fAST: Flattening Abstract Syntax Trees for Efficiency

Yijun Yu
School of Computing and Communications, The Open University
y.yu@open.ac.uk

***Abstract*—Frequently source code analysis tools need to exchange internal representations of abstract syntax trees (AST) with each other. Conveniently, and intuitively, the externalised representations are in the form of hierarchical trees. We argue, counter-intuitively, that hierarchical representation is not the most efficient way for source analysis tools to exchange parsed AST. In this work, we propose to speed up AST parsing whilst preserving the equivalence of hierarchies in binary forms: (1) AST could be saved as a *flat* one-dimensional array where pointers to tree nodes are converted into integer offsets, and (2) such *flattened* AST are more efficient to access by programming tools through the generated application programming interfaces (API). In programming language-agnostic evaluations, we show that parsing flattened AST becomes 100x faster than in textual form AST on a benchmark of open-source projects of 6 different programming languages.**

I. Introduction

Central to SE activities, automated tools need to parse the source code as intermediate representations (IR) and store them *persistently* to avoid parsing them again. However, none of existing persisted IR achieve three attributes together:

- **Expressive:** the IR shall express (recursive) source code structures of any mainstream programming language;
- **Efficient:** the IR shall be loaded into memory quickly;
- **Interoperable:** the IR shall be exchanged through common API.

In this work, we consider more efficient IR in binary forms because they are closer to machine, and we aim to encode hierarchically recursive structures into a form interoperable with other programming tools. The proposed binary IR is called "fAST" using for example protobuf [1] or flatbuffers [2],which are *expressive* enough to flatten the AST in order to load them back into memory without resorting to further parsing. As demonstrated later, such flattened AST are *efficient* to access by programming tools through the generated API.

II. Our Approach

Tools such as `srcML` can serialise AST into semi-structured data. To process semi-structured data efficiently, two open-source projects have been adopted: Protocol buffers (`protobuf`, or PB) [1] treats the data as a stream of messages according to a predefined protocol specification, and generates the API in different programming languages to integrate them with custom tools. Flattened Buffers (`flatbuffers`, or FBS) follows the standard of interface definition language (IDL) to specify data structures as a variable length one-dimensional array, where cross-referencing pointers to structured fields are encoded as integer offsets to the elements on the flattened array.

PB or FBS were designed as efficient substitution to XML's. Since AST in IR can be represented in XML (e.g., SrcML), it is reasonable to express them in PB. However, existing protobuf message protocols define records rather than trees. Therefore, we need to create a recursive schema by engineering the PB specification based on the ANTLR4 grammar production rules used by `srcML`.

Our approach supports any ANTLR4 grammar of over 170 different types of programming languages. Using generative compilers, our specification is translated into API to save or to load data from/into the corresponding IR. One advantage of reusing `srcML` is that the binary structures do not need to be at the root level of compilation units. A method, a statement, or even just an expression, could be the root element, even if the source is not syntactically correct. This flexibility also facilitates our case studies of code commits in `Git` repositories.

The approach has been applied to many program analysis tasks such as program slicing, tree-based diffs, clone detection, bug localisation, cross-language algorith classification, etc.

The 'fAST' tool has been provided as a precompiled binary in a docker image, which can be installed and used as follows:

```
1 docker pull yijun/fast:latest
2 docker run -e $(pwd):/e yijun/fast foo.cc foo.pb
3 docker run -e $(pwd):/e yijun/fast bar.java bar.fbs
4 docker run -e $(pwd):/e yijun/fast moo.fbs moo.cs
5 docker run yijun/fast
```

In the example, Line 1 installs the latest docker image from the dockerhub, Line 2 runs the command to convert the AST of a C++ program to PB, Line 3 runs the command to convert the AST of a Java program to FBS, Line 4 converts an FBS fAST back to C# code, and Line 5 shows all command line options.

Furthermore, these commands can be integrated through system calls to an IDE. For example, an extension for Visual Studio Code has been created to demonstrate the use of fAST for various applications:

`https://github.com/yijunyu/vscode-fast`

In this way, 'fAST' can be packaged together with programming tools in any programming language.

III. Performance Evaluation

For validation, we collected the most popular open source projects on the `GitHub`, one for each of the five programming language supported by `srcML` [3] (i.e., Java, C, C++, C#, Objective-C), as part of a benchmark for evaluating the efficiency of fAST. To assess the expressiveness of the proposed solution which could support any programming language defined by ANTLR4, we extend fAST to support `Smali` assembler for analysing Android apps, and evaluated its efficiency on the reverse engineered code of publicly available binary of the `Instagram` app.

Using `srcML` is our baseline, representative benchmarks include any number of C, C++, C#, Objective C, and Java projects. To evaluate parsing performance, from GitHub we chose the most starred projects in each language category. Additionally, we used a project to evaluate the extensibility of fAST to a new programming language `smali` and chose a reverse engineered Android app. In total 6 projects have been selected for evaluating the performance of different code analysis tasks.

For *interoperability*, one could generate both PB and FBS API in multiple programming languages. In this experiments, we chose the C++ API in order to provide a like-to-like performance comparison to the baseline tool `srcML` which was also implemented in C++. All experiments were carried out on a Macbook Air of Early 2015 build.

The first 5 programs in the benchmark have shown a significant speed up of parsing by using SrcML parser, which varies from 2x (AFNetworking) to 5x (linux). This observation suggests that `srcML` implementation has already considered efficiency in its design.

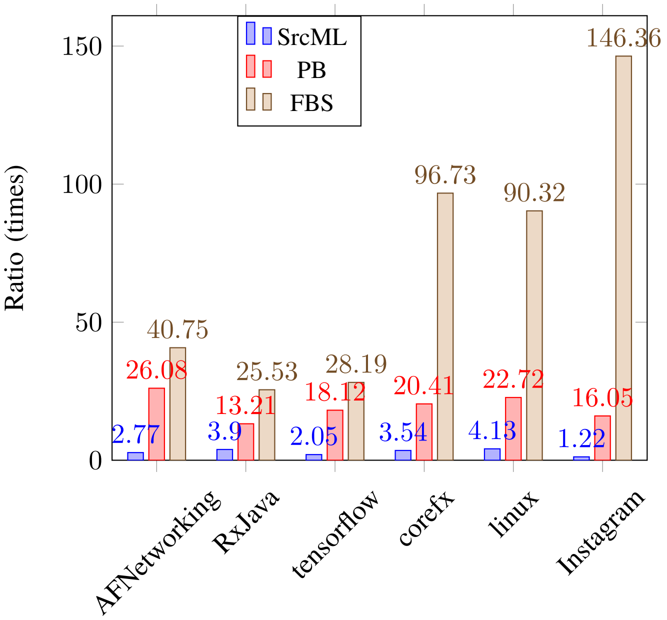

Fig. 1. Speed up parsing by loading respectively SrcML, PB, and FBS instead

After flattening all code structures into a stream of protocol messages in binary form, Figure 1 reveals that using PB, parsing speeded up by at least 13x, which is much more efficient than parsing SrcML. Storing fAST as FBS further, Figure 1 highlights that the speed ups outperform PB, registering at least 22x faster for all the projects. It is worth noting that Instagram shows a high speed up ratio to 150x.

IV. Conclusions and Future Work

As source code grows [4], it is essential to have their IR efficient, expressive, and interoperable for further analysis. To achieve these, we have proposed expressive binary IR and compared the performance of parsing the textual counterparts. The interoperability of our alternatives has also been examined, using 6 open-source projects in 6 different programming languages. One of them is non-open source project[1], to assess how effective it is to integrate the enhanced parsing process. According to these experiments, parsing is at least 13x faster using PB, or 22x using FBS. These demonstrate the efficiency benefits of flattening AST. The toolkit can already support the applications in cross-language algorithm classification using deep learning [5], bug localisation [6], meaningful changes detection [7].

Flattening AST is a general approach. Since natural language documents also have hierarchical structures similar to parsing trees, it is our future work to extend the approach to support natural language intensitve parsing tasks, which may be useful in NLP tasks such as security bug classification [8].

References

[1] Google, “Protocol Buffers, Google’s data interchange format,” 2017, https://github.com/google/protobuf.

[2] ——, “Flat Buffers, a cross-platform memory efficient serialization library,” 2017, http://google.github.io/flatbuffers/.

[3] M. L. Collard, M. J. Decker, and J. I. Maletic, “Lightweight transformation and fact extraction with the srcML toolkit,” in *11th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM 2011, Williamsburg, VA, USA, September 25-26, 2011*. IEEE Computer Society, 2011, pp. 173–184.

[4] R. Potvin and J. Levenberg, “Why Google stores billions of lines of code in a single repository,” *Commun. ACM*, vol. 59, no. 7, pp. 78–87, Jun. 2016. [Online]. Available: http://doi.acm.org/10.1145/2854146

[5] N. D. Q. Bui, L. Jiang, and Y. Yu, “Cross-language learning for program classification using bilateral tree-based convolutional neural networks,” in *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018.*, 2018, pp. 758–761. [Online]. Available: https://aaai.org/ocs/index.php/WS/AAAIW18/paper/view/17338

[6] T. Dilshener, M. Wermelinger, and Y. Yu, “Locating bugs without looking back,” *Autom. Softw. Eng.*, vol. 25, no. 3, pp. 383–434, 2018. [Online]. Available: https://doi.org/10.1007/s10515-017-0226-1

[7] Y. Yu, T. T. Tun, and B. Nuseibeh, “Specifying and detecting meaningful changes in programs,” in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, 2011, pp. 273–282. [Online]. Available: http://dx.doi.org/10.1109/ASE.2011.6100063

[8] F. Peters, T. Tun, Y. Yu, and B. Nuseibeh, “Text filtering and ranking for security bug report prediction,” *IEEE Transactions on Software Engineering*, p. 1. [Online]. Available: doi.ieeecomputersociety.org/10.1109/TSE.2017.2787653

[1] These 6 programming languages include C, C++, Java, C#, Objective C, and Smali.