interactive code checking with Cobra

a Tutorial

Gerard Holzmann
Nimble Research
gholzmann@acm.org

topics covered

1. *background* and principle of operation
   - installation and configuration
   - guide to online documentation
2. *pattern* queries and regular expressions
   - exercises
3. *interactive* queries
   - token attributes
   - sets and ranges
   - functions
   - reading files, libraries
   - exercises
4. *scripted* queries
   - recursive functions
   - associative arrays
   - the query libraries
   - using concurrency
   - exercises
5. *standalone* checkers
   - using concurrency: multi-threaded checkers
6. use of Cobra for *runtime verification*
   - using live data or event-logs

cobra inline programs

the scripting language

An inline program is enclosed in delimiters:

```
%{
        …
%}
```

which can be used like any other query command, e.g. in a query function:

```
def prog1
        %{
                …
        %}
end
```

and invoked by name:

```
: prog1
```

If stored in a file, these scripts can be invoked from the command line:

```
$ cobra –f file.cobra *.[ch]
```

A very simple example:

```
$ cat file.cobra
%{
    print .fnm ":" .lnr ": " .txt "\n";
%}
```

which prints the text of all tokens, each preceded by filename and linenumber

The syntax is described in online manual pages at:
http://spinroot.com/cobra/commands/progs.html

cobra inline programs

the scripting language

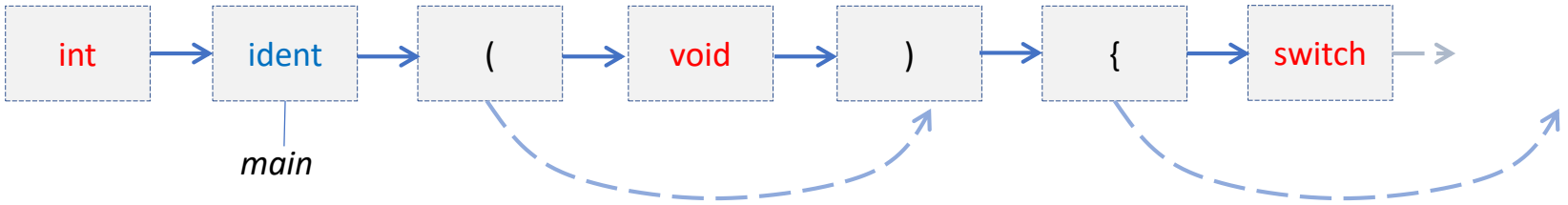
```
.fnm      # source filename (a string)
.lnr      # source line-number
.curly    # level of {…} nesting
.round    # level of (…) nesting
.bracket  # level of […] nesting
.len      # length of token text
.typ      # token type (a string)
.txt      # token text (a string)
.seq      # token sequence number
.mark     # marked value
```

the main control-loop:

cobra inline programs are, like all query commands, executed once for each token in the input sequence

but a cobra inline program can also take over control and navigation of the token sequence in any way that it wants

it can refer to any token attribute



cobra inline programs

keywords

control flow :

- if
- else
- while
- break
- continue
- goto
- for
- in
- return
- Next
- Stop

token references:

- Begin
- End
- .

two of the predefined functions:

- print …
- assert()

user-defined functions:

- function name (…) { … }

a fairly standard grammar:

- if (expr) { stmnt;+ } [else { stmnt;+ }]
- while (expr) { stmnt;+ }
- for (var in array) { stmnt;+ }
- L: goto L;
- q = .;
- array[expr] = expr;

variables (scalars and associative arrays):

- can be introduced without declaration
- the type is inferred from context, and
- may change dynamically

cobra inline programs

some simple examples, using print

```
%{
    print .fnm ":" .lnr ": " .txt "\n";
%}


%{
    print "hello world\n";
    Stop;  # don't continue printing this for every token…
%}
```

cobra inline programs

token attributes that can be referred to and modified

the *current* token is referred to as dot: .

manual page:
http://spinroot.com/cobra/commands/tokens.html

integer values:

- .round
- .bracket
- .curly
- .len
- .lnr
- .mark
- .seq
- .range

string values:

- .fct
- .fnm
- .txt
- .typ

token references:

- .
- .nxt
- .prv
- .jmp
- .bound

reference rule:

only one level of indirection is allowed,
so instead of writing:

```
q = .nxt.jmp;
```

you must use two hops:

```
q = .nxt;  q = q.jmp;
```

example1 (using a variable q):

```
q = .nxt;
q.mark++;
q.fnm = “hello”;
q = q.prv;
```

example2:

```
if (.txt == “{“)
{     q = .jmp;
      assert(q.seq != 0);
      r = q.jmp;
      assert(r == .);
}
```

cobra inline programs

the operators that can be used in expressions

binary operators:

+, -, *, /, %	arithmetic (integer operands)
>, <. <=, >=, ==, !=, ||, &&	Boolean (operands can be any type)

unary operators

!	Boolean, logical negation	
–	arithmetic, unary minus	
~	true if .txt contains a pattern	if (~yy) { … }
^	true if .txt starts with a pattern	if (^yy) { … }
#	true if .txt equals a pattern	if (#yy) { … }
@	true if .typ equals a pattern	if (@ident) { … }

provided that yy isn't a keyword of the scripting language…

regular expression matching of any token text (a predefined function):

match(s1, s2) true if s1 matches s2, where s2 can be a regex

if (match(q.txt, "/[Yy][Yy]")) { … }

comments start with # when followed by another # or a space

cobra inline programs

associative arrays (“hash-maps”)

name [expr [, expr]*]	associates a (possible comma separated sequence of) values, of any type with another value, of any type e.g.: name[.lnr, .txt] = .fnm;

predefined functions that work on associative arrays:

retrieve(A, n)	retrieves the nth element of associative array A
size(A)	returns the number of elements stored in array A
unset A[v]	remove associative array element A[v]
unset A	remove variable or array A

cobra inline programs

example

find the 10 most frequently occurring trigrams of type names

```
%{
    q = .nxt;
    r = q.nxt;
    if (.typ != "" && q.typ != "" && r.typ != "")
    {   Trigram[.typ, q.typ, r.typ]++;
    }
%}
track start _tmp_
%{
    for (i in Trigram)
    {   print i.txt "\t" Trigram[i.txt] "\n";
    }
    Stop;
%}
track stop
!sort -k2 -n < _tmp_ | tail -10; rm -f _tmp_
%{ unset Trigram; Stop; %}
```

```
const_int,oper,ident      157
key,chr,oper              177
oper,const_int,oper       180
oper,oper,ident           181
ident,oper,chr            185
storage,type,ident        263
type,oper,ident           541
ident,oper,const_int      739
oper,ident,oper          2342
ident,oper,ident          4197
```

cobra inline programs

example

count the number of cases in switch statements, taking into account that switch statements may be nested

```
$ cobra -f nr_cases_all cobra_lib.c | sort -n
    3 cobra_lib.c:173
    3 cobra_lib.c:1993
    3 cobra_lib.c:538
    3 cobra_lib.c:683
    4 cobra_lib.c:3450
    7 cobra_lib.c:3416
    8 cobra_lib.c:1717
    8 cobra_lib.c:583
   16 cobra_lib.c:1597
   27 cobra_lib.c:1546
```

```
%{
    if (.curly > 0 && #switch)
    { q = .;
      . = .nxt;
      if (.txt != "(" )
      {         . = q;
                Next; # move on to next token
      }
      . = .jmp;
      . = .nxt;
      if (.txt != "{")
      {         . = q;
                Next;
      }
      q.mark = 0;
      while (.curly >= q.curly)
      {         if (.curly == q.curly + 1
                &&  (#case || #default))
                {         q.mark++;
                }
                . = .nxt;
      }
      print q.mark " " .fnm ":" q.lnr "\n";
      . = q;
   }
%}
```

cobra inline programs

another example using associative arrays

```
%{
    if (.txt == "float")
    {   . = .nxt;
        if (@ident)
        {       Store[.txt] = .;        # store current location
                print .fnm ":" .lnr ": declaration of '" .txt "'\n";
        }
        Next;   # move on to the next token
    }
    if (@ident)
    {   q = Store[.txt];
        if (q.seq != 0)
        {       print .fnm ":" .lnr ": use of float '" .txt "' ";
                print "declared at " q.fnm ":" q.lnr "\n";
    }   }
%}
```

cobra inline programs

propagating data forward

```
%{      # check the identifier length for all tokens
        # and remember the longest in variable q

        if (@ident && .len > q.len)
        {       q = .;
        }
%}
# q holds its last assigned value
%{
        print q.fnm ":" q.lnr ": " q.txt " = " q.len " chars\n";
        Stop;   # stops after the line is printed
%}
```

```
cobra_links.c:487: switch_links_range = 18 chars
```

cobra inline programs

defining inline recursive functions

```
$ cobra file.c
: %{
        function fact(n)
        {   if (n <= 1)
            {     return 1;
            }
            return n*fact(n-1);
        }

        print "10! = " fact(10) "\n";
        Stop;
  %}
10! = 3628800
:
```

remember:
inline programs are
by default executed
once for every token
in the input stream

this has two consequences:

1. there has to be minimally one token to process – so using /dev/null as input will not work here
2. if something is meant to be executed only once, we need to explicitly Stop the control loop

cobra inline programs

creating new tokens, and modifying the token sequence

```
%{
        a = newtok();       # create 3 new empty tokens
        b = newtok();
        c = newtok();

        a.txt = "2";        # assign the .txt fields
        b.txt = "+";
        c.txt = "2";

        b.typ = "oper";     # assign the .typ field

        a.nxt = b;          # connect a to b
        b.nxt = c;          # and b to c
        set_ranges(a, c); # define a range from a to c
        Stop;
%}
%{
        print .txt "\n";    # scan the newly defined range
%}
```

running this program on arbitrary input, prints:

2
+
2

cobra inline programs

dealing with flow-sensitive properties

play/goto_links.cobra
collects goto statements and labels
and connects the .bound field of the gotos to the corresponding target label

similarly break_links.cobra, else_links.cobra, and switch_links.cobra use the .bound token attribute to set shortcuts, e.g. from case label to case label, or from if to else, etc.

a relevant fragment from else_links.cobra:

```
if (.txt == "if")
{       q = .;
        skip_cond();
        while (.typ == "cmnt")
        {       . = .nxt;
        }
        if (.txt == "{")
        {       . = .jmp;
                . = .nxt;
        } else
        {       skip_stmnt();
        }
        if (.txt == "else")
        {       q.bound = .nxt;
        } else
        {       q.bound = .;
        }
        . = q;
        Next;
}
```

cobra inline programs

using multiple cores – extending the earlier Trigram program

```
: ncore 8          # use 8 cores, or: cobra -N8 *.c
%{
        q = .nxt;
        r = q.nxt;
        if (.typ != "" && q.typ != "" && r.typ != "")
        {       Trigram[.typ, q.typ, r.typ]++;
        }
%}
track start _tmp_
%{
        if (cpu != 0)
        {       Stop;
        }
        a_unify(0);
        for (i in Trigram)
        {       print i.txt "\t" sum(Trigram[i.txt]) "\n";
        }
        Stop;
%}
track stop
!sort -k2 -n < _tmp_ | tail -10; rm -f _tmp_
```

post-process on cpu 0 only

unify the associative array data from all cores and make it available to cpu 0

sum the collected results

cobra inline programs

concurrency control: lock() and unlock()

```
$ cobra –N4 *.c       # cobra sources
4 cores, 10 files, 56546 tokens
%{
    if (.txt == “for”)  # for is a keyword, so #for doesn’t work here
    {   count++;        # { ... } braces always required
    }
%}
%{
    lock();
    print cpu “: my count = “ count “\n”;
    unlock();
    if (cpu == 0)
    {   print cpu “: total = “ sum(count) “\n”; # only 1 cpu gets here
    }
    Stop;
%}
```

```
0: my count = 54
0: total = 210
1: my count = 50
2: my count = 55
3: my count = 51
```

cobra inline programs

concurrency control: Begin, End, first_t and last_t

when multiple cores are used, each core scans part of the input sequence, so Begin and End refer to the local part of the sequence

when needed, the very first and very last token can still be accessed via first_t and last_t

```
# let cpu 0 scan all tokens backwards, for no good reason….
$ cobra –N4 *.c
4 cores, 10 files, 56546 tokens
%{
    if (cpu == 0)
    {    . = last_t;                # start at the end
         while (. != first_t)       # to the beginning
         {     .mark = .seq;        # do something pointless
               . = .prv;            # move backwards
         }
    }
    Stop;
%}
```

exercise

inline programs

1. write a Cobra inline program that removes any occurrence of the token sequence “else ;” (an else keyword followed by a semi-colon)

2. write a Cobra inline program that reports the total and the average number of gotos in the source files

```
# 1. one possible answer:
```

```
# 2. one possible answer:
```