L1VM - advanced topics

by Stefan Pietzonke

http://midnight-coding.de
26. March 2024
V1.3

1 The new linter

I developed a linter for my Brackets language for my L1VM. You can use it to declare the function call arguments variable types. And the return variable types can be defined too.

Both is declared in a special comment form. The linter runs before the compiler. The compiler does ignore the linter comments. To use the linter you have to build it first in the linter directory:

```
$ CC=clang ./make.sh
```

Now the linter should be build. You can also use the "gcc" compiler to build it.

Copy the linter in to your "~/l1vm/bin" directory (in your home directory). To use it call the new build script:

"l1vm-build-lint.sh".

1.1 Set the function call variable types

```
// (func args add int64 int64)
```

This declares that the function "add" has two variables of type int64 as arguments.

1.2 Set the function return variable types

To set the return values we can write this:

```
// (return args add int64)
```

This declares that the function "add" has one return value of type int64.

If you do a call of the function add, then the linter will check if the variable types and the number of the variables do match:

```
(x~ y~ :add !) (sum~ stpop)
```

And here you will see something new: the "stpop" command is in the same line as the function call.

You have to write the call and the stack pop of the return values this way!

1.3 Control the linter

You can turn the linter on and off by:

```
// lint-on
```

```
// lint-off
```

This can be useful when you want to switch it temporarily off in some cases.

2 Set legal ranges for variables

You can set the minimum and maximum value for a number variable.

The VM does runtime checks and returns an exception if a variable is in illegal range.

2.1 An example

```
(set int64 1 sum~ 0)
(set const-int64 1 sum_min~ 0)
(set const-int64 1 sum_max~ 1000000)

// set legal range for sum variable:
(sum~ sum_min~ sum_max~ range)
```

So here is the minimum value set to "0" and the maximum value to "1000000".

If at runtime the value of the variable "sum~" is out of range, then the VM will break with an exception.

3 Putting it all together

Here I will show a simple demo program with the new linter feature and the variable ranges check.

3.1 The demo program

Here is the demo program "hello-add-range.l1com":

```
// hello-add-range.l1com - with the new linter args
#include <intr.l1h>
#include <zero.l1h>
#include <misc-macros.l1h>
// this is the function declaration for the l1vm-linter:
// it has two int64 numbers as arguments:
//
// (func args add int64 int64)
//
// set the return variable type:
//
// (return args add int64)
//
(main func)
    #var ~ main

    (set int64 1 zero 0)
    (set int64 1 x~ 23)
    (set int64 1 y~ -30)
    (set int64 1 sum~ 0)
    (set const-int64 1 sum_min~ 0)
    (set const-int64 1 sum_max~ 1000000)

    // set legal range for sum variable:
    (sum~ sum_min~ sum_max~ range)
```

```
    (x~ y~ :add !) (sum~ stpop)

    // pull the function return value in sum~  and do the range check:
    pull_int64_var (sum~)
    print_i (sum~)
    print_n

    exit (zero)
(funcend)
(add func)
  #var ~ add

  (set int64 1 x~ 0)
  (set int64 1 y~ 0)
  (set int64 1 ret~ 0)

  (y~ x~ stpop)
  {ret~ = x~ + y~}
  (ret~ stpushi)
(funcend)
```

3.2 Run the demo program

You have to build the demo first:

```
$ l1vm-build-lint.sh hello-add-range
building: hello-add-range
second include path: '/home/stefan/l1vm-work/prog/linter/'
l1vm-linter 3.0.9  , no errors found!
code lines compiled: 32
[✓] out compiled
assembler args: ' '
assembling file: 'out'
codesize: 361 bytes
datasize: 669 bytes
filesize: 1155 , 1.128 KB
[✓] out assembled

build in 0.0040 seconds
copying files...
```

And now run it:

```
 $ l1vm hello-add-range -q
ERROR: int variable value in illegal range!
var: -7 : min: 0, max: 1000000

epos: 189
```

Now you see the runtime exception because the variable "sum~" is now "-7"!

The program added "23 + -30" = "-7". So the variable "sum~" is out of range.

In this example I used the "pull_int" function of the Brackets "misc.macros.l1h" include file:

```
pull_int64_var (sum~)
```

Here is the “misc-macros.l1h” include file:

```
>> misc-macros.l1h
>> some useful macros
>> variable pull macros, add zero to store register into variable
>> You must include "zero.l1h" first!
#func pull_byte_var (VAR) :{VAR = VAR + zero_byte}
#func pull_int16_var (VAR) :{VAR = VAR + zero_int16}
#func pull_int32_var (VAR) :{VAR = VAR + zero_int32}
#func pull_int64_var (VAR) :{VAR = VAR + zero}
#func pull_double_var (VAR) :{VAR = VAR + zero_double}
>> inc and dec integers
#func inc (VAR) :((VAR one +) VAR =)
#func dec (VAR) :((VAR one -) VAR =)
```

Here is the definition:

```
#func pull_int64_var (VAR) :{VAR = VAR + zero}
```

This just adds zero to an variable and stores it into the variable back.

It is needed to trigger the range out of bounds check here. Without this “pull_int” macro the return value of the function “add” would not be checked! Keep this in mind!

If you start the debugger you will see this execution position “189” assembly listing:

```
$ debug.sh 189
47
intr0, 28, 3, 4, 5
stpopi 0
loada summain, 0, 1
intr0 4, 1, 0, 0
intr0 7, 0, 0, 0
loada zero, 0, 2
intr0 255, 2, 0, 0
rts
:add
loada zero, 0, 0
stpopi 1
load yadd, 0, 2
pullqw 1, 2, 0
stpopi 2
load xadd, 0, 3
pullqw 2, 3, 0
addi 2, 1, 3
load retadd, 0, 4
pullqw 3, 4, 0
loada retadd, 0, 3
```

You can see the variable “summain” in the assembly listing above. This is the assembly code generated by the Brackets compiler.

4 Pure functions

Now you can mark functions as "pure". This means that the function returns the same values if it is called with the same input. And it is independent of global state.

4.1 Example code

```
(squareP func)
    #var ~ squareP
        (set double 1 num~ 0.0)
        (set double 1 ret~ 0.0)
        (num~ stpopd)
        {ret~ = num~ * num~}
        (ret~ stpushd)
(funcend)
```

The function name ends with an uppercase "P" to mark it as a pure function. You can only call other pure functions from this function type! To switch this off you can use:

```
(pure-off)
```

5 Documentation

You can output program documentation in to a markdown file by setting documentation blocks in your code. The start is marked by "#docustart" and the end by "#docuend".

5.1 An example

Here is an example (from the "math-circle-bignum.l1com" program:

```
(circle object)
    #var ~ circle

#docustart
circle object
==============
Calculate diameter, circumference and area of a circle.

init->circle
_____
Set the m_pi Pi variable.

calc_diam->circle
_____
Calculate the diameter of a circle.

calc_circ->circle
_____
Calculate the circumference of a circle.

calc-area->circle
```

Calculate the area of a circle.
#docuend

6 Variable scope

You can set the variable scope by some flags.

6.1 The flags

You can switch on to access only local and global variables only by:

(variable-local-on).

After this only the function local and global variables are allowed to access. The global variables must end by "main" in the name. To switch this off you can use:

(variable-local-off).

You can set a flag to allow only the function local variables with:

(variable-local-only-on).

7 Variable handling

You can set the variable settings with two flags.

7.1 The flags

(variable-immutable)
(variable-mutable)

The "variable-immutable" flag sets every new declared variable as immutable (not changeable). This is the same as with the "-const" ending for variables. You can switch this off by "variable-mutable" flag.

8 Register handling

There is a flag to set the register handling by the Brackets compiler.

8.1 reset-reg

With the "reset-reg" flag the current variables which are in the registers will be unset in the registers.

This is needed to load variables into the registers again to access them. I use this in deeep nested code with alternate branches that can be happen. You can use the flag like this:

(reset-reg).

This is useful in nested code and you can be sure that a variable is pushed again into the register.