A Artifact Appendix

A.1 Abstract

The artefact consists in Mlang, the compiler for the French Tax Code described in the research article.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** Virtualbox image (**Virtualbox 6.1**, Ubuntu 20.04). Source code is available.
- **Hardware:** Tested on an Intel Core i7-8650U with 32GB memory. The compiler is not resource-intensive, but test suites are run in parallel on all available cores. The Virtualbox image uses 12GB memory because the AFL fuzzer may need that much memory.
- **Experiments:** Provided make commands.
- **Time needed to prepare artefact:** 5 minutes (time to import virtualbox image).
- **Time needed to complete experiments:** Around 1 hour (plus 5 hours of compilation for one result).
- **Publicly available:** https://github.com/MLanguage/mlang/releases/tag/cc21-v1.0
- **Archived:** DOI: 10.5281/zenodo.4456774

A.3 Description

This artefact is provided as a compiled binary in a virtualbox image. Its sources are also publicly available.

A.4 Installation

We recommended to use the provided virtualbox image to avoid having to manage dependencies. Just import the appliance in virtualbox and run it (the root password is `cc21`). A terminal is opened, with the current directory being the `mlang` repository. Being at `mlang` root directory is necessary to run the commands provided the reproduce the results.

If needed, manual installation instructions are provided in Mlang's `README.md`.

To launch a single test from the test suite, use

```
$ TEST_FILE=path/to/test make test
```

To launch the default test suite, use:

```
$ make tests
```

A.5 Evaluation and expected result

Formal Semantics of Section 2 (est. time: < 10 minutes). The formal semantics of μM and the type safety theorem are written in Coq: `formal_semantics/semantics.v`. Correspondance between the Coq names and the ones used in the paper are given in `formal_semantics/README.md`. You can run `coqc semantics.v` to check that the semantics and proofs are correct.

Reproducing Figure 11 (est. time: < 10 minutes). The M specification files are located in folder `m_specs/`. Table 18 provides the equivalence between the names of Figure 11 and the M specification files.

To run Mlang with a given M specification file, just write:

```
$ M_SPEC_FILE=m_specs/your-file.m_spec make from_spec
```

Mlang then displays the number of inputs and outputs given by the specification file, as well as the number of instructions generated in the end. For example, `tests.m_spec` yields:

```
$ M_SPEC_FILE=m_specs/tests.m_spec make from_spec
[...]
[DEBUG] M_spec has 1732 inputs and 651 outputs
[...]
[DEBUG] Optimizations done! Total effect: 656719 → 115297
[...]
```

Note that the initial number of instructions differs from each M specification file, because some initialization assignments depend on the number of inputs.

Partially reproducing Figure 16 (est. time: 2 minutes with default `-O0`). Figure 16 shows 6 lines. The first 2 lines cannot be reproduced since the replication would require access to the private compiler and C code of the DGFiP. We were able to access the code after signing a non-disclosure agreement with the DGFiP, and doing the same is out of reach of artifact reviewers. The next two lines correspond to a old compilation scheme that corresponds to an obsolete state of our Mlang codebase, that we chose to get rid completely in order to keep the codebase clean. Indeed, this compilation scheme is less performant than the newer compilation scheme "Array" presented in the last 2 lines of Figure 16.

To replicate the last two lines from the figure, launch:

```
$ make test_c_backend_perf
```

This will run an executable that pass the same test 1000 times. To get the execution time for one run, divide the `time` result (in user category) by 1000. The last command should build with LLVM and option `-O0`. To get the `-O1` time, launch

```
$ C_OPT=-O1 make test_c_backend_perf -B
```

Be careful, `-O1` optimizations with LLVM currently take about 5 hours to complete, and will use approximately 10 GB of memory.

Reproducing the results of Section 5.2 (est. time: < 15 minutes).

1024-bit floats. In this mode, double-precision floats are replaced with arbitrary-precision floats, here 1024 bits. This mode uses the MPFR library and its equality function to test whether computed test values meet the expected. This equality function is stricter than the usual equality function; for instance `0` and `-0` are different in MPFR. This yields spurious test errors, which is why we have to allow an error margin in the comparison between expected and computed values. We set this value to a small number, here `0.0000001`. You can choose any reasonable $\varepsilon > 0$.

To pass all tests using 1024 bits precision, launch:

```
$ TEST_ERROR_MARGIN=0.0000001 PRECISION=mpfr1024 make tests
```

Spec. name	Spec file
All	`all_ins_and_outs_2018.m_spec`
Selected outs	`all_ins_selected_outs_2018.m_spec`
Tests	`tests.m_spec`
Simplified	`simulateur_simplifie_2018.m_spec`
Basic	`basic_case.m_spec`

Figure 18. Correspondance between the specification names and files

Rounding mode. Here, floats are replaced by floating-point intervals, with down rounding for the lower-bound and up rounding for the upper bound.

```
$ PRECISION=interval make tests
```

Some tests fail with "Tried to convert interval to float, got two different bounds": in those cases, the chosen rounding mode changes the results of the computation.

Fixed precision. To pass the tests using infinite-precision integers with a fixed points of 40 fractional bits, launch:

```
$ TEST_ERROR_MARGIN=0.0000001 PRECISION=fixed40 make tests
```

You can replicate the failure of some tests due to low fractional precision by launching something like:

```
$ TEST_ERROR_MARGIN=0.0000001 PRECISION=fixed30 make tests
```

Rationals. To pass the tests with infinite precision using MPFR rationals, launch:

```
$ TEST_ERROR_MARGIN=0.0000001 PRECISION=mpq make tests
```

Checking the results of Section 5.3 (est. time: 5 minutes). The randomized tests are provided in `tests/2018/randomized`. You can run the compiler on them with:

```
$ TESTS_DIR=tests/2018/randomized/ make tests
```

The fuzzing-based tests are used by default in `make tests`, they can be found in `tests/2018/fuzzing/`.

Reproducing Figure 17 (est. time: 10 minutes). To measure coverage, just add `CODE_COVERAGE=1` before the make command. The coverage results are given in the last three lines of the execution trace. On the fuzzed tests, this gives:

```
$ CODE_COVERAGE=1 make tests
[...]
[RESULT] Test results: 275 successes
[RESULT] No failures!
[RESULT] Here is the estimated code coverage of this set of test runs,
[RESULT] broken down by the number of values statements are covered with:
[RESULT] zero values → 14 (0.0021% of statements)
[RESULT] one values → 576923 (88.0561% of statements)
[RESULT] two or more values → 78074 (11.9165% of statements)
```

For the randomized tests, you need to run:

```
$ CODE_COVERAGE=1 TESTS_DIR=tests/2018/randomized/ make tests
```

The DGFiP private tests are not publicly available, due to secrecy and security reasons invoked by the DGFiP.

A.6 Experiment customization

Inspecting the generated code. You can test the C backend with the `test_c_backend` make target. The generated code will be left in files like `examples/c/backend_tests/ir_tests.c`. You can then inspect these files to get a sense of what Mlang generates.

Similarly for Python, use the `test_python_backend` target. The generated code is `examples/python/backend_tests/tests.py`.

Switching year. Mlang is also available on the 2019 version of the income tax computation. To use this year, simply prefix all your `make` calls by `YEAR=2019 make ...`.

Creating new fuzzer tests. To create new fuzzer tests, move to `examples/c/backend_tests`. Then, create the executable to fuzz with:

```
$ make fuzz_harness.exe
```

You can tweak the crash condition by modifying the code in `fuzz_harness.c`. Before running AFL, you need to run `echo core | sudo tee /proc/sys/kernel/core_pattern` (the root password is `cc21`). Fuzzing instances can then be created with

```
$ NO_JOB=<0,1,2...> make launch_fuzz
```