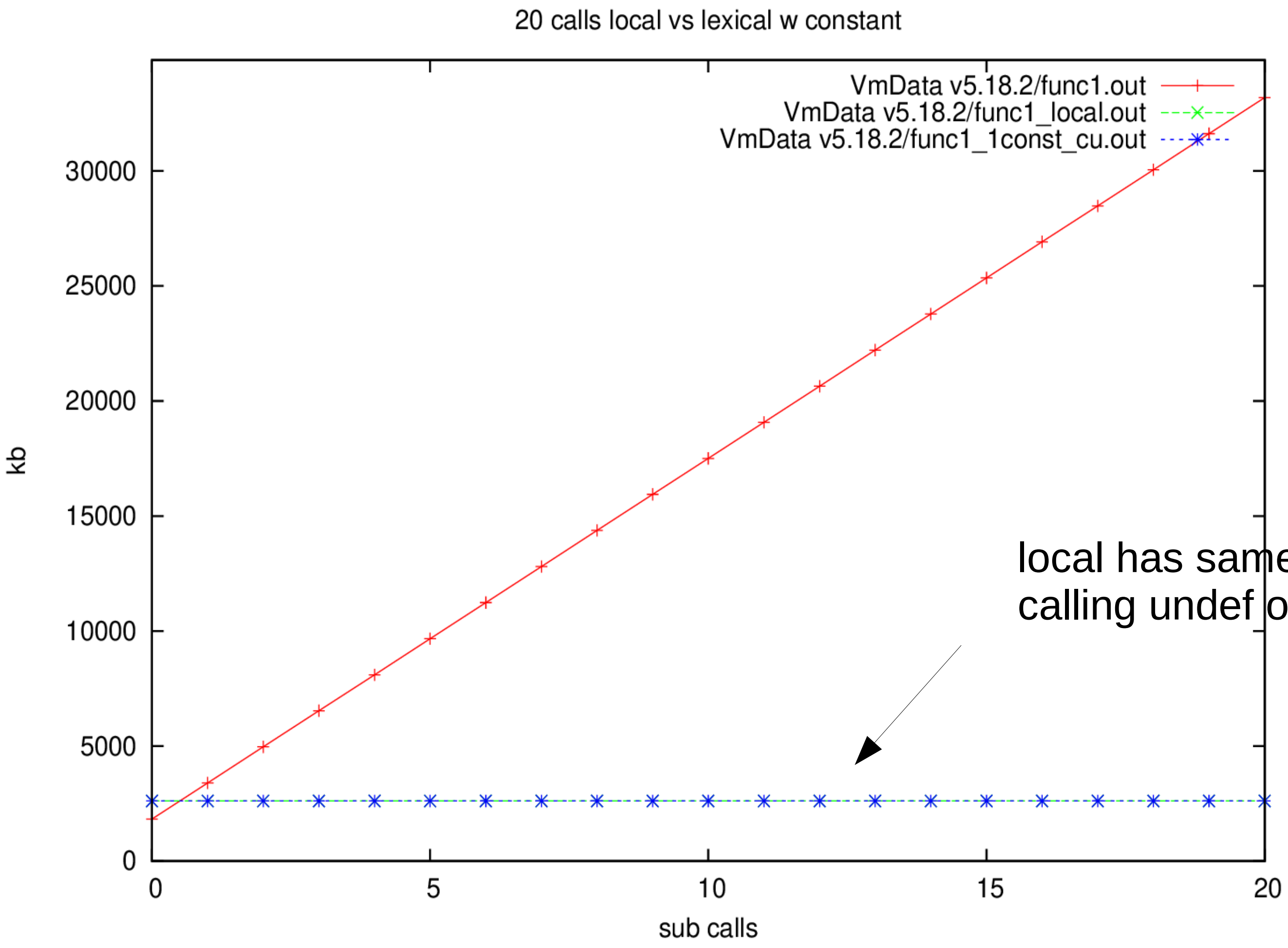
generated/funct1_1const_cu.pl

Segregating constant into function combined with calling undef minimizes memory use



func1_effect_1const.eps

Local with single constant



local has same footprint as calling undef on a lexical

templates/many_func1_local.template

func1_local.eps

Closures

Perl closures shouldn't cross contaminate unintentionally

Suggests that closure use of sub lexical variable will prevent reuse of allocated storage

```
sub sub_gen {
    my $offset = shift;
    print STDERR "gen $offset\n";
    my $s;
    Dump $s;
    $s = (chr(ord("a") + $offset)) x 800000;
    Dump $s;
    return {
        info => sub {
            print STDERR "info $offset\n";
            Dump $s;
        },
        clear => sub {
            print STDERR "clear $offset\n";
            undef( $s );
       },
    }
}
my $c1 = sub_gen( 1 );
my $c2 = sub_gen( 2 );
$c1->{info}->();
$c1->{clear}->();
$c1->{info}->();
$c2->{info}->();
$c2->{clear}->();
$c2->{info}->();
```

scripts/mem_test9.pl

```
my $c1 = sub_gen( 1 );
```

```
gen 1
SV = NULL(0x0) at 0x8c015dc
  REFCNT = 1
  FLAGS = (PADMY)
SV = PV(0x8be77b0) at 0x8c015dc
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x406c2008 "bbbbbbbbbbbbbbbb"...\0
  CUR = 800000
  LEN = 800004
```

```
my $c2 = sub_gen( 2);
```

```
gen 2
SV = NULL(0x0) at 0x8c39e6c
  REFCNT = 1
  FLAGS = (PADMY)
SV = PV(0x8be77b8) at 0x8c39e6c
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x40786008 "cccccccccccccccc"...\0
  CUR = 800000
  LEN = 800004
```

Lexical variable $s has different SV and PV for each call to sub_gen

```
gen 1
SV = NULL(0x0) at 0x8c015dc
  REFCNT = 1
  FLAGS = (PADMY)
SV = PV(0x8be77b0) at 0x8c015dc
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x406c2008 "bbbbbbbbbbbbbbbb"...\0
  CUR = 800000
  LEN = 800004
```

```
$c1->{info}->();
```

```
info 1
SV = PV(0x8be77b0) at 0x8c015dc
  REFCNT = 2
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x406c2008 "bbbbbbbbbbbbbbbb"...\0
  CUR = 800000
  LEN = 800004
```

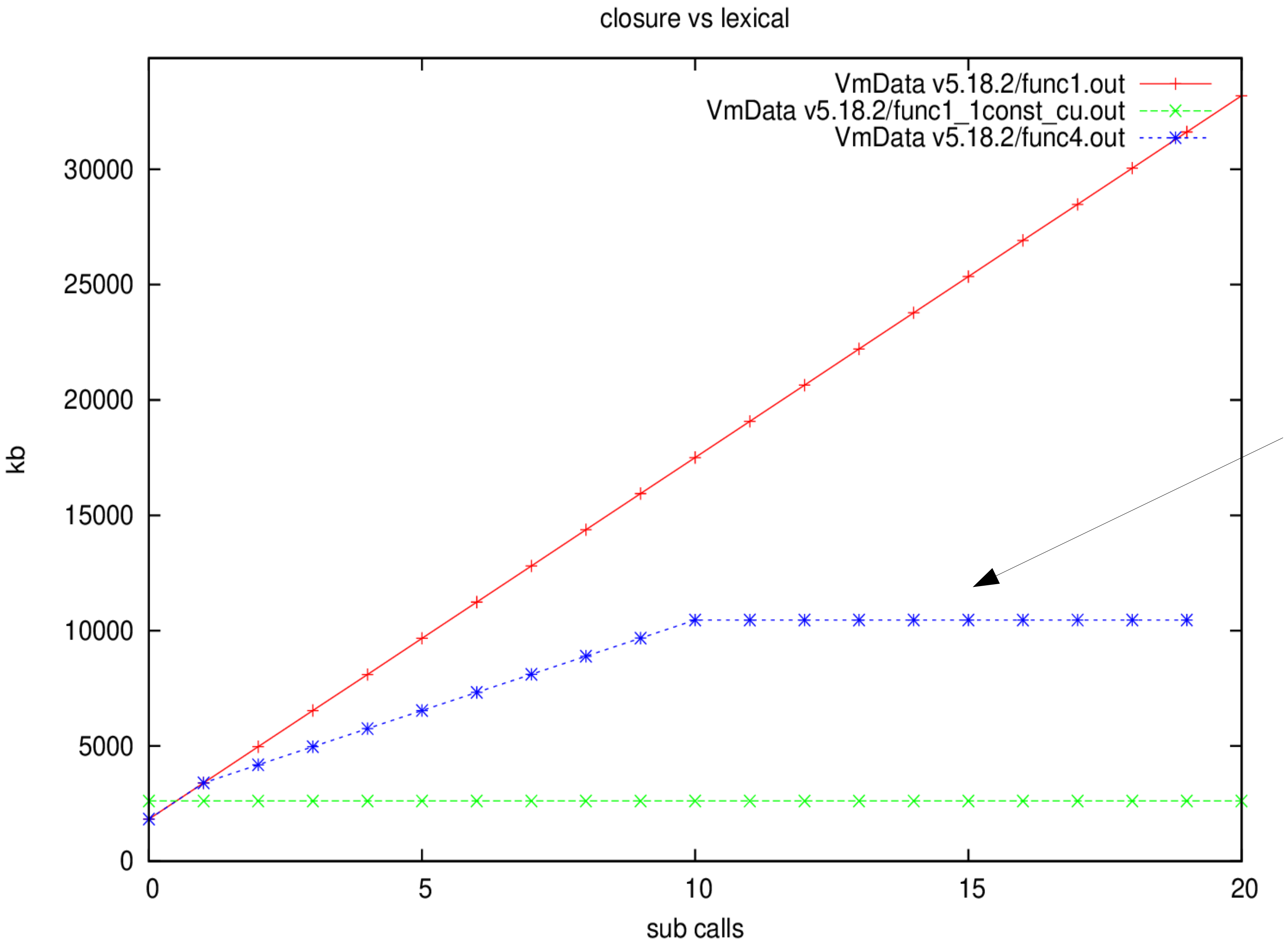
Closure takes ownership of lexical variable from generating function

Closure memory consumption test

- generate 10 closures
- generate 10 more, but invoke clear->() to undef closure variable

templates/many_func1.template

Closure memory use



func4.eps

Is Closure allocated memory released when ref count hits zero?

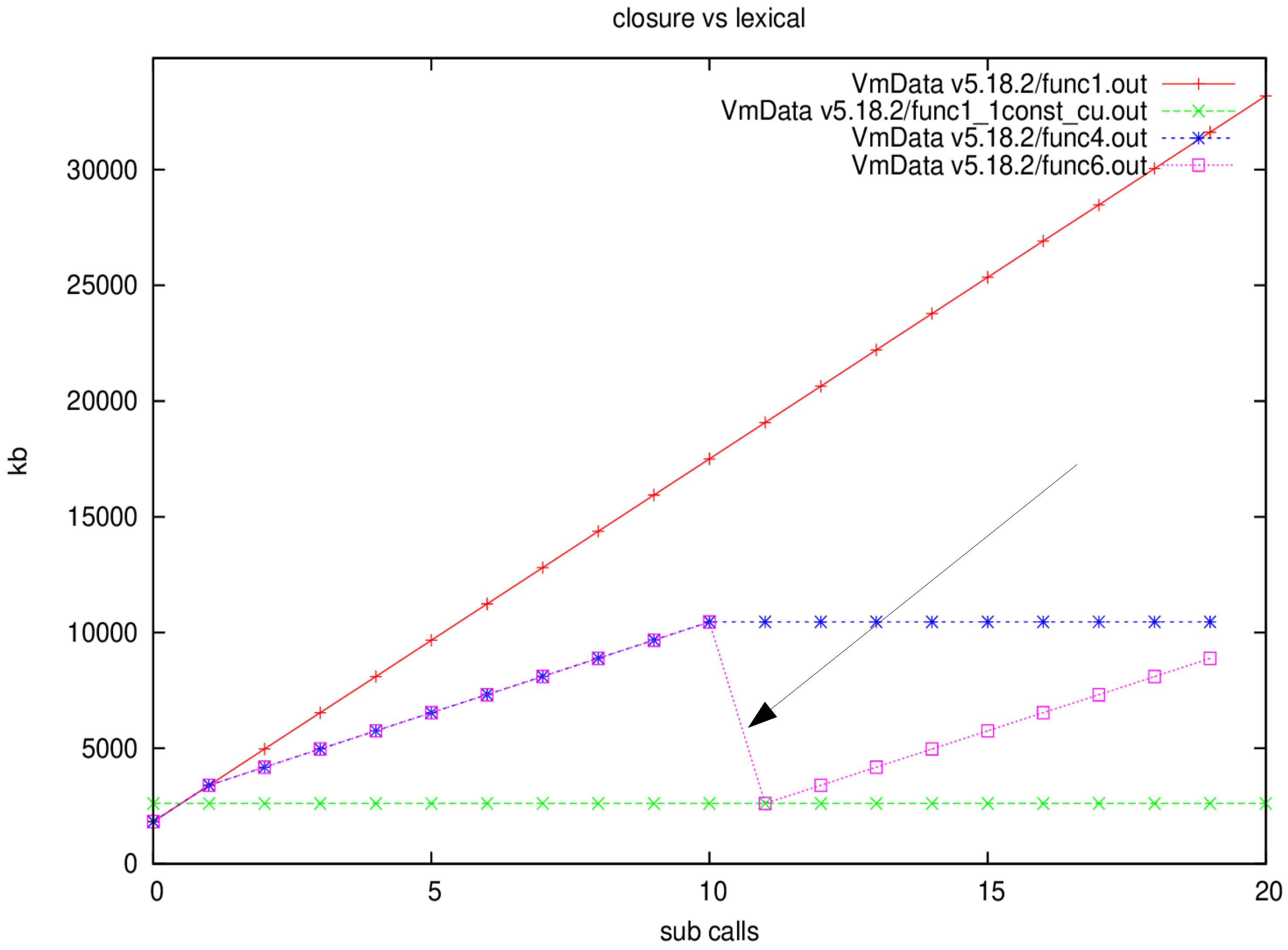
Expect allocated storage to be released once there are no more references

Otherwise this would be a source of memory leaks

- generate 10 closures and store in array
- empty the array
- generate 10 more closures

templates/many_func6.template

Closure memory is deallocated



func6.eps

Recursion

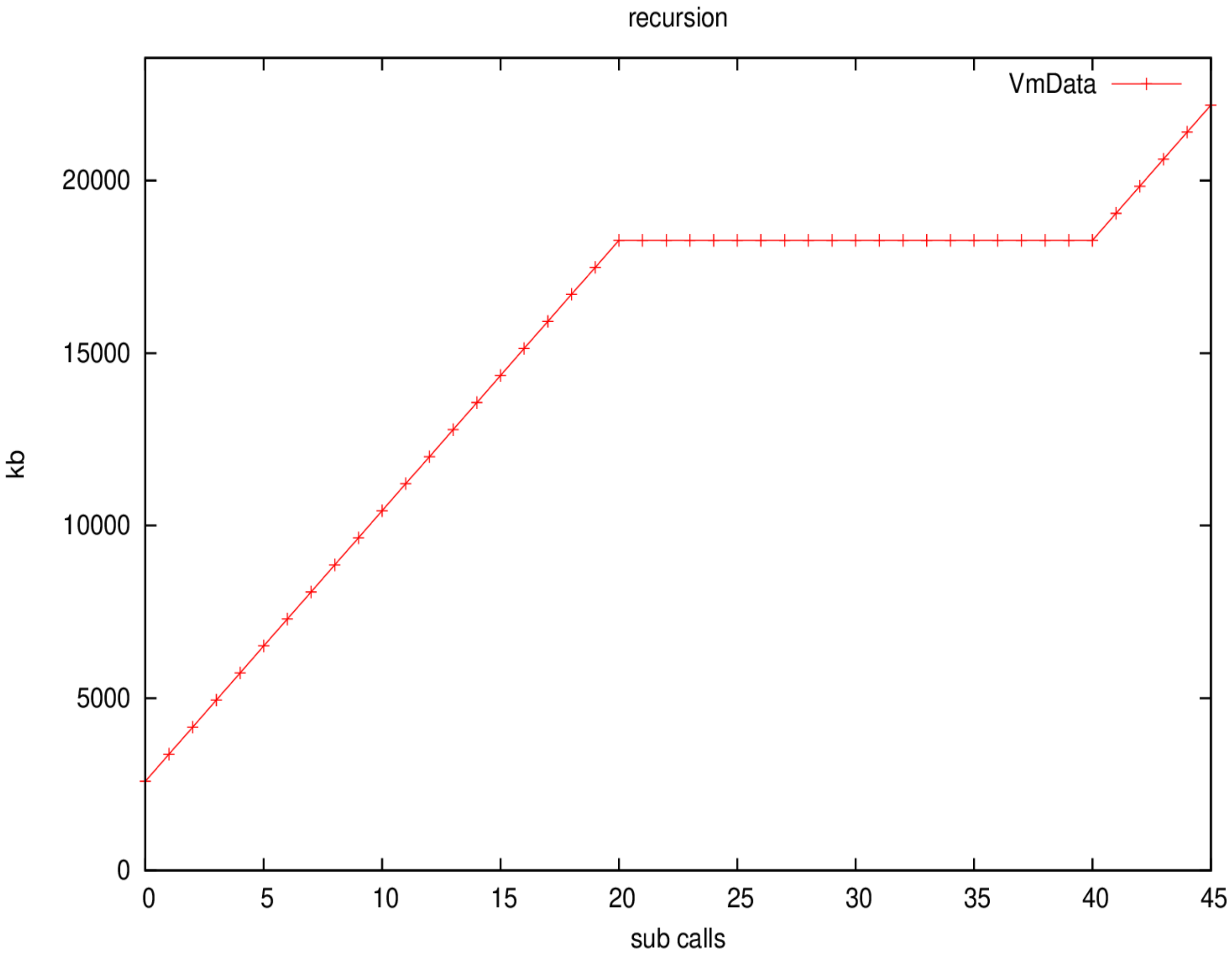
If functions hold on to allocated memory for lexical variables, how does this work with recursive functions?

```
sub get_const {
    return chr(ord("a")) x 800000;
}

sub recur {
    my $cnt = shift;
    my $limit = shift;
    return if $cnt >= $limit;
    $cnt++;
    ProcVM::print_proc_vm();
    print "sub $cnt\n";
    my $s;
    Dump $s;
    $s = get_const();
    Dump $s;
    return recur($cnt, $limit);
}

print "cycle 1\n";
recur(0, 20);
recur(20, 45);
ProcVM::print_proc_vm();
```

opportunity for tail recursion optimization

scripts/recur1.pl

Memory allocated and retained independently by recursion depth



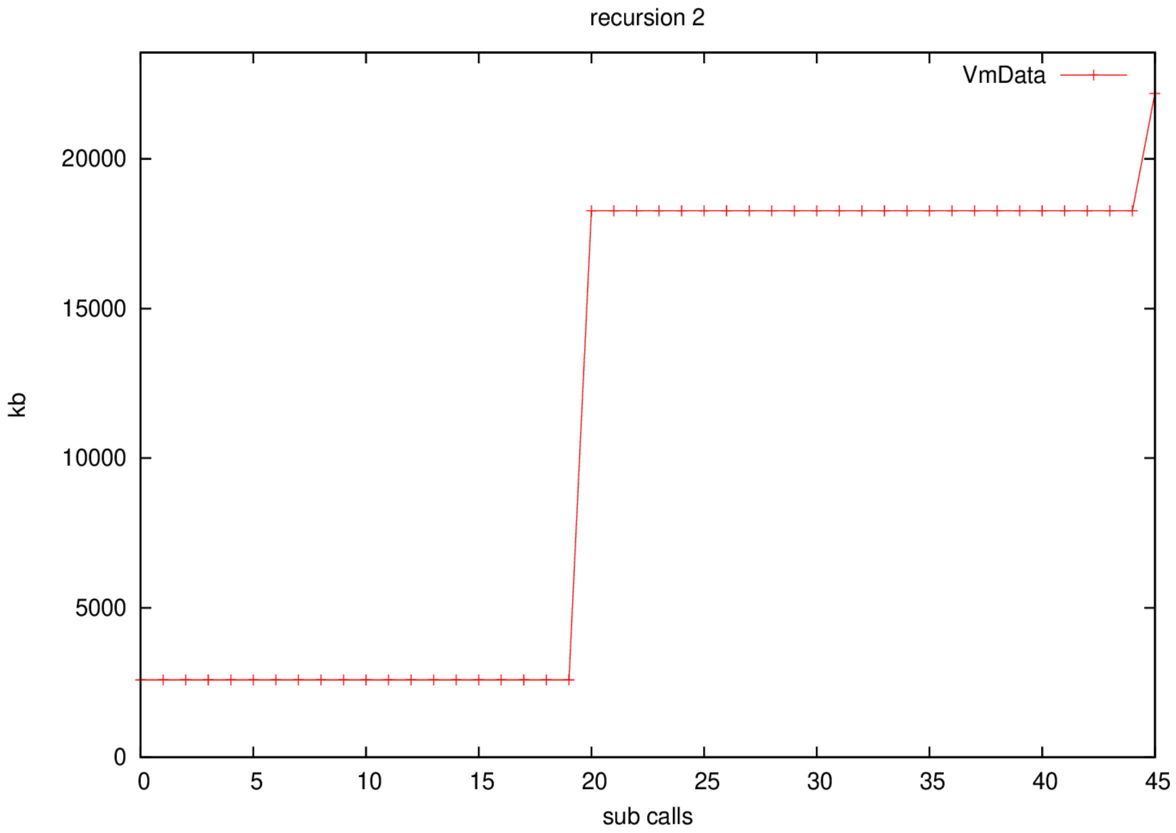
recur1.eps

What if scalar isn't defined until after recursive call?

```
sub get_const {
    return chr(ord("a")) x 800000;
}

sub recur {
    my $cnt = shift;
    my $limit = shift;
    return if $cnt >= $limit;
    $cnt++;
    ProcVM::print_proc_vm();
    print "sub $cnt\n";
    recur($cnt, $limit);
    my $s;
    Dump $s;
    $s = get_const();
    Dump $s;
}

print "cycle 1\n";
recur(0, 20);
recur(20, 45);
ProcVM::print_proc_vm();
```

scripts/recur2.pl

Doesn't make much difference



Measurement of memory usage precedes memory consumption. Allocation not apparent until second time through

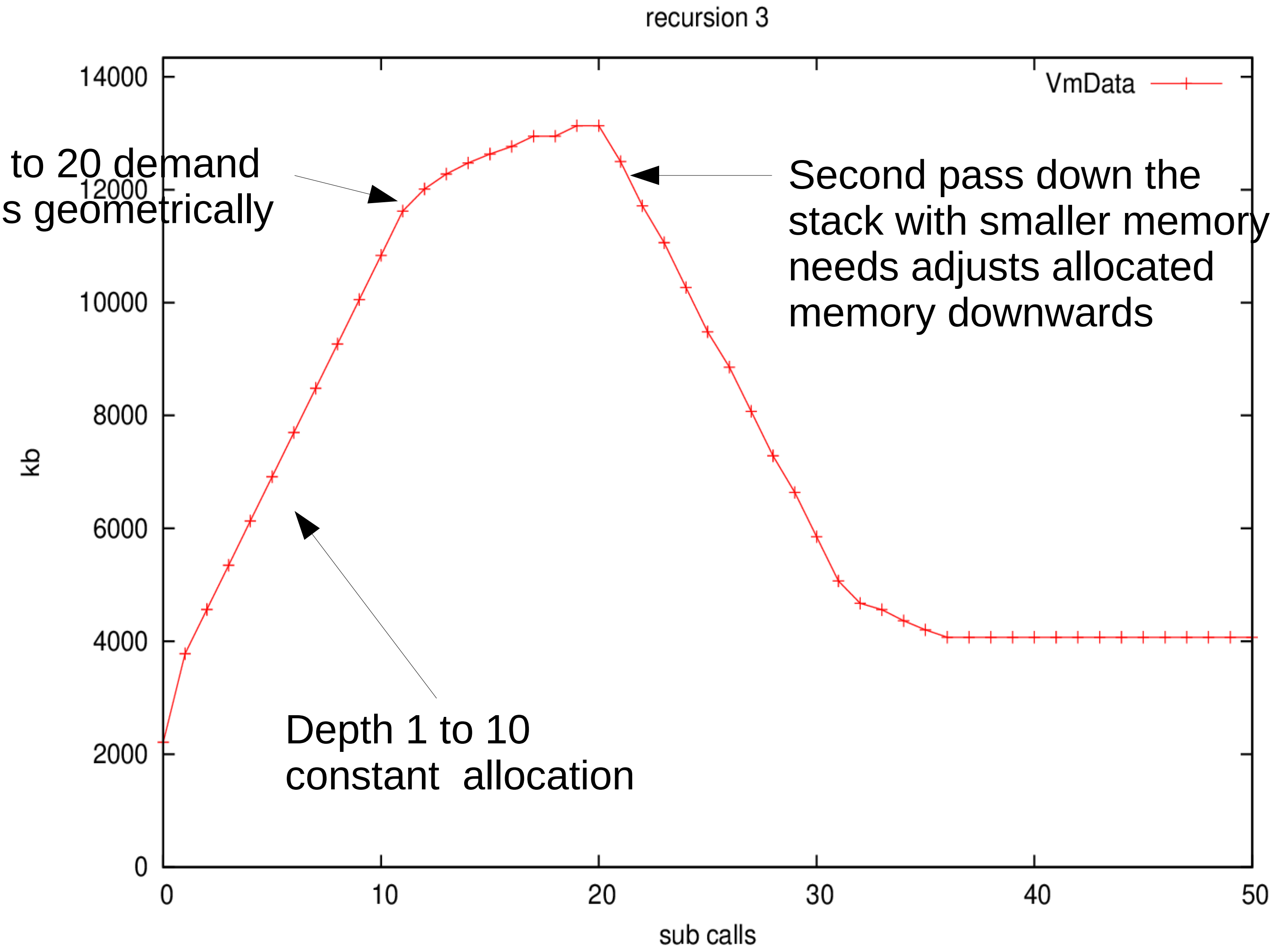
recur2.eps

Is memory allocation at given depth based on actual use at that depth?

```
sub get_const {
    my $descale = shift;
    $descale -= 10;
    $descale = 1 if $descale < 1;
    return chr(ord("a")) x ( 800000 / $descale );
}

sub recur {
    my $cnt = shift;
    my $limit = shift;
    return if $cnt >= $limit;
    $cnt++;
    ProcVM::print_proc_vm();
    print "sub $cnt\n";
    my $s;
    Dump $s;
    $s = get_const( $cnt );
    Dump $s;
    return recur($cnt, $limit);
}

print "cycle 1\n";
recur(0, 20);
recur(20, 45);
recur(45, 50);
ProcVM::print_proc_vm();
```

scripts/recur3.pl

Retained allocation based on usage and is adjusted if need decreases next time through



Depth 11 to 20 demand
decreases geometrically

Second pass down the
stack with smaller memory
needs adjusts allocated
memory downwards

Depth 1 to 10
constant allocation

recur3.eps

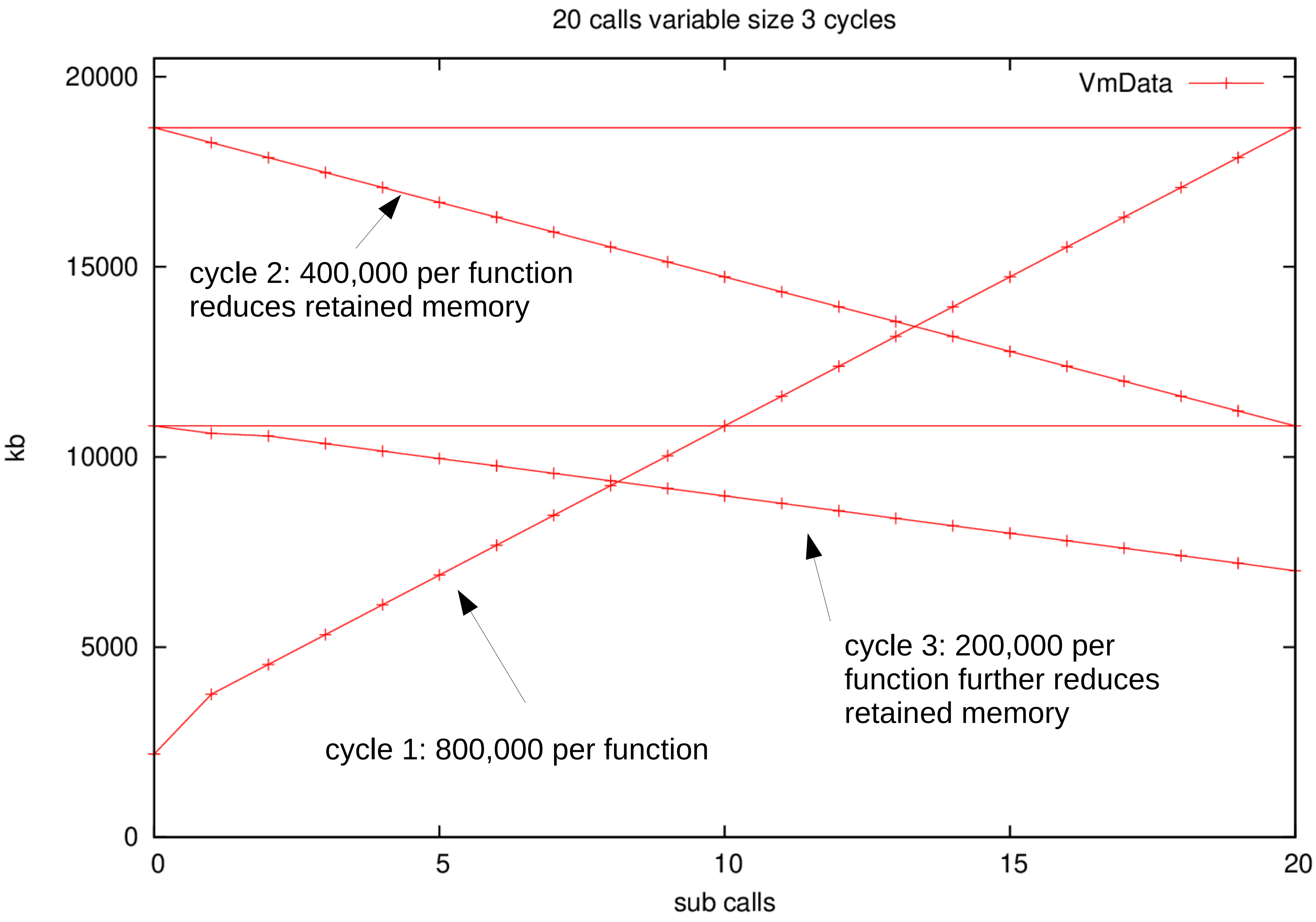
Is memory allocation adjusted in non-recursive case?

Revisit previous test but with variable memory needs

- 20 functions using separate sub to generate big scalar
- cycle 1 full 800,000 character scalar
- cycle 2 half size 400,000 character scalar
- cycle 3 quarter size 200,000 character scalar

templates/many_func1var.template

Allocated memory adjusts based on usage



cycle 2: 400,000 per function reduces retained memory

cycle 1: 800,000 per function

cycle 3: 200,000 per function further reduces retained memory

func1var.eps

What is happening? Is memory retained or not?

```
sub one_big {
    my $size = shift;
    return "a" x $size;
}
sub big_string {
    my $size = shift;
    my $x;
    Dump $x;
    $x = one_big($size);
    Dump $x;
}

big_string( 80_000 );
big_string( 40_000 );
```

scripts/mem_test12.pl

In at least some cases where scalar set via return value from another function, retained memory is not reused.

```
SV = NULL(0x0) at 0x8bdb704
  REFCNT = 1
  FLAGS = (PADMY)
SV = PV(0x8ba57a8) at 0x8bdb704
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x8c2a708 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
```

```
SV = PV(0x8ba57a8) at 0x8bdb704
  REFCNT = 1
  FLAGS = (PADMY)
  PV = 0x8c2a708 "aaaaaaaaaaaaaaaa"...\0
  CUR = 80000
  LEN = 80004
SV = PV(0x8ba57a8) at 0x8bdb704
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x8c3df90 "aaaaaaaaaaaaaaaa"...\0
  CUR = 40000
  LEN = 40004
```

mem_test12.err

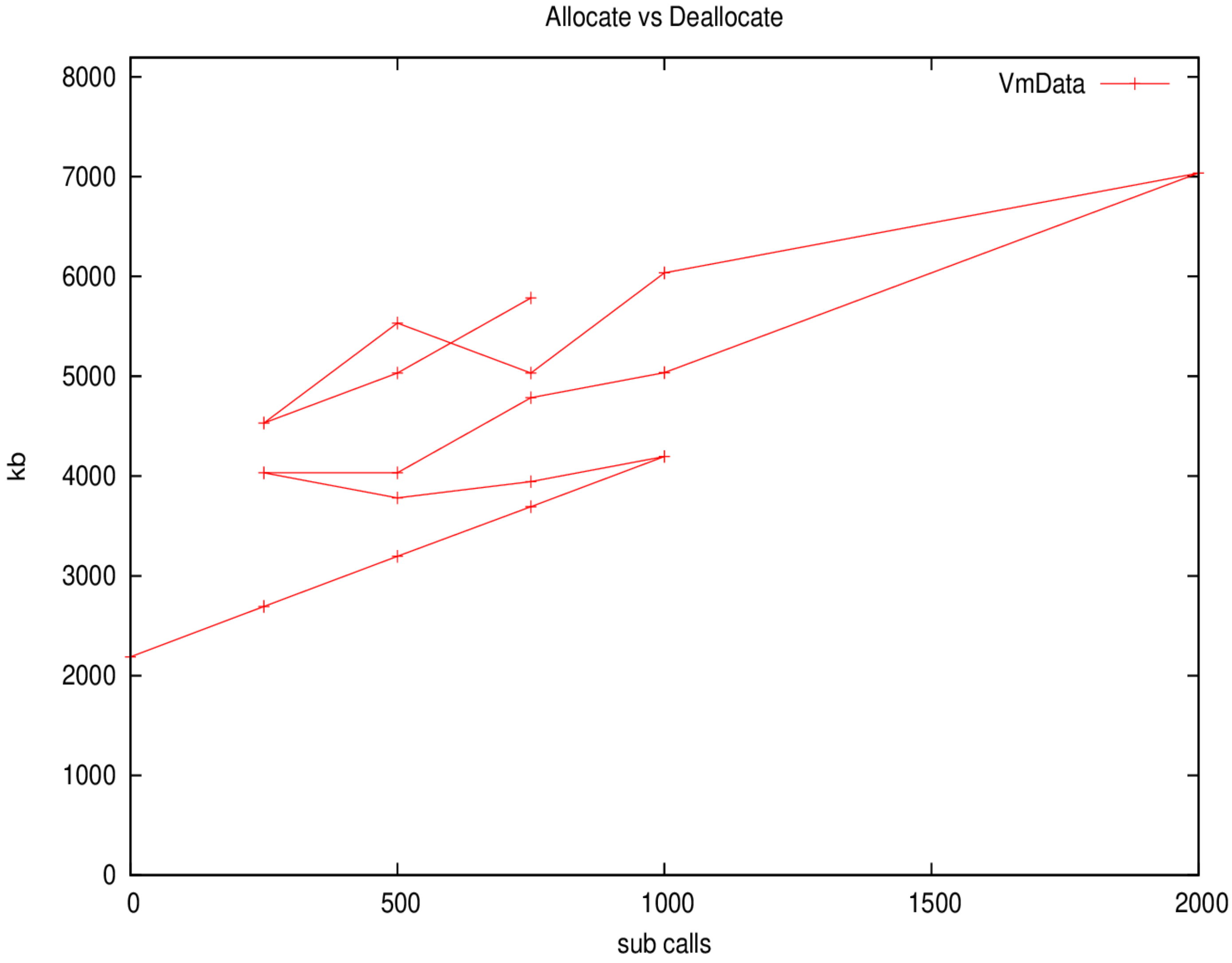
More complicated example of allocation / deallocation

```
sub big_string {
    my $size = shift;
    return "a" x $size;
}

sub mem_test {
    my $size = shift;
    print STDERR "mem_test $size\n";
    my $s;
    Dump $s;
    big_string( $size );
    $s = big_string( $size );
    Dump $s;
    return;
}
```

```
print "cycle 1\n";
my $i = 1;
for my $kb (
    250, 500, 750, 1000,
    750, 500, 250, 500, 750, 1000, 2000,
    1000, 750, 500, 250, 500, 750 )
{
    ProcVM::print_proc_vm();
    print "sub $kb\n";
    mem_test( $kb * 1024 - 4 );
    $i++
}
ProcVM::print_proc_vm();
```

scripts/mem_test10.pl

Allocation/deallocation interact with memory management in complcated ways



mem_test10.eps

Perl 5.28.1

5.28.1 demonstrates many of the same behaviours as 5.18.2 but there are some differences

- Storage allocated to lexical variables still persists between calls
- Since 5.20 perl adds copy on write (COW). Returning a scalar does not immediately create a new copy of a string.

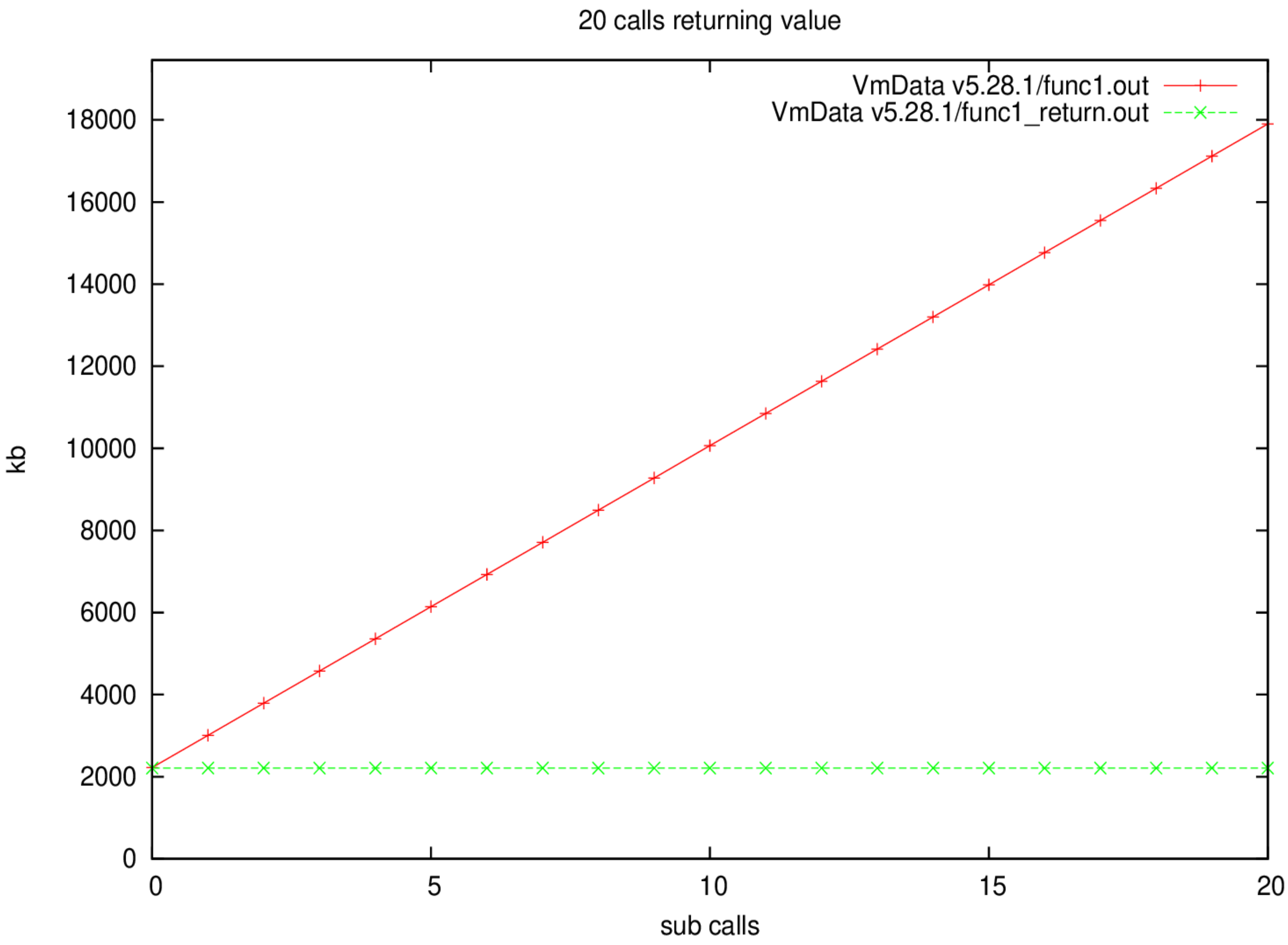
It takes some contortions to reduce persistent memory allocationvia returned value

```
sub f1 {
    my $offset = shift;
    print "sub 1\n";
    my $s = (chr(ord("a") + 1 + $offset)) x 800000;
    return $s;
}
...

{
    my $x =  f1(1);
}
ProcVM::print_proc_vm();
{
    my $x =  f2(1);
}
ProcVM::print_proc_vm();
...
```

templates/many_func1_return.template

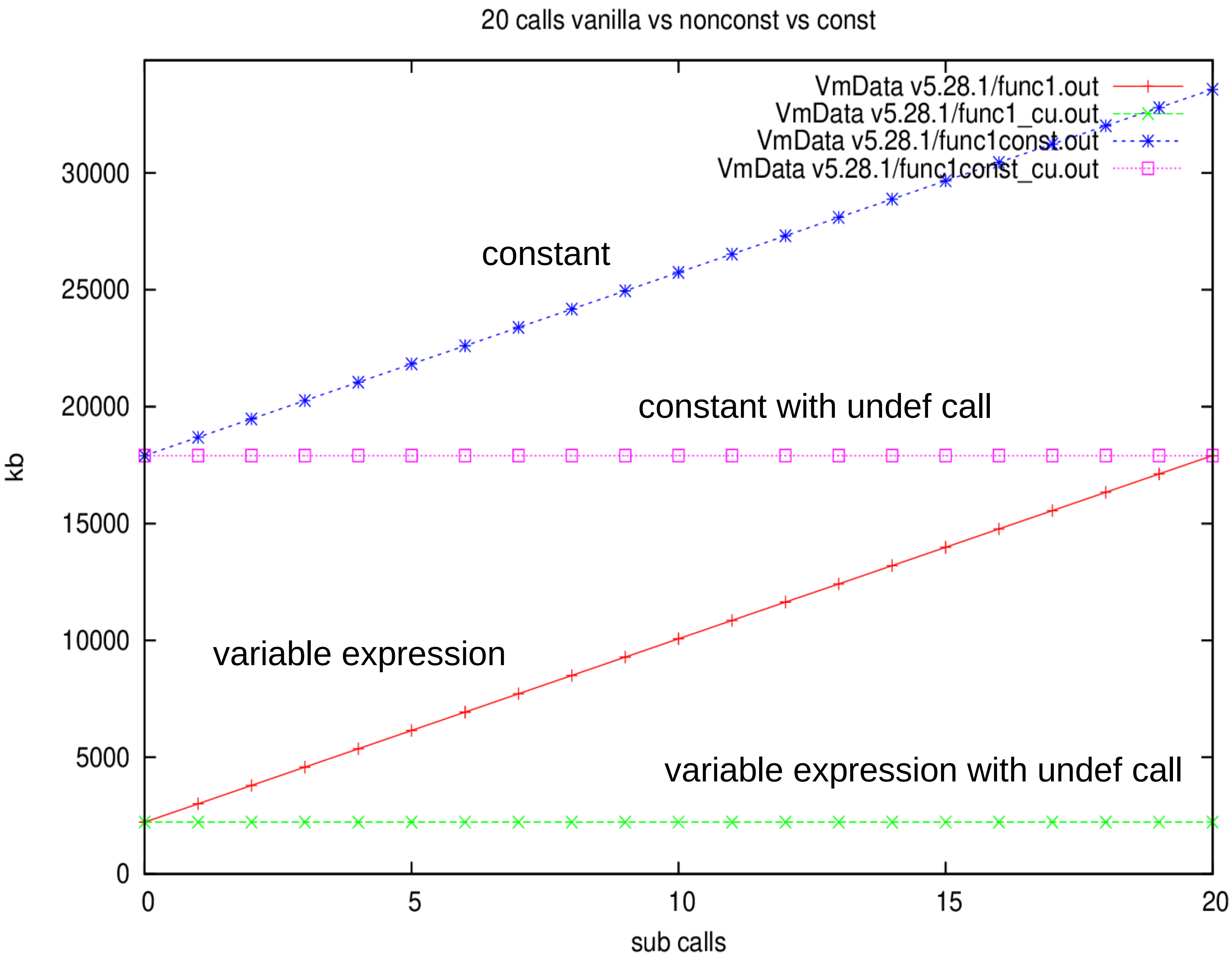
generated/func1_return.pl

5.28.1 returning value vs not returning



func1_return.eps

- Perl 5.28 does not consume memory for temporary expressions the way 5.18 did
- but, replacing a variable expression with a constant expression does consume memory for the constant



constant

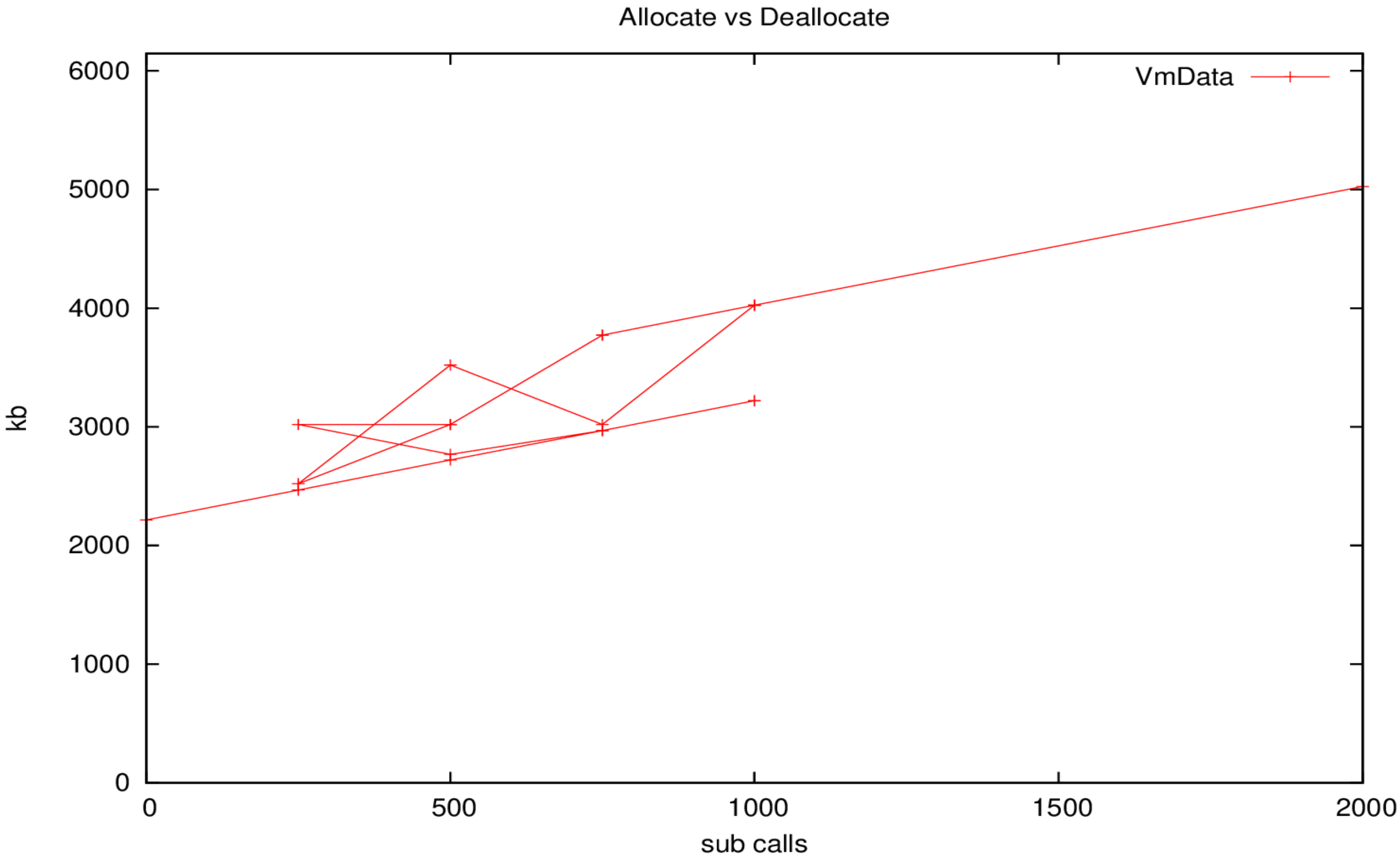
constant with undef call

variable expression

variable expression with undef call

func1_effect_const.eps

Allocation/deallocation still interacts with memory management in complicated ways



mem_test10.eps

summary

- Lexical variable memory retained between calls
- Assigning a new value doesn't “usually” release memory.
- Assigning undef doesn't release memory
- Calling undef does release memory
- Hitting a hard memory limit does not force release of memory

- Memory allocated for local variables is released (but we don't want to go there!)
- Return by value may release memory under some circumstances for recent versions of perl
- Returning a reference prevents retention of memory by function
- Memory may also be allocated for evaluating expressions. Less so for recent versions of perl
- Memory for constant expressions may be allocated at “compile” time.

- Closures hand of allocated storage as if they were references. Storage handed off for closures released when ref count hits zero.
- Recursion allocates and retains memory at each level of call depth
- Memory allocation may adjust in subsequent calls, but may depend on specifics of how scalar is populated

Conclusion

Memory management is complicated

“*We should forget about small efficiencies, say about 97% of the time:* ***premature optimization is the root of all evil****. Yet we should not pass up our opportunities in that critical 3%*"

Donald Knuth

man proc(5)

- VmPeak: Peak virtual memory size.
- VmSize: Virtual memory size.
- VmLck: Locked memory size (see mlock(3)).
- VmHWM: Peak resident set size ("high water mark").
- VmRSS: Resident set size.
- VmData, VmStk, VmExe: Size of data, stack, and text segments.
- VmLib: Shared library code size.
- VmPTE: Page table entries size (since Linux 2.6.10).
- VmSwap: Swapped-out virtual memory size by anonymous private pages; shmem swap usage is not included (since Linux 2.6.34).