Helium: An On-Demand Dynamic Analysis Framework

Hebi Li

Contents

1 Introduction

This is the manual for Helium framework. This document has a pdf version. You can directly go to the source code on GitHub at Helium and Helium2 (with advanced builder components).

Helium is a framework to generate partial programs and run dynamic analysis. It features a syntactic patching algorithm that find the extra code in addition to the user selection that is necessary for a valid partial program. It also features a demand-driven context search algorithm to find smaller partial programs that preserve a given program property. Please find our paper for technical details.

1.1 A Brief History of the Framework

In this section, we will describe the brief history of the development of the framework, and gives you a guideline of the versions and how to further develop your research ideas on top of it.

The first version is written in C++, with the choice of SrcML as the front-end compiler. We will refer to this version as Helium-srcml. This version is the most complete version, featuring AST data structures, syntactic patching, selector, builder, tester, and analyzer. Further developments are focused on "builder part" for the syntactic patching paper, and other components are not developed along the way. Thus, this version is appealing for readers that are concerned with *demand-driven context search* and *query resolving*. These components are not further developed, thus is up-to-date.

However, Helium-SrcML suffers from its low build rate and lack of applicability of complex projects. The major reasons are: (1) SrcML is not precise (2) SrcML does not provide library support as Clang, thus a lot of infrastructure are built from scratch, they are error prone and inefficient (3) SrcML does not support project build files, does not support compilation flags, does not respect header files, thus is impractical for handling large projects. (4) the SrcML project is lack of maintenance for bug fixes since 2015.

This motivates the transformation from SrcML to Clang. The second version, which we refer to as Helium, is developed *on top of* Helium-SrcML code base, but with Clang as the front-end. Most internal AST representations are changed, and *demand-driven context search* and *query resolving* code is *removed* to avoid compiling conflicts. This version also features a utility tool for visualizing Control Flow Graph, and a reader for iClones RCF file. This version is interesting for readers who are looking for implementing tools on top of clang.

The problem of *Helium* is the applicability of complex projects. The root problem we identified is the choice of generating completely stand-alone code. This involves tricky extraction and placement of tons of complex code snippets in case of complex benchmarks. Complex projects also require dedicated compilation flags, typically specified in project build files, to build source code precisely. *Helium* does not utilize it when parsing the source code.

To handle complex benchmarks, Helium2 is introduced. It advances from *Helium* in primarily two aspects: (1) generate only source code of main source code, rely on original source code to provide definitions and object linkage. (2) It utilize the build files, by introduce a dumping plugin into clang library, and invoke `make` using the instrumented clang as C compiler. The code base is *completely separate* from *Helium*, written in C++ (as Clang plugin for parsing) and mainly racket, though the parser code based on Clang are substantially similar to previous version. This version only implements the builder, focusing on build rate of complex projects, but it does not feature *test generation* and *demand-driven analysis*. The code is tested on several very large open source projects, and it supports more types of expressions and statements in its parser.

1.2 Future Development

Thus, to further develop demand-driven algorithm, you basically can choose to either work on C++ code base or racket code base. It is perfectly fine for either, since the two code bases are separate.

1. Develop Helium2: This is preferred way, as V2 has a more advanced builder component. As lisp code, it has less code, and the code is considered cleaner. Racket will probably save you tons of time, just as it saved mine. The downside is that, you might need to rewrite some functionality (specifically the demand-driven searching, test input generation, and query resolving) in racket.

2. Develop Helium: In case you are not comfortable with writing lisp code, you can work on the C++ code base. If you don't need to run on complex benchmarks, you probably don't even need the advanced builder component. If you do need it:

   - Developing the advanced builder system into V1.3 is not terribly hard, given V2 source code in hand.
   - It is also possible to just use V2 as a utility tool that accepts benchmark and selection criteria, and outputs generated code. You probably need to (1) play with the generated code for adding input/output instrumentation (2) develop the interface between the two.

2 Getting the code

The source code of the three versions:

- Helium-srcml (`srcml` branch of the repo)
- Helium
- Helium2

Docker images are prepared, including the original dockerfiles. The purpose is to get you quickly started, and also shows the exact steps to follow if you want to configure on your machine. You can build the images yourself using the dockerfiles, or obtain the prebuilt image from DockerHub.

- lihebi/helium (Arch Linux, xfce4 GUI). [dockerfile]: This is a GUI (via VNC) version of Helium based on Arch Linux, including both Helium and Helium2.
- lihebi/helium-nogui (Arch Linux, NO GUI) [dockerfile]: non-gui version of above
- lihebi/helium-srcml (Ubuntu 16.04, NO GUI) [dockerfile]: (SrcML does not work in Arch Linux)

Be sure to familiarize yourself with the concepts of docker container before using it, so that you don't lose your work by accident (the `--rm` option below means delete the container on exit). Basically you can create the GUI instance by:

```
docker run -t -d --rm lihebi/helium
```

The IP address will be printed, say `172.19.0.3`, you can connect to the server via

```
vncviewer 172.19.0.3:5901
```

with password "helium". You will be `root`. For non-gui versions, you can use interactive shell as:

```
docker run -it --rm lihebi/helium-nogui
docker run -it --rm lihebi/helium-srcml
```

If you prefer to compile Helium on your machine, it should be trivial to do that with the docker files in hand. Specifically, you need to install the relevant packages (or equivalent in other distributions), clone the source code, and do the compilations as normal. You can also configure the settings of Helium by following just the steps in the docker files. In case you are interested, here is a list of packages Helium requires:

- Helium2: `racket`, `clang`, `llvm`.
- Helium: `gtest`, `doxygen`, `graphviz`, `pugixml`, `rapidjson`, `boost`.
- Helium-SrcML: `sqlite3`, `boost`, `gtest`, `pugixml`, `clang`, `exuberant-ctags`, `r-base`, `python3`, `z3`

3 Running the tool

3.1 Helium-SrcML

The docker image has set up the environment for you, the command `helium` and `helium-preprocess.sh` are ready to use. In case you are setting it up, refer to the corresponding `Dockerfile`. In particular, simply run `setup.sh` in the root directory of Helium. Then, run `make systype.tags` in the directory as well to create ctags index for system types.

The config file is `helium.conf` at the root folder of Helium, also symbolically linked to `$HOME/.helium.conf`. You need to modify the `helium-home` directory to your local path.

Running the demand-driven bug signature search bug requires two steps. The first is to preprocess the benchmark using a c compiler, to remove conditional compilations and comments. To do that, issue the following script, and the output will be written into directory `helium-output/benchmark`.

```
helium-preprocess.sh /path/to/benchmark
```

In the second step, you provide *Point of Interest (POI)* via a file, containing the bug location and failure condition. It can be a csv file or a org file containing a table. The table must have the following columns, in whatever order:

- benchmark: should be the name of the benchmark folder
- file: the file name which contains POI
- linum: the line number of POI
- type: usually "stmt"
- failure-condition: the failure condition

For example, the `test/simple/poi.org` looks like this:

```
benchmark, file, linum, type, failure-condition,
simplebench, a.c, 8, stmt, output_int_str.strlen > output_int_buf.size
```

Running on such POI is simply:

```
helium helium-output/simplebench --poi-file poi.org
```

The tool should finally print out "query resolved" in green.

As another example, try to run on the `gzip-1.2.4` benchmark provided in the folder `benchmark/buffer-overflow/gzip-1.2.4.tar.gz`:

```
helium-preprocess.sh gzip-1.2.4.tar.gz
helium helium-output/gzip-1.2.4
```

To be easy to run experiment, you can provide a default POI file in the configuration file `helium.conf` (for example data/poi/poi-new.csv), so that you don't need to specify the `--poi-file` option every time.

3.2 Helium

The `helium` and `pyhelium` command should be available if you are using the docker image or setting up yourself following the `Dockerfile` (simply run `setup.sh`).

You can visualize the AST or CFG. For example, for the file `helium/test/cfgtest/call.c`, you can visualize AST and CFG via:

```
helium --dump-ast helium/test/cfgtest/call.c
helium --dump-cfg helium/test/cfgtest/call.c -o output
```

The visualized AST will be printed out to standard input. The procedure level CFG and ICFG will be generated into `output` folder, with format dot, ggx, grs, png. You can also supply `--cfg-no-decl` to suppress variable declaration nodes on generated CFG.

This version of code is used to run experiment for build rate of GitHub random projects, as well as the iClones experiments. The scripts to use is provided at scripts/pyhelium.

`pyhelium` is the main entry for the script system. Run `pyhelium --help` for available commands. Specifically, `--create-selection` is used to create random selection criteria. `--preprocess` is used to preprocess benchmarks, create header file dependencies (`include.json`) and code snippets (`snippets.json`). Finally `--run-helium` with `--selection` to run helium tests on created selection file. An example of running of build rate tests for random selected criteria can be done via the following command sequences:

```
pyhelium --create-profile /path/to/benchmarks -o profile.json
pyhelium --preprocess /path/to/benchmarks -o cpp
pyhelium --create-selection cpp -o sel
pyhelium --run-helium cpp -s sel -o output
```

Running iclones experiment requires `jdk` as well as the iclones tool and its RCF library. The script `pyhelium/iclones_process.py` is provided for this experiment. First, run iclones on preprocessed benchmarks. The result is a RCF file describing the clone pairs. The RCF is parsed into plain text, and used to generate selection criteria. Helium is invoked to run on those selection criteria to generate dynamic information. Finally, `compare` function in `iclones_process.py` is used for analyzing the result. The following script shows the detailed command for such experiment:

```
iclones_process.py --run-iclones --input /path/to/benchmarks --output result.rcf
iclones_process.py --read-rcf --input result.rcf --output result.txt
iclones_process.py --parse-result --input result.txt --output /path/to/seldir
iclones_process.py --run-helium --seldir /path/to/seldir --output output
```

3.3 Helium2

The `helium2` and `hcc` command should be ready to use.

Running Helium2 to build partial programs involves two steps. First, you need to invoke the compiler *Helium C Compiler (hcc)* on source code to the parse tree into files. It is basically a plugin inside Clang, and during the compilation of the source file (say `a.c`), also outputs a parse tree file (`a.c.ss`) to be loaded by the framework. By building this into the compiler, complex projects can be simply processed by calling `make` with `hcc` as the compiler, for example:

```
make CC=hcc
```

The second step is to load the parse tree, specify selection criteria, and generate partial programs. This is implemented in racket. The preferred way is to run it interactively. To do that, you may need to get yourself familiar with tools for developing racket code. As far as I know, Emacs is the ideal tool for that. Simply install racket-mode plugin in emacs, and you can

run the file `main.rkt` through `racket-run` (`C-c C-c`) command, and evaluate expressions with `racket-send-last-sexp` (`C-x C-e`). At the end of the file `main.rkt`, there are two test blocks containing two sets of experiments: random selection (`run-random-selection`) and selection from specification file (`run-from-selection`). Evaluate those expressions will run the tool.

If you prefer command line, a command line interface is also provided, in `helium.rkt`. However, you may need to extend the interface in case of advanced usage. A `helium2` executable is already built via `raco exe -o helium2 helium.rkt`. You can directly run the executable or run `racket helium.rkt`, they are equivalent. You can print help messages via `helium --help`. The input to the interface is the path to the benchmark. For example, running for the test project, using the following command:

```
helium2 /path/to/helium2/test/proj
```

The default mode is to run random selection, you can also manually provide code selection via a *selection specification file* (see the files in `cmt-data` folder for examples). For example, the `cmt-data/test.ss` folder contains two selection case, one with `foo.c` line 81, 83, another with `bar.c` line 15,16. Running the following command to run these two experiments:

```
helium2 -s /path/to/helium2/cmt-data/test.ss /path/to/helium2/test/proj
```

The above outputs `.` for success and `x` for failure. You can supply `-v` flag to enable verbose mode, in which case the output folder and compile success or failure message is displayed.

In case of complex benchmark, you might need to provide flags. However, the flags are typically complex (see `bench-data.rkt` for examples), thus you need to specify them in a file. The interface currently provide template based matching, and currently support linux, git, vim, ffmpeg, and openssl, and they are defined in `bench-data.rkt`. Note that the flags contain file system path information, so you need to modify to the path of your specific path. For example, you can specify the following command for git benchmark:

```
helium2 -f git /path/to/git
```

4 Developing the tool

This section provides a high level overview of important files in the code base.

4.1 Helium-SrcML

`main.cc` provides the entry point for the tool. It invokes functions defined in `helium_options.cc` for parsing command line arguments and configuration files.

`parser` folder contains parsing utilities of C source file, built upon SrcML. In particular,

- `ast-node.cc/h`: defines AST node classes used through the framework. Each node has corresponding function (`GetCode`) to generate C source code from AST node.
- `ast.cc/h`: defines the data structure of the AST. It also defines syntactic patching. The `AST::CompleteGene` function does the job for syntactic patching, which utilizes `ComputeLCA` function for LCA based syntactic patching.
- `cfg.cc/h`: defines CFG class, and the procedures to build CFG from AST. The CFG is used for demand-driven context search.
- `point_of_interest.cc`: parse point of interest files
- `xmlnode.cc`: defines wrapper classes using `pugixml` for each type of node in the output of SrcML
- `xmlnode_helper.cc`: helper functions for dealing with SrcML output, including traversing the parse tree, getting text, querying nodes.

`workflow` folder defines the logic of Helium framework.

- `helium`: accepts POI, implements the worklist algorithm, and handles query propagation
- `segment`: defines the class Segment, representing each query
- `generator`: generate partial programs, including main.c, support.h, makefile
- `builder`: compile and execute the generated program
- `tester`: test input generation, running the test
- `analyzer`: takes the CSV files generated by tester, run analysis using R scripts, and symbolically resolve query using Z3 (`Analyzer::ResolveQuery` function).

`type` folder defines types, both primitive and composite types, for input instrumentation and generation:

- `type.cc/h`: defines class `Type`, and all subclasses for different kinds of types. Each type has functions to generate input and output code, as well as generate random or pairwise input.
  - `primitive_type.cc`: defines `IntType`, `BoolType`, `CharType`, etc.
  - `composite_type.cc`: defines `StructType`
  - `sequential_type.cc`: defines `ArrayType` and `PointerType`
- `type_helper.cc`: defines functions for generating `scanf` and `printf` instruments for different types

`resolver` folder defines code snippet extractions, local and system headers.

- `snippet.cc/h`: defines `Snippet` class to hold code snippets. It parses ctags index file via ctags library to extract snippet information. For each code snippet, it look into the original source file to extract the corresponding C source code as string.
- `snippet_db.cc`: the snippets are extracted and inserted into a Sqlite database. Code snippets can be looked up using their type and keywords. Call graph is built for all function snippets.
- `snippet_registry.cc`: deprecated
- `system_resolver.cc`
- `header_resolver.cc`: manager header dependencies in the benchmark, used for deciding the order of code snippets
- `system_resolver.cc`: check on local system whether a header is available, and if yes, what are the library flags to use for compilation (-I and -L)

4.2 Helium

`main.cc` provides the main function to invoke Helium. It also defines the procedure to run Helium on a selection criteria.

`parser` folder deals with the parsing of C source code, based on Clang.

- `ClangParser.cc`: `ClangParser::parse` function is the entry point, and it accepts a file, and returns a `ASTContext*` pointer for the generated AST. This file is dominated with Recursive Descent style parsing functions.
- `AST.h`: defines all AST classes used through the framework
- `visitor.h`: defines visitor interface for general traversal of AST nodes
  - `CFGBuilder`: Build a CFG based on AST. It is a shell of visiting the AST nodes, and internally, it uses `utils/Graph.cc` for building the graph.
  - `Generator`: generate C source code from AST
  - `GrammarPatcher`: take an AST and a selection of nodes, this class traverse the AST and decide what are the other necessary nodes for valid partial program.
  - `Printer`: print out AST
- `IncludeManager`: parse and maintain the header file dependence used in original benchmark. This information is used to decide, for the generated partial program, what system header files to include, in what order.

- `SourceManager`: this class manage all the ASTs of all source files inside the benchmark. It defines the entry procedures for syntactic patch, def-use analysis, selection generation and loading, main program (`main.c`) and Makefile generation, and header file (`main.h`, containing code snippets) generation.

`type` folder contains type related code:

- `Cache.cpp`: defines the preprocessing of source file.
- `IOHelper`: instrumentation for test input
- `Snippet.cpp`: define class of snippets: `FunctionSnippet` `TypedefSnippet`, `VarSnippet`, `RecordSnippet`, `EnumSnippet`.
- `SnippetAction.cpp`: this class is based on clang for extracting code snippets inside benchmark. `clang_parse_file_for_snippets` is the entry function, which accepts a source file path, and return a vector of snippets.
- `SnippetManager.cpp`: the storage of snippets, can dump or load snippets into a file.

`utils` folder contains utilities functions and classes. Most of the files are intuitive for their purpose.

- `HeliumOptions.cc`: This file is invoked by main function for parsing all command line flags. It also deals with parsing of configuration files.

4.3 Helium2

The first part if a Clang plugin to parse and dump parse tree into a file, during invoking Clang on a source file. The related source files are:

- `plugin.cpp`, `plugin.h`: hook into clang compiler
- `sexp.cpp`, `sexp.h`: main work horse for parsing different kinds of AST nodes.

The second part is the main Helium logic.

- `ast.rkt`: This file defines the AST types and structure, and helpful functions, e.g. `children`, `get-callee`, `get-declared-var`, `if-leaf?`
- `gen.rkt`: This file deals with generate C source files from an AST. It is a simple recursive algorithm to traverse the tree, and generate string accordingly.
- `helium.rkt`: command line interface
- `main.rkt`: the working logic of Helium, including generate `main.c`, generate `Makefile`, invoke `make` to compile the generate code, high level schedule for generate random selection or load from selection file.

- `patch.rkt`: This file implements syntactic patching algorithm. The `syntactic-patch` function invokes `patch-lca` (algorithm step 1) and `patch-min` (algorithm step 2) to accomplish the algorithm.

5 Benchmarks

The benchmark used in paper are:

- linux-4.15: `https://github.com/torvalds/linux/releases/tag/v4.15`
- FFmpeg-n3.4.2: `https://github.com/FFmpeg/FFmpeg/releases/tag/n3.4.2`
- vim-8.0.1567: `https://github.com/vim/vim/releases/tag/v8.0.1567`
- git-2.16.0: `https://github.com/git/git/releases/tag/v2.16.0`
- openssl-1.0.2n: `https://github.com/openssl/openssl/releases/tag/OpenSSL_1_0_2n`

You can download these benchmarks using links above.
The script benchmark.py is used to query and download github projects.