Compile and execute CoBOL with GraalVM Community Edition and GnuCOBOL

Christophe Brun, PapIT (christophe.brun@papit.fr)

GraalVM the polyglot Virtual Machine by Oracle

The GraalVM supports various languages, Java, JavaScript, Ruby, Python, R, WebAssembly, C/C++[1]. In another document, Oracle explains the VM can be an interpreter from native codes using compiler called a Low-Level Virtual Machine (LLVM)[2]. Those native codes being C, C++, FORTRAN, Rust, COBOL, and Go. It is now getting even more exiting! As a former mainframe/CoBOL developer, I would like to see those billions lines of CoBOL modernized in a more open environments.

CoBOL modernization

This article's goal is to explain how the GraalVM could be running CoBOL on pretty much any platform **without package manager, only with the GraalVM and its LLVM package**. IT departments, mainly in financial institutions and governments are desperately seeking CoBOL and mainframe experts but the lack of training course and the repelling Z/OS TSO environment are not encouraging vocations. Another major issue with those technologies resides in the costs of mainframe licenses. I think, hope, both could be solved porting existing CoBOL in modern environments. In case you don't know, rewriting the code and just shutting down the mainframes is not an option, see this great article.

Native code in GraalVM

If not already installed, GraalVM installation is described on their website. Executing native code require a GraalVM package call llvm-toolchain. On my system I already have *clang* and *lli* so I created symlinks *g-clang* and *g-lli*. I prefer to create a symlink with a different name for *clang* and *lli* executables rather than extending the path which will required the use of *update-alternatives*. A great medium post from an Oracle collaborator detail the GraalVM llvm-toolchain. The installation gives the following LLVM version:

```
chrichri@chrichri-x470aorusultragaming:~/cobinatcci$ g-lli --version
LLVM (http://llvm.org/):
  LLVM version 10.0.0-4-g22d2637565-bg83994d0b4b
  Optimized build.
  Default target: x86_64-unknown-linux-gnu
```

[1]https://www.graalvm.org/docs/why-graal/

[2]https://www.graalvm.org/uploads/graalvm-language-level-virtualization-oracle-tech-papers.pdf

```
  Host CPU: znver1
```

GnuCOBOL

Using Flex for lexical parsing and Bison a compiler-compiler, GnuCOBOL can transpile CoBOL to C. It can directly compile CoBOL to an executable using your platform toolchain but it is not our goal here, as we want to execute it with GraalVM. There are many CoBOL compilers out there. This one implement major part of CoBOL 1985, 2002 and several extensions of other compilers[3]. This compiler and its library libcob can easily be compiled with formerly installed GraalVM Compiler. The latest release of the code can be found on their official SourceForge site. Auto configure the build with the provided shell script:

```
sh ./autogen.sh
```

> **Configure the built with the GraalVM Clang compiler** (and no Berkeley DB support in our example):

```
./configure --with-cc=g-clang --without-db
```

Build and install as usual:

```
make install
```

The installation gives me the following GNU CoBOL version:

```
chrichri@chrichri-x470aorusultragaming:~/cobinatcci/gnucobol-code-r4210-tags-gnucobol-3.1.2
cobc (GnuCOBOL) 3.1.2.0
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <https://gnu.org/licenses/gpl.html>
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
Written by Keisuke Nishida, Roger While, Ron Norman, Simon Sobisch, Edward Hart
Built     Mar 07 2021 21:44:51
Packaged  Mar 07 2021 20:31:04 UTC
C version "9.3.0"
```

Compiling CoBOL C intermediate, LLVM intermediate representation and its execution

Let's use the Mandelbrot set implemented in CoBOL as an example, see *mandelbrotset.cbl*:

```
identification division.
program-id. MandelbrotSet.

data division.
working-storage section.
01 ResolutionX              constant 240.
```

[3]https://sourceforge.net/projects/gnucobol/

```
01 ResolutionY            constant 100.
01 RealPlaneMin           constant -2.5.
01 RealPlaneMax           constant 0.8.
01 ImaginaryPlaneMin    constant -1.25.
01 ImaginaryPlaneMax    constant 1.25.
01 ProportionalX          pic S99V9(16) usage comp-5 value zeros.
01 ProportionalY          pic S99V9(16) usage comp-5 value zeros.
01 IterationsMax          constant 60.
01 Threshold              constant 10000.

01 ScreenX                   pic 999 usage comp-5.
01 ScreenY                   pic 999 usage comp-5.
01 MathPlaneX             pic S99V9(16) usage comp-5.
01 MathPlaneY             pic S99V9(16) usage comp-5.

01 PointX                    pic S9(7)V9(8) usage comp-5.
01 PointY                    pic S9(7)V9(8) usage comp-5.
01 XSquared               pic S9(10)V9(8) usage comp-5.
01 YSquared               pic S9(10)V9(8) usage comp-5.
01 Iteration              pic 999 value zero.
01 TempVar                   pic S9(5)V9(8) usage comp-5.

procedure division.

compute ProportionalX = (RealPlaneMax - RealPlaneMin) /
    (ResolutionX - 1)
compute ProportionalY = (ImaginaryPlaneMax - ImaginaryPlaneMin) /
    (ResolutionY - 1)

perform varying ScreenY from 0 by 1 until ScreenY is equal to
    ResolutionY

    compute MathPlaneY = ImaginaryPlaneMin +
           (ProportionalY * ScreenY)

    perform varying ScreenX from 0 by 1 until ScreenX is equal to
           ResolutionX

           compute MathPlaneX = RealPlaneMin +
               (ProportionalX * ScreenX)

           move zero to PointX
           move zero to PointY
           multiply PointX by PointX giving XSquared
           multiply PointY by PointY giving YSquared
```

```
          perform with test after varying Iteration from 0 by 1
              until Iteration >= IterationsMax or
                       XSquared + YSquared >= Threshold
                    compute TempVar = XSquared - YSquared + MathPlaneX
                    compute PointY = 2 * PointX * PointY + MathPlaneY
                    move TempVar to PointX
                    compute XSquared = PointX * PointX
                    compute YSquared = PointY * PointY
          end-perform

          if Iteration is equal to IterationsMax
              display "*" with no advancing
          else
              display " " with no advancing
          end-if
    end-perform

    display " "

end-perform
stop run.
end program MandelbrotSet.
```

Producing the C intermediate Using GnuCOBOL, the C intermediate can be produced with the following command:

```
cobc -C -x mandelbrotset.cbl
```

The project should look like:

```
benchmark
|- bin
|- mandelbrotset.c
|- mandelbrotset.c.h
|- mandelbrotset.c.l.h
|- mandelbrotset.cbl
```

Compiling C to LLVM Intermediate Reprensentation One point not completely clear from their documentation is the benefit of LLVM and how to execute code in GraalVM not just creating a binary like GNU CoBOL easily does. Using Clang to directly compile CoBOL into a executable is possible if you don't forget to include the *libcob* dependency with *-lcob*. But the real benefit of LLVM comes from the Intermediate Representation (IR) code that can run or compile on any platform running LLVM or in this case GraalVM LLVM.

Compiling to IR command is:

```
g-clang mandelbrotset.c -S -emit-llvm -o "bin/mandelbrotset.ll"
```

The project should look like:

```
benchmark
|- bin
|   |- mandelbrotset.ll
|- mandelbrotset.c
|- mandelbrotset.c.h
|- mandelbrotset.c.l.h
|- mandelbrotset.cbl
```

Execution in the LLVM interpreter The LLVM interpreter *lli* command can run the IR loading the *libcob* dependency:

```
g-lli -load /usr/local/lib/libcob.so ./bin/mandelbrotset.ll
```

comparison with the regular LLVM

The same version of LLVM can be downloaded from their github repository, under the *llvmorg-10.0.0* tag. It was compiled using Ninja, with the assertions disabled, as a release to get the same build as the GraalVm one. The compiling command is therefore:

```
mkdir build
cmake -GNinja ../llvm -DCMAKE_BUILD_TYPE=Release -DLLVM_ENABLE_ASSERTIONS=off
ninja
```

This give a similar LLVM interpreter as the GraalVM one:

```
chrichri@chrichri-x470aorusultragaming:~/cobinatcci$ lli --version
LLVM (http://llvm.org/):
  LLVM version 10.0.0
  Optimized build.
  Default target: x86_64-unknown-linux-gnu
  Host CPU: znver1
```

The previously generated IR code can be run with this LLVM interpreter:

```
lli -load /usr/local/lib/libcob.so ./bin/mandelbrotset.ll
```

Execution time with both LLVM interpreter is similar:

```
mandelbrotset>
LLVM ***************************************************************

real    0m0.356s
user    0m0.352s
sys     0m0.007s
GRAAL LLVM *********************************************************

real    0m0.371s
```

```
user    0m0.335s
sys     0m0.014s
```

Other programs indicated the GraalVM LLVM interpreter is always slightly slower. 1.39 slower in the worst case found with a program computing the first 1899 prime numbers. See the corresponding github repository for more details on this benchmark. Christophe Brun, https://www.papit.fr/