Artifact Documentation: MonNom

Fabian Muehlboeck and Ross Tate

July 23, 2021

1 Getting Started

1.1 Prerequisites

1.1.1 Virtual Machine Image

Download the newest Virtual Machine Image (.ova) from `https://drive.google.com/drive/folders/1UHlqkndnw3tVGy1xr1uI2nARuopY1pVx?usp=sharing`.

You'll need to install VirtualBox from `https://www.virtualbox.org/` - we were using Version 6.1 . Once you have installed it, start it, and in the VirtualBox Manager, select `File -> Import Appliance ...`. Search for the `.ovf` file you downloaded earlier.

Once the import process is complete, you can start the virtual machine. You should be automatically logged in, the password (for sudo) is "monnom". On the desktop, there is a link to the `monnom` directory, which contains all the files for our implementation and experiments. The rest of this guide assumes you have opened a terminal in that folder.

1.1.2 Installing from GitHub

You'll need a Linux distribution with APT as package manager (we tested this on Xubuntu and WSL). Clone `https://github.com/fabianmuehlboeck/monnom`, then run the scripts `./prerequisites.sh` and `./build.sh`.

1.2 Testing whether the benchmarks run

In the `monnom` directory, run `./run-single.sh` (if you are in a non-graphical interface, for example in WSL, run `./run-single-png.sh` instead; the difference is that at the end, the folder will contain three image files you can copy to somewhere else to look at, while the standard script opens a browser with somewhat interactive reports). It should run through a single run of all three benchmarks and generate the figures in the paper - either opening a browser window with three tabs, or leaving three `.png` files in the monnom folder - one for each of the benchmarks: `float`, `sort`, and `sieve`.

1.3 Updating from GitHub

Before running the benchmarks, you may want to check whether a newer version of the artifact is available. Simply run `git pull` in the `monnom` directory (you may have to `git stash` any changes you may have made). Bigger updates may include an `./updateN.sh` script to update the prerequisistes; if a new such script is added after updating, please run it, and afterwards re-run `./build.sh`.

1.4 Potential Bugs

1.4.1 Rebuilding

Technically, the update scripts include a call to `./build.sh`. However, we have noticed in test runs that the re-builds sometimes do not completely run through. Please run `./build.sh` anyway after updating, and also try it if you encounter any errors that may hint at build problems.

1.4.2 Grift Installation

We noticed that on WSL, Grift may not complete compiling the garbage collector when installed during the prerequisites step. If Grift fails to run, please try running

```
raco pkg remove grift
raco pkg install grift
```

to reinstall Grift.

1.4.3 Not All Figures Visible

In particular in the browser setting, it may happen that a tab does not load the figure it is supposed to. In this case, please run `./make_charts.sh` (or, `./make_charts_png.sh` for the image version) to re-create the figures from the already generated data.

2 Step-by-Step

2.1 Claims

The artifact is supposed to justify all experimental claims in the paper; all sources for the various experiments that have been run have been included, as well as scripts that regenerate figures in the style of the paper, which—modulo hardware and environment—should produce similar-looking figures overall.

2.2 Running Experiments

In the `monnom` folder, run `./run-all.sh` (or, if not in a graphical shell, run `./run-all-png.sh`). This will run each benchmark program 10 times, average

the running times, and produce figures as in the paper based on those running times; either shown directly in a browser, or as `.png` files to be viewed elsewhere – either way, there should be one tab/png-file per benchmark: `float`, `sort`, and `sieve`, where the first two are only run for MonNom, while `sieve` is also run for Racket, Nom, and Grift. You can adjust the number of runs by editing the `RUNS` variable in the script you are using.

2.3 Exploring the Artifact

After running the experiments, each configuration of each program is available for inspection in some subfolder of the various experiment folders. You may be particularly interested in the various configurations generated for MonNom; the meanings of the folder names (make sure to look for hidden folders, as the folders are named in the pattern `.BM_[configuration]`)—and in particular the way we generate the various configurations for our MonNom benchmarks—are explained in Section 2.4.4. In addition to the MonNom source code, you will also find `.lli` files in each folder after the benchmark has been run; these contain the LLVM IR generated by the MonNom Runtime that was used to generate the actually executed machine code (see also Section 2.4.3 for more details on those files).

You can re-run any individual benchmark configuration right away by (after setting up the environment as in the beginning of Section 2.4 typing `nom [Float/Sieve/Sort]` in a benchmark configuration folder. If you also want to recompile the program from source (say because you changed something), you will need to copy the relevant project file from the parent folder (see Section 2.4.1 for details on project files).

If you want to try out your own program, the next section explains the relevant parts of the MonNom tool chain that you will need to use. Note, however, that the current state of the standard library is quite barebones; it essentially contains only those classes and their methods that were necessary to make the benchmarks work. You can look at the source code of the `NomC/Runtime` projects to look at precisely what standard library classes and interfaces are available. Much of the design of MonNom is aspirational and irrelevant for the present artifact. For example, the language contains keywords for member visibilities, such as `public`, `private`, `protected`, etc.; the compiler's type-checking procedures mostly take these into account; however, there is currently no run-time enforcement mechanism for them.

Finally, if you want to look at how a particular feature of the tool chain is implemented, Section 2.6 gives an overview of the most important parts of the source code.

2.4 The MonNom Tool Chain

To use the MonNom Tool Chain manually, first make sure your environment is configured to use the right paths:

```
. /opt/intel/oneapi/setvars.sh
. [PATH-TO-MONNOM]/setpath.sh
```

Note that the toolchain is still very much in development status, and was developed on Windows. Thus, the implementation is incomplete in many places, and the Linux version created for the Artifact is missing several scripts to make the use of some tools more straightforward. This section describes the purpose and usage of the various points. Section 2.6 gives an overview of the most important places in the sources of these tools.

2.4.1 NomProject

The MonNom compiler works with project files (`.mnp`), which are XML files describing the source files (`.mn`) that should be compiled and any dependencies a project might have. The dependencies part is not fully implemented, but `NomProject` can still be used to manage the XML files more easily:

```
dotnet "$MONNOMBASE/monnombuild/NomProject/Release/netcoreapp3.1/NomProject.dll" \
  create MyProjectName *.mn
```

Creates a new MonNom project named `MyProjectName` and adds all `.mn` files in the current folder to it. The latter parameter can be omitted to create an empty project, files can be added later using

```
dotnet "$MONNOMBASE/monnombuild/NomProject/Release/netcoreapp3.1/NomProject.dll" \
  addfile MyProjectName File1.mn File2.mn ...
```

To be executable, a project needs a main class that contains a main method. The main class can be set with

```
dotnet "$MONNOMBASE/monnombuild/NomProject/Release/netcoreapp3.1/NomProject.dll" \
  setmain MyProjectName MyMainClass
```

To get around the missing implementation of dependencies for the purposes of benchmarking, you can use the following hack: manually edit the project file to contain a list of library files (which will not be inlcuded in the devolution scheme of the benchmark generator, see below). For example, in the `float` benchmark, the `Enumerable` class providing some standard functionality over arbitrary enumerables is specified as a library as follows:

```
<libraryfiles><file name="Enumerable.mn" /></libraryfiles>
```

2.4.2 NomC

The MonNom compiler takes a project file, type-checks the program, and generates MonNom Intermediate Language files (`.mnil`) and a manifest file (`.manifest`). Long-term, these are supposed to be enclosed in a singular zip-file, but for now they are just laid out flat in the project folder. To call the MonNom compiler, run

```
dotnet "$MONNOMBASE/monnombuild/NomC/bin/Release/netcoreapp3.1/nomc.dll" \
  -p path/to/my/project --project MyProjectName
```

The compiler will either output the files mentioned above, display an error message (say for a type error), or crash (it is research code, after all).

2.4.3 Runtime

The MonNom Runtime executes the intermediate code generated by the MonNom Compiler. It uses the LLVM JIT libraries to compile the intermediate code to machine code before actually running the program. You can call it by running

```
nom MyProjectName
```

In from of the project name, you can add a number of optional parameters to control the behavior of the runtime:

- `-d[N]`, controls the level of debug output, where `N` is optionally a number between 0 and 3 (if `N` is omitted, but `-d` is present, the default value is 2; without `-d`, it is 0). At level three, almost every intermediate value in the LLVM IR is printed, along with location information pointing to the relevant place in the IR. The runtime always outputs two files in the directory of the program: `llvmir.lli` and `llvmirPO.lli`. The latter is the pre-optimized llvm code that is generated by the code of the runtime, while the former contains the LLVM IR that is actually executed after running through the optimizer and possibly inserting debug print calls. If debug prints are inserted, look for lines that say `call void RT_NOM_STATS_DebugLine(...)`. The second parameter to this call is the line reference printed in the program's output, and the fourth parameter the value that is being printed.

- `-p path/to/my/project` works like the path argument for the compiler, effectively looking for the project's manifest in that path.

- `-v` turns on verbose mode, outputting a more readable form of the MonNom Intermediate Code as it is read in, the various states of LLVM IR, and the actual addresses of various symbols after the code is compiled and before actual program execution starts.

- `-o[N]`, controls the optimization level of the compiler, using the groups of optimizations clang uses for the various levels (0-3), default 2.

- `-s[N]`, controls the level of statistics collected about running the code (causes lots of overhead). `N` is a value between 0 and 4, but 0 or `s` alone already means that casting and call statistics are collected and displayed when the program finishes. At higher values, the runtime collects timings between instructions (extremely slow).

- `-f[N]`, controls the level of function timings (`N` between 1 and 3, default 1). At level 3, each LLVM-level function is timed, analogous to a profiler (high overhead; without `-f`, the level is 0, and no timings are collected)
- `-w[N]`, sets number of warmup runs (default 0, with just `-w`, 1, N within 32 bit integer range). Runs the program `N` times, suppressing its output, before running it a final time with output enabled. Between each run, the garbage collector is forced to collect, and, more importantly, cast and run-time type allocation caches are reset (see discussion in Section 2.6).
- `--stopatend`: lets the runtime wait for an input line after finishing running the program (mostly for debugging with Visual Studio)
- `--version`: displays version information at startup

2.4.4 BenchmarkGenerator

For the purposes of this Artifact, the benchmark generator is key to understanding how the various configurations for MonNom's benchmarks are created. While benchmarks for all other languages are created by mixing completely typed and completely untyped versions of various parts of the program, MonNom's benchmark generator starts from a completely typed program and generates various *devolutions*. Type annotations are still removed on a roughly per-module basis as in the other benchmarks, but the structural-to-nominal transition requires some more complicated changes to the code. The benchmark generator therefore runs all stages of the compiler except bytecode generation, and uses the type information generated by the type checker to generate source code for the various configurations we want to check.

A fully typed/nominal program consists of just fully typed classes and interfaces. Each configuration is described by a string describing their state in alphabetical order. Thus, fully typed/nominal configuration of the `float` benchmark is called `ICCC`, indicating that the interface `IPoint` is a fully typed nominal interface, and the classes `Main`, `Point`, and `PointMapFun` are fully typed classes. For interfaces, there exist two other states:

- `J` indicates an interface whose type annotations have been replaced with dyn. In this case, any class implementing the interface must adapt the signatures of its methods to match the interface's annotations.
- `K` indicates an interface has been removed from the program's source code altogether. This means that any class or interface implementing or extending it must remove the corresponding entry from its inheritance declarations, and any occurences of the interface in type annotations must be replaced with `dyn`.

For classes, another state `D` corresponds to the `J` state of interfaces - it's just the nominal class but without the type annotations. Furthermore, depending on the shape of the class (in particular, if it contains no static methods), more

states may exist: if it only contains a `this`-method (in the calculus in the paper, that is a λ-method), the additional states are `L` (typed lambda) and `M` (untyped lambda); otherwise, the additional states are `S` (typed record) and `T` (untyped record). In all cases, the class is removed from the code, and all calls to its constructor are replaced by expressions creating lambdas or records accordingly.

The benchmark generator generates all possible combinations of these stages for all classes or interfaces in the program. Thus, for example, `KDTM` is the most untyped/structural configuration of the `float` benchmark, as the `IPoint` interface has been removed, the main class contains static methods and thus can neither become a lambda or a record, the `Point` class becomes an untyped record, and the `PointMapFun` class becomes an untyped lambda.

To use the benchmark generator, you can use the `nombench` command with the following possible parameters:

- `--project MyProjectName` — the project from which to generate the benchmark configurations, as for `NomC`
- `-p path/to/my/project` — the path in which to find the project file, if it is not in the current directory, as for `NomC`
- `-r N` — how often each configuration should be run. Note that if this parameter is missing, the benchmark generator only generates the various configurations, but runs nothing.
- `-o N` — the optimization level that should be passed to the corresponding runtime flag (default: 3)
- `--byfile` — group classes and interfaces by file for devolution: all classes and interfaces defined in the same file are not varied independently, but devolve together. This cuts down on the number of configurations, which otherwise can explode quite quickly. We use this flag in the `sort` benchmark.
- `--priority` — try to run the actual benchmark runs with the highest process priority. Typically needs administrator-level permissions (i.e. `sudo`).
- `-w N` — use warmups for running the benchmark; forwarded to the corresponding runtime parameter (default: 0)

For a normal benchmark run, at least the `--project` and `-r` parameters are required, as in

```
nombench --project MyProjectName -r 1
```

For examples, see `run-monnom.sh` in the main folder.

2.5 The Artifact Scripts

The artifact comes with a number of scripts in the main folder:

- `build.sh`—updates the builds of all parts of the MonNom toolchain, say after pulling an update via `git`. If you need a complete re-build, delete all sub-folders of the `monnombuild` folder (leave at least the `nombench` script in it), and run `make clean` in `sourcecode/Runtime`, then run `./build.sh` again.
- `cleanup-benchmarks.sh`—deletes all intermediate results from running the benchmarks. This is particularly useful if you want to execute fewer runs than on a previous benchmarking run, as otherwise for some languages the later runs of the earlier benchmarking run would be included in the figures for the new run.
- `collekt.rkt/collect.sh path/to/benchmark` — collects benchmark results for individual configurations of the MonNom and Grift benchmarks and puts them into a `results.csv` file in the root folder of that benchmark.
- `make_charts.sh`—if all the benchmark data has been generated (i.e. also after running `collect.sh`), generates the figures from the paper, and displays them in browser tabs
- `make_charts_png.sh`—like `make_charts.sh`, but renders the figures as `.png` files.
- `prerequisites.sh`—installs the necessary programs and libraries needed to run all the benchmarks on a standard Ubuntu distribution
- `run-all.sh`—runs all the benchmarks 10 times and generates figures based on the average results
- `run-all-png.sh`—like `run-all.sh`, but figures are output as `.png` files.
- `run-[lang].sh`—runs all benchmarks available for that particular language
- `run-monnom-warmup.sh`—like `run-monnom.sh`, but lets each benchmark use two warmup runs.
- `run-single.sh`, `run-single-png.sh`—like `run-all.sh` and `run-all-png.sh`, respectively, but only uses a single run for each benchmark (used to test whether the benchmarks can run successfully)
- `setpath.sh`—adds the relevant directories to run the benchmarks to the `PATH` variable, and exports the root MonNom directory as `$MONNOMBASE`.

2.6 Exploring the Source Code

To be written