Artifact documentation for "Translating C to Rust"

We also put a text file version of this document inside the artifact archive to make copying the commands easier.

This document has 6 sections:

1. Claims supported by the artifact
2. Claims not supported by the artifact
3. Getting started with one benchmark
4. A more detailed step-by-step guide
5. A list of convenience scripts for running the experiments in the paper. We suggest reading this section if you are aiming to reproduce the results.
6. Reproducing the data needed for Figure 1. Reproducing figure 1 itself requires picking particular columns in this data, and using some plotting software. We used Google Sheets for these, and we can provide the spreadsheet upon request.

The artifact is structured as a tar archive `translating-c-to-rust.tgz` which contains the following:

- `docker-image.tar` – do not unpack this file, instead load it with Docker. The instructions to do so are provided later in the document.
- `README.md` – Markdown version of this document to make copying/pasting commands easier.
- `README.pdf` –
- `COPYRIGHT` – Text file describing the copyright and licensing terms. In general, all the code that is not part of the benchmark suite are dual-licensed under MIT and Apache 2.0 license. See the specific benchmark programs' directory for their copyright terms.
- `LICENSE-MIT` – MIT license file
- `LICENSE-APACHE` – Apache 2.0 license file

Claims supported by the artifact

1. The benchmark sizes (Table 1)
2. The claims on Tables 2–6 about our exploratory study
3. ResolveImports results on Table 7 (except the last 3 rows, see the comment about bugs below).

Claims not supported by the artifact

Results for libxml2 for Tables 7 and 8

As we found some bugs in our implementation and fixed them, the ResolveLifetimes pass does not halt for libxml2 without compiler errors. We found the underlying cause as not promoting some raw pointers that we were originally not converting to raw references, but we have not fixed that bug yet. We are planning to fix that bug by the time we submit the camera-ready.

Also, in our initial experimentation we ran our tools on the wrong directories for libxml2 when collecting data for Table 7, so the original column should read 210 and there should be 36 functions for Lifetime category after running ResolveImports on it (we describe how to obtain that statistic in the step-by-step guide). We are going to amend the libxml2 row of Table 7 with the updated numbers produced in this artifact and the fixed version of our tool.

Figure 1

We obtained Figure 1 using a brute-force search over the powerset of the features on the x axis, and used Google Sheets to prepare the figure. So, the specific spreadsheet could not be included in the artifact.

The following claims in the initial draft are not reproducible by the artifact because we discovered bugs since initial submission, and we are submitting the fixed version of the program as the artifact.

ResolveLifetimes results (Tables 7 and 8)

After our initial submission, we found some bugs in our implementation of ResolveLifetimes related to how we handle nested pointers and how we recognize malloc. The artifact contains the fixed version of the code whose results we are planning to submit for the revised draft.

The columns Original and Benchmarks in Table 7 are still produced by the artifact.

For Table 7 only the following benchmarks are affected by this: - The results of ResolveImports for tinycc and optipng - The results of ResolveLifetimes for urlparser, tinycc, optipng

For Table 8 the following benchmarks' results are affected by the bug fixes. We are going to amend the results during the revision process:

- Raw pointer declaration: genann, json-c, bzip2, tinycc, optipng
- Raw pointer dereferences: json-c, bzip2, tinycc, optipng

Performance results (Table 9)

The timing results may not be same on another specific machine. Also, as we fixed our technique, the iteration counts of the benchmarks have changed, and most of them finish in one iteration. You can count the number of iterations by using 'grep Applying fixes' on the output of resolve-lifetimes.

Getting started

Here, we describe the structure of our artifact, and how to run our tools on a benchmark to get results.

The docker image

We packaged our artifact as a Docker image (`docker-image.tar`). You can load it as follows.

```
docker load -i docker-image.tar
```

After this, you can start a container based on our image (named "c_to_safe_rust_artifact"):

```
docker run -it c_to_safe_rust_artifact
```

All of the remaining commands for invoking our tools

Special notes about benchmarks

Two benchmarks' names are abbreviated in the paper, in the artifact we use the full name for those benchmarks:

- RFK -> robotfindskitten
- TI -> tulipindicators

Structure of the code

Everything is in ~/lab directory with the following structure:

```
lab/
 + c2rust/
   -- the version of C2Rust we use
 + laertes/ -- our rewrite tools, as a Cargo project
    + test-inputs/ -- the initial Rust programs
    + invocations/
    + rewrite-invocations/
    + rewrite-workspace/   -- temporary workspace
 + c-code/ -- the original C code for the benchmarks
 + unsafe-counter/ -- unsafety-related statistics collection tool
```

*: Procedural macros are not supported by our implementation, so we inlined them and cleaned up other dependencies like the libc crate. We provide both the C and the Rust versions of the programs to produce the counts on

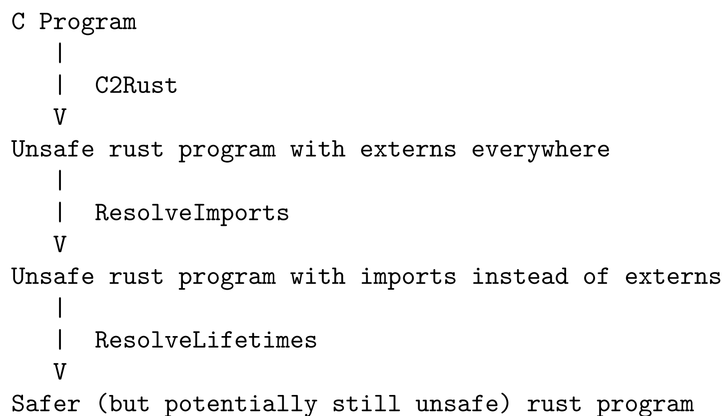
The compiler invocations needed for each benchmark are in lab/laertes/invocations. These are just command line arguments in text files. We have another copy of them to be used in the temporary workspace.

All of our commands can be run with cargo run --bin <command-name>. The internal naming of the commands are somewhat confusing:

- in laertes:
  - resolve-imports: this is ResolveImports in the paper
  - resolve-lifetimes: despite its name, this is the final part of our method and it is ResolveLifetimes in the paper
- in unsafe-counter:
  - unsafe-counter: this is the main tool we used for most of the data collection in Section 2
  - ...

Running our tools on a single program

The rough pipeline that our tools sit in looks like the following:



Running C2Rust on a random C project may be daunting, so we included Rust programs that are already processed Rust programs. Besides this, the original C code for the benchmarks are included for reproducing line-of-code metrics and comparing the C code to the Rust code.

In this guide, we will walk you through using our method for making the Rust program safer, and our data collection tool to measure different sources of unsafety. For this guide, we will focus on the bzip2 benchmark as it is a medium-sized self-contained program.

Before getting started, disabling logging can make the output much easier to

follow:

```
export RUST_LOG=off
```

First, let's measure the statistics about the original program:

```
cd ~/lab/unsafe-counter
cargo run --release --bin unsafe-counter -- `cat ../laertes/invocations/bzip2`
```

Here, we are running our main tool for counting up functions and sources of unsafety. After running the program above, we should get some output that ends with something like the following

```
function breakdown:
total function count: 128
unsafe function count: 120

raw pointer CSV
Benchmark,Statistic,{VoidPtr},{PtrArith},{Extern},{}
ip-names/bzip2,NonUniqueDecls,112,93,112,227
ip-names/bzip2,DeclsExact,43,24,9,37
ip-names/bzip2,NonUniqueDerefs,1192,627,1195,3764
ip-names/bzip2,DerefsExact,1704,173,11,679
ip-names/bzip2,NonUniqueFns,28,19,28,41
ip-names/bzip2,FnsExact,7,2,0,4

unsafe behavior CSV
Benchmark,Statistic,ReadFromUnion,MutGlobalAccess,InlineAsm,ExternCall,RawPtrDeref,UnsafeCas
ip-names/bzip2,Occurrences,0,700,0,424,3764,1,14
ip-names/bzip2,ExactUnsafeFns,0,3,0,7,23,0,2
ip-names/bzip2,UnsafeFns,0,79,0,85,82,3,26

Table 3, FP: 0

PROFILING RESULTS
constraint solving: 7 ms
call graph closure: 0 ms
```

Here, the first two lines give us the number of functions defined in the program, and the number of functions declared unsafe.

Then, there is a CSV table that contains the data in tables 4, 5, 6 for this benchmark. The first three data columns correspond to the columns on those tables. The last column "{}" corresponds to Lifetime or Total. Here is how to read the CSV table according to the Statistic column:

- DeclsExact: The unique columns in table 4. The last value is for the lifetime column.
- DerefsExact: The unique columns in table 5. The last value is for the lifetime column.

- FnsExact: The unique columns in table 6. The last value is for the lifetime column.
- NonUniqueDecls: The non-unique columns in table 4. The last value is for the total column.
- NonUniqueDerefs: The non-unique columns in table 5. The last value is for the total column.
- NonUniqueFns: The non-unique columns in table 6. The last value is for the total column.

After this, there is another CSV table showing the counts for different kinds of unsafe behavior. The columns are ordered in the same order as tables 2 and 3. Here is how to read the CSV table according to the Statistic column:

- Occurrences: The data in table 2.
- ExactUnsafeFns: The unique columns in table 3.
- UnsafeFns: The non-unique columns in table 3.

Finally, after this table there is the line "Table 3, FP: XXX". It shows the value on the FP column of table 3.

Resolving imports

Now, we can check the statistics related to unsafety. Let's make the program safer. The first step in our method (namely ResolveImports) is to resolve external functions and types, removing unsafe, and minimizing the use of mut. To run it, first go to the tool's directory, and make a copy that we can manipulate:

```
cd ~/lab/laertes
cp -r test-inputs/* rewrite-workspace/
```

Then, run the program `resolve-imports`:

```
cargo run --release --bin resolve-imports -- `cat rewrite-invocations/bzip2`
```

After this step, you can look at what has changed, e.g. by running diff:

```
diff -p -r test-inputs/bzip2 rewrite-workspace/bzip2
```

You can compile bzip2 by going to the directory `rewrite-workspace/bzip2/rust` and running `cargo build`. It is compiled as a library because this benchmark is originally generated on macOS and requires macOS libc bindings to link as a binary.

You can run unsafe-counter and inspect the changes in the results:

```
cd ../unsafe-counter
cargo run --release --bin unsafe-counter -- `cat ../laertes/rewrite-invocations/bzip2`
cd ../laertes
```

Resolving lifetimes

After the previous step is successfully finished, we can use our iterative method to rewrite single object pointers (the program to do so is called resolve-lifetimes, but it runs the whole method rather than the initial step when given the parameter -f).

First, let's make a copy to compare against:

```
cp -r rewrite-workspace/bzip2 bzip2-after-resolve-imports
```

Now, let's run ResolveLifetimes:

```
cargo run --release --bin resolve-lifetimes -- -f `cat rewrite-invocations/bzip2`
```

Upon finishing successfully, it will print the following message right before profiling information, it may print logging information before then.

```
DONE: The compiler successfully compiles the code
PROFILING RESULTS
...
```

After this step, we can use diff to inspect the changes:

```
diff -p -r bzip2-after-resolve-imports rewrite-workspace/bzip2
```

And, we can run unsafe-counter to see the changes in unsafety statistics:

```
cd ../unsafe-counter
cargo run --release --bin unsafe-counter -- `cat ../laertes/rewrite-invocations/bzip2`
cd ../laertes
```

Step-by-step guide

Each procedure below should be repeated for each benchmark listed under ~/lab/laertes/test-inputs/

The compiler invocations to give our tool are listed under ~/lab/laertes/invocations/

Although each tool finishes in a few minutes on smaller benchmarks, ResolveLifetimes takes a long time on the three largest benchmarks (optipng, tinycc, and libxml2).

Table 1

Producing line of code counts

You can use the command

```
tokei <benchmark-directory>
```

to get the line counts for a benchmark. We use the non-blank non-comment lines of code.

Producing function counts

You can get the function counts for a benchmark by running the following commands (notice the \ at the end of some lines showing that the line continues on the next line):

```
cd ~/lab/unsafe-counter
export benchmark=<benchmark name>
BENCHMARK=$benchmark cargo run --release --bin unsafe-counter -- \
  `cat ../laertes/invocations/$benchmark`
```

Then, the lines "total function count:" and "unsafe function count:" should report the relevant values from Table 1 (the numbers are highlighted in bold red in the command output). For example, on bzip2 benchmark, there should be two lines reading

```
total function count (foreign functions excluded): 128
unsafe function count: 126
```

The numbers below them are used in other parts of the paper.

Table 2

The values in Table 3 are obtained from the unsafe behavior CSV in unsafe-counter's output (marked with a blue header). The first part of the CSV table contains the occurrences of each feature.

Here is an example of the CSV table (it is more readable in the Markdown version):

```
unsafe behavior CSV
Benchmark,Statistic,ReadFromUnion,MutGlobalAccess,InlineAsm,ExternCall,RawPtrDeref,UnsafeCas
ip-names/bzip2,Occurrences,0,700,0,424,3764,1,14
ip-names/bzip2,ExactUnsafeFns,0,3,0,7,23,0,2
ip-names/bzip2,UnsafeFns,0,79,0,85,82,3,26
```

Table 3

The values in Table 3 are obtained from the occurrence CSV in unsafe-counter's output.

- The second part of the CSV table (where the second column is ExactUnsafeFns) contains the values in the unique columns (each row corresponds to one cause of unsafety that has the same name as the column in Table 3).
- The third part of the CSV table (where the second column is UnsafeFns) contains the values in the non-unique columns.

After this CSV, there is the line "Table 3, FP: XXX" which computes the functions that are marked unsafe ultimately for no reason.

Table 4

The values in Table 3 are obtained from the raw pointer CSV in unsafe-counter's output (computed during the taint analysis in ptr_provenance.rs). The CSV table may be easier to inspect visually if saved to a file and opened in a spreadsheet program. For the "VoidPtr", "PtrArith", "Extern" columns on Table3, the value on the unique column corresponds to the value in DeclsExact row in the CSV table, and the value in the non-unique column corresponds to the value on the NonUniqueDecls row on the CSV table.

The values on the Lifetime column correspond to the values on the "{}" column and the DeclsExact column on the CSV table, and the values in the Total column of Table 3 correspond to the values on the "{}" column and the NonUniqueDecls column on the CSV table.

Table 5

The numbers on this table are obtained similarly to Table 4 but focusing on the rows DerefsExact and NonUniqueDerefs.

The values in Table 3 are obtained from the raw pointer CSV in unsafe-counter's output. For the "VoidPtr", "PtrArith", "Extern" columns on Table3, the value on the unique column corresponds to the value in DerefsExact row in the CSV table, and the value in the non-unique column corresponds to the value on the NonUniqueDerefs row on the CSV table.

The values on the Lifetime column correspond to the values on the "{}" column and the DerefsExact column on the CSV table, and the values in the Total column of Table 3 correspond to the values on the "{}" column and the NonUniqueDerefs column on the CSV table.

Table 6

The numbers on this table are obtained similarly to Table 4 but focusing on the rows FnsExact and NonUniqueFns.

The values in Table 3 are obtained from the raw pointer CSV in unsafe-counter's output. For the "VoidPtr", "PtrArith", "Extern" columns on Table3, the value on the unique column corresponds to the value in FnsExact row in the CSV table, and the value in the non-unique column corresponds to the value on the NonUniqueFns row on the CSV table.

The values on the Lifetime column correspond to the values on the "{}" column and the FnsExact column on the CSV table, and the values in the Total column of Table 3 correspond to the values on the "{}" column and the NonUniqueFns column on the CSV table.

Tables 7 and 8

Original values

The values on the Original column on Table 7 comes from the Lifetime column on Table 6. The Orig. columns in Table 8 come from the Lifetime columns on Tables 4 and 5.

Running the transformations and obtaining other columns' values

First, enter the working directory:

```
cd ~/lab/laertes
```

Then, create a copy of the benchmarks to work on so the experiments are repeatable:

```
cp -r test-inputs/* rewrite-workspace/
```

From this point on, we are going to use `rewrite-invocations/<benchmark name>` as the invocation files to not destroy the original benchmarks.

Repeat the following steps for each benchmark:

1. Run ResolveImports

```
cargo run --release --bin resolve-imports -- `cat rewrite-invocations/<benchmark name>`
```

1.a. Applying and inspecting the tinycc patch and the optipng patch

In the paper, we mention that there is a small patch to apply at this step for the tinycc benchmark because of unnamed structs. This patch file is located at ~/lab/laertes/tinycc.patch. You need to apply it as follows to make sure that the resulting program compiles:

```
(in the directory ~/lab/laertes/rewrite-workspace)
patch -p1 <../tinycc.patch
```

Our implementation of the mutability reasoning misses a case in optipng, so we have a patch that adds the C-style casts necessary for that variable. This patch is located at ~/lab/laertes/optipng.patch . You can apply it as follows before proceeding:

```
(in the directory ~/lab/laertes/rewrite-workspace)
patch -p1 <../optipng.patch
```

2. Run unsafe-counter again to get the number of unsafe functions after ResolveImports

```
(from the directory ~/lab/unsafe-counter)
cargo run --release --bin unsafe-counter -- \
  `cat ../laertes/rewrite-invocations/<benchmark_name>` \
  | grep '^unsafe function count:'
```

Should output something like

```
unsafe function count: 42
```

Remember this value as AfterRI

3. Run ResolveLifetimes (don't forget the `-f` flag in the command below, it needs to come right after the `--`)

Special note about urlparser: The urlparser benchmark consists of only one file, which changes how we need to inject our helper functions. So you need to pass the –single-mod flag to resolve-lifetimes to run it on this benchmark.

```
cargo run --release --bin resolve-lifetimes -- -f \
  `cat rewrite-invocations/<benchmark name>`
```

After ResolveLifetimes finishes, it should print the message "DONE: The compiler successfully compiles the code" followed by some profiling statistics.

4. Run ResolveImports again to remove the `unsafe` tags (this is a limitation of our specific implementation where we do not remove the `unsafe` tag in ResolveLifetimes)

```
cargo run --release --bin resolve-imports -- `cat rewrite-invocations/<benchmark name>`
```

5. Run unsafe-counter to get the remaining statistics in Tables 7 and 8:

```
cargo run --release --bin unsafe-counter -- \
  `cat ../laertes/rewrite-invocations/<benchmark_name>`
```

The line "unsafe function count: XXX" should give the value on column Remaining on Table 7.

The raw pointer CSV table contains the values on the Remaining columns of Table 8.

- For the Declarations section, the value at column {}, row DeclsExact contains the value on the Remaining column of Table 8.
- For the Dereferences section, the value at column {}, row DerefsExact contains the value on the Remaining column of Table 8.

Fixed, Fixed (%), Total made safe (%), and other columns

We calculated these values out of the values above:

- Fixed = Orig - Remaining
- Fixed (%) = Fixed / Orig

On Table 7,

- ResolveImports = Original - AfterRI
- ResolveLifetimes = AfterRI - Remaining
- Total made safe (%) = 1 - Remaining / Original

Inspecting what ResolveLifetimes is doing

All of our tools can report an extensive amount of logging information by setting the environment variable RUST_LOG before running the tool:

```
export RUST_LOG=info
cargo run ...
```

Besides this logging infrastructure, ResolveLifetimes supports some diagnostics options passed via the command line. These arguments must be passed before the invocations file, right after the `--` on the command line:

- print the compiler errors as JSON files as it is processing them, if passed the -i flag.
- with –print-suggestions flags, you can see the specific edits that are performed by the tool
- by default, it performs only one iteration, that is why we pass the -f flag to iteratively fix the compiler errors
- with –dry-run flag, you can run it for one iteration without performing any edits

Scripts for running the tools in bulk

We included a script (`run-all-experiments.sh`) that runs all experiments (except running resolve-lifetimes on libxml2) and saves the results of running unsafe-counter before and after each stage. The script must be run from ~/lab/laertes under the container after compiling our tools. After running the script, following files are created for each benchmark:

- `results/<benchmark-name>/before.txt` – results of unsafe-counter before running our tools
- `results/<benchmark-name>/after-resolve-imports.txt` – results of unsafe-counter after running resolve-imports
- `results/<benchmark-name>/after-resolve-lifetimes.txt` – results of unsafe-counter after running resolve-lifetimes

We chose not to run `resolve-lifetimes` on libxml2, as ResolveLifetimes does not halt for that benchmark without compiler errors.

We also included two other scripts (`unsafe-behavior-stats.sh` and `raw-ptr-stats.sh`) that you can run after running `run-all-experiments.sh`.

- `unsafe-behavior-stats.sh` gives the statistics about occurrences and effect of unsafe behaviors (tables 2 and 3) except the FP column of table 3
- `raw-ptr-stats.sh` gives the statistics about use of raw pointers (tables 4–6, also you can derive tables 7 and 8 out of the results of this script)

You can run these scripts as follows:

```
# in ~/lab/laertes
```

```
bash raw-ptr-stats.sh before  # this gives the statistics before running our tools
bash raw-ptr-stats.sh after-resolve-imports  # this gives the statistics after ResolveImport
bash raw-ptr-stats.sh after-resolve-lifetimes  # this gives the statistics after all stages
```

You can also run `unsafe-behavior-stats.sh` with the same arguments as `raw-ptr-stats.sh`. You can use `grep` on the output of these commands to single out a statistic. For example, the following command will print out only the statistics for Table 5.

```
bash raw-ptr-stats.sh before | grep Derefs
```

Reproducing the data for figure 1

We created figure 1 with the following process, our concern was sharing the spreadsheet that produces the figure from the data in step 3 anonymously: 1. We changed `unsafe-counter` to print number of functions affected by each combination of unsafe behaviors. We attached the changes to `unsafe-counter` as a patch file (`fns-affected-by-sets-of-ub.patch`). After applying this patch, `unsafe-counter` prints a 2-row table after its normal output before printing the performance information. It prints an output like the following right before the performance statistics: (it is a 2-line output but the lines are really long):

```
Benchmark,Statistic,"[]","[ReadFromUnion]","[MutGlobalAccess]","[ReadFromUnion, MutGlobalAcc
bzip2,UnsafeFns,6,6,9,9,6,6,9,9,13,13,41,41,13,13,41,41,29,29,39,39,29,29,39,39,42,42,100,10
```

2. The output of `unsafe-counter` with the sets of unsafe behaviors is a CSV file with 130 columns (2^7 for each subset of unsafe behaviors, 1 for benchmark name, and 1 for the statistic name), we picked the columns corresponding to cumulatively and re-ordered the sets to get cumulatively affected functions. The columns we picked from the output of `unsafe-counter` were:
   - [RawPtrDeref]
   - [RawPtrDeref, Alloc]
   - [ExternCall, RawPtrDeref, Alloc]
   - [MutGlobalAccess, ExternCall, RawPtrDeref, Alloc]
   - [MutGlobalAccess, ExternCall, RawPtrDeref, UnsafeCast, Alloc]
   - [ReadFromUnion, MutGlobalAccess, ExternCall, RawPtrDeref, UnsafeCast, Alloc]
   - [ReadFromUnion, MutGlobalAccess, InlineAsm, ExternCall, RawPtrDeref, UnsafeCast, Alloc]
3. Then, we normalized the column values by the last column (the total number of affected functions)
4. We used Google Sheets to create a line chart out of values from step 3. Unfortunately, we could not find a way to automatically reproduce the graph from the data in a container. If the reviewers are interested, we can create a temporary Google account to share the spreadsheet where the reviewers can paste the data they obtain from the artifact to see the figure

change.