The Extra Project Assignment - Reproducibility of Published Results

Name: Thaís Regina Damásio | **ID:** [redacted]

1. Choose a paper presented in a recent compiler related conference, such as CGO, PLDI or CC.

I chose the best paper of CGO 2020: "*Testing static analyses for precision and soundness*" by Jubi Taneja, Zhengyang Liu and John Regehr, available at: https://doi.org/10.1145/3368826.3377927.

---

2. Download the material that the authors have made publicly available for the paper. If there is no such material, but the student still wants to reproduce the paper's results, he/she can write to the authors of the paper, asking for their implementation.

Done. The materials are publicly available.

---

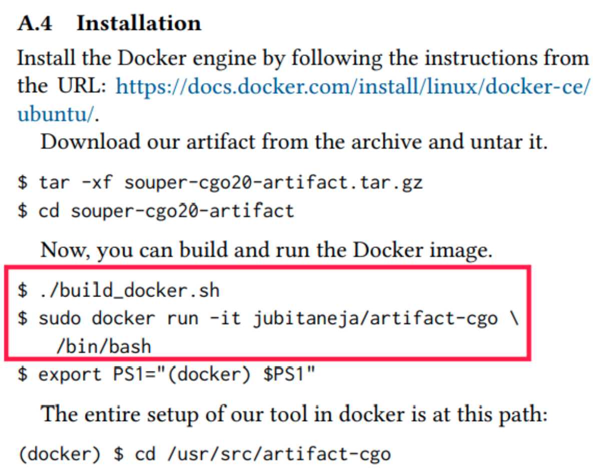
3. Run the implementation, trying to reproduce at least one of the experiments in the paper. Even if it is not possible to reproduce the experiments, the student can still write a report about his/her tries, to claim the extra points.

Done. It was possible to reproduce some of the experiments. I talk about this procedure and the results in the next section.

---

4. Write a short report about the entire procedure. This report must contain the URL of the material that has been downloaded, plus a brief description of the student's experience with it. If the student has not been able to reproduce those experiments, than the report must explain the reasons for this failure.

The artifacts to reproduce the experiments in the paper are publicly available at https://doi.org/10.1145/3373122. The paper contains an appendix with instructions on how to use the scripts, but the same URL provides a README with even more detailed directions. To prepare the environment we have two options: download a docker image with everything ready, or create the docker image yourself, with a script installing every prerequisite. The paper only cites the latter:



A.4 Installation

Install the Docker engine by following the instructions from the URL: https://docs.docker.com/install/linux/docker-ce/ubuntu/.

Download our artifact from the archive and untar it.

```
$ tar -xf souper-cgo20-artifact.tar.gz
$ cd souper-cgo20-artifact
```

Now, you can build and run the Docker image.

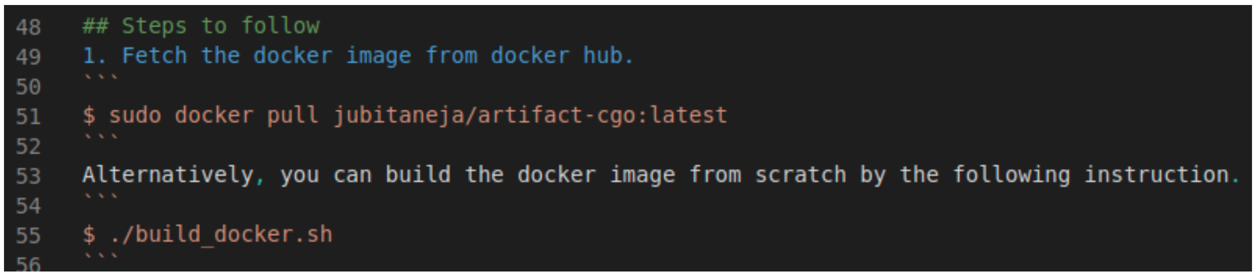
```
$ ./build_docker.sh
$ sudo docker run -it jubitaneja/artifact-cgo \
   /bin/bash
$ export PS1="(docker) $PS1"
```

The entire setup of our tool in docker is at this path:

```
(docker) $ cd /usr/src/artifact-cgo
```

But it is currently broken. It uses a specific commit on their github repository that does not exist anymore, and if we choose to use the last commit, the version of the other prerequisites starts changing and everything becomes a mess.

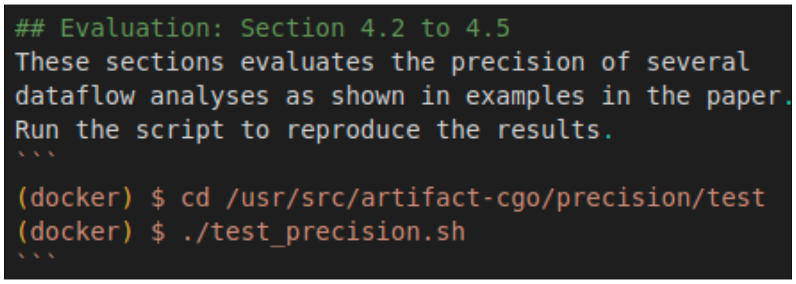
Fortunately, the readme available is more clear, and it states that we can download the image directly:



```
## Steps to follow
1. Fetch the docker image from docker hub.
```
$ sudo docker pull jubitaneja/artifact-cgo:latest
```
Alternatively, you can build the docker image from scratch by the following instruction.
```
$ ./build_docker.sh
```
```

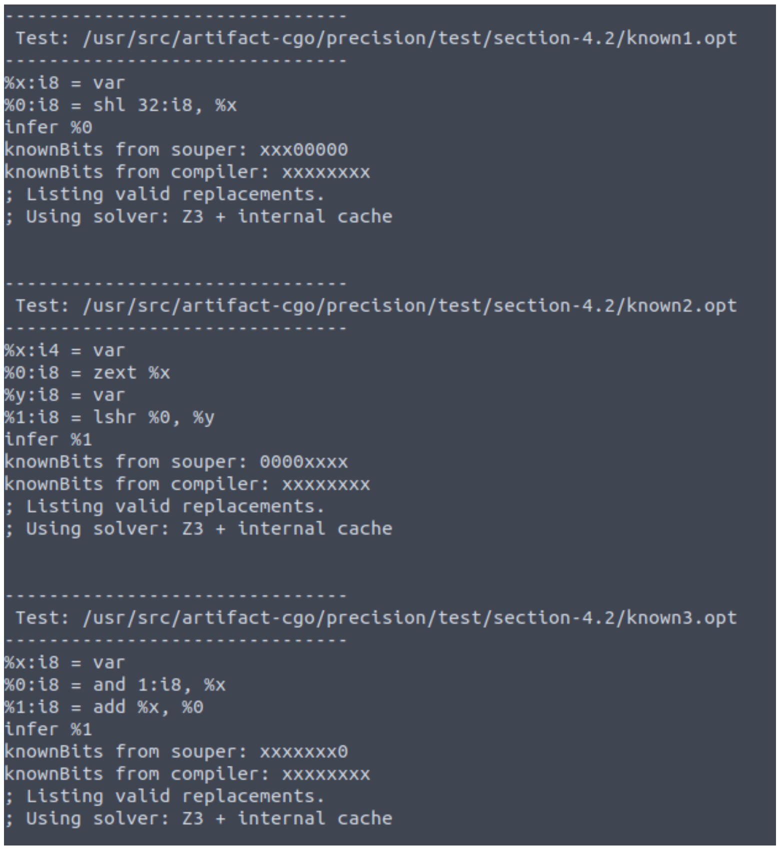
From this point and beyond I thought everything was very smooth. The README file contains over 700 lines of instructions on how to reproduce the experiments and what to expect from them.

To reproduce the experiments in sections 4.2 to 4.5, the guidelines are the following:

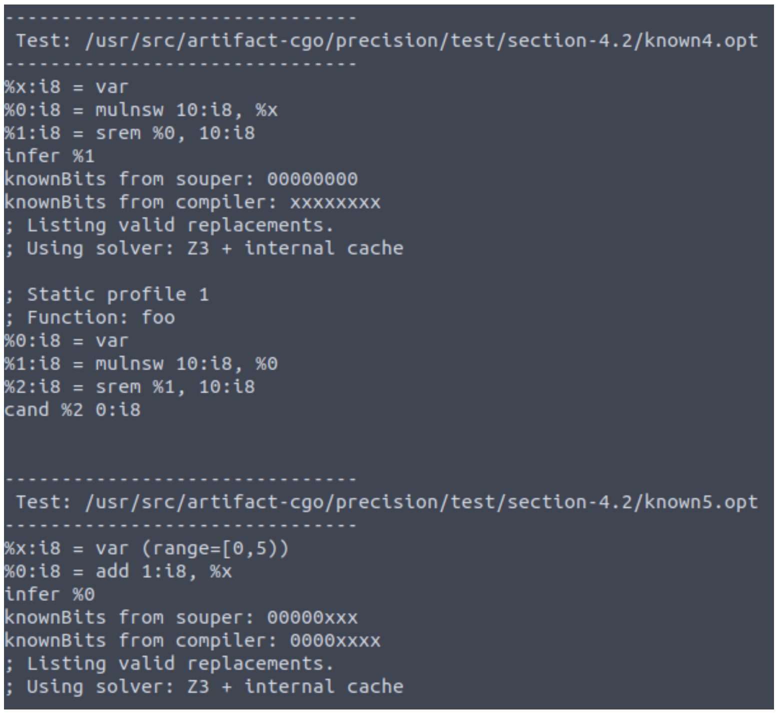


~~~
## Evaluation: Section 4.2 to 4.5
These sections evaluates the precision of several
dataflow analyses as shown in examples in the paper.
Run the script to reproduce the results.
```
(docker) $ cd /usr/src/artifact-cgo/precision/test
(docker) $ ./test_precision.sh
```
~~~

- **Reproducing the experiments of section 4.2 (Known bits)**

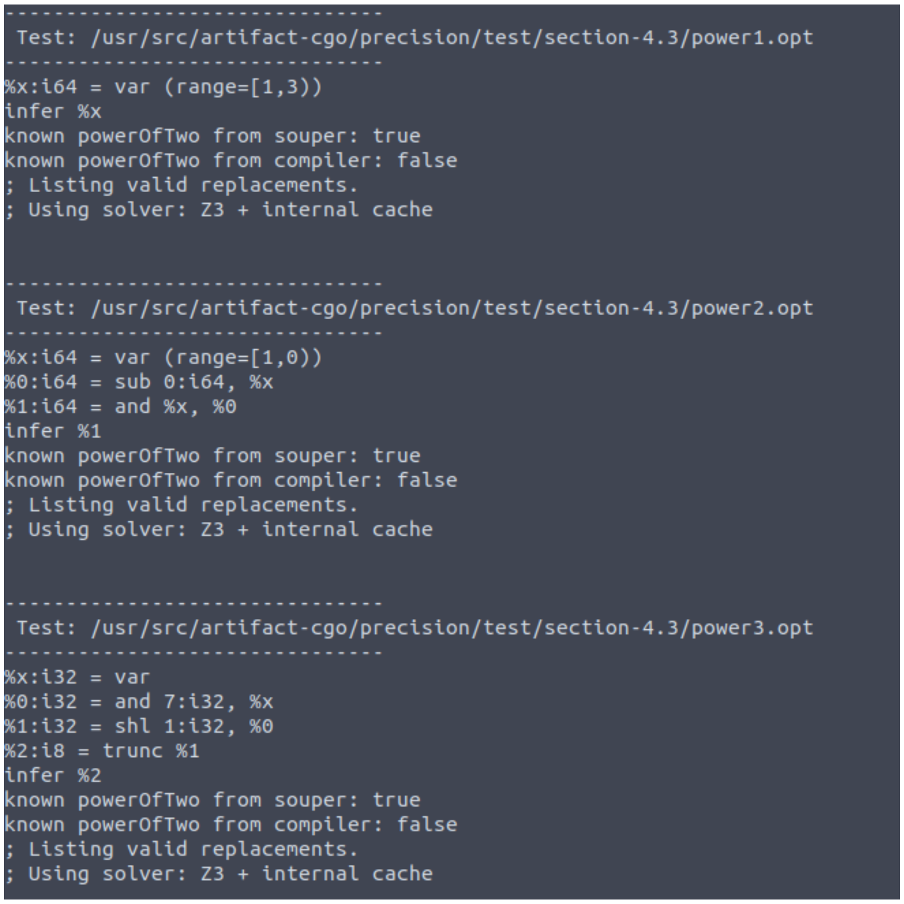


```
------------------------------
 Test: /usr/src/artifact-cgo/precision/test/section-4.2/known1.opt
------------------------------
%x:i8 = var
%0:i8 = shl 32:i8, %x
infer %0
knownBits from souper: xxx00000
knownBits from compiler: xxxxxxxx
; Listing valid replacements.
; Using solver: Z3 + internal cache


------------------------------
 Test: /usr/src/artifact-cgo/precision/test/section-4.2/known2.opt
------------------------------
%x:i4 = var
%0:i8 = zext %x
%y:i8 = var
%1:i8 = lshr %0, %y
infer %1
knownBits from souper: 0000xxxx
knownBits from compiler: xxxxxxxx
; Listing valid replacements.
; Using solver: Z3 + internal cache


------------------------------
 Test: /usr/src/artifact-cgo/precision/test/section-4.2/known3.opt
------------------------------
%x:i8 = var
%0:i8 = and 1:i8, %x
%1:i8 = add %x, %0
infer %1
knownBits from souper: xxxxxxx0
knownBits from compiler: xxxxxxxx
; Listing valid replacements.
; Using solver: Z3 + internal cache
```



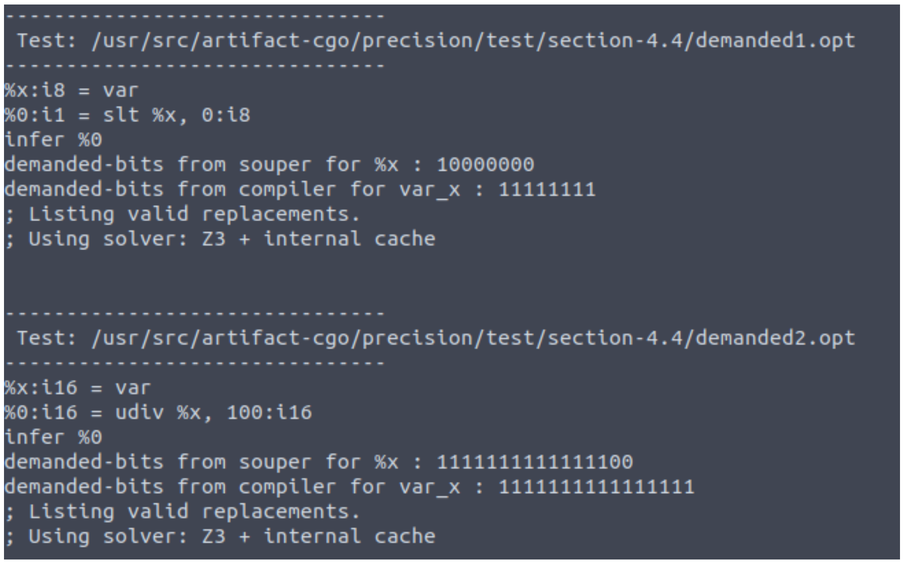
```
------------------------------
 Test: /usr/src/artifact-cgo/precision/test/section-4.2/known4.opt
------------------------------
%x:i8 = var
%0:i8 = mulnsw 10:i8, %x
%1:i8 = srem %0, 10:i8
infer %1
knownBits from souper: 00000000
knownBits from compiler: xxxxxxxx
; Listing valid replacements.
; Using solver: Z3 + internal cache

; Static profile 1
; Function: foo
%0:i8 = var
%1:i8 = mulnsw 10:i8, %0
%2:i8 = srem %1, 10:i8
cand %2 0:i8


------------------------------
 Test: /usr/src/artifact-cgo/precision/test/section-4.2/known5.opt
------------------------------
%x:i8 = var (range=[0,5))
%0:i8 = add 1:i8, %x
infer %0
knownBits from souper: 00000xxx
knownBits from compiler: 0000xxxx
; Listing valid replacements.
; Using solver: Z3 + internal cache
```

- **Reproducing the experiments of section 4.3 (Power of two)**



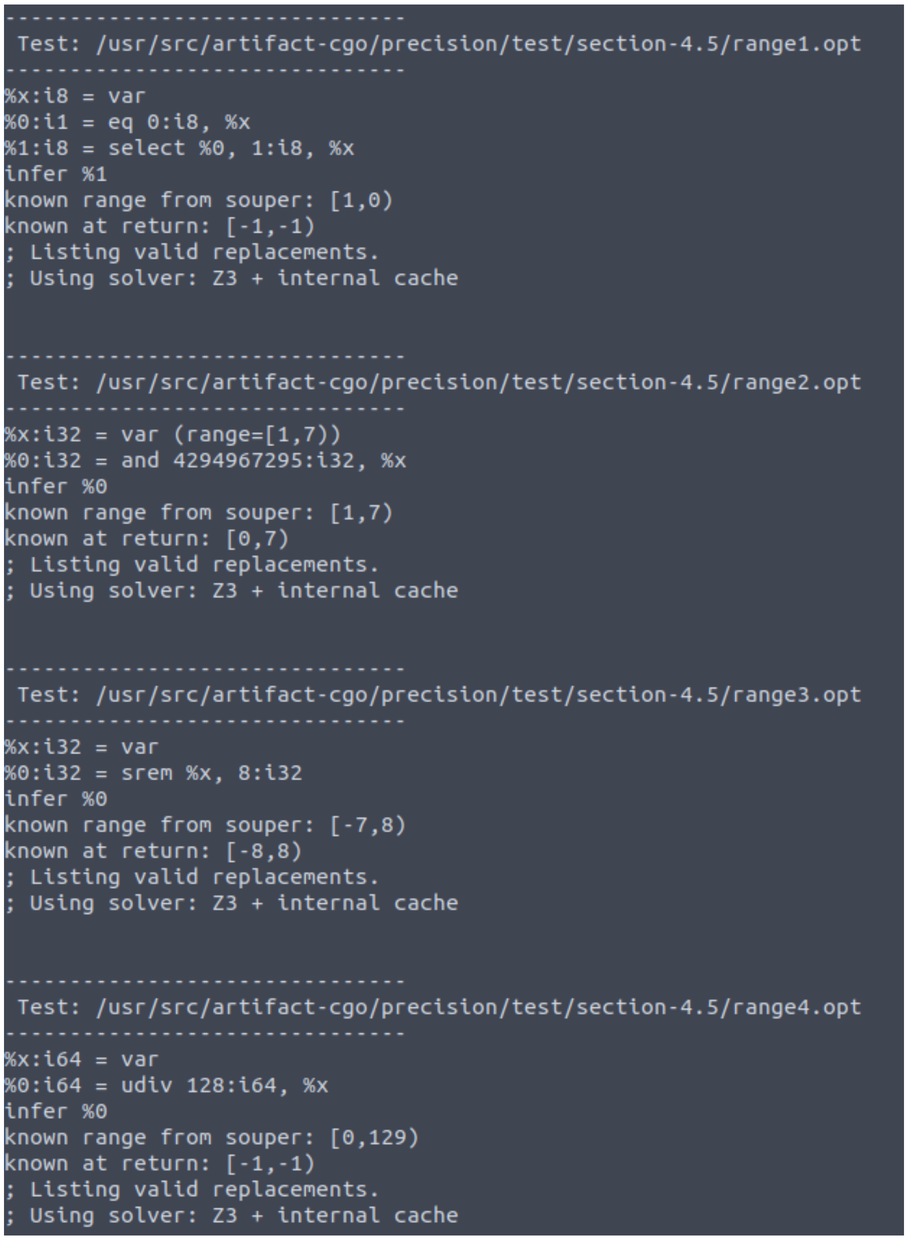
```
-------------------------------
 Test: /usr/src/artifact-cgo/precision/test/section-4.3/power1.opt
-------------------------------
%x:i64 = var (range=[1,3))
infer %x
known powerOfTwo from souper: true
known powerOfTwo from compiler: false
; Listing valid replacements.
; Using solver: Z3 + internal cache


-------------------------------
 Test: /usr/src/artifact-cgo/precision/test/section-4.3/power2.opt
-------------------------------
%x:i64 = var (range=[1,0))
%0:i64 = sub 0:i64, %x
%1:i64 = and %x, %0
infer %1
known powerOfTwo from souper: true
known powerOfTwo from compiler: false
; Listing valid replacements.
; Using solver: Z3 + internal cache


-------------------------------
 Test: /usr/src/artifact-cgo/precision/test/section-4.3/power3.opt
-------------------------------
%x:i32 = var
%0:i32 = and 7:i32, %x
%1:i32 = shl 1:i32, %0
%2:i8 = trunc %1
infer %2
known powerOfTwo from souper: true
known powerOfTwo from compiler: false
; Listing valid replacements.
; Using solver: Z3 + internal cache
```

- **Reproducing the experiments of section 4.4 (Demanded bits)**



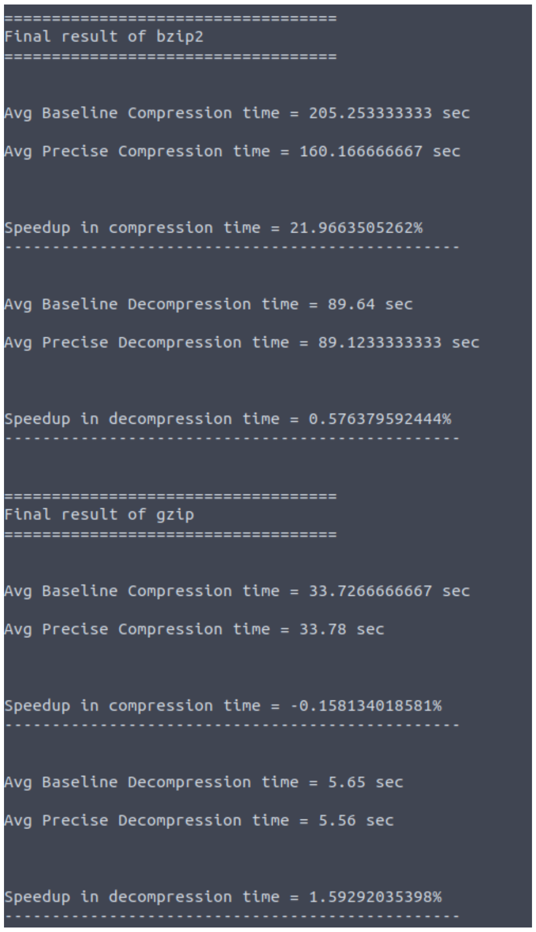
```
-------------------------------
 Test: /usr/src/artifact-cgo/precision/test/section-4.4/demanded1.opt
-------------------------------
%x:i8 = var
%0:i1 = slt %x, 0:i8
infer %0
demanded-bits from souper for %x : 10000000
demanded-bits from compiler for var_x : 11111111
; Listing valid replacements.
; Using solver: Z3 + internal cache


-------------------------------
 Test: /usr/src/artifact-cgo/precision/test/section-4.4/demanded2.opt
-------------------------------
%x:i16 = var
%0:i16 = udiv %x, 100:i16
infer %0
demanded-bits from souper for %x : 1111111111111100
demanded-bits from compiler for var_x : 1111111111111111
; Listing valid replacements.
; Using solver: Z3 + internal cache
```

- **Reproducing the experiments of section 4.5 (Range)**

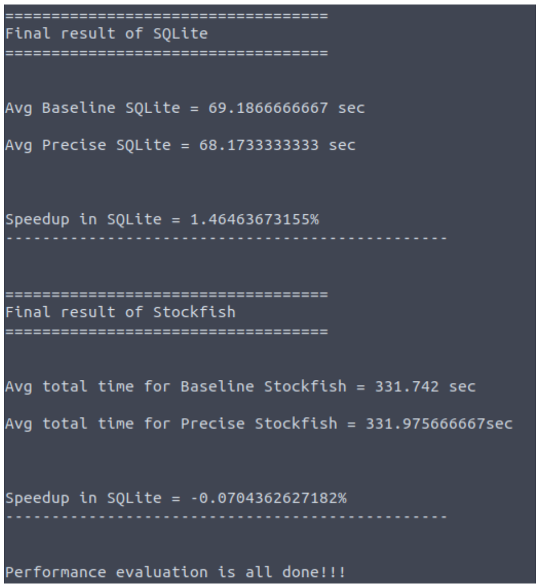


```
-------------------------------
 Test: /usr/src/artifact-cgo/precision/test/section-4.5/range1.opt
-------------------------------
%x:i8 = var
%0:i1 = eq 0:i8, %x
%1:i8 = select %0, 1:i8, %x
infer %1
known range from souper: [1,0)
known at return: [-1,-1)
; Listing valid replacements.
; Using solver: Z3 + internal cache


-------------------------------
 Test: /usr/src/artifact-cgo/precision/test/section-4.5/range2.opt
-------------------------------
%x:i32 = var (range=[1,7))
%0:i32 = and 4294967295:i32, %x
infer %0
known range from souper: [1,7)
known at return: [0,7)
; Listing valid replacements.
; Using solver: Z3 + internal cache


-------------------------------
 Test: /usr/src/artifact-cgo/precision/test/section-4.5/range3.opt
-------------------------------
%x:i32 = var
%0:i32 = srem %x, 8:i32
infer %0
known range from souper: [-7,8)
known at return: [-8,8)
; Listing valid replacements.
; Using solver: Z3 + internal cache


-------------------------------
 Test: /usr/src/artifact-cgo/precision/test/section-4.5/range4.opt
-------------------------------
%x:i64 = var
%0:i64 = udiv 128:i64, %x
infer %0
known range from souper: [0,129)
known at return: [-1,-1)
; Listing valid replacements.
; Using solver: Z3 + internal cache
```

- **Reproducing the experiments of section 4.6 (Impact of Maximal Precision on Code Generation)**



```
===================================
Final result of bzip2
===================================


Avg Baseline Compression time = 205.253333333 sec

Avg Precise Compression time = 160.166666667 sec


Speedup in compression time = 21.9663505262%
------------------------------------------------


Avg Baseline Decompression time = 89.64 sec

Avg Precise Decompression time = 89.1233333333 sec


Speedup in decompression time = 0.576379592444%
------------------------------------------------


===================================
Final result of gzip
===================================


Avg Baseline Compression time = 33.7266666667 sec

Avg Precise Compression time = 33.78 sec


Speedup in compression time = -0.158134018581%
------------------------------------------------


Avg Baseline Decompression time = 5.65 sec

Avg Precise Decompression time = 5.56 sec


Speedup in decompression time = 1.59292035398%
------------------------------------------------
```



```
===================================
Final result of SQLite
===================================


Avg Baseline SQLite = 69.1866666667 sec

Avg Precise SQLite = 68.1733333333 sec


Speedup in SQLite = 1.46463673155%
-----------------------------------------------


===================================
Final result of Stockfish
===================================


Avg total time for Baseline Stockfish = 331.742 sec

Avg total time for Precise Stockfish = 331.975666667sec


Speedup in SQLite = -0.0704362627182%
-----------------------------------------------


Performance evaluation is all done!!!
```

All of the results are in consonance with those presented in the paper.

I thought it was easy to reproduce the experiments. Although the instructions in the paper are outdated, the authors carefully described the process to reproduce them in the README file. I liked that they used docker, I had the opportunity to use it in the past and it really makes it easy to set up an environment. I hope to use everything I learned during this homework on my next projects :)