Projection Boxes in CodeProber

Vera Paulsson
D20, Lund University, Sweden
ve2675pa-s@student.lu.se

Klara Tjernström
D20, Lund University, Sweden
kl4257tj-s@student.lu.se

Abstract

We introduce projection boxes, a live programming feature, into the client-server application CodeProber. CodeProber provides its user with live feedback to visualize and investigate the AST created from the analysis tool. Examples of CodeProber's use cases are the compiler and program analysis courses at LTH. Our addition of projection boxes extends CodeProber's liveness by displaying runtime values inline with the code, eliminating workflow interruptions caused by existing hover-based interactions. The projection boxes are evaluated with a user study, with the aim to explore what impact the feature has on a programmer's development process. Results indicate that projection boxes enhance usability and effectiveness while maintaining minimal performance overhead.

1 Introduction

Developing tools for programming languages, such as compilers and static analyzers can be a challenging task. To help developers with this task, several tools and techniques have been developed, such as CodeProber. CodeProber is a tool which provides developers a way to explore their analysis results, helping bridging the gap between complex implementations and their underlying behavior. [15] It achieves this through several live features, such as property probes which acts as live observers of properties of Abstract Syntax Tree (AST) nodes.

Liveness, as defined by Tanimoto, is "the ability to modify a running program" [16]. In addition, the programming environment provides the programmer with immediate information on what they are changing. In other words, it always presents up-to-date values when the user makes changes. For example, the values in the CodeProber probes are automatically updated whenever the user makes edits in the editor or rebuilds the underlying compiler or static analysis tool.

Kramer et al. [9] performed an experiment in which they compared the same development environment with and without feedback. This experiment gave insight in the positive effects live programming has from a debugging aspect. The result showed that programmers in the experimental environment more often switched between developing and debugging, reducing the time spent fixing bugs.

A study of students using CodeProber revealed that liveness is one of the tool's most appreciated aspects [14]. In this paper, we aim to further improve CodeProber's liveness by extending it with Projection Boxes. Projection Boxes, as presented by Lerner [10], are a novel visualization technique which displays runtime values in a program. Lerner's paper includes a formative study where projection boxes were implemented in a tool called VERSABOX. It was demonstrated that users found projection boxes and their configurability useful for example for finding and fixing mistakes right when they were introduced and guiding the code writing. It was also shown to be useful for testing the entire code after it was written to check if it would output the right results. Furthermore, they found that a programmer's experience with a programming language affected how much immediate feedback the person would want. More experienced programmers would in general want less information.

When presenting live feedback it is important to take interface design into consideration. Lerner's study [10] shows that the previous programming experiences, together with language knowledge creates various conceptual models which affect how visualisation should be implemented. Another aspect covers the amount of information shown in the boxes. Some participants in the study wanted more information to be presented when they entered a debugging phase while other participants always wanted a large amount of information. There were also participants that wanted the opportunity to hide the boxes entirely. This creates a difficulty when presenting projection boxes. They should be presented in such a way that it helps the programmer rather than being a distraction.

In CodeProber, developers can already extract the information shown in the projection boxes by using the squiggly lines feature. By hovering over a variable, users can view relevant runtime data. This approach interrupts the developer's workflow as it requires the use of a mouse or touchpad to access the information. Projection boxes address this by displaying the information directly on the same line as the variable, eliminating the need for mouse interactions. The data remains visible at all times, creating a more seamless and immediate debugging experience.

One use case of CodeProber is the Program Analysis course at LTH. In the course, students implement analyses on a JastAdd-based compiler for TEAL, a gradually typed imperative language, following provided compilers and lab instructions. Students in the Program Analysis course would benefit from projection boxes to visualize computed analysis

Course paper, EDAN70, Lund University, Sweden
January 15, 2025.

results directly [14]. The projection boxes would simplify the comparison between expected and actual values. Another use case of CodeProber is the Compiler course at LTH, which will be further discussed in the Use Case section of this report.

Finally, we believe that the feature could be useful beyond academia, for example in industry. By integrating projection boxes, we aim to make CodeProber a more comprehensive tool for programming language developers.

2 Background

2.1 Attribute Grammar

Knuth introduced attribute grammars (AGs) as a technique for defining semantics of programming languages [8]. The formal structure of a language can be defined by a context-free grammar, which the attribute grammar builds on. Attribute grammars specify the 'meaning' of languages by storing semantic information in attributes associated with terminal and non-terminal symbols of the grammar.

The intrinsic meaning in the code is unveiled during the parsing of a program, which determines the structure of the parse tree by describing the parent-child relationships between nodes or different program elements. These intrinsic relationships provide the foundation for deriving non-trivial values with attributes.

Knuth defined two types of attributes in his paper: synthesized and inherited. Synthesized attributes compute information based on a given tree node and its descendants. In contrast, inherited attributes are defined by the ancestors of a given node. The inherited attribute can for example be a symbol table which will associate variable names with their types and by propagating this information down in the tree, it can be used for checking type correctness of expressions.

2.2 Reference Attribute Grammars

Reference Attribute Grammars (RAGs) is an extended type of canonical attribute grammars. RAGs is non-circular and it has the addition of references in its attributes. References make it possible for nodes to create direct links between each other as well as access its properties.

By making use of references in the grammar, other graph structures can be built on top of the AST. This makes is possible for information to traverse these edges instead of the original edges of the AST. The benefits of this is the possibility to pass unwanted nodes or attributes and access certain nodes and attributes directly [7].

When RAGs are used in JastAdd, it is specified in modules called Jrag. Jrag supports a modified Java version and these modules includes class declarations which consists of equations and attributes. Attributes are specified with a keyword "syn", for synthesized, or "inh", for inherited, to specify where in the AST the execution is to be done.

2.3 JastAdd: *Just add to the ast*

JastAdd is a Java based system with support for RAGs. This is done by a combination of imperative and declarative structures. The combination of these makes it possible to divide and work with smaller problems within the compiler [6].

JastAdd uses different modules to weave together the compiler. These modules consists of the abstract grammar as well as behavioural aspects. These parts are combined in jastadd to generate AST classes written in ordinary Java. When these files and corresponding classes are generated, a tree hierarchy is created. The AST classes also includes methods which makes the tree traversable [1].

JastAdd, and RAGs in general, uses lazy dynamic evaluation which means that dependencies are evaluated on demand. Rewrites of AST nodes are only performed when the traversal API is used to access the node. The benefits of this is that there are only a cost if the attribute are accessed, which is a efficient way of evaluation. [7]

The structure of JastAdd and its Java based system makes it possible to "just add to the ast" [7].

2.4 ExtendJ

ExtendJ, "The JastAdd Extensible Java Compiler"[2], is a Java based compiler that works equal to a normal Java compiler. ExtendJ is built with JastAdd which makes it possible to easily extend the compiler into static analysis tools as well as extending the Java language constructs[1].

Ekman and Hedin added support for Java 1.4 and Java 5 for ExtendJ [1]. They also performed tests on the compiler with the conclusion that the structure of JastAdd and the system of only using the AST as data structure has shown to be a workable way of handling compiler structures.

Öqvist added support for Java 7 to ExtendJ[12]. Öqvist compared the extension of Java 7 to javac and ExtendJ with the conclusion that ExtendJ was easier to extend. This was compared through added amount of source code, source line of code (SLOC).

3 Projection Boxes in Codeprober

The addition of projection boxes would make CodeProber more comprehensive as it introduces a new way to present diagnostic information. These boxes visualize data such as execution traces, offering a side-by-side format alongside the corresponding lines of code. The feature enhances clarity and usability by complementing existing diagnostics such as squiggly lines.

3.1 CodeProber Architecture and Protocol

CodeProber is a web-based client-server application designed to facilitate live feedback for code analysis and debugging. It is implemented using Java and TypeScript and comprises two main components: the client-side interface and the server-side backend. The client side, built on the Monaco editor,

provides users with an interactive environment to explore code, visualize runtime values, and investigate diagnostics such as variable states or errors. The server side, integrated with analysis tools like ExtendJ, handles requests from the client to deliver live feedback and computations.

Diagnostics in CodeProber are computed on the server side, where the analysis tool parses the edited text into an Abstract Syntax Tree (AST) and populates it with functionalities that can be explored using probes.

Probes act as live observers, dynamically evaluating and displaying properties of specific AST nodes. For example, a probe might show the type of an expression or the runtime value of a variable, presenting this data directly within the editor.

To expand the existing diagnostic infrastructure, projection boxes were introduced as a new diagnostic type. A projection box displays information side-by-side with the corresponding line of code.

3.2 Implementation and Design Considerations

Projection boxes use Monaco's `deltaDecorations` API to dynamically display relevant information. When a user modifies code, the client sends an updated representation of the code to the server. The server processes the changes, computes the new state of the AST, and generates diagnostic data. This data is then rendered as projection boxes.

On the client side, projection boxes are styled using CSS to ensure clarity and avoid overwhelming the user. Neutral colors such as gray and white were chosen to maintain simplicity and formality, avoiding unintended emotional associations linked to diagnostic colors like red or yellow. Placement on the right-hand side of the code editor follows established design principles, such as the Golden Ratio theorem, to present additional information in a way that minimizes cognitive load and distractions [4].

While projection boxes could offer always-on feedback, such an approach risks cluttering the interface and detracting from the user experience. By using the existing toggling option in CodeProber, the user can choose to enable or disable projection boxes as needed, adapting the tool to their workflow.

3.3 Use Cases

Two of the use cases covers different courses given at LTH. One of them are the program analysis course and the other is the compiler course. These two courses presents CodeProber as a tool to be used during the lab assignments. Depending on the goal of the assignment, CodeProber presents different attributes or data to be presented but has not before presented live feedback in the format of projection boxes.

During the program analysis course, the goal is to improve the understanding of the language as well as being able to analyse the executed value of the code in comparison to the expected.

In the compiler course, the goal is to be able to follow the interpretation steps when SimpliC is developed. Figure 1 illustrates how CodeProber appears without projection boxes, while Figure 2 demonstrates its appearance with projection boxes.

Figure 1. Example of how CodeProber looks without projection boxes

```
1   int main() {
2
3     int n = 0;
4     int i = 1;
5
6     while (i < 4){
7       if (n <= 5) {
8         n = n + 1;
9       }
10      i = i + 1;
11    }
12    return 0;
13  }
```

Figure 2. Example of how CodeProber would look with projection boxes in the compiler course

```
1   int main() {
2
3     int n = 0;    0
4     int i = 1;    1
5
6     while (i < 4){
7       if (n <= 5) {
8         n = n + 1;   1  2  3
9       }
10      i = i + 1;   2  3  4
11    }
12    return 0;
13  }
```

Another use case apart from the academic studies presented above are the industry. The development of new compilers is an ongoing process and the implementation process can be complex and difficult. By using CodeProber and projection boxes, several steps of development could be easily improved due to the opportunity of live feedback and data presentation. The version of projection boxes developed in this paper is primarily designed to address after the needs of students. It remains future work to identify more concrete examples of industry use cases where projection boxes could help, and whether our design addresses these cases.

4 Evaluation

4.1 Introduction

This section evaluates the impact of projection boxes on usability and performance. The evaluation combines performance benchmarks, analysis of required lines of code, and qualitative feedback from students who used CodeProber during the 2023 and 2024 compiler courses.

4.2 Method

The evaluation was conducted in two stages: a pre-questionnaire followed by hands-on sessions where participants actively engaged with CodeProber and its projection box feature. During these sessions, participants were tasked with creating a program and debugging deliberately introduced errors, enabling an assessment of how projection boxes support various tasks such as interpreting program behavior and identifying issues.

Participants for the study were drawn from students who had taken the compiler course in 2023 or 2024. The pre-questionnaire focused on understanding participants' prior usage of CodeProber during laboratory assignments. Specific attention was paid to their experience with Lab 4, which involved implementing name analysis for the SimpliC compiler, and Lab 5, which focused on developing an interpreter.

The pre-questionnaire included two key questions:

1. How much did you use CodeProber in Lab 4 (name analysis) in the compiler course? (1 - Not at all, 5 - A lot)
2. How much did you use CodeProber in Lab 5 (interpreter)? (1 - Not at all, 5 - A lot)

Performance metrics were measured by comparing the time required to complete specific tasks with projection boxes enabled versus disabled to complete 1,000 iterations. Additionally, the lines of code required for projection box implementation were analyzed, with distinctions made between unique and boilerplate code.

4.3 Results

The pre-questionnaire revealed varying levels of CodeProber usage among participants, with responses indicating moderate to high reliance on the tool during labs. Notably, the average usage of CodeProber was higher in Lab 4 (4.33) compared to Lab 5 (3.33). These findings established a baseline for assessing projection boxes' impact during the hands-on sessions.

Participants valued the projection boxes' ability to provide immediate feedback on variable values and program behavior, particularly in illustrating how values were evaluated across different namespaces. This functionality was highlighted as especially useful in debugging custom language implementations.

However, several areas for improvement were identified. Users suggested increasing the spacing between code and boxes for improved readability and requested more detailed runtime error explanations. Additionally, concerns were raised about the limited scope of the projection boxes, as they only visualized the main program and did not account for functions with static variables. The handling of long outputs was also highlighted as an issue, with users suggesting features like prioritizing the display of final values or managing overflow to improve navigation.

To incorporate projection boxes in the analysis tool, 70 lines of code (LOC) are needed, including 20 lines which could be made into boilerplate code.

The results of the performance metric demonstrated a negligible performance impact, with total times of 30.29 seconds without projection boxes and 30.71 seconds with projection boxes. This slight difference reflects the additional rendering and processing overhead introduced by projection boxes, amounting to an average server-side increase of 0.1 milliseconds per evaluation.

4.4 Discussion

The results show that there are practical benefits of projection boxes, particularly for the debugging experience. While quantitative performance impacts were negligible, the qualitative feedback underscores the value of projection boxes in making diagnostics more accessible. The ability to visualize executed values alongside corresponding code lines was appreciated, aligning with the intended goals of improved usability and comprehension.

The performance evaluation demonstrated that Projection Boxes introduce only a negligible overhead to the system. With an average server-side increase of 0.1 milliseconds per evaluation, the total runtime difference between tasks completed with and without projection boxes was minimal (30.29 seconds versus 30.71 seconds for 1,000 iterations). This finding suggests that projection boxes can be integrated into CodeProber without significantly compromising system responsiveness or performance.

The user study exclusively involved students who had prior experience with CodeProber, which could introduce bias into the findings. Their familiarity with the tool may have influenced their appreciation of projection boxes, as they were likely already accustomed to its interface and workflows. This makes it potentially misleading to generalize the results to broader programming contexts or to users without prior experience with CodeProber. Future studies should aim to include participants with varying levels of familiarity, including those with no prior exposure to CodeProber. This would provide a more balanced perspective on the usability and effectiveness of projection boxes and help identify whether the observed benefits are equally applicable to first-time users. Additionally, this approach would allow for better assessment of the learning curve associated with projection boxes, ensuring the tool is intuitive and accessible for all users.

The user study revealed varying opinions on how the projection boxes should be designed, emphasizing the importance of tailoring their design to accommodate individual user preferences and workflows. As highlighted in prior studies [10], excessive information flow can overwhelm users, particularly during debugging phases. This issue is evident in the current implementation of projection boxes in CodeProber, where loops generate visualizations spanning a large number of iterations. Currently, all values are displayed on a single row, making it challenging for users to locate the most relevant information—often the final value of a variable. Scrolling to the far right of the screen to find this value undermines the purpose of projection boxes, which are intended to remain readily visible to the programmer without requiring a switch from the keyboard to the mousepad. Future iterations could address this by prominently highlighting the final value and positioning it first, thereby reducing both cognitive and physical effort.

Finally, to further validate the utility of projection boxes, future evaluations should involve larger and more diverse participant groups, exploring complex use cases and real-world industry scenarios. Such studies would provide deeper insights into the broader applicability and effectiveness of projection boxes across various programming contexts.

5 Related work

Lerner [10] discusses design choices when implementing projection boxes. Lerner also presents different aspects to take in consideration when designing them, such as information overload.

Ferdownsifard et. al. presents live programming and their new paradigm called "Small-Step Live Programming by Example" [3]. Their tool, SNIPPY, exemplifies this approach by leveraging projection boxes to dynamically generate code based on changes in output specifications. The paper presents the benefits of live programming and the innovative capabilities of their method.

McDirmid [11] presents live programming as a method which increases the flow in programming experiences. Furthermore, they present a solution by probing information within the editor together with the benefits of this type of live programming experience.

Research papers covering CodeProber and its corresponding property probes was initially published by Risbers Alaküla [14]. Moreover there has been several publications such as Hardt et.al[5] and Riouak[13] that includes the usage of CodeProber.

6 Conclusion

This study presents the integration of projection boxes into CodeProber, in the educational context of compiler and program analysis courses at LTH. By extending CodeProber with projection boxes, the tool now offers a new visualization of runtime values alongside corresponding code.

The evaluation demonstrates that projection boxes improve usability by providing immediate, contextual feedback, which aids debugging and understanding program behaviour. Performance analysis indicates a negligible impact on processing speed, with an average increase of only 0.1 milliseconds per evaluation on the server side.

Additionally, the implementation effort to integrate projection boxes in another compiler is modest. It requires 70 lines of code, including 20 lines of reusable boilerplate, making it feasible to integrate into similar analysis tools.

User feedback highlights several strengths of projection boxes, particularly their ability to facilitate error identification and program analysis. However, areas for improvement were also identified, such as increasing output readability, improving the handling of long visualizations, and expanding the functionality to support additional program features beyond the main function. These observations provide a foundation for targeted design refinements in future iterations.

In conclusion, projection boxes constitute a valuable enhancement to CodeProber, improving its usability and effectiveness in educational contexts. Although currently designed for compiler courses, the feature's implementation and functionality suggest potential for adaptation to broader programming and debugging scenarios, subject to further development and evaluation.

Acknowledgments

We wish to thank Anton Risberg Alaküla for being our supervisor during this project. We really appreciate the helpful feedback and guidance the development and writing of this report. We would also like to thank Görel Hedin for the opportunity to work with a compiler project during this semester.

References

[1] Torbjörn Ekman and Görel Hedin. 2007. The jastadd extensible java compiler. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 1–18. https://doi.org/10.1145/1297027.1297029

[2] ExtendJ. [n. d.]. ExtendJ: The JastAdd Extensible Java Compiler. ([n. d.]). https://jastadd.cs.lth.se/web/extendj/

[3] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-Step Live Programming by Example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*. Association for Computing Machinery, New York, NY, USA, 614–626. https://doi.org/10.1145/3379337.3415869

[4] Interaction Design Foundation. 2016. What is the Golden Ratio? *Interaction Design Foundation* (08 2016). https://www.interactiondesign.org/literature/topics/golden-ratio

[5] Johannes Hardt and Dag Hemberg. 2023. JastAdd Bridge: Interfacing reference attribute grammars with editor tooling. (2023).

[6] Görel Hedin and Eva Magnusson. 2003. JastAdd—an aspect-oriented compiler construction system. *Science of Computer Programming* 47, 1 (2003), 37–58. https://doi.org/10.1016/S0167-6423(02)00109-0 Special Issue on Language Descriptions, Tools and Applications (L DTA'01).

[7] JastAdd. [n. d.]. JastAdd Concept Overview. ([n. d.]). https://jastadd.cs.lth.se/web/documentation/concept-overview.php

[8] Donald Ervin Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2.2 (1968), 127–145. https://api.semanticscholar.org/CorpusID:5182310

[9] Jan-Peter Kramer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. 2014. How live coding affects developers' coding behavior. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 5–8. https://doi.org/10.1109/VLHCC.2014.6883013

[10] Sorin Lerner. 2020. Projection boxes: On-the-fly reconfigurable visualization for live programming. *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Apr 2020). https://doi.org/10.1145/3313831.3376494

[11] Sean McDirmid. 2013. Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 53–62. https://doi.org/10.1145/2509578.2509585

[12] Jesper Öqvist and Görel Hedin. 2013. Extending the JastAdd extensible Java compiler to Java 7. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '13)*. Association for Computing Machinery, New York, NY, USA, 147–152. https://doi.org/10.1145/2500828.2500843

[13] Idriss Riouak. 2024. *Towards Declarative Specification of Static Analysis for Programming Tools*. Doctoral Thesis (compilation). Department of Computer Science. Defence details Date: 2024-11-22 Time: 13:15 Place: Lecture Hall E:A, building E, Klas Anshelms väg 10, Faculty of Engineering LTH, Lund University, Lund. External reviewer(s) Name: De Roover, Coen Title: Prof. Affiliation: Vrije Universiteit Brussel, Belgium. —.

[14] Anton Risberg Alaküla. 2024. *Property Probes: Live Exploration of Source Code Analysis*. Licentiate Thesis. Department of Computer Science.

[15] 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)*. Association for Computing Machinery, New York, NY, USA, 148–160. https://doi.org/10.1145/3567512.3567525

[16] Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*. 31–34. https://doi.org/10.1109/LIVE.2013.6617346