OOPSLA 17 Artifact - Step By Step Guide

Fabian Muehlboeck, Cornell University
Ross Tate, Cornell University
({fabianm,ross}@cs.cornell.edu)

Paper: Sound Gradual Typing is Nominally Alive and Well

Note: We provide our Artifact for both a Windows and a Ubuntu environment. In cases where the Windows setup differs from the Ubuntu setup, the text applying to Windows is added between [brackets] (except for discussions of command-line arguments, where brackets have the usual semantics of denoting placeholders). The instructions are phrased assuming you are using on of the provided Virtual Machines. Specific locations of things may differ if you use the standalone setup.

1 Running the Benchmarks

1.1 Steps

- Run `run_benchmarks.sh` [`run_all_benchmarks.bat`] on the Desktop
- NOTE: the output of the scripts may contain some error messages about files either not or already existing. These are most likely due to the simple way we tried to ensure to make the benchmarks re-runnable - we move all result files from previous runs (which may not exist) into a new backup folder (which may already exist).

1.2 Description

The purpose of this artifact is to first and foremost allow others to reproduce the results of our benchmarks. Hence, the user's home folder [the desktop] contains the folder `BENCHMARKS` containing all the benchmark programs running reasonably on Ubuntu [all our benchmark programs] (it turns out that the C# programs are written in a way that neither Mono nor the newer Core CLR deal with very well. This is probably because no one would seriously write C# programs like that. In our case, we wrote the programs this way because we are trying to stay close to the Racket originals). The shell [batch] script on the desktop should run all programs and leave csvs with the running times and bar graph/scatterplot images (depending on the benchmark) in the `BENCHMARKS` folder. There is a shell [batch] script for each kind of benchmark in the `BENCHMARKS` folder that can be run individually.

On a modern desktop, the whole benchmark suite should take around 1-3 days to complete (mostly due to Snake; Sieve takes a few hours and the Python-related benchmarks should be completed within minutes). The results usually show little variance, so the process can be sped up by just running the Snake and Sieve benchmarks 1 instead of 10 times. For that, all the lines calling `Benchmark.exe` in the files {sieve,snake}.sh[.bat] need to have the argument "-n 10" changed to "-n 1".

2 Comparing the Programs in Different Languages

2.1 Steps

- Racket:Nom : compare the files `NG_sieve/annotated/*.ng` with their counterparts in `racket_sieve/annotated` (similar for `snake`).
- Python:Nom : compare the file `NG_bm_go/annotated/bm_go_main.ng` with its counterpart `py_bm_go/typed-source/bm_go.py` (similar for other python-based benchmarks).
- C#:Nom : compare the files `NG_sieve/annotated/*.ng` with their counterparts in `csharp_sieve/annotated` (similar for `snake`).

2.2 Description

Since we took our Benchmarks from Benchmarking suites created for other gradually typed languages and compare the results of Nom with the results for the respective languages, an obvious question is how the source and behavior of our programs compare.

2.2.1 Nom and Racket

The two Racket programs we adapted from the benchmark suite of Takikawa et al., `Snake` and `Sieve`, are relatively simple programs that do not rely on very complicated types or contracts, thus, the basic layout of the programs should strike the reader as similar. However, the main point for why we think that there is a reasonable comparison to make is that we took care to make the Nom programs imiate the number of transitions between typed and untyped code. This can be verified in two steps: first, the reader should observe that the control flow of the program is the same in the Nom versions as in the Racket versions, in the sense that there is a corresponding method for each Racket function in the corresponding file and control flow works the same way. The only exception is a recursive function in `Sieve` – `stream-get` – that we had to convert into an iterative loop because we don't have tail call elimination. The function does call itself, thus there is no typed/untyped module transition that is lost this way. Second, since Nom's gradual typing is more fine-grained than Racket's (i.e. Nom allows to changed "typedness" for individual values, while in Racket a whole module is either typed or untyped), type information from typed code might "leak" into untyped code and thus eliminate checks. We combat this by casting values where type information would play a role that are coming from other modules to `Dyn` in the unannotated versions of each module to mimic Racket's behavior more closely.

2.2.2 Nom and Reticulated Python

Python is a lot closer to Nom, so the programs we adapted from Vitousek et al. are a lot closer to their source material. Note that the python files used for the benchmarks are already compiled down to Python from Reticulated Python, so we included the Reticulated python files in a folder called "typed-source" in each of the python benchmark folders. The biggest differences you should see are: first, that Nom doesn't feature generics, so the "raw" list class ArrayList has to suffice for all list-like types. Second, since Nom does not have implicit Nulls, static fields and constructors are a little more restricted. In particular, while all newly introduced fields have to be initialized before the super-constructor call, the use of the `this` pointer is disallowed until afterwards. Thus, as an example from the `go` benchmark, the classes `EmptySet` and `Board` each have a field of the other type, and `EmptySet` uses a structural type specifying a particular method of `Board` (which we model with an interface) as a constructor argument. Therefore, we created a dummy class implementing said interface, an instance of which is given to the `EmptySet` constructor that is called within `Board`'s constructor when it initializes its fields. An alternative to this strategy would have been to use the type modifier ! for `Board`'s field to be able to leave it `null` until the `this` pointer is available. This modifier essentially indicates that the field may contain `null` or an `EmptySet`, but can optimistically be treated as an `EmptySet` – however, uses of it will always have a run-time type check inserted to make sure that the value is indeed not `null`. Some other locations use the ! modifier, such as the `reference` field in the `Square` class in `Go`.

2.2.3 C#

The C# programs are essentially translated 1:1 from Nom. C# does have lambdas and generic types, but they do not interact well with the gradual guarantee (which is important for the arbitrary mixing of modules to work). In any case, a "real" C# program would be written with a completely different structure, but the same goals as for Nom apply (have the same behavior in terms of typed/untyped transitions), and the translation from Nom should satisfy that goal.

3 Checking the Output of the Benchmarks

3.1 Steps

- Look at images corresponding to the benchmarks in the `BENCHMARKS` folder. They should match the information in the graphs in our paper (the python-related benchmarks are separate by language for scaling reasons and separate by benchmark to make the scripts more modular; the important part is whether the typed or untyped version of the program is faster).
- Look at the two additional images `sieve_stats.png` and `snake_stats.png` to verify our claims about matching the Racket version's level of typed/untyped interaction.

3.2 Description

The first set of images simply plots the running times of each configuration for the different benchmarks and languages. As Snake has 256 configurations, the configurations are grouped by the number of typed modules in a scatter-plot, just like in the paper.

The last two images are not in the paper, but we mention that we verified that the translated programs have levels of interaction between the modules and thus typed and untyped code that match those reported by Takikawa et al. (they put the number of interactions between modules in those two benchmarks into a range between one million and one billion). We don't know how to easily produce those counts for Racket, so we are just trying to get into the right ballpark. The figures are based on the number of times any object's type identifier is checked by the run time. This also happens in some cases in typed code (such as for interface method table lookups), but the increase of such cases over typed code means some sort of type checking or type-based decision making is going on, and for both `sieve` and `snake`, this difference is in the hundreds of millions (note the scales of the figures). A single transition between typed and untyped code may involve multiple checks and type lookups, but as the number of arguments of a function is typically low, the multiplier is probably 2 or 3, which still remains in the upper part of the range suggested by Takikawa et al. Together with the structural match between the Nom and Racket versions of the programs, these numbers should give us confidence that the programs are comparable with respect to being a stress-test for gradual typing.

4 Technicalities

Refer to this section if things go wrong or you want to try out things that are not included in the benchmark by default. These explanations should be enough to understand what the shell [batch] scripts are doing and how to edit them if necessary.

4.1 Important Programs and Scripts

4.1.1 Benchmark.exe

Takes care of creating all configurations of a program, compiling and running them, interpreting their timing output and creating standardized result csvs. It expects there to be at least subfolder “annotated” and “unannotated” in a benchmark folder, containing the same file names. There can be a “shared” folder whose contents are copied into every configuration. `Benchmark.exe` will create another subfolder called “benchmark” that will in turn contain a subfolder for every configuration, plus the “.csv” files containing the timings of every run. It’s command-line arguments are as follows:

- “-l [Language]”: one of:
  - `ng-bash` [`ng`]: Nom
  - `py3` [`py`]: Python (NOTE: the `Benchmark.exe` shipped with the original Artifact only knows `py` and interprets it as using the “python3” command as a hack. That is, it will not work with Python on Windows, nor will it work with the new Ubuntu-scripts)
  - `cs-mono` [`cs`]: CSharp
  - `ng-bash-stats` [`ng-stats`]: Nom with run-time statistics collection turned on (do not use to measure timings)
- “-r [Benchmark-Root-Dir]”: the root directory of the benchmark to run. Usually a subfolder of `BENCHMARKS`, e.g. “-r NG_sieve”
- “-n [num-of-runs]”: the number of runs of a particular benchmark (presumably to take the average timing afterward).
- “-s”: instructs the benchmarker to skip the compilation step. It will assume that the `benchmark` subdirectory of the benchmark is already in a state where all configurations have been compiled and are ready for execution
- “-c”: cleanup. The benchmarker will delete files that would be rebuilt in another compilation step, such as compiled executables, after running the benchmark.
- “-w”: nowait. By default, the benchmarker waits for some user input at the end of its run. This is undesirable for scripts that run multiple benchmarks, so “-w” should be used for scripts.
- “–no-parallelization”. By default, the benchmarker tries to run several compiler instances for different configurations in parallel. This has caused problems for `snake`, so be advised to use this flag there. The actual timed runs of compile programs are never run in parallel.
- “–toolroot [racket-binary-dir]”. Can be used to use a different version of racket than the system default. Should point to wherever `raco` and `racket` can be found.

4.1.2 NominalGradual.exe

This is the Nom compiler. It takes all ".ng" files in a folder and generates C code corresponding to the program composed of those ".ng" files. A C compiler will need to compile those C files into an actual executable. The Nom compiler has the following command-line arguments (it might show more in the usage information, but most of those are defunct):

- "-r [root-dir]": the directory in which the compiler should look for ".ng" files
- "-m [main-class-name = Main]": the name of the class that contains the main method. By default it is `Main`.
- "-s": turns on the collection of run-time statistics (used by the "-stats" versions of Nom in `Benchmark.exe`). The final executable generated by the C compiler will create a file called `statsout.txt` [`statsout-win.txt`] in the directory where it was run at the end of its run containing those statistics. They may not necessarily all be easy to interpret.

4.2 plot.py and scatterplot.py

These scripts are used to generate the images representing the benchmark timings. `plot.py` creates bar graphs, `scatterplot.py` creates scatterplots, meant for benchmarks with large numbers of configurations that should be grouped by the number of typed modules (i.e. `snake`). Both use the following scheme for command-line arguments: the first argument is either "SHOW" or a file name for an image (e.g. "image.png") that tells the script where to direct the output. "SHOW" will let the script immediately display the graph, otherwise it will save the graph to the given file name. After the first argument, arguments come in pairs. The first is a csv file as output by `Benchmark.exe` containing timing data. The second is the display name of the language to be used in the graph's legend. An example of such a pair is "ng_sieve.csv Nom".

4.3 process-stats.py

This script collects the run-time statistics generated by using a "-stats"-version of Nom in `Benchmark.exe` from all the configurations of a given benchmark (second argument, as in "-r" for `Benchmark.exe`) and creates a graph of one of the lines of the statistics (the fourth argument is the 0-based line number in the statistics file). If there are more than 16 configurations, the graph is a scatterplot, otherwise a bar chart. The first argument is the file name of the image that the graph should be saved to or "SHOW", as in `plot.py` and `scatterplot.py`. The third argument is the file name of the statistics file, which depends on the platform. It is `statsout.txt` [`statsout-win.txt`].

4.4 [lang]ship folders and benchmark-util

The folders `csship`, `ngship`, and `pyship` contain files that are copied into every benchmark configuration of the respective languages. Racket needs the folder benchmark-util to be registered as a module, using the command `raco link benchmark-util` in the `BENCHMARKS` folder (the shipped VMs already have this done, and the scripts that run all benchmarks make sure it is done, too). The `csship` and `ngship` may be important if issues arise, especially when setting up on Windows. They contain the scripts to compile C# and C code, which on Windows have to set up the compiler environment first (`call vcvarsall x64`). In some Windows setups, this may need to be adjusted. The benchmarker usually uses the `fastcompile` scripts. The `compile` scripts don't optimize and add debugging symbols in case those are needed.