Metavis: Visual debugger for MetaDL

Emil Blennow
E19, Lund University, Sweden
em8055bl-s@student.lu.se

Abstract
This paper introduces a tool for visualizing static bug reports generated by MetaDL. By utilizing a terminal UI the analysis results from MetaDL become easier to understand, debug and make use of. Taking inspiration from other terminal based UI the tool aims to be easy to learn and quickly usable.

1 Introduction

MetaDL [2] is a collection of tools for writing declarative static bug checkers. The tool mainly targets Java, C and Datalog programming languages and the analyses are written in the MetaDL language which is an extension of Datalog.

MetaDL outputs the analysis result in a combination of comma separated values (CSV) or JavaScript object notation (JSON) files. These are great for parsing and analyzing for computers but not as good for humans. An example of the current outputs can be seen in Listing 1, Listing 2 and Listing 3.

```
1,png.c,438,12,438,12
2,png.c,438,7,438,20
3,pngmem.c,252,9,252,9
4,pngmem.c,252,4,252,12
5,png.c,427,4,427,27
6,pngmem.c,246,47,246,57
36,/usr/include/x86_64-linux-gnu/bits/uintn-
identity.h,35,10,35,10
...
```

Listing 1. `Debug_Loc.csv`

```
[
  {
    "name": "Assign",
    "file": "Assign.csv",
    "locs": [ 0, 1, 2, 3, 4 ],
    "locFile": "DEBUG_Loc.csv"
  },
  ...
]
```

Listing 2. `debug.json`

```
6373,18214,6373,17055,6373
6378,18319,6378,18239,6378
6414,6412,7675,7676,7677
6544,18218,6544,17215,6544
```

Listing 3. `Assign.csv`

To the trained eye the output files might be readable and understandable but not without effort and when having relations referring to multiple files it becomes very hard to keep track of everything.

The process for understanding the files in Listings 1-3 is to start at the `debug.json` file and find what type of relation you are interested in. Then you navigate to the file, in this case `Assign.csv` make note of the nodes on a line and then navigate to `Debug_Loc.csv`. From here you find the corresponding nodes (first column) and their file and the location in code the AST node represents where the fields in `Debug_Loc.csv` are start row, start column, end row, and end column.

According to a study conducted by Glen Wylie and Alan Allport [5], humans have a harder time switching between different tasks than just continue working on the same. In their study, test subjects were subjected to words and colors separately and asked to name what they saw. When switching between stimuli, subjects where on average slower to react and name the correct thing they saw compared to when being subjected to the same type of stimuli.

This effect can also be observed when trying to manually parse these relations since they are number based and need to be translated to source code locations. One might even say the overhead is higher since there is more work and context required to link the information together.

2 Introducing `metavis`

Metavis tries to alleviate the confusion of these files by introducing a terminal user interface (TUI) to visualize the analysis results with the source code as context. The tool was designed to be familiar to how a regular editing environment while following the conventions of similar terminal based UIs such as GitUI (https://github.com/extrawurst/gitui) and lazydocker (https://github.com/jesseduffield/lazydocker). The primary use case of this tool is to help debugging and verifying that analyses point towards the correct nodes and that they are located correctly within the source code. Though

Course paper, EDAN70, Lund University, Sweden
January 9, 2024.

this might be the main use case, the program can also be used to consume the final analyses and aid debugging and/or fixing the source code being analyzed. A screenshot of the program can be found in Figure 1.

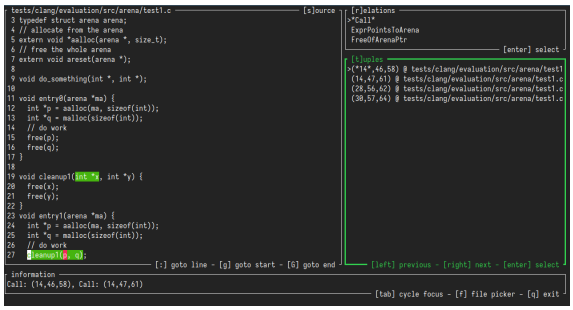

Figure 1. Metavis program

The following parts of this paper will detail some main parts of the program, how they are implemented and the program can be used.

3 Implementation

First and foremost the output from MetaDL must be parsed, and all necessary connections must be made. This report will use an example project provided in the source repository at GitHub (https://github.com/blennster/metavis).

3.1 MetaDL output

MetaDL outputs relations in CSV format where each row in the CSV file represent a tuple as seen in Listing 3. Relations of MetaDL are typed and each column in a CSV file can either be an integer, a string or an AST node.

To differentiate which fields contains AST nodes and integers, MetaDL outputs the `debug.json` (Listing 2) file. The `locs` field defines which columns should be parsed as AST nodes and not integers and the corresponding location can be matched in the file defined in `locsFile`.

The `DEBUG_Locs.csv` contains the id of the AST node, the source file and the location within that file (start row, start column, end row and end column).

3.2 Parsing the output

Metavis starts at the `debug.json` file and parses all the files it references, taking care to de-duplicate when parsing the location file and source file since they may be referenced multiple times.

The results are then combined into a struct named `MetaInfo` which contains all the raw analysis information and methods for extracting information for individual files or relations.

3.3 Terminal UI

A TUI is an application which uses simple text only to render a user interface. These have many benefits compared to regular graphical user interfaces in that they can be very fast and efficient. Since most terminals are already highly optimized, rendering the UI is done fast and efficiently. It is also easier to work with layouts since the smallest unit is a character and since this is a text heavy interface no smaller unit is needed. There is a minimal amount of clutter and the design is streamlined to its simplest parts since the UI is only text based.

3.4 Ratatui and immediate mode rendering

Ratatui (https://ratatui.rs) is the rust library used to render the terminal UI. It uses immediate mode rendering for the UI. It is a special render paradigm where you redraw the UI on every frame. This means that rendering can be delayed until something new needs to be rendered which leads to more efficient code. It is also very simple to get stared since you just need to set up a loop and issue render calls. The drawbacks are that the render loop and event loop needs to be set up by a programmer which can be a lot of work and there is no framework to help developers keep a sane and flexible structure when building larger applications. [1]

3.5 Highlighting

For every line inside a file there can be one or many nodes, and they may be nested within each other as seen i Listing 4. Highlighting may also span several lines and within these lines highlighting may also be nested.

```
int void main(int argc, char *argv[]) {
    return 0;
}
```

Listing 4. An example how highlighting may look

The Ratatui library only allows setting the style of text before rendering it and not after. Therefore, every line is checked if it contains any highlight and if it does, every character will be checked if it is part of a highlight. Nesting is checked by comparing the number of highlights a position matches and coloring is based on this number. Listing 5 details a pseudo algorithm for how the highlighter works where the `n_highlights()` method works in a similar way to the algorithm detailed in Listing 6.

```
for line in file
  let line_to_render = ""
  if has_highlights(line)
    // Accumulator
    let acc = ""
    let prev_n = 0
    for (char, index) in line
      let n = n_highlights(index)
      if n != previous_n
        // Map an integer to a color
        let color = get_color(previous_n)
        // Get a styled span from string
        let span = new_span(acc, color)
        line_to_render.append(span)
        acc = ""
        prevous_n = n
      acc.append(char)

    // Get the last of the accumulator
    let color = get_color(previous_n)
    let span = new_span(acc, color)
    line_to_render.append(span)
  else
    line_to_render = line

  render(line_to_render)
```

Listing 5. Highlighter pseudo code

The algorithm in Listing 5 produces a result which can be seen in Figure 2.

```
19 void cleanup1(int *x, int *y) {
20   free(x);
21   free(y);
22 }
23 void entry1(arena *ma) {
24   int *p = aalloc(ma, sizeof(int));
25   int *q = malloc(sizeof(int));
26   // do work
27   cleanup1(p, q);
28 }
```

Figure 2.

3.6 The different panes

As seen in Figure 1, the program is divided into four main panes. These can be selected using the shortcut in the brackets within pane title. Switching can also be done using `<Tab>` and `<Shift-Tab>` also known as back tab. The *file selector* can be summoned by pressing `f`. The currently focused pane is highlighted with a thicker green.

3.6.1 Relations

The *relations* pane represents the different analysis items present within a project. These contain one or many rows of one or many *tuples*. This pane helps with usability for large projects since having all tuples in one list. This list can be very long and not very friendly to navigate but grouping by relations alleviates this.

3.6.2 Tuples

Tuples are the main driving force of the program. They represent one or more nodes that contribute to a named analysis. Navigation within this pane causes the source view to jump to the selected node within a list. Nodes within one row can be navigated by using the `left`- or `right` arrows and since analyses can refer to nodes across files the source view will load the correct file and jump to the location of the node. To not surprise the user as much when a different file is loaded, the list displays what files are included within one diagnostic as seen in Figure 3

```
[t]uples
(2,5) @ png.c
(4,6) @ pngmem.c
(992,5) @ png.c
(993,5) @ /work/projects/llvm-project/build-release/l
(994,5) @ png.c
(7429,5) @ png.c
(7430,5) @ png.c
(20427,5) @ png.c
(20428,5) @ /work/projects/llvm-project/build-release
(20429,5) @ /work/projects/llvm-project/build-release
>(*20430*,5) @ ./pngpriv.h & png.c
[left] previous - [right] next - [enter] select
```

Figure 3. Tuple spanning multiple files

3.6.3 Source

The source view displays the analyzed source code with the selected nodes rendered as highlights. This view can be navigated in a similar way to how most other text editors work. It will preserve column selection when navigating up and down and will highlight items within the diagnostic view when the cursor is placed on a node contained within the item.

Evaluating which nodes are under the cursor is simply done with a containment check as filter detailed in Listing 6. The containment check is also used in Listing 5 at the `n_highlights()` function call.

```
let nodes
let c_col = cursor.location().x
let c_row = cursor.location().y
for diagnostic in diagnostics
  let loc = diagnostic.location
  let in_range = false
  if (loc.start_line <= c_row &&
      c_row <= loc.end_line)
    if loc.start_line == loc.end_line
      in_range = loc.start_col <= c_col &&
          c_col <= loc.end_col
    else if loc.start_line == c_row
      in_range = loc.start_col <= c_col
    else if loc.end_line == c_row
      in_range = c_col <= loc.end_col
    else
      in_range = true

  if in_range
    nodes.append(diagnostic.node_id)
```

Listing 6. Range filtering for nodes under cursor

3.6.4 Information

The information view displays all the analysis contributions for the current cursor position within the code.

3.6.5 Files pop-up

For easy jumping between files, a file picker was added. This can be used to easily jump to an interesting file within the project without having to find a diagnostic pointing towards that file. An example of this can be seen in Figure 4.

Figure 4. File picker

3.6.6 Goto line

To easily navigate to a specific line, the user can input : and a pop-up will show where the user enters a number and the cursor will jump to that line within the current loaded file. This binding is only active when the user has the *source* view focused.

4 Evaluation

Usability was evaluated with one usability target: all features within the program should be discoverable and usable without the user having to ask for help or reading a manual. Additionally, there should not be any unexpected behaviors and the program also implement most of the expected behaviors based on users previous knowledge.

I tested the usability by placing a computer with the program running in front of users and asking them to use the program. Some background about what the overall goal of the program was described beforehand.

The user was then observed and to see that all features was used and some simple informal questions where asked such as *"what did you think of the program?"* and *"was there anything that felt unnatural/unexpected?"*.

This testing is based on the methods described by X. Ferre et al. [3] and was evaluated continuously during development.

Three users where part of this testing, an expert user, an programmer with some TUI experience and a non programmer.

The expert user was part of the design of the program user interface and guided what felt usable and not. The main complaint was that some parts of the program behaved unexpectedly but these complaints where resolved.

The programmer that used the program got up to speed easily since he had some experience with similar programs and keyboard navigation in programs. He found that some expected keyboard bindings such as backtab were missing but these were later implemented.

When the non programmer who did not have any TUI experience was asked to test the program and felt that the lack of mouse support was a limitation for their ability to use the program and did not really understand why the interface was text based.

Performance is good with no massive spikes of CPU usage, low memory usage and fast rendering. Input response is instant and there is no loading delays with one caveat that when compiling in debug mode there can be slowdowns in some cases but these are not present in release mode. Robustness was improved by handling missing source files instead of crashing.

5 Related work

Terminal based UIs are still very common with development tools since many other tools utilize a CLI and a TUI provides a more intuitive way to interact with these tools. TUIs are also great for portability since basically every machine has a command line interface and therefore can render the UI without external dependencies. Some other examples of TUIs are htop, vim, lazygit and the GDB TUI mode.

The TUI mode found in GDB is the most similar to this work. It offers a syntax highlighted source view, registers assembly and a command window. It also fea-

tures mouse scrolling in the views for easier navigation.

An often used tool at LTH when working with compilers and program analysis is CodeProber [4]. It can be used similarly as this tool by analyzing code but is restricted to Java implementations. It also focuses on a different type of analysis where it is often the compiler being analyzed but MetaDL and Metavis is more focused on the Datalog program that runs the analysis and the source code.

Metavis follows some ideas from Codechecker https://codechecker.readthedocs.io/en/latest/ but on a smaller scale. Codechecker reads analyses from many static analysis tools such as Cppcheck, ESLint and golint and displays these in a web GUI with the respective code. Codechecker is used for displaying the final output of analyzers while Metavis focuses on the intermediate steps also to help debug the analyses itself.

6 Conclusion and future work

The current implementation is functioning well but there is always room for improvements.

Some possible improvements are:

- Mouse support
- Implement a fuzzy finder to search and select files
- Easier selection of a specific node
- Improve the source view with better scroll into view behavior and syntax highlighting
- Allow searching text in files
- Scrollbars to indicate scrollable views
- Improve the performance when selecting relations
- Allow different directories for the source files and analysis outputs

7 Acknowledgments

I would like to thank Alexandru Dura for continuously evaluating and guiding development of this tool and my classmates for usability testing and especially Axel Froborg for excellent feedback.

References

[1] Immediate mode rendering in ratatui. Retrieved December 16, 2023 from https://web.archive.org/web/20231216235836/https://ratatui.rs/concepts/rendering/

[2] Alexandru Dura, Hampus Balldin, and Christoph Reichenbach. 2019. MetaDL: Analysing Datalog in Datalog. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP 2019)*, 2019. Association for Computing Machinery, Phoenix, AZ, USA, 38. https://doi.org/10.1145/3315568.3329970

[3] X. Ferre, N. Juristo, H. Windl, and L. Constantine. 2001. Usability basics for software developers. *IEEE Software* 18, 1 (2001), 22–29. https://doi.org/10.1109/52.903160

[4] Anton Risberg Alaküla, Görel Hedin, Niklas Fors, and Adrian Pop. 2022. Property Probes: Source Code Based Exploration of Program Analysis Results. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2022)*, 2022. Association for Computing Machinery, Auckland, New Zealand, 148. https://doi.org/10.1145/3567512.3567525

[5] Glenn Wylie and Alan Allport. 2000. Task switching and the measurement of 'switch costs'. *Psychological Research* 63, 3 (August 2000), 212–233. https://doi.org/10.1007/s004269900003