

Property Probes: Live Exploration of Source Code Analysis

Anton Risberg Alaküla



Licentiate Thesis, 2024

Department of Computer Science
Lund University

ISBN 978-91-8104-222-1 (electronic version)
ISBN 978-91-8104-221-4 (print version)
ISSN 1652-4691
Licentiate Thesis 3, 2024

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: anton.risberg_alakula@cs.lth.se

Typeset using L^AT_EX
Printed in Sweden by Tryckeriet i E-huset, Lund, 2024

© 2024 Anton Risberg Alaküla

ABSTRACT

Source code analysis is ubiquitous in the development of software tools, for example in compilers to detect compile-time errors and possible optimizations, in IDEs to provide interactive coding assistance, and in stand-alone analysis tools to detect bugs. There are several techniques that have been developed to help the analysis developer, including one called *Reference Attribute Grammars* (RAGs). With RAGs, the developer specifies functionality for their analysis as a set of high level equations. The concern of how to apply the equations is abstracted away, and handled by a RAG evaluation system. This abstraction can enable more concise and efficient implementations, but also calls for adequate debugging tools. When things break down, being able to see how things execute in terms of these abstractions helps the developer identify and fix issues.

The aim of this thesis is to provide source code analysis developers with a live, exploratory view of the inner workings of their analyses. We are particularly interested in helping developers using the RAG formalism. In this thesis, we introduce the concept of *property probes* to support this goal. Property probes enable efficient and robust interaction with computation associated with nodes on abstract syntax trees (ASTs). The different kinds of probes, and associated algorithms for creating and applying them, are presented in this thesis. We also present benchmarks showing that performance scales well for real-world development tasks.

We have implemented property probes in a tool called CODEPROBER. CODEPROBER has been integrated into two university courses on compilers and program analysis, as an aid during lab assignments. We wanted to determine the use- and user experience of the students that used it, and to this end we performed a mixed-method user study with students from the program analysis course. The focus of the study is a set of in-person interviews, and the overall feedback from students has been very positive.





CONTRIBUTION STATEMENT

This thesis consists of an introductory section followed by two papers.





Included Publications

Paper I Anton Risberg Alaküla, Görel Hedin, Niklas Fors and Adrian Pop “Property Probes: Live Exploration of Program Analysis Results” In *Journal of Systems and Software*, Volume 211, 2024, Elsevier.
DOI: 10.1016/J.JSS.2024.111980.

This paper is the extended version of a conference paper with the title “Property Probes: Source Code Based Exploration of Program Analysis Results.”, presented in *SLE 2022: Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, pp. 148-160. 2022.
DOI: 10.1145/3567512.3567525.

<i>Concepts</i>	<i>Implementation</i>	<i>Evaluation</i>	<i>Writing</i>
			

Paper II Anton Risberg Alaküla, Niklas Fors and Emma Söderberg “Study of the Use of Property Probes in an Educational Setting”. Submitted for publication.



<i>Design</i>	<i>Execution</i>	<i>Analysis</i>	<i>Writing</i>
			

The dark portion of the circles in the tables above represents the amount of work and responsibilities assigned to Anton Risberg Alaküla for each individual step:

- ◐ Anton Risberg Alaküla was a minor contributor to the work
- ◑ Anton Risberg Alaküla was a contributor to the work
- ◒ Anton Risberg Alaküla led and did a majority of the work
- Anton Risberg Alaküla led and did almost all of the work

Artifacts and Demonstrations Related to the Publications

Artifact I Anton Risberg Alaküla, Görel Hedin, Niklas Fors and Adrian Pop
“Property Probe Artifact (1.1.0)” DOI: 10.5281/zenodo.7259115.

This artifact was submitted alongside the conference paper version of **Paper I**. It contains an early version of CODEPROBER, and instructions for reproducing the experimental results of the paper. It was awarded with the ACM badges “Available”  and “Reusable” .

- Anton Risberg Alaküla led and did almost all of the work.

Artifact II Anton Risberg Alaküla, Görel Hedin, Niklas Fors and Adrian Pop
“CodeProber repository” URL: <https://github.com/lu-cs-sde/codeprober/>.

This is the source repository of CODEPROBER. It contains several new features developed after **Artifact I** was published.

- Anton Risberg Alaküla led and did almost all of the work.

Demonstration Anton Risberg Alaküla, Görel Hedin and Niklas Fors
“CodeProber: Live Compiler Exploration”.

URL: https://www.youtube.com/watch?v=_ZTv_Li1f_g.

This is a presentation and demonstration of CODEPROBER at *The Ninth Workshop on Live Programming, at ACM SIGPLAN SPLASH, Cascais, Portugal, 2023*.

- Anton Risberg Alaküla led and did almost all of the work.



Artifact Available: “Author-created artifacts relevant to this paper have been placed on a publically accessible archival repository. A DOI or link to this repository along with a unique identifier for the object is provided.”



Artifact Reusable: “The artifacts associated with the paper are of a quality that significantly exceeds minimal functionality. That is, they have all the qualities of the Artifacts Evaluated – Functional level, but, in addition, they are very carefully documented and well-structured to the extent that reuse and repurposing is facilitated. In particular, norms and standards of the research community for artifacts of this type are strictly adhered to.”

Source: <https://www.acm.org/publications/policies/artifact-review-and-badging-current>.

ACKNOWLEDGEMENTS

There are many people who have helped me in writing this thesis. First and foremost, I would like to thank my main supervisor Görel Hedin. We first met after I, some random guy from industry, emailed you and wanted to meet and discuss the compiler I was working on. I presented my work, and you immediately understood what I was trying to accomplish. You then rephrased my presentation in a way that actually made me better understand what I had been working on for the last 4 years. To this day, I am very thankful for the wisdom and guidance you provide.

I also want to thank my co-supervisors Niklas Fors and Adrian Pop, for helping me with my work. A special thanks to Niklas for the many discussions at the weekly meetings, and all the effort you put into the user study. I would also like to thank Christoph Reichenbach for integrating CODEPROBER into your course, and Emma Söderberg for helping with the user study.

There are a number of people who I haven't written papers with, but who still brightens my days at work. I want to thank my work roommate Idriss Riouak, for bouncing ideas with me and teaching me about how different vegetables would wear ties. Also thanks to Momina Rizwan, Peng Kuang and all others at the SDE group and the broader computer science department. Among other things, I am thankful to this group for the many fun and weird discussions we had around the lunch table.

Finally, I want to thank my girlfriend Maria Bark. You make every day better, and I love you very much.

The work performed in this thesis was funded by ELLIIT - Excellence Center at Linköping-Lund on Information Technology, and supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP). Thank you for letting me do research in the field which I find so fascinating.

CONTENTS

Abstract	iii
Contribution Statement	v
Acknowledgements	vii
I Introduction	1
1 Introduction	1
2 Background	3
2.1 Attribute Grammars	3
2.2 Reference Attribute Grammars	5
2.3 JASTADD	5
2.4 Attribute Evaluation	6
3 CODEPROBER	8
3.1 Debugging RAGs	8
3.2 Property Probes	8
3.3 Usage	9
3.4 Spanning Tree	10
3.5 JASTADD integration	11
3.6 Usage in Research and Education	11
3.7 Combination of Liveness and Attribute Exploration	11
4 Node Locators	12
4.1 Data structure	12
4.2 Algorithms	14
5 User Study	16
5.1 Methodology	16
5.2 Results	17
6 Future Work	17

6.1	Expanding CODEPROBER With Test Support	18
6.2	Applying Property Probes in Code Review	19
6.3	Side Effect Detector	20
	Detecting Side Effects Statically	20
	Detecting Side Effects Dynamically	20
6.4	LSP Workbench	21
7	Conclusion	22
	References	22

Included Papers 29

II	Property Probes: Live Exploration of Program Analysis Results	31
1	Introduction	31
2	Property Probes	33
2.1	Properties	33
2.2	Property Probes	35
2.3	Use Cases	36
2.4	Tool Architecture	37
3	CodeProber	38
3.1	Creating a Probe	38
3.2	Advanced Probes	38
	References and Nested Probes	39
	Search Probes	40
	AST Probes	40
	Probes Contributing Diagnostics	41
3.3	Liveness	42
3.4	Synthetic Nodes	42
4	Node Locators	43
4.1	Node Locator Steps	44
4.2	Example Node Locators	45
4.3	Kinds of Node Locators	46
4.4	Adapting to Changes	47
4.5	Known Limitation: False Positives	47
5	Node Locator Algorithms	48
5.1	Creating Locators	48
5.2	Applying Locators	50
5.3	Optimizing Node Locator Construction	53
	APPLYTALSTEP	53
	CANEXPANDTAL	53
	CREATEROBUSTLOCATOR	54
	Achieving Linear Time	54
5.4	TAL Adjustments	55

5.5	Handling Multiple Files	56
	CREATEROBUSTLOCATOR	57
	Probe Creation Node List	58
	CreateTAL	58
6	Implementation	58
6.1	Overall Architecture	58
6.2	Client↔Server API	59
6.3	Server↔Analysis Tool API	60
6.4	Desirable AST Features	62
6.5	Program Representation	63
7	Case Studies	64
7.1	Java Compiler	64
7.2	Flow Analysis	65
7.3	Compilers for Other Languages	65
7.4	Compiler Course	66
7.5	Program Analysis Course	67
7.6	Cloud Server	69
	Delayed Requests	69
	Optional WebSocket	70
7.7	Summary	70
8	Performance Evaluation	70
8.1	Methodology	71
8.2	Results	72
	Creating a Probe	72
	Evaluating a Probe	73
	Full Parse Time	74
8.3	Discussion	74
9	Related Work	76
10	Conclusion	78
	References	79
III Study of the Use of Property Probes in an Educational Setting		83
1	Introduction	84
2	Language Tools with On-Demand Evaluation	86
2.1	Debugging RAGs	87
3	CODEPROBER	88
3.1	Basic Usage	88
4	Study Overview	90
4.1	Study Objective and Design	91
4.2	Educational Setting	92
	Student Population	92
	The Compilers Course	93
	The Program Analysis Course	93

5	Interviews	94
5.1	Method	94
	Data Collection	95
	Data Analysis	96
5.2	Results	97
	Participants Experience and Skill	97
	CODEPROBER Feature Usage	97
	Techniques and Scenarios	98
	Interview Themes	98
6	Log File Analysis	103
6.1	Method	103
	Data Collection	104
	Data Analysis	104
6.2	Results	105
	Amount of CODEPROBER Use	105
	Spread of CODEPROBER Use	106
7	Survey	108
7.1	Method	108
7.2	Results	109
8	Summary of Results	110
8.1	RQ1 What is the user experience of using CODEPROBER in an educational setting?	110
8.2	RQ2 How is CODEPROBER used during the development of compilers and static analysis tools in an educational set- ting?	111
8.3	RQ3 How does the use of CODEPROBER compare to other tools used by students during the development process (e.g. debuggers, test cases, print-statements, AI, etc.)?	111
8.4	Threats to Validity	112
9	Related Work	113
9.1	Liveness	113
9.2	User Studies	115
9.3	Language Tooling Development	116
9.4	Debugging	117
10	Conclusions	117
	References	118
	Appendices	122
	A Interview Design	122
	B Background Information Form	124
	C CODEPROBER Feature Form	125
	D Tool/Scenario Form	126
	E Coding Scheme	127

INTRODUCTION

1 Introduction

It is hard to overstate the importance of software in the modern world. We use software to socialize, educate, manage our time, handle money, play games, control our vehicles, and much more. All software is created with some sort of programming language. Consequently, the construction of programming languages and their supporting tools is a broad and active research area, with a large impact on industry.

An important part of implementing programming languages consists of creating source code analyses. A compiler needs to analyze code to find compile-time errors and possible optimizations. It also needs to determine how to transform each language construct into its target code, and this transformation is non-trivial. For example, in many languages the '+'-operator can be used for both addition of numbers and concatenation of strings, and it is with source code analysis that the compiler decides which operation is applicable. Code analyses can also be found in IDEs, where it is used to provide interactive coding assistance in the form of code completion, hover information, etc. There are also a number of non-interactive analyses that are often run in continuous integration pipelines [Zam+17; Sad+15]. A popular example is SonarQube [Son], which contains hundreds of analyses, covering security issues, code smells, and more.

Source code analyses can be implemented as a set of smaller analyses that depend on each other. For example, to detect compile-time errors, it is typically necessary to compute types of all expressions, which in turn requires computing name bindings. Developing an analysis is non-trivial, and debugging it can be difficult as there may be thousands of smaller analyses that are linked together. For example, the specification for Java 8 [Gos+15] is 788 pages long, and it contains many semantic rules that interact with each other.

In this thesis we present work on trying to help the source code analysis developer. To this end we have developed CODEPROBER, an exploration tool of

analysis results. In CODEPROBER, the analysis developer begins by editing an example source file and selecting a program element in the file, such as an expression or function declaration. Then they select a (sub-)analysis step from their analysis tool, and CODEPROBER presents the result of running the analysis with the program element as input. The result is presented in a small floating window, which we call a *property probe*. CODEPROBER is *live*, which means that displayed values (i.e., property probes) on screen stay up to date, even as input values change. For example, if the user updates the example source file, or makes changes to their analysis tool implementation, CODEPROBER will immediately display updated values. In general, liveness in development environments refers to the ability to modify a running program [Tan13]. This can enable a tighter develop-test-fix development loop [Krä+14], as the time between making changes and getting feedback is lowered.

Live development tools have to respond quickly to user interactions in order to maintain a good user experience. One of the challenges with building live development tools is getting performance to scale well in larger systems and codebases. As there is more code to compile and/or execute on each keystroke, performance will naturally degrade, which reduces the benefit of liveness. Benchmarks for CODEPROBER [Ala+24] show that interactions typically respond in less than 100 milliseconds when exploring codebases containing 100k lines of code, which means that it is fast enough to appear instant [Nie93] even in real-world codebases. The exact response time depends on the analysis step itself, so an analysis implementation style that can quickly compute individual values works best.

CODEPROBER is a tool for exploring individual computations on a tree-like structure, which fits well with Reference Attribute Grammars [Hed00] (RAGs), an extension of Attribute Grammars [Knu68b]. RAGs is a formalism for analyzing source code which separates the specification and evaluation of language semantics. The analysis developer writes a specification for their tool as a set of high-level equations, and a RAG system handles evaluating the specification. The evaluation can be performed on-demand, meaning that if only a small sub-analysis step is requested, then no other values are computed. This is an important characteristic that helps CODEPROBER achieve liveness even when exploring larger, real-world analysis tools. There are several implementations of RAGs available, including Silver [VW+10], Kiama [SKV09] and JASTADD [HM03]. CODEPROBER has been specifically integrated with JASTADD, but in theory it can be used with any tool that associates computation with nodes on a tree-like structure.

There are other tools that support attribute exploration, and there is even one specifically built for JASTADD called DrAST [LTH16]. However, to our knowledge, none of the other exploration tools support liveness. One of the biggest technical challenges in implementing CODEPROBER was how to track the program element the user selected even as they make changes, i.e. how to make the computation stay *live*. Most live tools perform computation associated with a line of code, for example showing values of variables on the lines where variables are

updated. In CODEPROBER, the computation is associated with a position in a tree-like data structure, which is significantly more difficult to track. Our solution is a data structure which we call *node locator* and it is presented in Section 4.

We performed a mixed-method user study on the use and experience of using CODEPROBER, in order to determine if it was useful and how it compares to traditional debugging tools such as breakpoint/step-debuggers. The study focused on students in a program analysis course, where they use CODEPROBER and RAGs in a series of labs. We found that the students found CODEPROBER to be useful and enjoyable to use, and they seem to prefer it over traditional tools like debuggers and print debugging, at least for the task of creating program analyzers with RAGs.

The next sections of this thesis introduction are the following:

- In Section 2 we give background on RAGs and JASTADD.
- In Section 3 we present CODEPROBER in more detail, and how it compares to similar tools.
- In Section 4 we present *node locators*, the data structure that enables the quick and robust exploration found in CODEPROBER.
- In Section 5 we present a summary of the user study.

Finally, in Section 6 we discuss ideas for future work, and in Section 7 we conclude.

2 Background

The purpose of source code analysis is to compute values from source code. These values may be compile-time errors, machine code, refactoring suggestions, etc. Knuth introduced Attribute Grammars [Knu68b] (AGs), which he described as “a simple technique for specifying ‘meaning’ of languages defined by context-free grammars”. Much of the work performed in this thesis is indirectly built on top of AGs. In this section we present AGs, an extension to AGs called Reference Attribute Grammars [Hed00] (RAGs), and a metacompiler that supports RAGs called JASTADD [HM03]. Finally, we discuss some challenges with attribute evaluation.

2.1 Attribute Grammars

Attributes are a technique for computing “meaning” from source code. The process of parsing a program will reveal some intrinsic meaning in the code, such as the values of number literals, and parent-child relations between the program elements (i.e. the structure of the parse tree). Attributes use this intrinsic meaning to derive non-trivial values. To exemplify how attributes can be used, we will use a

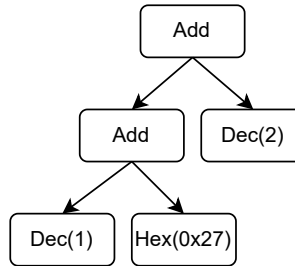


Figure 1: The parse tree for the HEXCALC expression $1 + 0x27 + 2$.

Listing 1: Implementation of HEXCALC in canonical attribute grammars, using slightly modified syntax to fit an abstract syntax tree and assuming a `base_16_to_10` operation exists.

$$\begin{aligned}
 v(\text{Add}(\text{Lhs}, \text{Rhs})) &= v(\text{Lhs}) + v(\text{Rhs}) \\
 v(\text{Dec}(\text{Value})) &= \text{Value} \\
 v(\text{Hex}(\text{Value})) &= \text{base_16_to_10}(\text{Value})
 \end{aligned}$$

simple calculator language we call HEXCALC. HEXCALC is a tiny language that only supports additions of decimal (base-10) and hexadecimal (base-16) numbers. The desired output (“meaning”) of a HEXCALC expression should be a single decimal value. For example, for the input $1 + 0x27 + 2$, the output should be 42.

Figure 1 shows the parse tree for the HEXCALC expression $1 + 0x27 + 2$. In the figure, a few intrinsic values are visible, such as the values of `Dec` (base-10) nodes and the structure of the tree. However, it is not immediately obvious what the value of the `Hex` (base-16) or `Add` nodes are. Attributes can be used to describe how to compute these unknown values.

In Knuth’s paper he defines attributes on top of a context-free grammar, but he does also write that his ideas “blends well with McCarthy’s idea of ‘abstract syntax’”. By modifying his syntax slightly to fit with abstract syntax trees (ASTs), we can implement value computation for HEXCALC as shown in Listing 1. The specification is a declarative set of equations that describe how to compute the value for each AST node type. The values of `Add` nodes depend on the values of their child nodes. `Dec` node values are intrinsically known, so there is nothing to compute. `Hex` nodes require conversion from base-16 to base-10, using an operation `base_16_to_10` which we assume is available for this example.

There are two kinds of attributes defined in Knuth’s paper: *synthesized* and *inherited*. Synthesized attributes compute information based on a given tree node and its descendants. The attribute definitions in Listing 1 are synthesized, as each attribute only depends on the nodes themselves or their children. Inherited attributes are instead defined in terms of the ancestors of a given node. This can

be used to implement non-local information lookup, such as finding the type and location of variable declarations.

2.2 Reference Attribute Grammars

Reference Attribute Grammars [Hed00] (RAGs) extend AGs with the capability of attributes resolving into references to other AST nodes. RAG specifications are able to specify computation of non-local information in a more concise manner, by allowing AST nodes that are far away from each other in the tree to directly *reference* each other, similar to how a parent node can directly reference its child nodes.

In AGs, non-parent-child nodes cannot reference each other. What they *can* do is to communicate with a common ancestor, using a combination of synthesized and inherited attributes. This ancestor can act as a glue layer that transports information on behalf of its descendants. For example, to make variable type information available to variable uses, an AG specification should contain the following two attributes:

1. A synthesized attribute that collects a table of variable names mapped to variable types under a common ancestor. In languages like Java, variable uses may be connected to declarations in other files via e.g. subtyping or static imports, so the common ancestor in this case is located far up in the tree, possibly the root of the tree.
2. An inherited attribute that makes the table from the synthesized attribute available to all descendants of the common ancestor.

For each new piece of information that should be made available, the variable table needs to be expanded, or a new table attribute can be defined. The resulting AG specification can become one big table attribute, which makes the code less modular and/or extensible. Alternatively, it may become a set of smaller table attributes that contain very similar looking (i.e. duplicated) code for collecting and transporting information. With RAGs, a single reference attribute that connects variable uses and declarations can be reused for transporting all variable-related information. The resulting RAG specification is therefore a set of smaller attributes, which retains modularity and extensibility while avoiding the need to duplicate code.

2.3 JASTADD

There are several tools that implement support for RAGs. For example, Kiama [SKV09] adds RAG support to Scala with a library. Silver [VW+10] and JASTADD [HM03] are both metacompilers that compile RAG specifications into Java source code. An example of a tool built using RAGs is AbleC [Kam+17],

Listing 2: Implementation of HEXCALC in JASTADD.

```

/* AST structure */
abstract Expr;
Add: Expr ::= Lhs:Expr Rhs:Expr;
Dec: Expr ::= <Value:int>;
Hex: Expr ::= <Value:String>;

/* Attribute definition */
syn int Expr.v();
eq Add.v() = getLhs().v() + getRhs().v();
eq Dec.v() = getValue();
eq Hex.v() = Integer.parseInt(
    getValue().substring(2), 16
);

```

an extensible C compiler frontend implemented in Silver that is used for building extensions to the C programming language. In this thesis we work with ExtendJ [EH07b], an extensible Java compiler implemented in JASTADD.

JASTADD compiles RAG specifications into normal Java code. The input to JASTADD contains both attributes and a description of an AST structure. In Listing 2, a JASTADD implementation of HEXCALC can be seen. The implementation starts with a description of the AST structure, which declares that there are three node types (Add, Dec & Hex), which are all subtypes of `Expr`. It also declares which fields each node type contains, such as `Add` having two child nodes called `Lhs` and `Rhs`. JASTADD uses this to generate Java classes with appropriate constructors, fields and accessors for those fields (e.g. `getLhs`, `getValue`, etc.). The attribute definition declares that all `Expr` nodes have an attribute `v`, which should return an `int`. Then, there is a definition of `v` for each AST node type in order, similar to Listing 1. The expression on the right of the equals sign is an arbitrary Java expression. Attributes must not have any observable side effects, but there are otherwise no restrictions, which is why it is able to use the standard library function `Integer.parseInt`. JASTADD translates the attribute expressions into normal Java methods and weaves them into the generated AST classes, similar to static aspect weaving in AspectJ [Kic+01].

2.4 Attribute Evaluation

One of the challenges with evaluating attributes is how to handle dependencies. For example, the attribute `v` for `Add` in Listing 2 depends on the value of `v` in its child nodes. The attribute evaluator must make sure that those child values are computed before the addition takes place. There are two main approaches to

evaluating attributes: data-driven and demand-driven.

Data-driven attribute evaluation involves scheduling the evaluation of individual attributes according to their dependencies, and in this way evaluating all attributes on all nodes. Ordered Attribute Grammars is an example of such an algorithm [Kas80].

Demand-driven attribute evaluation involves only evaluating a requested attribute value, along with all attributes it transitively depends on. Values may be cached after they are evaluated, to ensure that no value is evaluated more than once [Jou84]. This guarantees that a demand-driven evaluation does not perform any more work than a data-driven evaluation. In fact, the opposite is more likely since a demand-driven evaluation only evaluates the minimum necessary to fulfill a given request. For example, if we wanted to compute the value of the `Hex` node in Figure 1, then that value alone would be calculated with a demand-driven evaluation strategy. If we later wanted to know the value of the parent of the `Hex` node, then the base-16 to base-10 conversion would not be performed again, since the result is cached. Conversely, a data-driven evaluator would have to evaluate the values of all nodes in the AST before being able to safely access individual values, such as the value of the `Hex` node.

RAGs use a demand-driven attribute evaluation strategy [Hed00]. This is safe to do, because of the previously mentioned rule that attributes must not have any side effects, so it does not matter if only a subset of attribute values are evaluated.

Liveness-focused tools such as `CODEPROBER` sometimes rely on incremental computation in order to make the live feedback update quickly enough [KEV16]. For RAG-based tools, one such improvement that can be made is to add incremental parsing, i.e. when the user modifies a piece of source code, only parse the modified code and merge the result with a previously parsed AST. Then, the RAG-based tool may also want to also incrementally evaluate an attribute to display to the user, such as a set of compile-time errors, or a list of code completion items, etc. However, incrementally evaluating attributes is non-trivial, as each attribute may indirectly have dependencies to any other attribute in the tree. Even if only a tiny part of the AST was incrementally parsed, the cached values in all other parts of the AST can become stale. `JASTADD` is able to generate code that dynamically tracks attribute dependencies to allow for incremental evaluation [SH12]. However, this dynamic tracking adds overhead which in some cases may exceed any potential performance improvements. Therefore, in this thesis we only make use of incremental parsing and on-demand evaluation, but not incremental attribute evaluation. In order to avoid using stale values in an incrementally parsed AST, `CODEPROBER` always flushes (removes) all cached values before any attribute is evaluated.

3 CODEPROBER

CODEPROBER helps debugging RAG-based analyses. In this section we present the ideas behind the tool, and some of its functionality. CODEPROBER is presented in more detail in both **Paper I** and **Paper II**. CODEPROBER is open source¹, and there is video available that demonstrates some of its features.²

3.1 Debugging RAGs

When creating a debugger for RAG-based analyses such as those built with JAST-ADD, there are several possible approaches. Perhaps the most straightforward approach would be to create a traditional breakpoint/step-debugger. However, this would expose the developer to the internal attribute evaluation code, which they are likely not interested in. We believe the on-demand characteristic of RAGs enable new and interesting opportunities for debugging.

As mentioned in Section 2.4, attribute dependencies in RAGs are evaluated on-demand. This means that individual values can often be computed quite rapidly, without spending time on computing unnecessary values. This rapid evaluation of individual attributes enabled the creation of CODEPROBER, a live exploration tool for RAG-based analyses. CODEPROBER provides the developer with an interactive environment for exploring attributes on an AST. The idea is that by being able to quickly explore each individual attribute, the developer is able to build understanding of how their analysis works, and therefore be able to pinpoint errors. This does not replace a traditional debugger, but is able to fulfill a similar need.

3.2 Property Probes

CODEPROBER has support for *property probes*, which were initially presented in **Paper I**. A property probe is presented to the user in the context of a source code text editor, and acts as a live observer of a property on an AST node. A *property* in this case means any form of computation associated with an AST node. This can be a RAG attribute, a visual representation of the structure of the AST, a list of nodes of a certain type, etc. Probes are live, which means that they should stay up to date when its input values change. There are two main input values: a RAG analysis implementation, and an example source file which is used as input to the analysis.

CODEPROBER can be seen in Figure 2, where it is used to inspect an error in a HEXCALC implementation. The error is that the value computed by $0 \times 12 + 3$ is 15, but it should be 21. Through the help of probes (the four floating windows on top of the code), the developer is able to investigate each intermediate step of the computation, i.e. the value of all numbers and the addition. This reveals that

¹<https://github.com/lu-cs-sde/codeprober/>. Accessed September 2024.

²A demo video of CODEPROBER: <https://youtube.com/watch?v=1kTJ4VL0xtY>.

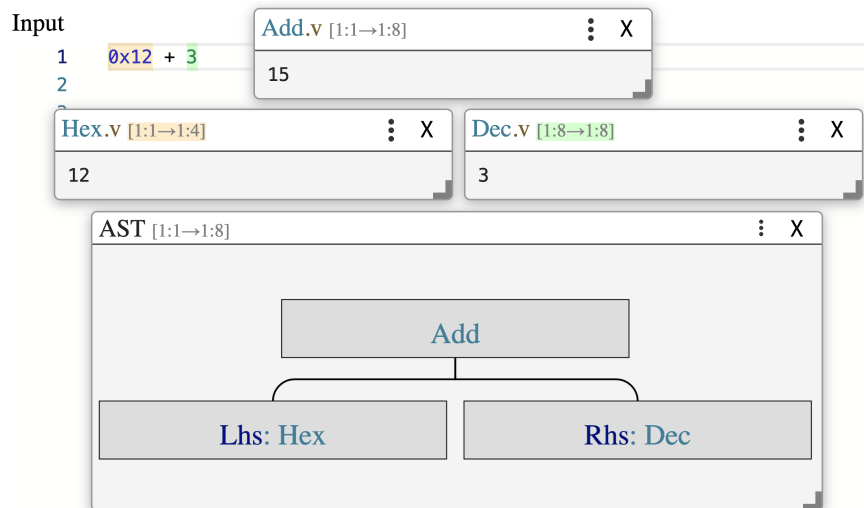


Figure 2: CODEPROBER being used to investigate an implementation error in a HEXCALC compiler. The developer has opened four different probes. One shows the structure of the AST and the rest shows intermediate steps of a HEXCALC expression evaluation.

the error is located with the HEX type, as its value is computed to be 12, rather than the expected 18. CODEPROBER supports several kinds of property probes, and two of them are shown in the figure. Three of the windows show attribute values, which we sometimes refer to as “value probes”, or just probes. The last window shows a visual representation of the structure of the AST, which we refer to as an AST probe.

3.3 Usage

Upon starting CODEPROBER, the user is presented with a text editor where they can type an example source file to be used as input to their analysis implementation. Once a file is specified, the user can right-click to create a probe, as seen in Figure 3. In the figure, the user creates a probe showing the compile-time constant value of an addition expression in the Java compiler ExtendJ [EH07b].

There are several kinds of probes supported, and different ways to explore the AST produced by the underlying analysis implementation. In addition to the previously mentioned value probes and AST probes, there are search probes that show all nodes that pass a given predicate, and nested probes that chain several value probes together. The output of a probe can be a variety of values, including references to other nodes in the AST. If the output is a reference, then it can be

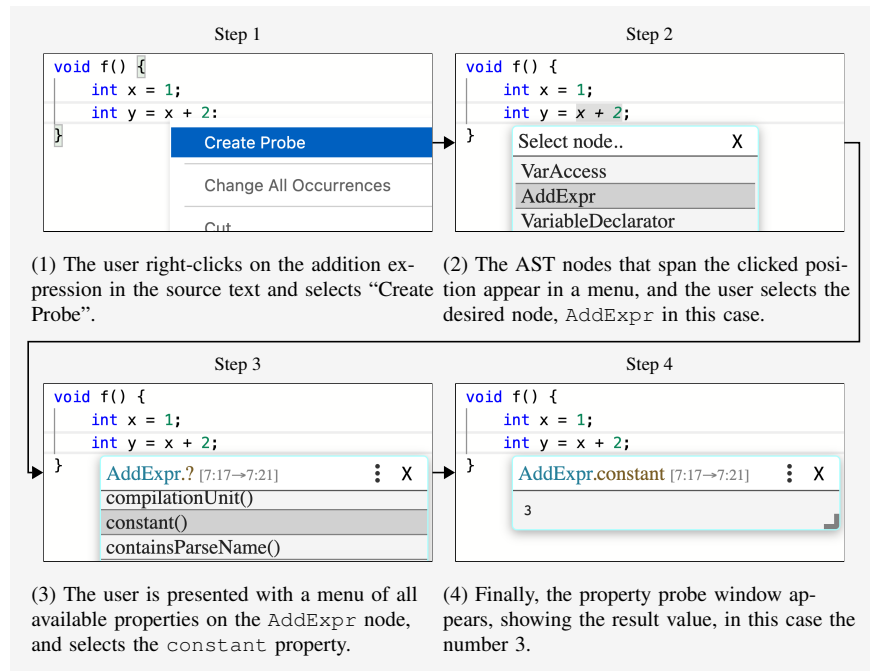


Figure 3: Steps to create a probe for the `constant` property of the addition expression `x + 2`.

hovered to cause the corresponding piece of the source code inside the text editor to highlight. Additionally, the reference can be clicked to create a new probe with the clicked node as the probe target. The feature set is described in more detail in both **Paper I** and **Paper II**.

3.4 Spanning Tree

CODEPROBER presents probe results in connection to locations in source code. For example, in Step 4 of Figure 3, the probe shows that it is connected to an `AddExpr` node on line 7, columns 17 → 21. This is possible to do because analysis tooling commonly parses source code into an AST, and associates source locations with each node in the AST. In this thesis we present property probes as being used for ASTs, but this definition is technically too narrow. Some analysis tooling, such as those using RAGs can technically represent source code as graphs due to the addition of non-local references, but they still have a *spanning tree* that can be used for traversal purposes. A more accurate definition of probes is therefore that they can be used with any data structure that has a spanning tree over nodes that have location information.

3.5 JASTADD integration

CODEPROBER can in theory be integrated with any tool that associates computation with a tree structure. However, in this thesis we have mainly integrated with JASTADD. The code that JASTADD generates follows a predictable structure, and it additionally contains some annotations that describe where the generated methods came from, which CODEPROBER uses to filter and present information in its UI. CODEPROBER has been used to explore several JASTADD-based tools. This includes the Java compiler ExtendJ [EH07b], and IntraJ [Rio+21] which adds intraprocedural control-flow and dataflow analysis on top of ExtendJ. We have also used CODEPROBER with implementations of Oberon-0 [FH15] (a tiny procedural language), Bloqqi [FH16] (a visual language for automation), and ChocoPy [PSH19] (a subset of Python commonly used for educational purposes), and more. Very few modifications had to be made to these tools to make them compatible with CODEPROBER. In most cases only 2 lines of code had to be added in order to provide CODEPROBER with an entry point into the code base. In the Implementation section in **Paper I** we more concretely describe the API used for integration.

3.6 Usage in Research and Education

CODEPROBER has been used in both research and education. In research, it has been used to help develop intraprocedural analyses for Java [Rio23]. In a project course, students have used CODEPROBER to help create and debug IDE extensions [HH24; LB24]. It has also been used by some students performing master's thesis works, in order to help build analyses for Java [Sol24; AW24]. Finally, it has been integrated into two courses. One is a course on compiler construction, where students build a compiler for a C-like language, and they have access to CODEPROBER to help debug their implementations. The other is a course on program analysis, where the students are handed an already working compiler at the start of each lab session, and are tasked with extending it with some analysis. These students are also able to use CODEPROBER to help debug their implementations. The students in the program analysis course are the main focus of the user study performed in **Paper II**.

3.7 Combination of Liveness and Attribute Exploration

There are other development tools that focus on liveness, i.e., instantly presenting updated values in response to user input. There are also tools that offer different forms of attribute exploration on ASTs. However, we are not aware of any previous work that combines liveness and attribute exploration in the same way that CODEPROBER does. In this section we briefly present the features of typical liveness-focused tools, and of typical attribute exploration tools. In the Related Work section in **Paper II** we perform a more thorough comparison to other tools.

A typical liveness-focused tool contains a code editor where the user can type the code they wish to explore. When changes are made, the tool runs the code and extracts runtime values [Ler20; Krä+14; Dub+16; McD13; HWX23]. The values are then presented to the user in connection to the code. For example, variable values may be rendered on lines where the corresponding variable assignments happen. One thing that differentiates CODEPROBER here is that the code inside CODEPROBER's editor typically does not run, it is instead used as input to a program analysis implementation in order to compute a static property of the code.

A typical attribute exploration tool contains some way to specify an analysis tool, or analysis tool specification as input. Additionally, the user can specify an example source file to be parsed into an AST. Then the user is able to explore attribute values associated with the different nodes in the AST [Slo99; LTH16; Ike+00; RCHS14]. Most of the user interaction in these tools is usually presented in terms of the AST, i.e., the values are rendered on top of- or in connection to- a graphical AST view. Sometimes the tools also connect the AST view and source code, for example so that the AST nodes can be hovered to highlight the corresponding code. What these tools typically do not do, however, is to allow for live updates of the example source file, which is one of CODEPROBER's most important features.

4 Node Locators

A tool that simultaneously supports liveness and attribute exploration on an AST must have some way to consistently track a position in an AST while the user is making changes to the corresponding source text. Tracking an AST position is difficult in part because even minor syntactic differences can have large impacts on the structure of the tree. In CODEPROBER, the user is able to change source text and get live feedback of computations that are associated with AST nodes. One of the greatest technical challenges involved in creating this is how to track AST nodes across user input changes. This section presents both the data structure and algorithms we designed for this purpose, which we call *node locators*. Node locators are presented in more detail in **Paper I**.

4.1 Data structure

An ideal data structure has good performance and is good at handling user changes. However, achieving these two qualities at the same time can be a challenge. To exemplify why, we first consider two simple structures that we call `FullPath` and `TargetNode`.

FullPath In an AST it should be possible to reach any AST node by starting from the root node and accessing child nodes in a specific order. In an object-

oriented language with a method `GETCHILD(N)` for accessing the N :th child, consider the following sequence of calls:

```
root.getChild(2).getChild(5).getChild(0).getChild(3)
```

This sequence of calls can be encoded as a data structure by extracting the child indexes, like this:

```
[2, 5, 0, 3]
```

The benefits of the `FullPath` structure is that it is unambiguous and quick. There will always be at most a single node that a given list of indexes identifies, and accessing children by index should be a constant time operation. The main downside of `FullPath` is that it is fragile to changes. If the user makes a change somewhere in the AST, then previously valid child index sequences may now point to different nodes, or no nodes at all.

TargetNode Instead of storing the full sequence for reaching the target AST node, it is possible to only store some identifying information that describes the target node. For example, the following may be used to identify an addition expression on line 5 and columns 8 \rightarrow 14 in a program:

```
{
  type = AddExpr
  location = 5:8  $\rightarrow$  5:14
}
```

Any number of fields may be stored, for example the types of neighboring nodes, how deeply nested the node is inside the AST, etc. This structure can later be used to search through an AST to find the node that best fits the stored information. A naive implementation of such a search algorithm could be quite slow, but it can be significantly improved by pruning the search based on location information. Many AST nodes contain information about which line and column they exist on. Additionally, parent nodes normally completely encapsulate their child nodes, i.e. no node has a greater span than its parent. Given this information, it is possible to avoid searching through subtrees with line/column spans that do not overlap with the location of the target node.

When the user makes a change, such as inserting a newline on the beginning of line N , then a `TargetNode` that targets line M where $M \geq N$ can simply increment its line by one. Applying the incremented `TargetNode` on the changed AST should result in finding the correct target node.

The benefits of the `TargetNode` approach is that it is robust, and can be fast enough if suitably implemented. Even after the user changes the source text, it is quite likely that a target node can be found.

The downside of `TargetNode` is that it can be ambiguous, as there may be multiple nodes that have the same identifying information. For example, a developer working on top of a control-flow graph (CFG) may introduce synthetic AST nodes to represent the entry and exit of a CFG in a method. Synthetic nodes such as these often do not have any position information. This means that there can be multiple entry and exit nodes in the same file with identical location information. In addition, it is possible that the developer makes mistakes during development and associates incorrect location information with some AST nodes. This can break the assumption that parent nodes completely encapsulate their child nodes, which would prevent the search pruning optimization. Given that we are interested in designing a tool that should be used during development, our chosen data structure must be tolerant to some level of implementation errors.

Combined Approach As mentioned earlier, an ideal data structure has good performance and is good at handling user changes. It should additionally be able to handle some level of ambiguity in the AST. Neither `FullPath` nor `TargetNode` is able to achieve this individually. However, it is possible to achieve by combining aspects of both. We have done so, and we call our approach *node locators*. A node locator is a sequence of steps, where each step moves a “current node” from one node to the next, starting from the AST root. There are three kinds of steps: `Child`, `TAL` and `FN`.

- A `Child` step contains a child index, just like one of the indexes in `FullPath`.
- `TAL` stands for “Type At Location” and a `TAL` step contains an AST node type and location information, similar to `TargetNode`. Since the steps are applied in sequence, a `TAL` step is not necessarily applied from the root, which avoids the ambiguity issues described for `TargetNode`.
- `FN` stands for “Function” and a `FN` step contains the name of a property and optionally a list of arguments. This step type is mainly created to support accessing children that are not “normal” indexed children, but rather higher-order attributes [VSK89a], such as the synthetic entry and exit nodes for a CFG. `FN` can be thought of as a more advanced `Child` step, and is described in more detail in **Paper I**.

4.2 Algorithms

By correctly combining a sequence of the three step types, any node can be reached in an efficient and robust manner. However, deciding on what combination to pick is not trivial. Each individual node can usually be found by many different combinations of steps. For example, consider the following line of Java code:

```
int x = y + z;
```

Assume we are debugging some Java compiler implementation, and we want to reference the addition expression $y + z$. There are multiple ways to reach the expression. In our theoretical Java tool, any of the following locators would work:

```
[TAL("AddExpr", 2:10 → 2:15)]  
[Child(0), Child(1), TAL("AddExpr", 2:10 → 2:15)]  
[TAL("DeclStmt", 2:2 → 2:16), FN("rhs")]
```

The first example locator might be best in a small tool with a single source file. However, when working on a larger tool that supports multiple source files in the AST, such a locator would risk searching through files other than the one our target node exists in, which wastes time and increases risk of identifying the wrong node.

The second example locator could solve the issue of searching in incorrect files by taking a few `Child` steps from the root down into the correct file, and then applying `TAL` there. However, use of `Child` steps tend to make the locator less robust towards changes.

The third example locator is similar to the first one, but adds a layer of indirection by finding the declaration statement first and then taking the “rhs” step, i.e. going to the right-hand side of the declaration. This increases the risk of identifying the wrong node when the user makes changes. For example, consider what happens if the user changes the source code to:

```
int x = f(y + z);
```

The third example locator would now find the function call to `f`, not the addition expression.

Through a process of iterative development, we found a few heuristics that seem to work well:

- Shorter locators are generally more robust.
- `TAL` should almost always be used if possible. In cases where it would be ambiguous or for some other reason would not work, fall back to `Child` or `FN`.
- `TAL` should be avoided when traversing from the root of the AST to the desired source file (in multi-source file ASTs).

Combining these heuristics results in the second locator in our example above. Creating these locators efficiently while handling `TAL`-related ambiguity correctly is a challenge. Similarly, efficiently applying the created locators also requires use of techniques like the search pruning described for `TargetNode` above. Our developed algorithms are described in detail in **Paper I**.



Figure 4: Room used during the interviews.

5 User Study

We created CODEPROBER because we believed it could provide a useful way of debugging analysis implementations. In order to help determine to what extent this is true, we performed a mixed-method user study on the use and user experience of CODEPROBER in an educational setting. Our research questions for the study are the following:

- RQ1** What is the user experience of using CODEPROBER in an educational setting?
- RQ2** How is CODEPROBER used during the development of compilers and static analysis tools in an educational setting?
- RQ3** How does the use of CODEPROBER compare to other tools used by students during the development process (e.g. debuggers, test cases, print-statements, AI, etc.)?

In this section we summarize the methodology and results of the user study. The full study is presented in **Paper II**.

5.1 Methodology

The user study is a mixed-method study with three parts, focusing on students in the 2024 instance of a program analysis course. The three parts are:

1. In-person interviews with 9 students that participated in the course. We also interviewed two students that were involved in the course as TAs. The interview setup is shown in Figure 4.
2. Event log files gathered from 24 students in the course, containing a total of 576167 interactions in CODEPROBER.

3. Course surveys from two courses where CODEPROBER was used during labs. This includes two instances of the program analysis course, and one instance of a compiler construction course.

We include these three parts in order to enable *triangulation*, i.e., the process of investigating a phenomenon from at least two different perspectives in order to draw conclusions with more confidence [RSP23, pp. 179–180].

The focus in the user study is on the in-person interviews, as they help us answer all research questions. The other two sources help corroborate what was said during the interviews, and can help prevent some bias in the results, such as participant responder bias [Del+12]. We had interacted with the students in the interview in some form prior to the program analysis course, and they knew that we were working on CODEPROBER, so they could want to say positive things to be kind, even though that would taint the results. The event logs and course surveys are both anonymous, so they help detect if the interviews were unfairly positive.

5.2 Results

The overall findings from the user study were that students found CODEPROBER to be a useful tool, and they voluntarily used it continuously throughout the labs. We further found that they seem to use CODEPROBER more than traditional development tools such as breakpoint/step debugging and print debugging. In **Paper II** we go through each data source in detail. The summary of the research questions results were:

- RQ1** The students found CODEPROBER to be a useful tool that is enjoyable to use, despite some technical issues.
- RQ2** The students made continuous use of CODEPROBER, and they mainly rely on standard probes, squiggly lines and liveness.
- RQ3** Students in our study used CODEPROBER to partially replace print debugging, breakpoint/step-debuggers and test cases.

These claims come with a set of caveats and limitations which are discussed in the Threats to Validity section in **Paper II**. However, the results are still overall quite positive, and we believe they show that CODEPROBER is promising. More work is needed to investigate if the findings generalize to other contexts, such as different universities, hobbyists, industry, etc.

6 Future Work

We see several possibilities for future work, some focused on expanding CODEPROBER, and others focused on applying property probes in new domains. Our main ideas for future work are presented in this section.

```

String code = "import java.util.*;"
+ "public class Test {"
+ "  void m1(List<? extends String> names) {"
+ "    names.stream().mapToInt(name -> name.length());"
+ "  }"
+ "}";

CompilationUnit cu = parseCompilationUnit(code);
MethodDecl m1 = (MethodDecl) cu.getTypeDecl(0).getBodyDecl(0);
Dot dot1 = (Dot) ((ExprStmt) m1.getBlock().getStmt(0)).getExpr();
Dot dot2 = (Dot) dot1.getRight();
MethodAccess mapToInt = (MethodAccess) dot2.getRight();
LambdaExpr lambda = (LambdaExpr) mapToInt.getArg(0);
TypeDecl anonType = lambda.toClass().type();
for (BodyDecl decl : anonType.getBodyDeclList()) {
  if (decl instanceof MethodDecl) {
    MethodDecl method = (MethodDecl) decl;
    String typeSignature = method.methodTypeSignature();
    System.out.format("Type signature of %s: %s\n", method.name(),
typeSignature);
  }
}
}

```

Figure 5: Parts of a test case for the ExtendJ Java compiler. The code prints the type (“signature”) of a lambda expression. The test runner (not shown in the figure) captures the printed value and compares against an expected value. Large parts of the test code is dedicated to traversing from the file level (`CompilationUnit`) down to the `MethodDecl` corresponding to the lambda expression.

6.1 Expanding CODEPROBER With Test Support

In the user study (**Paper II**), students said that they wrote fewer test cases due to CODEPROBER. We hypothesize that this is because it is possible to manually verify functionality by exploring it inside CODEPROBER, and writing unit tests becomes a comparatively large hurdle, so students tend to avoid it. This is a problem, since CODEPROBER is not able to perform automatic regression testing. There is a possible solution however: allow probes to be converted to test cases. The probes contain input code, an action to be performed (evaluating a property) and a resulting value. These can be mapped to the arrange, act and assert pattern of typical unit tests [Kho20].

Probe tests could make it easier to create more fine-grained unit testing. For example, to assert that type inference is computed correctly, a test must parse source code, find a specific expression node in the AST, extract the inferred type and then compare against an expected value. Finding the expression node can be a cum-

bersome task. For example, consider the test case shown in Figure 5. More than half the statements in the test only exist to traverse through the AST. Additionally, the statements contain several typecasts and accesses child nodes by hard-coded indexes. If a developer wanted to refactor the compiler in a way that changes AST node names or child indexes, then the test will need to be updated. We believe that tests created in CODEPROBER would not suffer as much from this issue as the traversal code could be replaced by node locators which are automatically generated, and possibly even maintained by the tool.

6.2 Applying Property Probes in Code Review

It would be interesting to apply property probes in a code review setting. When performing code review, the reviewer must build an understanding of the change being made in order to decide if it is good, or if additional changes are necessary. Code review UIs typically only display a textual diff of the change being made. Sometimes this is enough, but other times the reviewer needs more information. For example, when a method is changed, it can be beneficial to see all the locations where that method is called. Those locations might not appear in the textual diff. What the reviewer can do is to fetch the change and start reviewing it in their local IDE, where navigating through the code is easier. This takes time and further increases how much code review can disrupt ongoing work. If more information was displayed in the code review UI, the need for fetching changes to a local IDE would be reduced.

One problem with deciding how to enrich a code review UI is that the kinds of information a reviewer needs can be highly personalized and context dependent [Söd+22]. Someone in an architectural role may be more concerned with the high-level structure of the code, while a more junior developer may want to focus more on lower-level details, such as ensuring that a coding standard is maintained. A code review UI should ideally support these different roles with the information they need, without making the experience cluttered. Property probes may be of use here, by making it possible to explore a large set of information from an analysis tool running on the server, just like when using CODEPROBER to explore the analysis steps. By following a similar design as CODEPROBER of not showing any extra information by default, but letting the user pick and choose which analysis results to display, the reviewer will be able to create a rich, custom reviewing experience. Additionally, by using a tree-diffing algorithm such as *truediff* [ESP21], it would be possible to identify the “same” AST node in the before and after version of a code change. This would enable the creation of a new kind of probe: *diff probe*. These could evaluate the same property on the two AST nodes, and display a diff of the output. This could be combined with search probes (see Figure 7 in **Paper I**) to enable queries like “show all nodes where the property X has changed”.

6.3 Side Effect Detector

Another interesting direction of future work would be to help JASTADD developers find side effect bugs in their analysis implementations. There is an informal contract between JASTADD and the developers who use it: attributes must not contain any observable side effects. Because of this contract, JASTADD is free to schedule attribute evaluation in any order, and may cache results of those attributes if deemed necessary for optimization purposes. There are no guarantees given to the developer regarding in which order attributes are evaluated, and how many times they are invoked. Side effects naturally depend on the order of operations, so their existence can cause incorrect values to be computed. Side effects also diminish the value of exploring with CODEPROBER, as the assumption that each property can be inspected in isolation while still producing the same value no longer holds.

JASTADD assumes that there are no side effects, but it does not verify that it is true. Each attribute body is able to reference global memory, interact with the file system, make network requests, etc. If JASTADD was able to detect these side effects, then it could give a warning to the developer and/or change how it generates evaluation code. We see two main approaches to detecting side effects: static and dynamic analysis.

Detecting Side Effects Statically

The code of attributes in JASTADD is regular Java code, which means that statically detecting side effects in JASTADD involves statically detecting side effects in arbitrary Java code. While this is an undecidable problem, it is still possible to produce interesting results with some level of overapproximation. JPure [Pea11] is a purity checker for Java, which is able to detect side effects with a relatively high degree of certainty. We could try adapting such a checker to work on JASTADD code.

Detecting Side Effects Dynamically

Side effects may be detected at runtime by invoking the same attribute multiple times in different configurations. For example, invoking a single attribute multiple times in a row, or a set of attributes in different orders. Dynamic testing could also detect performance issues, like memory leaks and/or slower evaluation times.

Simply exploring an analysis tool with CODEPROBER will automatically lead to some level of dynamic testing being performed, as the developer will notice if crashes occur, or if they run out of memory after a while. During the development of CODEPROBER, we usually tested functionality against the Java compiler ExtendJ [EH07b]. In this process we discovered two caching issues that would leak memory over time. These issues had previously gone undiscovered because ExtendJ is typically used and tested as a command-line compiler, where a memory

leak is not necessarily an issue since the compiler usually finishes execution quite quickly. However, for longer running processes like language server plugins in IDEs, a memory leak can be a problem, so the near-accidental dynamic testing by CODEPROBER was useful here. It would be beneficial to make the dynamic testing more structured and automated. This could be created as an extension to the probe tests mentioned above: once a set of tests are defined, they could be evaluated in different configurations and monitored for performance issues. Alternatively, a dynamic testing tool could be a simple command-line tool where the developer specifies a number of attribute names which should be tested, and then the tool could evaluate those attributes for all AST nodes. In such a tool there would be no “expected value”, but it could still detect values that change over time.

6.4 LSP Workbench

The Language Server Protocol (LSP) is a protocol that defines how clients (IDEs) and servers (programming language implementations) communicate about IDE interactions. For example, the protocol specifies how to describe a hover event, how to encode a list of code completion items, etc. The protocol is designed to allow a single language server to be used in multiple IDEs, rather than having to design a language extension for every single IDE. Almost all modern languages have a language server implementation available.³

Language servers often need to compute very similar information as other programming language tools, e.g. types of expressions, set of available variables at a point in the program, etc. For this reason, it makes sense to share a significant amount of code between e.g. a compiler and a language server. In this thesis we present CODEPROBER as a tool to be used during development of source code analysis tools, and we focus on compilers and bug detectors. However, we believe that CODEPROBER could be useful when developing language servers too.

When developing a language server, a natural way to test the implementation is to load the server into an actual IDE and check if the IDE interactions behave as expected. Any observed issues could be debugged using CODEPROBER. For example, the developer could copy the current code from the IDE into CODEPROBER and use probes to inspect intermediate values in order to locate the underlying cause of the issues. This works, but having to jump back and forth between the IDE and CODEPROBER for testing and debugging is not convenient. An alternative solution could be that CODEPROBER implements support for parts of LSP itself. This could make CODEPROBER into a tool that supports both testing and debugging LSP implementations.

Furthermore, there is interesting research opportunities in reducing the complexity of implementing the LSP protocol. We could for example build a translation layer that converts all LSP interactions into attributes in a RAG-based tool.

³<https://microsoft.github.io/language-server-protocol/implementors/servers/>. Accessed September 2024.

This would mean that in order to implement a certain interaction, the developer would no longer have to parse the raw protocol messages, but could instead implement an attribute with a certain signature. Such abstraction/translation layers that simplify LSP implementations have been done in the past, such as LSP4J⁴ which simplifies writing language servers in Java, and MagpieBridge [LDB19] which builds upon LSP4J to simplify connecting language servers to IDEs. An initial attempt at exploring this possibility for JASTADD was done by students we supervised in a project course, and the result is a Visual Studio Code extension called JASTADDBRIDGE [HH24]. JASTADDBRIDGE uses CODEPROBER in order to traverse and interact with the AST. The implementation shows good promise, but also has some room to improve. For example, it supports quite a small subset of LSP, only works with one editor, and does not support multiple simultaneous files. All of these issues are relatively easy to overcome though, which makes this a promising option for future work.

7 Conclusion

In this thesis, we presented CODEPROBER, a tool supporting live exploration of source code analysis results. CODEPROBER aims to help the analysis developer by providing an interactive view into the intermediate steps of their analysis implementation. CODEPROBER implements *property probes*, which are live observers of computation associated with nodes in a tree-like structure, such as attributes in an AST. The probes are able to consistently display updated values when the user makes changes to the input to CODEPROBER. Making probes efficiently and robustly handling user changes was one of the main technical challenges in implementing CODEPROBER. In this thesis we present *node locators*, which is our solution to this challenge.

CODEPROBER is a mature tool that has been used in both research and education at our university. Two courses make use of CODEPROBER as a supporting tool during lab assignments: one course on compiler construction, and one course on program analysis. We performed a mixed-method user study that focuses on the students in the program analysis course. The study found that the students make heavy use of CODEPROBER and seem to enjoy using it. Additionally, they seem to prefer using it over traditional debugging tools and techniques, such as breakpoint/step-debugging and print debugging.

Overall, the study shows that CODEPROBER is a promising tool for debugging analysis implementations. Looking forward we see several interesting opportunities for future work, including extensions to CODEPROBER itself, and making use of property probes in a new domain.

⁴<https://www.eclipse.org/lsp4j>. Accessed September 2024.

References

- [Ala+24] Anton Risberg Alaküla et al. “Property probes: Live exploration of program analysis results”. In: *Journal of Systems and Software* 211 (2024), p. 111980.
- [AW24] Johan Arrhén and Ruben Wiklund. “Pointer Analysis for Interactive Programming Environments”. <https://lup.lub.lu.se/luur/download?func=downloadFile&recordId=9169110&fileId=9169111>. MA thesis. 2024.
- [Del+12] Nicola Dell et al. ““Yours is better!”: participant response bias in HCI”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’12. Austin, Texas, USA: Association for Computing Machinery, 2012, 1321–1330.
- [Dub+16] Patrick Dubroy et al. “Language hacking in a live programming environment”. In: *Proceedings of the LIVE Workshop co-located with ECOOP 2016*. 2016.
- [EH07b] Torbjörn Ekman and Görel Hedin. “The Jastadd Extensible Java Compiler”. In: *SIGPLAN Not.* 42.10 (2007), 1–18.
- [ESP21] Sebastian Erdweg, Tamás Szabó, and André Pacak. “Concise, type-safe, and efficient structural diffing”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 406–419.
- [FH15] Niklas Fors and Görel Hedin. “A JastAdd implementation of Oberon-0”. In: *Sci. Comput. Program.* 114 (2015), pp. 74–84.
- [FH16] Niklas Fors and Görel Hedin. “Bloqqi: Modular Feature-Based Block Diagram Programming”. In: *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, 57–73.
- [Gos+15] James Gosling et al. *The Java® Language Specification: Java SE 8 Edition*. 2015.
- [HH24] Johannes Hardt and Dag Hemberg. *JastAdd Bridge: Interfacing reference attribute grammars with editor tooling*. Tech. rep. <https://fileadmin.cs.lth.se/cs/Education/edan70/CompilerProjects/2023/Reports/hardt-hemberg.pdf>. 2024.

- [HWX23] Devamardeep Hayatpur, Daniel Wigdor, and Haijun Xia. “CrossCode: Multi-level Visualization of Program Execution”. In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, CHI 2023, Hamburg, Germany, April 23-28, 2023*. Ed. by Albrecht Schmidt et al. ACM, 2023, 593:1–593:13.
- [Hed00] Görel Hedin. “Reference attributed grammars”. In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.
- [HM03] Görel Hedin and Eva Magnusson. “JastAdd—an aspect-oriented compiler construction system”. In: *Science of Computer Programming* 47.1 (2003), pp. 37–58.
- [Ike+00] Yohei Ikezoe et al. “Systematic Debugging of Attribute Grammars”. In: *Proceedings of the Fourth International Workshop on Automated Debugging, AADEBUG 2000, Munich, Germany, August 28-30th, 2000*. Ed. by Mireille Ducassé. 2000.
- [Jou84] Martin Jourdan. “Strongly non-circular attribute grammars and their recursive evaluation”. In: *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, Montreal, Canada, June 17-22, 1984*. Ed. by Mary S. Van Deusen and Susan L. Graham. ACM, 1984, pp. 81–93.
- [Kam+17] Ted Kaminski et al. “Reliable and automatic composition of language extensions to C: the ableC extensible language framework”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017), 98:1–98:29.
- [Kas80] Uwe Kastens. “Ordered Attributed Grammars”. In: *Acta Informatica* 13 (1980), pp. 229–256.
- [Kho20] Vladimir Khorikov. *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster, 2020.
- [Kic+01] Gregor Kiczales et al. “An Overview of AspectJ”. In: *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*. Ed. by Jørgen Lindskov Knudsen. Vol. 2072. Lecture Notes in Computer Science. Springer, 2001, pp. 327–353.
- [Knu68b] Donald E Knuth. “Semantics of context-free languages”. In: *Mathematical systems theory* 2.2 (1968), pp. 127–145.
- [KEV16] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. “Towards Live Language Development”. In: *Proceedings of the LIVE Workshop co-located with ECOOP 2016*. 2016.

- [Krä+14] Jan-Peter Krämer et al. “How live coding affects developers’ coding behavior”. In: *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2014, Melbourne, VIC, Australia, July 28 - August 1, 2014*. Ed. by Scott D. Fleming, Andrew Fish, and Christopher Scaffidi. IEEE Computer Society, 2014, pp. 5–8.
- [Ler20] Sorin Lerner. “Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming”. In: *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*. Ed. by Regina Bernhaupt et al. ACM, 2020, pp. 1–7.
- [LTH16] Joel Lindholm, Johan Thorsberg, and Görel Hedin. “DrAST: an inspection tool for attributed syntax trees (tool demo)”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*. Ed. by Tijs van der Storm, Emilie Balland, and Dániel Varró. ACM, 2016, pp. 176–180.
- [LB24] Mark Lundager and Andreas Bergqvist. *Finding and Resolving Common Code Style Errors in Java*. Tech. rep. <https://fileadmin.cs.lth.se/cs/Education/edan70/CompilerProjects/2023/Reports/lundager-bergqvist.pdf>. 2024.
- [LDB19] Linghui Luo, Julian Dolby, and Eric Bodden. “Magpiebridge: A general approach to integrating static analyses into ides and editors (tool insights paper)”. In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.
- [McD13] Sean McDirmid. “Usable live programming”. In: *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*. Ed. by Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld. ACM, 2013, pp. 53–62.
- [Nie93] Jakob Nielsen. “Response times: the three important limits”. In: *Usability Engineering* (1993).
- [PSH19] Rohan Padhye, Koushik Sen, and Paul N Hilfinger. “Chocopy: A programming language for compilers courses”. In: *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. 2019, pp. 41–45.

- [Pea11] David J. Pearce. “JPure: A Modular Purity System for Java”. In: *Compiler Construction - 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. Ed. by Jens Knoop. Vol. 6601. Lecture Notes in Computer Science. Springer, 2011, pp. 104–123.
- [Rio23] Idriss Riouak. *Declarative Specification of Intraprocedural Control-flow and Dataflow Analysis*.
<https://portal.research.lu.se/en/publications/declarative-specification-of-intraprocedural-control-flow-and-dat>. 2023.
- [Rio+21] Idriss Riouak et al. “A Precise Framework for Source-Level Control-Flow Analysis”. In: *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2021, pp. 1–11.
- [RCHS14] Daniel Rodríguez-Cerezo, Pedro Rangel Henriques, and José-Luis Sierra. “Attribute grammars made easier: EvDebugger a visual debugger for attribute grammars”. In: *2014 International Symposium on Computers in Education (SIIE)*. 2014, pp. 23–28.
- [RSP23] Yvonne Rogers, Helen Sharp, and Jennifer Preece. *Interaction Design: Beyond Human-Computer Interaction, 6th Edition*. John Wiley, 2023.
- [Sad+15] Caitlin Sadowski et al. “Tricorder: Building a Program Analysis Ecosystem”. In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. Ed. by Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum. IEEE Computer Society, 2015, pp. 598–608.
- [Slo99] Anthony M. Sloane. “Debugging Eli-Generated Compilers With Noosa”. In: *Compiler Construction, 8th International Conference, CC’99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS’99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*. Ed. by Stefan Jähnichen. Vol. 1575. Lecture Notes in Computer Science. Springer, 1999, pp. 17–31.
- [SKV09] Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. “A Pure Object-Oriented Embedding of Attribute Grammars”. In: *Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications, LDTA 2009, York, UK, March 27-28, 2009*. Ed. by Torbjörn Ekman and Jurgen J. Vinju. Vol. 253. Electronic Notes in Theoretical Computer Science 7. Elsevier, 2009, pp. 205–219.

- [SH12] Emma Söderberg and Görel Hedin. “Incremental evaluation of reference attribute grammars using dynamic dependency tracking”. In: (2012).
- [Söd+22] Emma Söderberg et al. “Understanding the Experience of Code Review: Misalignments, Attention, and Units of Analysis”. In: *EASE 2022: The International Conference on Evaluation and Assessment in Software Engineering 2022, Gothenburg, Sweden, June 13 - 15, 2022*. Ed. by Miroslaw Staron et al. ACM, 2022, pp. 170–179.
- [Sol24] Max Soller. “SinfoJ: A simple Information Flow Analysis with Reference Attribute Grammars”. <https://lup.lub.lu.se/luur/download?func=downloadFile&recordId=9149210&fileId=9149211>. MA thesis. 2024.
- [Son] SonarSource. *Code Quality, Security & Static Analysis Tool with SonarQube | Sonar*. <https://www.sonarsource.com/products/sonarqube/>. Accessed: 2024-10-02.
- [Tan13] Steven L. Tanimoto. “A perspective on the evolution of live programming”. In: *Proceedings of the 1st International Workshop on Live Programming, LIVE 2013, San Francisco, California, USA, May 19, 2013*. Ed. by Brian Burg, Adrian Kuhn, and Chris Parnin. IEEE Computer Society, 2013, pp. 31–34.
- [VW+10] Eric Van Wyk et al. “Silver: An extensible attribute grammar system”. In: *Science of Computer Programming 75.1-2 (2010)*, pp. 39–54.
- [VSK89a] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. “Higher-Order Attribute Grammars”. In: *Proceedings of the ACM SIGPLAN’89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*. Ed. by Richard L. Wexelblat. ACM, 1989, pp. 131–145.
- [Zam+17] Fiorella Zampetti et al. “How open source projects use static code analysis tools in continuous integration pipelines”. In: *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*. Ed. by Jesús M. González-Barahona, Abram Hindle, and Lin Tan. IEEE Computer Society, 2017, pp. 334–344.

INCLUDED PAPERS

PROPERTY PROBES: LIVE EXPLORATION OF PROGRAM ANALYSIS RESULTS

Abstract

We present *property probes*, a mechanism for helping a developer explore partial program analysis results in terms of the source program interactively while the program is edited. A node locator data structure is introduced that maps between source code spans and program representation nodes, and that helps identify probed nodes in a robust way, after modifications to the source code. We have developed a client-server based tool CODEPROBER supporting property probes, and argue that it is very helpful in debugging and understanding program analyses. We have evaluated our tool on several languages and analyses, including a full Java compiler and a tool for intraprocedural dataflow analysis. Our performance results show that the probe overhead is negligible even when analyzing large projects.

1 Introduction

Modern software tooling includes many kinds of program analysis. For instance, compilers do type analysis, IDEs support type-based navigation and editing, and bug-finding tools may use analyses based on dataflow and effects. However, de-

veloping new analyses can be difficult. There are often many subanalyses, and they might need to handle many corner cases of the analyzed language.

In this paper we propose a new interactive mechanism, *property probes*, to help the analysis developer. The main idea is to allow the developer to inspect and display *properties*, i.e., (partial) analysis results tied to specific parts of an editable source code (as plain text). Examples of properties include name bindings, types, generated code, propagated constant values, control-flow edges, and data flow properties like live variables. The developer can interactively explore analyses by creating probes for different program elements, and the results are updated live as the source code is edited, and even after updates to the analysis tool itself.

It is a challenge to match a probe for a particular property to the corresponding program element after source code changes, and we provide a robust algorithm for this purpose. Our approach also supports the probing of properties of implicit program elements that are not directly visible in the edited source code, e.g., imported libraries or predefined elements built into the programming language, like `class Object` in Java. We see property probes as a complement to traditional development support such as automated tests and traditional breakpoint/step-debuggers or “print debugging”.

We have implemented a property probe tool, CODEPROBER, specifically targeting analyses built with Reference Attribute Grammars (RAGs) [Hed00]. The attributes of an attribute grammar match the probed properties, and the interactive probing fits the demand evaluation used in RAGs. However, the concept of property probes can in principle be applied to any analysis that uses an abstract syntax tree (AST) as the spanning tree over its program representation, and that associates partial analysis results with nodes of the tree. We therefore expect the ideas to be useful for analyses built with a much wider range of approaches than RAGs.

We have applied CODEPROBER to a number of different languages and analyses implemented using the JastAdd metacompiler [EH07a] that supports RAGs and demand evaluation. In particular, we have applied it to ExtendJ [EH07b], a full Java compiler, and to IntraJ [Rio+21], an extension of ExtendJ that supports intraprocedural control-flow and dataflow analysis. Furthermore, we have used CODEPROBER in a course on compiler construction and in a course on program analysis.

Our contributions are as follows:

- We introduce the concept of property probes (Section 2).
- We present the CODEPROBER tool and the different kinds of property probes it supports (Section 3).
- We present a data structure *node locator*, used for robust identification of the probed AST nodes after changes of the source code (Section 4).
- We present the algorithms that are needed for implementing node locators.



Figure 1: Probe for the compile-time `constant` property of a selected addition expression.

We also present optimizations for improving performance and user experience when using node locators in larger projects (Section 5).

- We present the architecture of CODEPROBER, and the requirements an AST must satisfy to be used with CODEPROBER (Section 6).
- We present experiences from using CODEPROBER in case studies (Section 7), and performance measurements to explore its limitations with regards to the size of the edited code and number of active probes (Section 8).

Finally, we present related work in Section 9, and then conclude in Section 10.

This paper is an extension of a previous conference paper at SLE 2022 [RA+22]. Major additions include detailed descriptions of CODEPROBER (Section 3), algorithms (Section 5), and more extensive case studies (Section 7).

2 Property Probes

In this section, we explain what we mean by *properties* and present the concept of a *property probe*. We also briefly discuss different use cases for property probes.

2.1 Properties

We use the term *property* for a named compile-time value associated with an AST node, and computed by some compile-time (static) analysis. For example, an addition expression could have a property `int constant`, holding the compile-time integer value of the expression, as computed using constant propagation. Figure 1 exemplifies this. Here, an addition expression `x+2` has been selected, identifying an AST node of type `AddExpr`. Its property `constant` is shown, having the value `3`.

We distinguish between *intrinsic* and *computed* properties. An *intrinsic* property is part of the AST constructed by a parser or an editor, and is directly available without further computation. Examples include token values like variable names and literal values. A *computed* property is computed by an analysis of the AST. Examples include name bindings, types, and the `constant` property mentioned

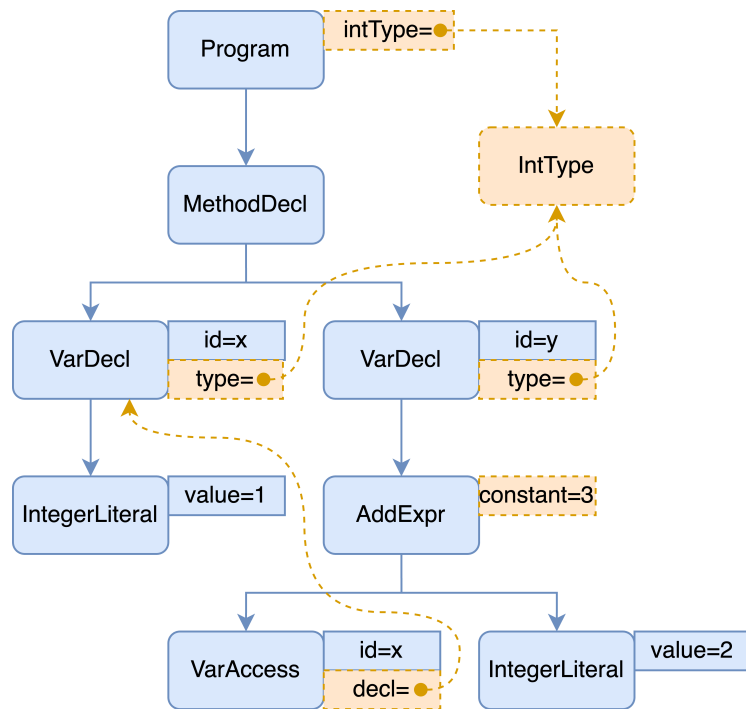


Figure 2: AST with intrinsic (solid blue) and computed (dashed orange) properties. `IntType` is a synthetic AST node. Straight lines are child edges, and curved lines are references.

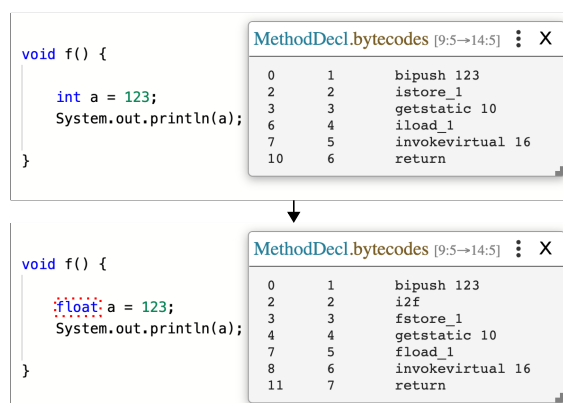


Figure 3: Probe for the bytecode of a method. The probe result is automatically updated after editing the type from `int` to `float` (dotted box).

above. A computed property may be *higher-order* in that its value can be a fresh AST subtree whose nodes may have their own properties. We refer to such computed AST nodes as being *synthetic*. An example is an AST node for a primitive type that is not present in the parsed program, but that is useful to reify as an AST node.

Figure 2 illustrates a (slightly simplified) AST with intrinsic and computed properties for the example in Figure 1. Here, intrinsic properties include the `id` properties of variable declarations and accesses, and the `value` property of an integer literal. Computed properties include `decl`, `type`, `constant`, and `intType`. Here, the `decl` of a variable access is a reference to an AST node representing the declaration, `type` is a reference to an AST node representing a type, `constant` is the constant value of an expression as discussed above, and `intType` is a reference to a synthetic AST node representing the primitive type `IntType`.

Optionally, a property can take arguments, i.e., serve as a function on a given AST node. An example could be a property `boolean visible(String id)` for a statement node, that returns `true` if there is a declaration named `id` visible at the position of the statement.

Different compiler and static analysis tools use different strategies to compute property values. For tools built using RAGs, the properties are computed on demand: if a client asks for a particular property of a particular node, that property will automatically be evaluated, and any other properties it depends on will be recursively evaluated, memoizing subresults for efficiency. After an edit, the memoized values can be thrown away and new results are recomputed when the values are asked for the next time. This often gives a very short response time after an edit, but the time of course depends on what property is being asked for. More traditional tools are often pass-oriented, traversing the complete program several times, computing all properties of all AST nodes. After an edit, the whole computation needs to be redone, thus giving long response times. We do not prescribe any particular strategy to be used, but assume that there is an automatic way to trigger computation of the properties, so that their up-to-date values can be presented.

2.2 Property Probes

A property probe is an interactive element, presented in the context of a source code text editor, and acting as a live observer of a property of an AST node. In CODEPROBER, each property probe is displayed as a small window. An example can be seen in Figure 1.

Internally, a probe is represented by a *node locator* (a way of identifying a particular node in the AST), a *property name*, optionally a number of *arguments* (if the property takes arguments), and a *result value*, i.e., the most recently computed value of the property. The result value is a collection of primitive values, like string or integer, and AST node references (represented as node locators).

A probe has two main responsibilities:

1. It adjusts the node locator after source code edits.
2. It presents up-to-date results to the user, reevaluating the property as needed.

Evaluating a property means, in practical terms, invoking a function in the context of an AST node. Keeping the result up-to-date means re-invoking the same function whenever needed, such as when the source code is modified. Figure 3 shows an example in CODEPROBER of how a probe is updated when the user edits the source code. In this case, the user has created a probe for the bytecode of a method. When the user edits the source code, changing the type of the variable from `int` to `float`, the bytecode in the probe result is updated, for example, changing the `istore_1` and `iload_1` instructions to `fstore_1` and `fload_1`.

The user can create a probe starting from a location in the text, selecting the desired AST node in case several nodes match the same location. It is also possible for a user to create a new probe starting from the result of another probe. This allows property exploration not only directly related to the edited text, but also by exploring probe results, which can again be explored further, supporting an interactive way of investigating partial results of an analysis.

Exploration of probe results opens for exploring properties of nodes that have no matching location in the edited source text. One example is nodes corresponding to ASTs of imported libraries. Another example is synthetic nodes created to represent implicit program entities. Examples include built-in types like the `IntType` in Figure 2 or `class Object` in Java, desugared representations of language constructs, and computed complex properties resulting from a static analysis. Attribute grammars can use higher-order attributes for computing synthetic nodes. A higher-order attribute is an attribute whose value is a new AST node subtree, and where these new nodes may themselves have attributes [VSK89b].

2.3 Use Cases

Property probes help understanding the inner data structures of compilers and program analysis tools through interactive exploration. We see a variety of situations when property probes can be useful, both in education and for production work. In education, they can help students understand core compiler concepts such as ASTs, name bindings and type checking, as well as program analysis concepts like control flow and data-flow analysis. In extending an existing compiler, for example to extend the supported language, property probes can be useful for understanding the existing functionality and internal APIs. For program analysis tools, there are similar use cases, when constructing a new analysis that build on existing ones. When there are bugs in a tool, property probes can be used for interactively pinpointing what computations are correct and which are faulty. Furthermore, property probes

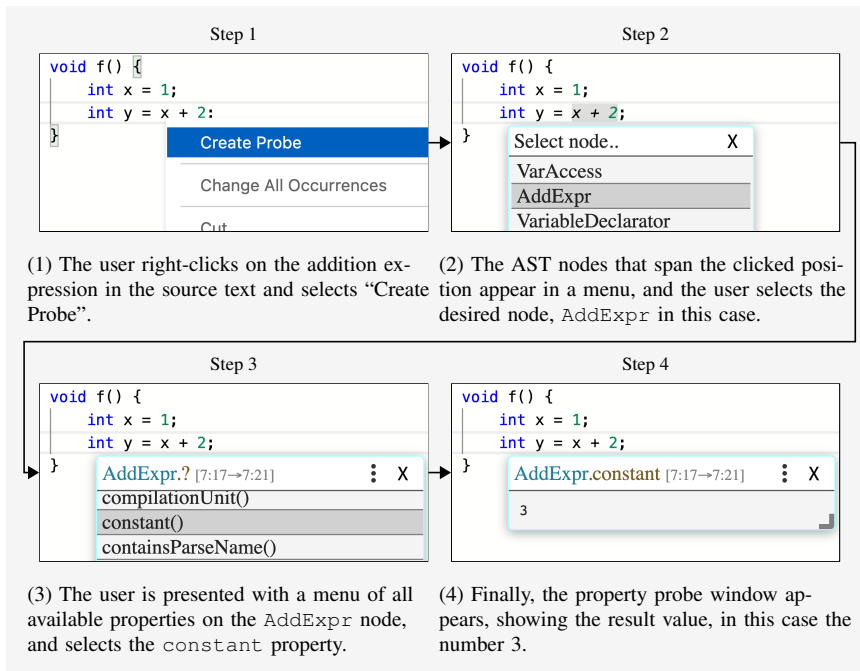


Figure 4: Steps to create a probe for the `constant` property of the addition expression `x + 2`.

can be useful for prototyping interactive language services, like code completion, semantic navigation, etc., by expressing the service data as computed properties.

Property probes do not replace ordinary control-flow focused debugging that uses step/breakpoints or print statements, since they do not address the order in which properties are computed. However, they provide a useful complement to such debugging, and might reduce the need for it.

We have used property probes in two university courses and in development of compilers and static analyzers, and have had overall positive results. This is discussed further in Section 7.

2.4 Tool Architecture

To implement property probes for a specific analysis tool, we propose a client-server architecture, see Figure 14. The client side uses a customizable code editor such as Monaco [Mic] or similar. The server side consists of two components; a server and an analysis tool.

It is the responsibility of the analysis tool to parse the edited text into an AST (as well as any other files needed for the analysis), and to populate the AST with



Figure 5: Example of a reference result. The `decl` property of the variable `x` has been probed, resulting in a reference to a `VariableDeclarator` node. The user hovers the result, causing the corresponding span in the source code (`x = 1`) to be highlighted in gray.

the functionality to be explored using probes. The server uses an API to access the analysis tool, e.g., to get the root of the current AST and to query properties on different AST nodes. The server also handles the communication with the client-side code editor.

The probes are stored on the client-side, using node locators as references to node objects. This allows the AST to be reparsed at any time, or even the analysis tool to be restarted from scratch during the editing session. It also allows the server to be completely stateless.

3 CodeProber

CODEPROBER supports property probes on an underlying compiler or analysis tool. In this section, we present its key features, using the ExtendJ Java compiler as the underlying tool.

3.1 Creating a Probe

The user can create a probe interactively via CODEPROBER’s text editor. Figure 4 shows an example. The user right clicks in the source code and selects the menu option “Create Probe” (1). A list appears, showing AST nodes that overlap with the clicked location. The user selects one of them (2). A new list appears, showing all properties available on the selected AST node. The user selects a property (3). A window (property probe) appears which shows the result of evaluating the selected property on the selected node (4). The user can click the position indicator (`[7:17→7:21]`) to get the same green highlighting as displayed in Figure 1.

3.2 Advanced Probes

The probe window created in Figure 4 is simple in that it shows a single property for a single AST node, and the probe result is displayed as plain text (the value 3). Based on our experience from using the tool, we have developed support for

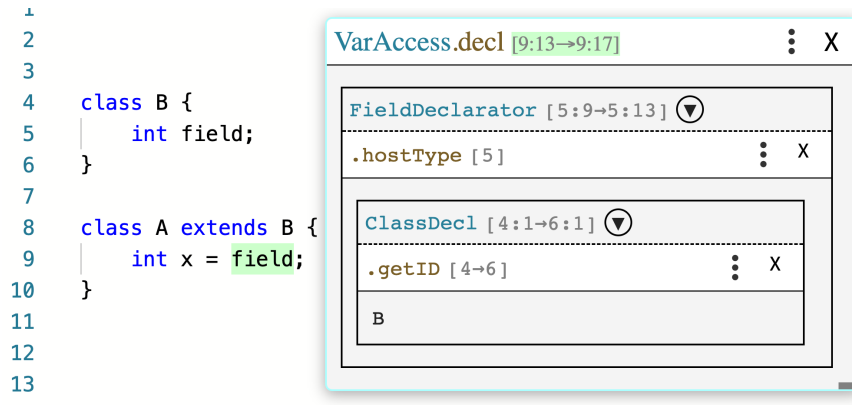


Figure 6: Nested probes to show what class the field variable is declared in. The user has first created a probe for field’s property decl, resulting in a FieldDeclarator. Then a nested probe was created, showing the FieldDeclarator’s hostType property, resulting in a ClassDecl. Finally, the user has probed its name (the getID property), which is B.

several more advanced kinds of probes: *nested probes* (local probes created from probe results), *search probes* (showing a set of properties), *AST probes* (visual presentation of the AST), and *probes contributing diagnostics* (that is shown in the source text). We will now discuss these in turn.

References and Nested Probes

A probe result can be a reference to an AST node. The user then can hover over the result to see the corresponding source text for that node. Figure 5 shows an example where the decl property of the variable access x is a reference to a VariableDeclarator node, highlighted in gray.

The user can explore a result reference by clicking on it to create a new probe for the referenced node. This can be done iteratively to follow chains of AST node references. Instead of creating a new window for the new probe, it can be nested inside the original probe. In addition to saving screen space, this retains the link between the probes, so that an inner probe will be re-evaluated on the result of the outer one when the user edits the program.

Figure 6 shows an example where a chain of nested probes is created to show what class the instance variable field is declared in. Starting at the AST node v for the field variable, the class name can be accessed via the chain of calls v.decl().hostType().getID(). The user has first created a probe for the decl property for the field variable, then a nested probe for hostType, and finally another nested probe for getID. If the user were to change field on line

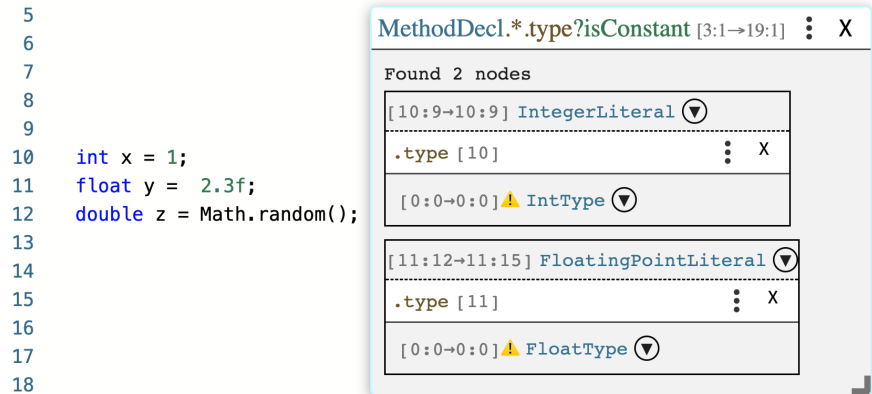


Figure 7: Search probe that finds all nodes under a `MethodDecl` where `isConstant` is true, and opens a nested probe for `type` on them.

9 to another variable, that would cause the nested probes to display potentially different values.

Search Probes

The user may want to see a given property for multiple AST nodes. CODEPROBER supports this through *search probes*. A search probe lets the user specify a query to find all AST nodes that pass a given filter. Figure 7 shows an example. Here, the query `MethodDecl.*.type?isConstant` selects all AST nodes in the subtree of a given method declaration for which the property `isConstant` is true, selecting two nodes in this case. Furthermore, the query includes the `type` property, which is shown as nested probes. If desired, the user can then investigate details of the probe results, using more nested probes.

Search queries don't have to specify both a property and a filter. For example, the query `*.type` on the root node would show the `type` property for all nodes in the whole AST. The query `?isConstant` would select all nodes that are constant, but not evaluate any property on them.

AST Probes

To select what nodes to create probes for, the user needs an understanding of the AST structure and the different AST node types used in the compiler or program analysis tool. To support this, CODEPROBER provides *AST probes* where the AST is rendered graphically. Figure 8 shows an example, showing the AST for the method call `f(x, y + 2)`. The nodes can be hovered to highlight their corresponding span in the source code. They can also be clicked to create new probes.

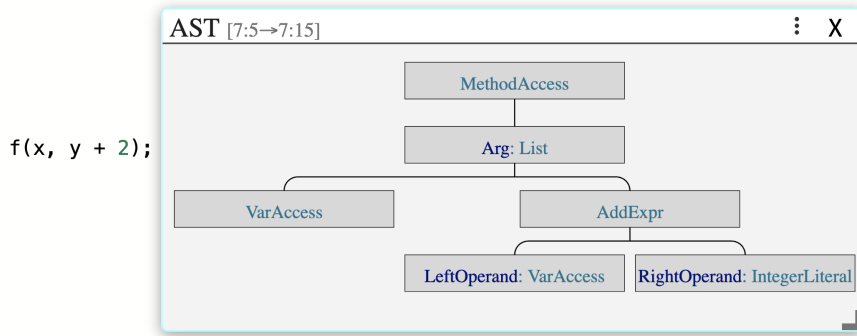


Figure 8: AST probe showing the AST of a method call. More probes can be created by clicking on individual nodes in the AST.

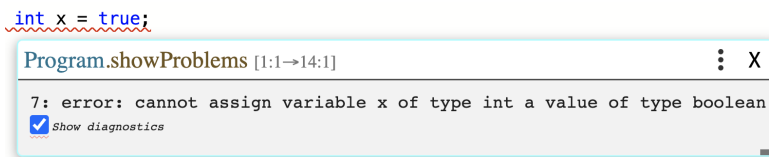


Figure 9: Squiggly line diagnostics contributed by a probe.

This can be useful when learning about the internal structure of a compiler.

Probes Contributing Diagnostics

A probe can be used for contributing diagnostics information that is displayed directly in the text editor, for example, squiggly lines, arrows, and hovering behavior. This can be used for prototyping language service support. Figure 9 shows an example of a probe for `showProblems`, a property holding a set of error messages. The error messages are displayed in the probe. In addition, each error message object contains information about where to place squiggly lines in the text editor. The visibility of the diagnostics can be toggled on and off with a checkbox (*Show diagnostics*).

Figure 10 shows another example where the property is a set of call graph edges. Here, each edge object contains information about where to draw an arrow in the text editor.

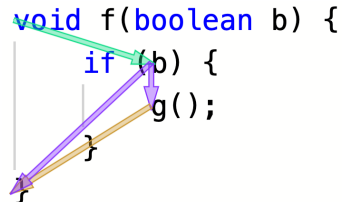


Figure 10: Arrow diagnostics representing a control-flow graph contributed by a probe. The colors are random.

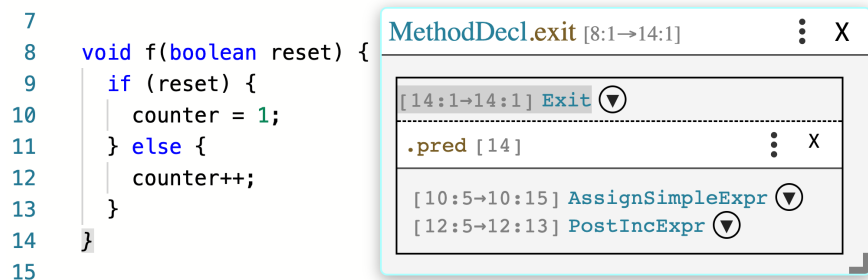


Figure 11: Probe on a synthetic Exit node of a control-flow graph, listing the predecessors of the node.

3.3 Liveness

CODEPROBER probes are live in that their results are updated after each edit to the source code, as was shown in Figure 3. Probe results are updated also if the underlying compiler or static analysis tool is rebuilt. These updates happen automatically, as CODEPROBER listens to changes in the file system, and can detect when the underlying tool has been replaced. This supports a tight development loop, for example, when the developer fixes bugs in the underlying tool. In principle, a modified underlying tool might imply that existing probes can no longer be matched. For example, if the abstract grammar is changed, the new AST structure might differ from the previous one, even if the source code is unchanged. CODEPROBER makes a best effort to match probes anyway, as will be discussed in Sections 4 and 5.

3.4 Synthetic Nodes

As discussed in Section 2, the value of a property might be a reference to a synthetic AST node, i.e., a node that is computed rather than constructed by the parser. Through the use of probes, these nodes can be investigated even if they do not have

any source code representation. Figure 11 shows an example from a control-flow graph analysis. The analysis constructs synthetic `Entry` and `Exit` nodes for each method so that the control-flow graphs have well-defined starting and ending points. In the figure, the user has created a probe for the synthetic `Exit` node, accessed via the `exit` property of the method. The user has also created a nested probe for the `Exit` node's `pred` property, to investigate its predecessors.

If a synthetic node is given artificial line and column information, this can be used for mapping the node to a source code position. In the figure, the analysis has set the line and column information for the `Exit` node so that it appears to be located at closing brace for the method. This allows the user to hover over the reference to the synthetic node, and see which method it belongs to.

4 Node Locators

As mentioned earlier, so-called *node locators* are used at the client side to identify the AST nodes referenced by probes. These node locators need to be updated after changes of the source code in the text editor. There are many potential ways to identify where an AST node is located. Some examples in plain English are:

1. “The call expression on line 12, column 9”
2. “The third child of the fifth child of the root AST node”
3. “The class declaration with ID set to ‘Foo’”

These ways of identifying nodes might work, but they are fragile to changes: The first example will break if, for example, a statement is added at the beginning of the source code; The second example will break if, for example, the construct of interest is nested inside a new statement; The third example will break if, for example, the class is renamed to ‘Bar’.

Furthermore, there can be probes on synthetic nodes that have no textual representation, and that require more sophisticated identification methods.

We have designed the node locators with the goal of making them both resilient to different kinds of changes of the source code, and efficient to apply, i.e., to resolve them to actual object references. Another design goal is that node locators should be as language agnostic as possible, and not make assumptions of how source text is parsed. This prevents potential solutions that rely on inserting tracking markers in the code, for example using annotations or block comments, as annotations and block comments are not supported in all languages. In addition, such tracking markers would not be possible to use with synthetic nodes.

Our current design is the result of an iterative development process where we have tried out probes on many different properties for several different analysis tools and for different languages. In our experience the design works very well in practice, although there will always be corner cases when node locators can

fail, for example when the corresponding code is completely removed. This is discussed in more detail in Section 4.5.

4.1 Node Locator Steps

A node locator is a list of *steps* where each step moves a current position to a new position in the AST. The steps are applied in order, starting at the root of the AST. Any of the steps can fail, in which case the whole application of the node locator fails, and no node could be identified. The following steps are supported:

Child has the form

$$\text{Child}(i)$$

It means go to the i -th child node.

TAL stands for *Type At Location* and has the form

$$\text{TAL}(t, d, l_s : c_s \rightarrow l_e : c_e)$$

Here, t is an AST node type, d is a number of steps down in the AST from the current node (the result of the previous step), and $l_s : c_s \rightarrow l_e : c_e$ is a line/column start/end span in the text. This step moves the current position to the “best” node of type t in the subtree of the current position and whose text span has at least one character overlap with $l_s : c_s \rightarrow l_e : c_e$.¹ If there are multiple compatible nodes in the AST, then they are sorted by how closely they match d and $l_s : c_s \rightarrow l_e : c_e$ (see Section 5.2 for implementation details). If multiple nodes are equally good matches, then the first one in a depth-first traversal is chosen.

FN stands for *Function* and has the form

$$\text{FN}(f, a_1, \dots, a_n)$$

where f is the name of a function on the current node, and a_1, \dots, a_n are arguments to the function ($n \geq 0$). The function is expected to return an AST node reference, and the current position is moved to that node.

The `Child` steps provide a simple way of locating a node in an AST, but it is not very resilient to changes. Even a small change, like changing something in the beginning of the source code, would result in old node locators failing or resolving to the wrong node in the new AST.

The `TAL` steps were introduced to provide a resilient solution. They can handle variations in the placement of the nodes due to additions, deletions, and nesting changes. The `TAL` steps also make use of text spans in the edited source code. For this to work, the editor needs to adapt the `TAL` text spans in its stored probes as the text is edited. If, for example, the user inserts a newline between lines N and M , then for all `TAL` steps on line $L \geq M$, the line count is increased by 1. Similar adjustments are made for lines and columns on all insertions and removals.

¹Nodes that are not given explicit spans by a parser should have the span $0 : 0 \rightarrow 0 : 0$ and are considered to overlap any other span.

The FN steps were introduced to handle synthetic nodes, constructed by the analysis tool in a different stage than parsing. In particular, they can be used for higher-order attributes (HOAs), in which case the function is simply the name of the HOA, and returns the root of the HOA subtree. In Reference Attribute Grammars, the HOAs are evaluated on demand and memoized, so in case the attribute had not already been accessed for other reasons, calling the function will result in the HOA being created before its root is returned.

The FN steps might be useful also for other purposes. They are very versatile as the function may return any node in the AST. One possible use might be to introduce application-specific steps, such as jumping to a particular source file or declaration node. However, we have not explored this possibility, since our main goal has been to provide an algorithm that works out of the box for any language and analysis tool.

4.2 Example Node Locators

In the Java compiler ExtendJ, the root AST node is of type `Program`. `Program` has a `List`, which in turn contains a number of `CompilationUnit` nodes, each corresponding to a single source file. Most AST nodes for a source file can be identified by first identifying a `CompilationUnit`, and then using a TAL step within that file.

As an example, assume we have the following variable declaration at line 5 in a given source file:

```
int a = 1;
```

An example node locator for the variable declarator (“a = 1”) is:

```
[ Child(0),  
  Child(131),  
  TAL("VariableDeclarator", 10, 5:13 → 5:17) ]
```

Here, `Child(0)` goes from the root AST node (`Program`) to the `List` node. `Child(131)` goes to the 132:nd `CompilationUnit`, which happens to represent the source file. “10” is the number of steps from the `CompilationUnit` down to the `VariableDeclarator` node. “5:13” is the starting line and column and “5:17” is the ending line and column.

For library compilation units, we rely on FN instead, since libraries are implemented using higher-order attributes in ExtendJ.

For example, to identify the `Integer` class, we would use:

```
[ FN("getLibCompilationUnit",  
    "java.lang.Integer"),  
  TAL("ClassDecl", 2, 0:0 → 0:0) ]
```

Here, the FN step represents the function call

```
getLibCompilationUnit("java.lang.Integer")
```

on the root node, and results in a `CompilationUnit` node. The TAL step then locates the `ClassDecl` two steps down from the `CompilationUnit`. The text span in this case is $0:0 \rightarrow 0:0$, which is expected for AST nodes that do not get created from a normal source file.

4.3 Kinds of Node Locators

There are two kinds of node locators that can be created: *naive* and *robust*.

Naive A list of steps corresponding to the path from the root down to the target node. For each edge on the path, the corresponding step is either a `Child` (for a normal child), or an FN (for a higher-order attribute).

Robust A naive locator where sequences of one or more `Child` steps are replaced with a single TAL step if possible, i.e., if applying it results in exactly the same node as the `Child` steps would.

The key difference between naive and robust locators is how good they are at locating their target node after the user has made changes to the source document. Naive locators are not good at handling changes, which is why we call them “naive”. Still, naive locators have their uses. Construction and application of naive locators is quite fast, so whenever a locator does not have to be resilient to changes, a naive locator is preferable. Both the input and output of a probe may be a collection of AST node references, which are represented by node locators. When a change happens, the input is used to re-calculate the output. Therefore, only the locators in the input need to be resilient to changes. The output nodes can be naive, which is good for performance.

Main scenarios when `Child` steps cannot be converted to TAL are when interacting with built-in or rewritten parts of the AST. Built-in AST nodes are often missing position information, i.e., their span is $0:0 \rightarrow 0:0$, so any two nodes at the same depth and type in the AST will overlap. Rewrites can also create nodes with overlapping position, for example due to multiple rewritten nodes taking the position from a single source node. For example of why overlaps are an issue, consider the expression $f(x, y)$. If both x and y have the same span, then a TAL cannot be used to reference the y node.

There is a third scenario where TAL cannot be used, and that is with ASTs that have content from multiple source files. Since TAL steps only consider line/column and not “file path” or similar, nodes from different files cannot be reliably differentiated. See Section 5.5 for how we solve this problem.

4.4 Adapting to Changes

As mentioned, the client keeps track of the active probes with their node locators, and adjusts the text spans of the TAL steps as the code is edited. However, the client does not have any knowledge of the syntax, so it only adjusts according to textual changes. The consequence is that the node locator might not fit exactly with the AST of the reparsed code. The flexibility of the TAL step usually makes it possible to find the node, but this also means that there can be a better, more exact node locator that should be used in the future. For this reason, when the server sends updated probe results to the client, it also sends the updated node locator for the probe.

As an example, assume we have the following source code:

```
if (x) {
    a();
}
if (x) {
    b();
}
```

The node locator for the first if-statement contains a TAL step `TAL("IfStatement", ..., 1:1 → 3:1)`. Suppose now that the user decides to clean up some duplicated code, so they remove the two lines in the middle. The source code now looks like the following:

```
if (x) {
    a();
    b();
}
```

Because of the removed lines, the client adjusts the TAL step to `TAL("IfStatement", ..., 1:1 → 2:9)`, covering only the first two lines (up to and including the `a()` statement). The client then informs the server that there are changes, and the server then sends back the updated probe result, together with a new more appropriate node locator with a `TAL("IfStatement", ..., 1:1 → 4:1)`.

4.5 Known Limitation: False Positives

The proposed design for node locators has a known limitation relating to false positives. In particular, when the code has been edited, it is not always clear which node a user would perceive as the best match. For example, assume we have the following expression:

```
a(b(c))
```

The user adds a probe for `b(c)`. The locator for that probe looks like this:

```
[ TAL("CallExpr", 7, 5:13 → 5:16) ]
```

Then the user changes the source code to:

```
a(c)
```

After the change, the probe will be re-evaluated on `a(c)`, since it is the only AST node that matches `CallExpr` and overlaps with the original TAL position. If the user wanted the probe to keep matching the first argument to `a`, then matching `a(c)` is a false positive. The user would rather match `c`.

One potential solution to this scenario is to make use of subtyping: The call `b(c)` has the AST node type `CallExpr`, and according to the abstract grammar, its parent expects any node of the supertype `Expr` at that position. By using `Expr` instead of `CallExpr` in the TAL step, the new expression `c` would match, solving the user's problem.

Another potential solution is to make the users intent more explicit in the locator, and include a FN step that selects the first argument to `a`. Such a locator could look like this:

```
[ TAL("CallExpr", 5, 5:11 → 5:17),  
  FN("getArgument", 0) ]
```

Node locators need to balance resilience and risk of false positives when deciding how strictly they should match nodes. We found that being strict with types and permissive with locations seems to work well. Being less strict with types (for example by using subtyping) could potentially introduce more false positives, even if it would help the specific example above. Shorter locators also seem to be more resilient, so extra steps like `FN("getArgument", 0)` should be avoided.

A more robust solution to the problem could be to use multiple kinds of locators simultaneously, and use some heuristic to pick the best result among them.

5 Node Locator Algorithms

Creating and applying node locators involves a significant amount of traversal in the AST. Doing so efficiently can be a challenge. In this section we present algorithms for node locator construction and application. We also discuss which algorithms have potential performance problems, and how to mitigate them.

5.1 Creating Locators

We will present two algorithms for creating node locators; `CREATENAIVELOCATOR` and `CREATEROBUSTLOCATOR`.

Algorithm 1 Naive node locator creation

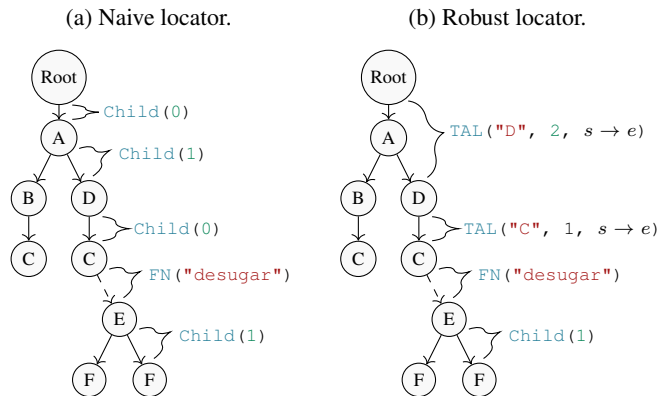
```

1: procedure CREATENAIVELOCATOR(target)
2:   ▷ Create a naive locator for the target AST node.
3:   list ← []
4:   node ← target
5:   while parent(node) ≠ null
6:     list ← [CREATEPARENTSTEP(node)] + list
7:     node ← parent(node)
8:   return list
9: end procedure

```

CREATENAIVELOCATOR is presented in Algorithm 1. It assumes the existence of a procedure CREATEPARENTSTEP(N) that returns a Child or FN step which represents how to get from the parent of N to N. A Child step can be used when the child is accessed by index, i.e., for most AST nodes. FN steps can be used for other AST nodes, for example children that are roots of HOA subtrees. CREATENAIVELOCATOR creates a locator consisting only of Child or FN steps by iteratively calling CREATEPARENTSTEP until it reaches the root of the AST.

Figure 12: Naive and robust node locator steps for the bottom-right “F” node.



CREATEROBUSTLOCATOR is presented in Algorithm 2. It steps through the result of CREATENAIVELOCATOR and tries to identify subsequences of Child steps that can be substituted with a TAL step. To explain the algorithm, we will show how a naive locator in Figure 12a is transformed to a robust locator visible in Figure 12b. The node labels A, B, ..., F indicate node types. CREATEROBUSTLOCATOR traverses the naive locator in reverse order and computes a shorter locator, where some Child subsequences are replaced by a TAL step. To keep track of the

current subsequence, it uses two pointers, *src* and *dst* that represent the source and destination of a potential TAL step. When *src* and *dst* point to the same node, it means that no TAL is in progress. Initially, they both point to the target node of the locator, i.e., F in Figure 12a. In each loop iteration, `CREATEROBUSTLOCATOR` does the following in order:

1. If a TAL step is currently being built ($src \neq dst$) and cannot be grown any further, finish building it and move *dst* up to *src* (lines 8 \rightarrow 11).
2. If we cannot start building a new TAL step, take the incoming (naive) step as-is. Also move *dst* up one step past *src* to avoid starting a new TAL step (lines 12 \rightarrow 14).
3. Move *src* one step closer to the root of the AST (line 15).

Finally, a TAL step is added from the root of the AST if needed (the final “ $src \neq dst$ ”, lines 17 \rightarrow 18).

The AST in Figure 12a contains multiple steps where a TAL might not be possible to use, depending on whether nodes have unique text spans or not. In the worst case, where all nodes have identical text spans $s \rightarrow e$, there are three such cases:

1. The bottom-right F node cannot be identified by a TAL since its left sibling would take precedence.
2. FN steps, such as between C and E, always prevent TAL steps.
3. The bottom right C can be identified by a TAL from its parent (D), but not from its grandparent (A). Such a TAL step would instead match the bottom-left C node.

The output of `CREATEROBUSTLOCATOR` can be seen in Figure 12b, where three `Child` steps are replaced with two TAL steps. If all AST nodes had unique spans, then the two TAL steps could be replaced by a single `TAL("C", 3, $s \rightarrow e$)` step. Figures 12a and 12b are intentionally constructed to showcase how multiple levels of overlap are handled. In most normal cases a single TAL step can cover a majority of the tree.

5.2 Applying Locators

Algorithm 3 shows the algorithm for applying a node locator. The key part is the procedure `APPLYTALSTEP`. It finds the best match for a TAL step in the subtree starting with *src*. When multiple potential matches are found, `ISBETTERMATCH` is used to determine which node is the best match. In case there are two equally good matches, then whichever one is found first is returned.

Algorithm 2 Robust node locator creation

```

1: procedure CREATEROBUSTLOCATOR(target)
2:   ▷ Creates a robust locator for the target AST node.
3:   locator ← CREATENAIVELOCATOR(target)
4:   src ← target
5:   dst ← target
6:   res ← []
7:   for each step in reverse(locator)
8:     if src ≠ dst and                                     ▷ Ongoing TAL?
9:       (step is FN or ¬CANEXPANDTAL(src, dst))
10:      res ← [CREATETAL(src, dst)] + res
11:      dst ← src
12:     if step is FN or ¬CANEXPANDTAL(src, dst)
13:       res ← [step] + res
14:       dst ← parent(src)
15:       src ← parent(src)
16:     end for
17:   if src ≠ dst                                           ▷ Ongoing TAL?
18:     res ← [CREATETAL(src, dst)] + res
19:   return res
20: end procedure
21:
22: procedure CREATETAL(src, dst)
23:   ▷ Create a TAL step that can be applied to src to get to dst. src is an
   ancestor of dst in the AST.
24:   t ← type(dst)
25:   d ← distance(src, dst)
26:   s ← span(dst)
27:   return TAL(t, d, s)
28: end procedure
29:
30: procedure CANEXPANDTAL(src, dst)
31:   ▷ Check if a TAL step from the parent of src would identify dst.
32:   expandedSrc ← parent(src)
33:   tal ← CREATETAL(expandedSrc, dst)
34:   return APPLYLOCATOR([tal], expandedSrc) == dst
35: end procedure

```

The *disjoint* check in APPLYTALSTEP exists to avoid traversing through subtrees that do not overlap with the TAL step. We assume that parent nodes fully cover all their children.

When comparing types, we currently consider only exact type matches. There

Algorithm 3 Node locator application

```

1: procedure APPLYLOCATOR(locator, astRoot)
2:   n ← astRoot
3:   for each step in locator do
4:     n ← APPLYSTEP(step, n)
5:     if n == null
6:       fail
7:   end for
8:   return n
9: end procedure
10:
11: procedure APPLYSTEP(step, node)
12:   case step of
13:     Child(i)
14:       return getIthChild(node, i)
15:     FN(fn, a0, ..., an)
16:       return invoke(node, fn, a0, ..., an)
17:     TAL(..)
18:       return APPLYTALSTEP(step, node, node)
19:   end procedure
20:
21: procedure APPLYTALSTEP(step, node, src)
22:   if span(node) ≠ 0 : 0 → 0 : 0
23:     and disjoint(span(node), span(step))
24:     return null
25:   if type(node) == type(step)
26:     best ← node
27:   else
28:     best ← null
29:   for each child in children(node)
30:     match ← APPLYTALSTEP(step, child, src)
31:     if ISBETTERMATCH(step, src, best, match)
32:       best ← match
33:   end for
34:   return best
35: end procedure
36:
37: procedure ISBETTERMATCH(step, src, lhs, rhs)
38:   ▷ Check if rhs is a better match than lhs when applying step from src.
39:   if lhs == null or rhs == null
40:     return lhs == null
41:   sl ← abs(span(step) - span(lhs))
42:   sr ← abs(span(step) - span(rhs))
43:   if (sl == 0) ≠ (sr == 0)                                     ▷ One perfect span match?
44:     return sr == 0
45:   dl ← abs(depth(step) - distance(src, lhs))
46:   dr ← abs(depth(step) - distance(src, rhs))
47:   if dl ≠ dr                                                 ▷ One side closer to ideal depth?
48:     return dr < dl
49:   return sr < sl
50: end procedure

```

are arguments for and against permitting subtypes here, see Section 4.5 for a discussion on this.

ISBETTERMATCH compares potential matches against an ideal match, which is determined by the span and depth values on the TAL step. Three criterias are considered in descending order of importance:

1. Perfectly matching span.
2. Least deviation from ideal depth.
3. Least deviation from ideal span.

If APPLYTALSTEP is changed to permit subtyping matches, then ISBETTERMATCH could also use type comparisons to select the best match. For example, a perfect type match might be less important than perfectly matching the intended span, but more important than the other criteria.

5.3 Optimizing Node Locator Construction

There are cases when the performance of CREATEROBUSTLOCATOR (Algorithm 2) can be a problem. To understand why, we must analyze the worst case time complexity of some of the procedures that are involved. Assume a node locator is being created for a node that is d steps down in the AST from the root, and the full AST contains N nodes.

APPLYTALSTEP

CREATEROBUSTLOCATOR makes use of the APPLYLOCATOR algorithm which in turn uses the procedure APPLYTALSTEP. This procedure iterates over every node that overlaps with the span of the TAL step. The time this takes depends on the shape of the AST, and the span information available on its nodes. If the AST is a balanced binary tree and each node has non-overlapping text spans, then this runs in $O(\log(N))$ time. In the worst case, however, the AST is shaped like an upside-down T (\perp) and each node has overlapping text spans. In this case, almost the entire tree will be visited when applying TAL from any point along the vertical part of the \perp . This means that in the worst case, APPLYTALSTEP runs in $O(N)$ time.

CANEXPANDTAL

The CANEXPANDTAL procedure invokes APPLYLOCATOR with a TAL step, so it also runs in $O(N)$ time in the worst case.

CREATEROBUSTLOCATOR

The main loop of CREATEROBUSTLOCATOR runs for d iterations. The first iteration can invoke CANEXPANDTAL once, and all subsequent iterations can invoke it up to 2 times. In the worst case, this means CANEXPANDTAL is invoked $(2 * d) - 1$ times. The total worst-case time complexity for CREATEROBUSTLOCATOR is therefore $O(d * N)$. Because $d = N$ in the worst case, the time complexity is quadratic with respect to the size of the AST.

Achieving Linear Time

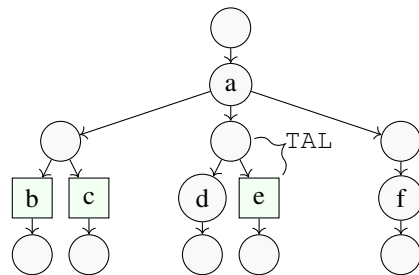
In practice, the performance of CREATEROBUSTLOCATOR is often good enough if implemented as Algorithm 2. The APPLYTALSTEP procedure skips over entire subtrees if their spans do not overlap with the TAL. In addition, APPLYTALSTEP does not search through “FN” connections in the tree, which further reduces the number of nodes that have to be visited. Still, performance can noticeably degrade when using locators in a larger context. In Section 8 we show benchmarks where probes update within tens of milliseconds for a project with a hundred thousand lines of code. Of those milliseconds, only a small fraction ($\sim 20\%$) comes from node locator related functionality. To get these numbers we had to optimize CREATEROBUSTLOCATOR. There are two relatively simple and very effective optimizations that can be done.

The first and most important optimization makes sure that nodes aren't visited more than once when CREATEROBUSTLOCATOR runs. The process of creating TAL steps involves iteratively applying TAL steps further and further up in the AST, as long as the application results in the expected target node. This can result in APPLYTALSTEP traversing through the same subtree multiple times. However, only the first time is necessary, and in all subsequent visits to the same subtree the outcome of visiting that tree is already known.

The second optimization uses the knowledge that there is a perfect TAL match at $depth(tal)$ steps down in the AST. Any node at a different depth cannot possibly be the best match, so APPLYTALSTEP can also stop upon reaching a depth further down than the expected perfect depth. Also, a better match can only be before the target node in a depth-first search (DFS), due to the DFS traversal in APPLYTALSTEP.

The optimizations allow CREATEROBUSTLOCATOR to avoid visiting large parts of the AST when creating or extending a TAL step. Figure 13 aims to illustrate these optimizations. A TAL step is being created for the target node e and is being extended one step upwards in the tree, to include node a . First, only nodes on the same depth need to be considered. Then, node d can be excluded since it has already been visited (when creating the current TAL step). Node f can also be excluded since it is after e in a depth-first search. Thus, only the nodes b , c and e need to be considered for a perfect match (e is a perfect match since it is the TAL target).

Figure 13: Visualization of which nodes might be matched by a TAL step if it expands one step. Green squares are the potential matches. All other nodes are impossible to match, which enables the optimizations described in Section 5.3.



The two optimizations brings the time complexity for `CREATEROBUSTLOCATOR` down to $O(N)$, even in the worst case of a ' \perp '-shaped AST with fully overlapping text spans.

5.4 TAL Adjustments

At the client side, `CODEPROBER` keeps track of the node locators of all probes, and adjusts spans of TAL steps when the users makes changes in the editor. Each span is represented by two positions; start and end. The two positions are adjusted independently.

`CODEPROBER` uses the Monaco text editor, which reports changes as a range of text being replaced with some new text. The cases of typing or deleting characters correspond to the replaced range or inserted text being empty. These cases are relatively simple to handle. The more special case of text being removed and inserted at the same time is more challenging.

For example, assume that `CODEPROBER` has a node locator containing a position for the b inside the expression $a + b$. The user has $c * d$ in their clipboard. They mark b and paste, and the resulting text is $a + c * d$. It is not immediately clear where `CODEPROBER` should move the position that previously pointed at b . The edit could be treated as two independent edits in a sequence; first a removal of b , and then insertion of $c * d$. In this case, the removal of b should move the position to the space just after the $+$ sign. The insertion of $c * d$ should then move the position up to the end of the inserted text, i.e the d . Another possibility, which `CODEPROBER` uses, is to try to retain the original position whenever simultaneous removals and insertions occur. In this case, it means that the position should end up at c .

Algorithm 4 shows `HANDLEEDITORCHANGE` which computes a new position based on an original position and a change event. It assumes that there are

two functions `HANDLEREMOVAL` and `HANDLEINSERTION` that adjust a single position based on just a removal or insertion. Based on their outputs it can detect simultaneous insertion and removal and decide whether the updated position should be used, or if the original position should be retained instead.

Algorithm 4 Handle simultaneous removal and insertion

```

1: procedure HANDLEEDITORCHANGE(pos, change)
2:   ▷ Adjust a position (pos) based on an change event (change) from Monaco.
3:   del ← HANDLEREMOVAL(pos, change)
4:   ins ← HANDLEINSERTION(del, change)
5:   if pos ∈ removedSpan(change)                                ▷ pos inside removal?
6:     and ins ≠ del                                              ▷ Was there an insertion?
7:     and ins > pos                                             ▷ Insertion larger than removal?
8:     return pos                                               ▷ If yes, retain original position
9:   return ins
10: end procedure

```

5.5 Handling Multiple Files

Most language tools work with multiple files, for example through imports or passing a list of source files on the command line. However, `CODEPROBER` assumes that all relevant information is available in a single AST. This means that the AST can be quite large, since it will typically include content from many files, and not only the edited text within `CODEPROBER`. It also means that nodes created from different files may have overlapping line/column spans. If no special measures are taken, this will lead to the following problems:

1. Accuracy of `TAL` steps will degrade due to the overlapping nodes.
2. Performance of creating and applying locators scales with the size of the AST, so a larger tree will be slower.
3. The list of nodes that is presented when creating a probe (see step 2 of Figure 4) might include irrelevant nodes from external files, due to span overlaps.
4. Adjustments to `TAL` steps in external files cannot be reliably done, as `CODEPROBER` only has access to change information for the single edited file.

We have chosen to solve the above problems by introducing an optional boolean property `externalFileRoot` for AST nodes. This can be set by the analysis tool when building the AST. When this property is defined on a node, it means that the node represents a file. If set to false, then the node is the root

of the edited file inside CODEPROBER. If set to true, then the file is any external file, for example an imported file. Several algorithms are then enhanced to take the `externalFileRoot` property into account, as will be described in the following subsections.

CREATEROBUSTLOCATOR

The issue of TAL having low accuracy across multiple files can be solved by not using TAL steps above files in the AST. Any node with `externalFileRoot` defined is assumed to represent a file. Therefore, `CREATEROBUSTLOCATOR` in Algorithm 2 can be enhanced to forbid TAL steps after a node with `externalFileRoot` is found. In more concrete terms, add a local boolean variable *foundFileRoot* to `CREATEROBUSTLOCATOR`, and update it in the main loop with:

- 1: `foundFileRoot` \leftarrow `foundFileRoot`
- 2: **or** `src` **implements** `externalFileRoot`

Then, add “`foundFileRoot or`” as a prefix to the two instances of “`step is FN`”. This stops any ongoing TAL step from being expanded further, and prevents any TAL from being built further up in the AST.

By not using TAL above files in an AST, the performance problems related to multiple files are solved as well. The performance of creating and applying TAL is strongly related to the size of the (sub-)tree where the creation/application starts. By forbidding TAL above files, the size of the (sub-)trees are limited to a single file.

Algorithm 5 Listing nodes that overlap with a position

- 1: **procedure** LISTNODES(`node`, `pos`)
 - 2: \triangleright *Get a list of all AST nodes from node and down that overlap with pos.*
 - 3: **if** `node` **implements** `externalFileRoot`
 - 4: **and** `invoke`(`node`, `externalFileRoot`) == `true`
 - 5: **return** []
 - 6: **if** `span`(`node`) \neq (0 : 0 \rightarrow 0 : 0) **and** `pos` \notin `span`(`node`)
 - 7: **return** []
 - 8: `res` \leftarrow [`node`]
 - 9: **for each** `child` **in** `children`(`node`)
 - 10: `res` \leftarrow `res` + LISTNODES(`child`, `pos`)
 - 11: **end for**
 - 12: **return** `res`
 - 13: **end procedure**
-

Probe Creation Node List

As was shown in Figure 4, when the user starts creating a probe, a list of nodes overlapping the clicked position is shown in a menu. Algorithm 5, LISTNODES, shows how this list is computed. It is relatively similar to APPLYTALSTEP in Algorithm 3, in that it recursively traverses through the AST and has an early return condition if the position looked for is outside the span of the currently examined node. To avoid descending into nodes for an irrelevant file, an additional return condition is added, checking if the examined node has the `externalFileRoot` property set, and with the value `true`. In this case, an empty list is returned.

CreateTAL

To make TAL step adjustments more reliable for external files, we extend TAL with a new boolean field, `external`. This field should be set to `true` whenever the target node of a TAL, or any of its ancestors, has `externalFileRoot` set to `true`. In more concrete terms, replace the last line of CREATETAL with:

```
1: e ← ISEXTERNAL(dst)
2: return TAL(t, d, s, e)
```

ISEXTERNAL recursively searches upwards in the AST for the closest implementation of `externalFileRoot`:

```
1: procedure ISEXTERNAL(node)
2:   if node == null
3:     return false
4:   if node implements externalFileRoot
5:     return invoke(node, externalFileRoot)
6:   return ISEXTERNAL(parent(node))
7: end procedure
```

CODEPROBER's client does not adjust the values on a TAL step with `external` set to `true`.

6 Implementation

The overall architecture of CODEPROBER has three components: a client, a server, and an analysis tool, see Figure 14. The implementation is open source and available on <https://github.com/lu-cs-sde/codeprober>. This section describes the three components and the API between them, together with rules and recommendations on how the AST produced by the analysis tool should work.

6.1 Overall Architecture

In CODEPROBER, the client is a web page, mostly written in TypeScript, and using the Monaco code editor [Mic]. The server is written in Java, and can use

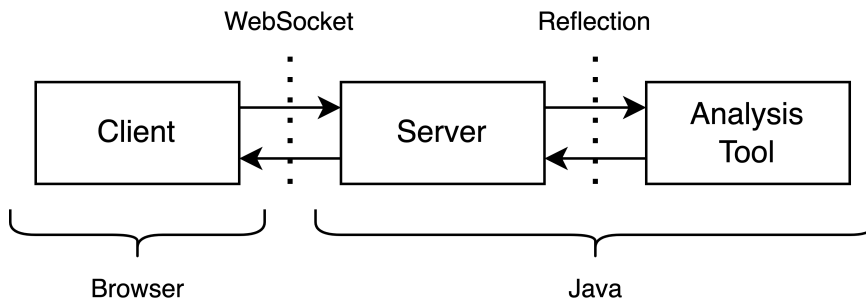


Figure 14: High level architecture

any analysis tool written using the JastAdd metacompiler (packaged as a jar file, following certain conventions).

For practical purposes, CODEPROBER packages the client and the server together as a single jar file which takes a path to the analysis tool as its argument. When started, CODEPROBER opens a local HTTP server that serves the webpage, and a local WebSocket[MF11] server for all dynamic requests. The user can then simply go to the webpage and start editing and creating probes.

6.2 Client↔Server API

CODEPROBER's client is a browser application, so the options for inter-process communication with the server is quite limited. CODEPROBER supports HTTP requests and WebSocket, but default to using WebSocket for its better performance. There are cases where HTTP requests are preferable, see Section 7.6. The client and server communicate with remote procedure calls (RPC). There are three different request types sent by the client: LISTNODES, LISTPROPERTIES and EVALUATEPROPERTY, corresponding to the three steps when the user creates a probe, as in Figure 4. The client sends the EVALUATEPROPERTY request also when the user edits the text, as in Figure 3.

There is one message sent from the server to the client, REFRESH. This is sent when the server detects that the underlying analysis tool has been updated. The client is not expected to respond to this message, but will instead re-evaluate all active probes by sending EVALUATEPROPERTY requests.

In all requests sent from the client to the server, the client includes the current editor state, i.e., the full text. The server will then ask the underlying analysis tool to parse this text into a new AST. Sending the full text in each request allows the server to be stateless. For performance, the server can, however, cache the latest text used for parsing, to avoid reparsing if the text has not changed. With this optimization, we have not seen that sending the full text in each request gives any

performance problems, even if the text is large. Any imported files can be similarly cached. Additionally, it is possible to buffer changes until a period of inactivity has passed, see Section 7.6.

For responses to requests on node locators, the server includes updated node locators in the response, based on the new AST. The client then uses the new locator in subsequent requests. This way, the node locators on the client side are constantly adapted to the most recent AST, as was discussed in Section 4.

Example. Figure 15 shows a sequence diagram of the messages sent when the user creates the probe in Figure 4. In the first step (1 → 2), the user clicks in the text to create a probe. The client then sends the `LISTNODES` request, with the full text and the cursor position as arguments. In response, the server sends a list of node locators, corresponding to the nodes that match the cursor position.

In the second step (2 → 3), the user selects one of the nodes. The client then sends the request `LISTPROPERTIES`, with the full text and the node locator as arguments. In response, the server sends an updated node locator NL'_2 , along with a list of property identifiers, each containing a property name and its argument types.

In the third step (3 → 4), the user selects one of the properties. The client then sends the request `EVALUATEPROPERTY`, with the full text, the node locator NL'_2 , the property identifier and any arguments to the property. (If the property has arguments, the client prompts the user to supply them interactively.) In response, the server sends the updated node locator NL'_2 , as well as the probe result. Any AST nodes in the result are encoded as node locators that can be used for future `LISTPROPERTIES` requests.

For the scenario in Figure 3, the client will send one `EVALUATEPROPERTY` request for each of the active probes.

6.3 Server ↔ Analysis Tool API

The concept of property probes can in theory be used for analysis tools implemented in any language. `CODEPROBER` is written in Java and currently requires the analysis tool to run on the JVM. To use `CODEPROBER` with analysis tools not running on the JVM, a bridge implementation running on the JVM is needed. In our experiments we have used analysis tools running on the JVM. Most are implemented with the `JastAdd` metacompiler.

The server communicates with the analysis tool using reflective calls. Heavy use of reflection can have a negative effect on performance, but this is not a problem for `CODEPROBER`, as seen in Section 8.

The AST is parsed or reparsed by calling the `main` method on the analysis tool jar file, with the path of a temporary file containing the client state (the full text). The `main` method should store the parsed AST in a static field in the main class, that should be declared as follows:

```
public static Object CodeProber_root_node;
```

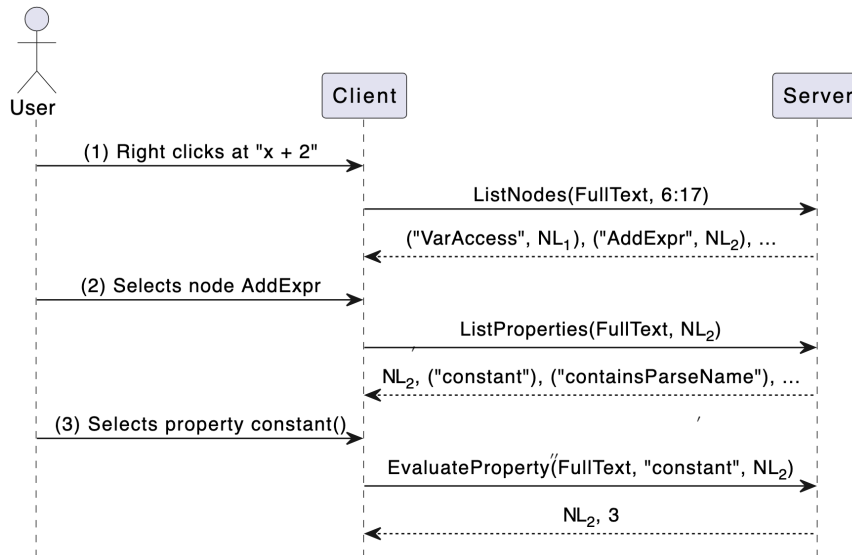



Figure 15: Sequence diagram for Figure 4

Once an AST is produced, the server uses reflection to access and traverse it. The server assumes that the AST follows a certain structure. A number of methods should be available on each AST node for traversal purposes:

1. `getNumChild()` - returns the number of children on this node.
2. `getChild(int)` - returns a child at a given index.
3. `getParent()` - returns the parent node for this AST node, or *null* for the root AST node.
4. `getStart()` / `getEnd()` - returns the start/end position as line/column pairs for this AST node.

When listing available properties, the default implementation is to again use reflection, via `java.lang.Class.getMethods()`. The full list is filtered before being returned to the client. Methods that are non-public are removed. Methods with arguments can only contain arguments of type `int`, `boolean`, `String` or AST node references. Methods with other argument types are removed.

To compute node locators, it must be possible to determine the connection between a parent and child AST node in the form of either a `Child` or `FN` step.

`Child` steps are determined by iterating over all children in the parent node. Using identity comparison, we can detect the index of the child node.

If a node is a higher-order attribute (HOA), an FN step should be used, including the name and the arguments of the HOA. This can be determined thanks to the fact that JastAdd memoizes the arguments to, and results of, all HOA invocations. To construct an FN step for a HOA, CODEPROBER selects the parent of the HOA, and then iterates through all the parent's HOA memoizations, again using identity comparison to find the name and arguments of the appropriate child HOA.

In case no parent/child connection can be determined, the child node is considered to not be attached to the AST. This causes node locator creations to fail, and the server sends an error code to the client.

6.4 Desirable AST Features

To use property probes, some design choices for the analysis tool are desirable to help improving the user experience: good source locations in the AST, on-demand evaluation of properties, and pure properties. We will discuss these in turn.

Source locations. For property probes to work well, it is desirable that the parser captures line and column positions and stores them in the AST nodes at parsing. Also, the positions should honor the AST hierarchy: CODEPROBER assumes that a node with an explicitly set position has an equal or larger span than all nodes in its subtree. Otherwise, our TAL algorithm might miss the best matching node, since it uses positions to prune subtrees in its search.

In many tools, nodes do get appropriate positions, in order for the tool to be able to report locations of errors and warnings. However, for a given tool, there might be nodes that lack this information. For instance, locations may have been added only for nodes with associated error messages. Another reason might be that the AST has been transformed, without carrying location information over to the transformed parts.

Missing locations will degrade the user experience, as features like highlighting and right clicking to select AST nodes will not work. In the client of CODEPROBER, a small warning triangle is shown next to each AST location that has its line and column set to zero.

However, CODEPROBER also tries to compensate for missing location information. It uses a *position recovery strategy* to infer suitable location information in case it is missing. There are multiple supported strategies, and the user can select which (if any) to use. The default strategy is called RECOVERZIGZAG and is shown in Algorithm 6.

RECOVERZIGZAG looks at nearby parent and child nodes, progressively searching further up and down in the AST until an explicitly set location is found, and which is then used as a replacement position. A recovered position usually covers a slightly larger or smaller span than the real span of the AST node. Therefore, position recovery should be seen as a temporary solution, and it is better if the analysis tool is updated so that all AST nodes carry their own position instead.

Algorithm 6 Default position recovery strategy

```

1: procedure RECOVERZIGZAG(node)
2:   ▷ Find a replacement position for node from its parent and children.
3:   up ← node
4:   down ← node
5:   while up ≠ null or down ≠ null
6:     if up ≠ null
7:       if span(up) ≠ (0 : 0 → 0 : 0)
8:         return span(up)
9:       up ← parent(up)
10:    if down ≠ null
11:      if span(down) ≠ (0 : 0 → 0 : 0)
12:        return span(down)
13:      down ← firstChild(down)
14:    return (0 : 0 → 0 : 0)
15: end procedure

```

On-demand evaluation. The user can create probes for any property the AST supports, but usually only a tiny subset of the functionality is ever probed for at the same time. This fits well with on-demand evaluation. Rather than computing all properties up front, it is advantageous if properties are lazy and their values computed only when demanded. On-demand computation is not strictly necessary, but if all potentially probed values are computed up-front, as soon as the source text is edited and the AST is reparsed, re-evaluation of all these values might take a long time and negatively impact the user experience. Of course, if the user does not edit the source text, but only explores properties, the probes can still be very valuable for tools that do up-front computations.

Pure properties. The probed properties should be observationally pure, i.e., without visible side-effects when accessing them. If accessing properties has side-effects, they may behave differently depending on in which order they are invoked, and the benefits of property probes then diminishes. In addition, there is a caching setting in CODEPROBER that greatly improves performance by reusing the AST whenever possible, to avoid unnecessary reparsing. If properties can cause changes in the AST, then caching is not reliable.

All our evaluations have been performed with JastAdd-based tools, where all property evaluation is on-demand and all properties (attributes) are observationally pure.

6.5 Program Representation

In this paper, we have assumed that the program representation is an AST. However, it is sufficient if the representation has a spanning tree, with the traversal

interface discussed above, and where source text locations can be attached to the nodes in the spanning tree. In fact, this is the case for JastAdd tools: Many of the JastAdd attributes are node references, so the program representation is actually a graph, but with the AST as the spanning tree.

We have also performed brief experiments with non-JastAdd tools like SpotBugs [Spo], PMD [Pmd] and WALA [Wal]. SpotBugs and WALA both perform their analyses on the bytecode level, which is common among analysis tools for Java. Bytecode level tools can also be used with CODEPROBER, but the experience is slightly inferior. This is because the text editor in CODEPROBER shows source code, but the AST that the user interacts with is that of the bytecode, which is not as intuitive. Still, we were able to explore the properties of those tools, which shows that CODEPROBER is not necessarily restricted to tools that put analyses directly on top of source code AST's.

7 Case Studies

We have performed a number of case studies of CODEPROBER in order to qualitatively evaluate its usefulness, and find opportunities for improvements. In particular, we have run it on a full Java compiler, on a tool for intrafLOW analysis of Java, and on compilers for several other smaller languages. We have used the tool in teaching two different courses (compiler construction and program analysis). Furthermore, we have deployed it on a cloud server, so it can be run directly in the browser without the need for installing any local tools.

7.1 Java Compiler

During development of CODEPROBER we continuously tested its functionality on the Java compiler ExtendJ. Based on this experience, we added a few features that we think are useful also for many other analysis tools.

For example, the *position recovery strategy* mentioned in Section 6.4 was added specifically because some node types in ExtendJ do not carry their own position information. The support for *multiple files*, described in Section 5.5, was also added based on our experience from ExtendJ.

During development we also identified and fixed two different caching issues in ExtendJ, which we contributed back to ExtendJ. We hypothesize that these issues had not been discovered before because ExtendJ had not before been used in the live, incremental way that CODEPROBER uses its underlying analysis tools.

In general, our experience is that using property probes with ExtendJ has been an excellent way of building understanding of the compiler. One common use case for ExtendJ is to write program analysers or experimental extensions to Java. To do this you need to know what AST node types are available, and what properties they contain. This can be accomplished by consulting the official API documentation [Ext]. However, we soon found ourselves using CODEPROBER more often

than the documentation. For example, if you want to know what properties are available on a *for each* statement, then you can write such a statement, right click on it and see the list of properties. If any property looks interesting, you can click it to immediately see how it works. With the API documentation you need to first find the name of the node (`EnhancedForStmt`) and then you get a list of property names, but these cannot be directly invoked since there is no concrete code attached.

7.2 Flow Analysis

IntraJ [Rio+21] is an extension to ExtendJ that adds intraprocedural control-flow and dataflow analysis. The main developer of IntraJ used CODEPROBER in developing new types of flow analysis. Before using CODEPROBER, the IntraJ developer had used “print debugging” when developing new analysis features:

- Write code for the new feature.
- Add print statement(s) to check that the feature works correctly.
- Iteratively modify the code and print statements until you get the expected behavior.
- Remove the print statement(s).

Now, property probes have replaced most of the “print debugging” steps, since it is much faster and simpler to open/close probes than it is to add/remove print statements and recompile IntraJ.

The IntraJ developer also mentioned that they use the ExtendJ API documentation less, since it often is quicker to explore functionality via the property probes in CODEPROBER.

One new feature was added specifically for IntraJ, *arrow contributions*. This is a feature that allows property probes to contribute arrows to be drawn between two positions in the source code, overlaying the text. IntraJ uses this feature to visualize the control-flow graph directly in the source code. Previously, the control-flow graph was usually inspected in textual form, which was inconvenient, or using a `dot` visualization of the AST with control-flow edges, which became very large even for a small program. The visualization using arrows is shown in Figure 10. The idea was inspired by the bug explanations in Clang Static Analyzer [Llv].

7.3 Compilers for Other Languages

We tried out CODEPROBER with compilers for a number of different additional languages: Oberon-0 [FH15] (a very tiny procedural language), Bloqqi [FH16] (a visual language for automation), and SimpliC (a simple C-like language, used in teaching). The experience worked well for all compilers. We did, however,

discover that each of the compilers had a few AST nodes that did not carry correct location information. Nodes that produced errors/warnings generally had correct locations. Missing locations were usually attributed to either desugaring or that multiple nodes were created by the same parser production (the parser generator attached location only to the return node of a production). This was no major issue, however, since the *position recovery strategy* developed for ExtendJ worked fine here, as well.

We also used CODEPROBER with a student implementation of a ChocoPy compiler. ChocoPy is a subset of Python, commonly used for educational purposes [PSH19]. Here we had more severe location-related issues. One of the challenges with parsing Python is the indentation sensitivity. The student ChocoPy implementation we used had solved this by making the parser a two-step process; first it transforms all indentation into whitespace-insensitive indentation tokens, and then it parses the transformed source code. AST nodes produced from the transformed sources always had line numbers that matched the original source code, but their columns were usually wrong by a few characters, since the whitespace-insensitive tokens did not match the width of the original indentation. It was possible to create probes, but the user experience was not very good. Of course, these problems could easily have been solved by fixing the student ChocoPy implementation to set proper line and column numbers, but it illustrates the kinds of problems a user can run into.

7.4 Compiler Course

We used CODEPROBER in a course on compiler construction which is taken by around 70 students each year. In the course, the students build a compiler for a C-like language, step by step over six lab sessions. The first two labs cover scanner and parser generation. The remaining four labs cover name analysis, type analysis, call graphs, and code generation, all using the JastAdd metacompiler. For most students, this course is their first encounter with terms like “grammar”, “abstract syntax tree”, etc.

For many years, an AST exploration tool, DrAST [LTH16], has been used successfully in the course, allowing the students to visualize the AST of a program, and to look at attribute values of individual nodes. For the 2022 edition of the course, we introduced CODEPROBER as an additional opportunity for the students to use, and gave a brief introduction to the tool early in the course. After each of the lab sessions, we performed short informal interviews with the students, to ask about their experiences with both DrAST and CODEPROBER.

The students’ tool preferences fell into three main groups of roughly equal size. Either they used CODEPROBER, DrAST, or neither of the tools. Very few students used both tools.

The students who preferred using CODEPROBER generally liked the ease and speed of testing their compilers: Adding/removing code and probes could be done

much quicker than, for example, writing test cases. While this was positive feedback, it also highlights one downside we noticed with CODEPROBER: it can lead to students writing fewer ordinary unit tests. Because it is so quick and convenient to open a probe, the students did not feel motivated to write the tests. They had the opinion that if something breaks in the future, it is easy to just start probing again. While this may work in the short term, having normal regression tests help in the long term. To improve on this, we plan on adding a *test probe* feature in CODEPROBER, i.e., a mechanism that allows a probe to be saved as a test case. We think such a feature can provide a very convenient way of constructing test cases.

A common point of feedback from those students that preferred DrAST was that they liked being able to see the AST visually. When working with tools like JastAdd, students need to have a good understanding of what the AST looks like, to know where and how to declare attribute equations. Even when the students had understood the concept of an AST, it was helpful to them to see the individual nodes visually for a given example. AST Probes (see e.g. Figure 8) were added based on this feedback. The implementation is inspired by the DrAST view.

The students who did not use either tool said they did not “see the point” of using either tool. They used more traditional methods to aid the development, like test cases and “print debugging”. When students asked for help during the labs, we sometimes asked them to inspect a few properties inside CODEPROBER. We needed to guide them step-by-step on how to do this. Afterwards, we noticed that some of those students started using CODEPROBER much more. This tells us that there is a barrier to getting started with CODEPROBER, and we need to work on improving usability for first-time users. We believe the portion of students using CODEPROBER will be larger in next iterations if we add some of the planned improvements.

7.5 Program Analysis Course

The program analysis course included labs on type inference, interval-based dataflow analysis to detect array out of bounds errors, and type analysis using points-to analysis. For these labs a small teaching language, TEAL, is used, and implemented using the JastAdd metacompiler. The core language implementation is provided as part of the course material, so the students only need to implement the analysis parts. The course leader forked CODEPROBER and modified it to be a more special-purpose tool for this course. The fork had a number of small changes, such as adding syntax highlighting for TEAL and hiding some of the options in the tool that were not necessary for the course. However, one of the main changes was the support for *background probes*, which are probes that are always on, and whose results are presented via squiggly lines at various points of interest in the code, and supporting extra hover information. A unique set of background probes was then configured for each lab. Figure 16 shows an example

```

1  fun f(arg) = {
2    | var x: int := arg;
3  }
4  fun g() = {
5    |
6    |
7    |
8    | f("Hello");
9  }

```

Type Mismatch: int \neq string

[View Problem \(\F8\)](#) No quick fixes available

Figure 16: CODEPROBER being used in a program analysis course.

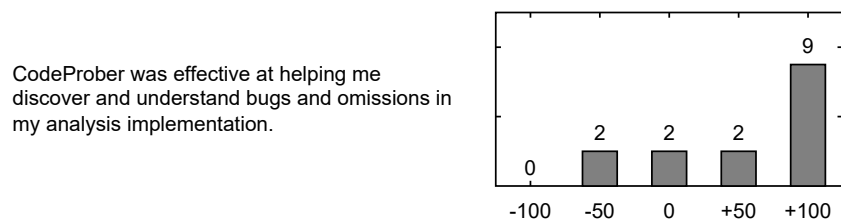


Figure 17: Result from course evaluation from a program analysis course. +100 is “fully agree”, and -100 is “fully disagree”.

of this from a lab on type inference. The figure contains several places with hoverable information that are extracted from the students’ analysis tool by the background probes. Hovering over the tiny dots (...) will show type equations for the related program element. Yellow squiggly lines appear wherever the type equations conflict and produce a type mismatch. Hovering over the lines, as is done in the figure, shows details about the mismatch. As the students progressed through the lab, more dots and yellow squiggly lines appeared. Whenever type equations or type mismatches seemed wrong, students could resort to normal, manually created probes to investigate why. The preconfigured background probes made it easy for the students to get started, and is definitely something we will use in the future.

A few analyses the students wrote needed to run in a loop until some value converged. Implementation mistakes could lead to infinite loops, which manifested in CODEPROBER as a probe loading forever. This is an area where CODEPROBER can be improved. For example, each probe can in theory be evaluated in a separate

process, and that process can be killed after a certain timeout. The server could also periodically sample and report stack trace information back to the client, to give hints as to where the loops/deadlocks are occurring.

Every student used CODEPROBER to some degree. It was also through CODEPROBER that teachers reviewed the student implementations at the end of the labs. However, some students only used the preconfigured background probes, and did not explore any extra properties on their own. We suspect this group of students has some overlap with those who used neither DrAST nor CODEPROBER in the compiler course.

It was not originally planned for CODEPROBER to become a special purpose tool like this, but the general feedback from students was positive: In an anonymous course evaluation questionnaire completed by 15 students, 9 agreed strongly with the statement that CODEPROBER was effective in helping them discover and understand bugs and omissions in their analysis implementations (see Figure 17). To support use cases like this, CODEPROBER will merge in some of the fork's changes, and add more configuration support so that it can be adapted to future use cases without needing to fork the entire repository.

7.6 Cloud Server

The normal way of running CODEPROBER is to run the server on the local machine, and browse to localhost to run the client. We were also interested in hosting the CODEPROBER server on a cloud server so that users can run the client directly on the web, without installing any local software. This can be very useful for demonstrations and also for providing playgrounds for users before deciding to install it on their own computer.

As an experiment, we decided to adapt CODEPROBER to run in GitHub Codespaces. Codespaces is a service that hosts development environments in the cloud, and allows you to connect to them directly in the browser. GitHub Codespaces was made generally available in 2022². Anybody with a GitHub account can try it for free, which makes it a good candidate for running demonstrations. A few minor changes were made to CODEPROBER to make it run well in Codespaces. The most notable ones were to delay requests and to make WebSocket optional.

Delayed Requests

We noticed that during heavy use, the requests to Codespaces would sometimes fail. When inspecting our server logs we couldn't see any trace of the request that failed. Our guess is that Codespaces has some hidden throttling limits. To overcome this limitation we added a small delay to probe updates when running

²<https://github.blog/2022-11-10-whats-new-with-codespaces-from-github-universe-2022/>

in Codespaces. Instead of immediately updating when the user makes a change, the client will wait for a period of inactivity before sending update requests to the server. The delay defaults to 0.5 seconds. After adding this delay we haven't had any issue with random request failures.

Optional WebSocket

The WebSocket connection between CODEPROBER's client and server is normally able to stay connected indefinitely. When running in Codespaces however, it automatically disconnects after a few minutes of inactivity. The exact inactivity time varies, but seems to be around 1 to 5 minutes. During normal use, CODEPROBER might be idle in the background for several minutes while the user is working on their tool, and only occasionally will the user come back to inspect something in CODEPROBER. This usage pattern does not work well with the automatic WebSocket disconnections.

To overcome this issue we added support for sending WebSocket-related requests as HTTP PUT requests instead. We replaced server-initiated messages (e.g. REFRESH in Section 6.2) with long polling. The result is a slightly slower time per request, since each request has to establish a new connection. The upside is that this allowed CODEPROBER to stay idle in the background for a long time without disconnection issues.

7.7 Summary

The case studies show that property probes are useful for a variety of analysis tools and different languages, both for development and in teaching. By running CODEPROBER for a full Java compiler, we made sure it works for real program analysis development. By running it in courses, we found that students appreciated the tool, and used it voluntarily to solve problems, which we take as an indication of its usefulness. Furthermore, by trying it out in many different scenarios, we have found several opportunities for adding new useful support. Examples include the position recovery strategy (fixing missing line and column information), support for multiple files (necessary for big projects), arrow diagnostics (visualizing graphs on top of the code), AST probes (visualizing the AST in a probe), and background probes (that are always active, and manifested directly in the code as squiggly lines). We also successfully adapted CODEPROBER to run on a cloud server, allowing users to try it out without needing to install any software.

8 Performance Evaluation

In the previous section we saw examples of CODEPROBER being used for a few different tools and in teaching scenarios. These examples show that CODEPROBER works well in practice, at least in smaller contexts. In this section we investigate

the performance of CODEPROBER for larger projects, to ensure that the response time is low enough for interactive use also in these cases. We are interested in finding out the administrative overhead of creating and evaluating probes, how it relates to other overhead like parsing, and how these overheads scale with project size.

8.1 Methodology

The processing time it takes to use property probes can be divided into three main parts:

- Parsing
- Property evaluation
- Probe administration

The processing times for parsing and property evaluation depend on the underlying analysis tool, which is ExtendJ in our benchmarks. As mentioned earlier, ExtendJ is a full Java compiler, implemented using the JastAdd metacompiler, which uses on-demand evaluation for properties. Different properties may naturally take different time to evaluate. Since we are interested in investigating the probe overhead rather than property evaluation, we have chosen simple properties with constant evaluation time in our setup.

To evaluate properties, the code needs to be parsed into an AST, and reparsing is needed whenever the user edits the code. Initially, all source files on the source path will be parsed, including the one edited in CODEPROBER. Additional imported class files are parsed on-demand, depending on what properties are evaluated. ExtendJ has support for incremental parsing at the file level, so when the user edits code, only the edited file is reparsed, and ASTs for other files on the source path are reused. As part of the parsing cost, we also count the administrative cost of flushing memoized properties—their values might be inconsistent since the AST has changed. (There is an incremental attribute evaluation mode in JastAdd, but it carries its own overhead, and is not used in our experiments.)

For property probes, all time-consuming work happens on the server side, so this is what we measure in the probe administration part. A headless client is used for measurements, and it runs on the same machine as the server in order to avoid any network latency in the data.

The probe administration part contains all the server-side functionality that is required to support property probes. This includes listing nodes that overlap with the user's cursor, creating and applying node locators, serializing probe results to send them to the client, etc.

We have run measurements on both a high-end benchmark machine and on a normal development laptop. The results we report are from the benchmark ma-

chine, in order to get as noise-free results as possible. The results from the laptop were roughly 1.5 times slower than those from the benchmark machine.

The benchmark machine ran Java 11 on Ubuntu 21.20 with an Intel i7-11700K CPU and 128GB DDR4 RAM. The machine was configured with a minimal amount of background services to reduce noise in the data. The laptop ran Java 17 on Mac OS 12.2.1, an Apple M1 Pro CPU and 16GB LPDDR5 RAM.

Benchmarking is done with three variables: project configuration **P**, action type **T**, and number of actions **N**.

P is one of six project configurations. The minimum configuration is a single file containing five lines of code. The largest configuration is the source code of Apache FOP [Apa], which contains over 900 source files and 96K lines of code.

The action type **T** is either creating or evaluating probes. The number of actions **N** is 1, 5, 10 or 15.

Whenever CODEPROBER performs more than one action for the same source code, it will reuse the AST. Therefore, the parsing cost only needs to be paid once per source code version. The cost for a subsequent action_{*k*} ($k > 1$) is therefore only the property evaluation time and probe administration.

For each combination of **P**, **T**, and **N**, we performed the following sequence:

1. Simulate a change to the source code.
2. Perform **N** actions of type **T**.

This sequence was performed in a loop until steady state had been achieved. After that we performed the sequence an additional 5000 times, and recorded the average time to finish the **N** actions.

We also measured full parsing time, since this is relevant both when the user starts up CODEPROBER, and in case the user reconfigures the analyzed project, in which case a full reparse of all source files is performed. For this reason, we measured full parsing both as a start-up cost (without JVM warmup) and as steady state (after warmup of the JVM).

8.2 Results

All results are shown in Table 1. Furthermore, the time for creating and evaluating probes are plotted in Figures 18a and 18b. We will now discuss these results in more detail.

Creating a Probe

Creating a probe involves two operations:

1. List all AST nodes overlapping with the user's cursor.
2. List all properties available on a given AST node.

Table 1: Performance measurements, all times in milliseconds.

The data for creating and evaluating probes is plotted in Figure 18.

Project configuration names are shortened in the table to A, B, . . . , F, their full names are “Mini”, “Probe Server”, “Commons Codec”, “NetBeans”, “PMD” and “FOP” respectively. “Hot” represents a steady state, e.g. the JVM has warmed up.

“Cold” represents startup, e.g. the JVM has just been started.

Proj	Size	Time to create N probes				Time to evaluate N probes				Full parse	
		1	5	10	15	1	5	10	15	Hot	Cold
A	5	0.8	3.4	6.2	9.5	0.3	0.9	1.7	2.5	0.1	128.5
B	2K	1.5	4.5	8.1	11.7	0.9	1.7	2.6	3.5	9.0	198.2
C	10K	4.0	7.3	11.0	15.0	3.4	4.2	5.2	6.2	44.1	290.6
D	18K	6.3	11.2	14.7	18.4	5.6	8.5	9.3	10.3	57.7	345.7
E	50K	15.7	22.2	29.6	37.5	14.4	16.2	18.1	20.1	155.1	493.2
F	96K	32.0	40.9	50.9	61.6	30.5	32.8	35.4	38.3	348.4	695.1

The two operations correspond to the LISTNODES and LISTPROPERTIES requests in Section 6.2. The result of these actions is a probe with a so far empty result. The actual evaluation of the probe is triggered by the EVALUATEPROPERTY request which is benchmarked separately.

If a user creates a probe based on an existing probe result, the first operation will be skipped. In our benchmarks we always measure both operations, which gives us a worst case time for creating a probe. The results can be seen in Table 1, and are plotted in Figure 18a.

Evaluating a Probe

A probe is evaluated for one of two reasons:

- either a new probe was created,
- or some underlying data changed, and existing probes must be re-evaluated

Evaluating a probe corresponds to the EVALUATEPROPERTY request in Section 6.2. In the scenario where a user plays around with code to see how properties update, the probes will be evaluated significantly more often than they are created. In another scenario, where the user keeps the code fixed, and only explores new properties by creating new probes, there will be a single evaluation each time a probe is created.

Since the property evaluation cost is kept very small in our benchmarks, the measurements should consist almost entirely of parsing time and probe administration time.

The results can be seen in Table 1, and plotted in Figure 18b.

Full Parse Time

The results shown in Figures 18a and 18b are using incremental parsing. The costs shown include a single re-parse of the edited file, regardless of how many probes were created or evaluated. A full parse is sometimes needed, such as when initially starting CODEPROBER, or when changing project configuration while CODEPROBER is running. We have benchmarked the full parse in two scenarios. Once where the parsing code had reached steady state, and once where the parsing code had just been loaded into the JVM (“Startup”). Full parse is measured for all project configurations **P**. The time for the full parse (steady state and start up) can be seen in Table 1.

8.3 Discussion

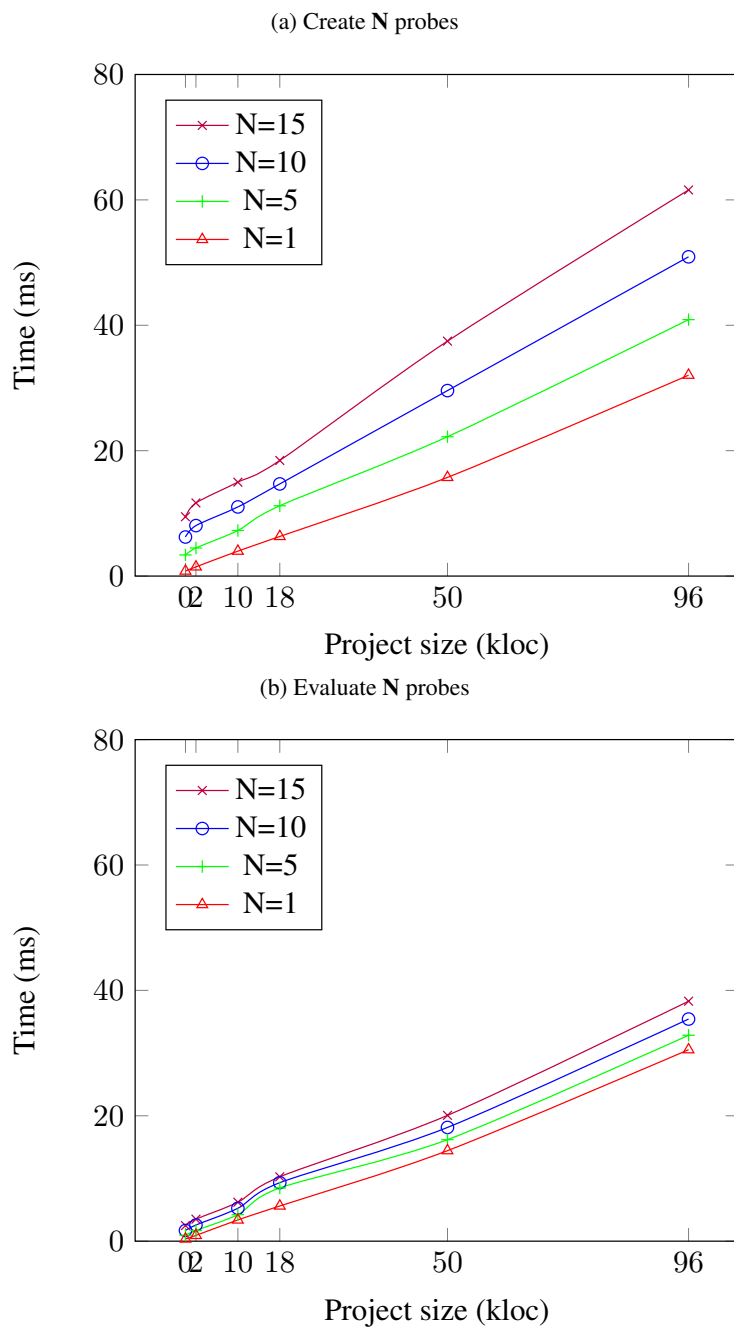
J. Nielsen defined three time limits concerning responsiveness [Nie93]. According to that definition, responding in less than 0.1 seconds is enough to appear instant, responding in less than 1 second is good enough to not interrupt the user’s flow of thought, and responding in less than 10 seconds is the limit for keeping users’ attention so they do not start on other tasks.

As can be seen from Table 1, all overheads for creating a new probe or reevaluating one or many probes are well below 0.1 seconds. For example, creating a probe in the largest project (FOP) takes 32 ms, and reevaluating 15 probes after an edit takes 38.3 ms, so they are all within Nielsen’s category of appearing as instant.

From the plots in Figures 18a and 18b, we can see that the time difference between creating or evaluating 5, 10, or 15 probes is fairly constant for a given project. For example, for FOP, the time to evaluate 5, 10, and 15 probes is 32.8, 35.4, and 38.3 ms respectively. This means that the administration time per probe is only around $(38.3-32.8)/10 = 0.55$ ms. The bulk of the total time is for reparsing the edited file and flushing memoized properties, e.g., around $32.8-0.55*10 = 27.3$ ms for FOP. This means that creating many probes is not a problem for performance. It will rather be the screen size that limits how many active probes the user would like to have.

We can also see that the time grows linearly with project size. The main reason for this is not the probe administration, but the parsing and the flushing of memoized properties. The parsing is proportional to the size of the edited file, which should be rather similar in all projects. The flushing of memoized properties is proportional to the size of the AST of the complete project, including all source files. This is because of the way JastAdd implements memoization (as fields in AST node objects). This might be possible to optimize in JastAdd, e.g., by storing memoized properties in a way that is faster to flush, or by using incremental flushing. While the growth is linear, it is still fairly slow, and fairly large projects can

Figure 18: Time to create and evaluate probes



be handled without problems.

The benchmarks show results for properties that are very quick to compute. Naturally, in case the user probes a property that takes a long time to compute, the response time will be correspondingly longer. However, if the underlying analysis tool is demand-driven, like the JastAdd-generated tools we have used, the response time can still be very short, even for fairly advanced computations. For example, we have experimented with probes showing generated bytecode for a method and the arrow diagnostics showing control-flow, and they do update directly when editing the code.

The measurements of the full parse is an effect of the underlying analysis tool, rather than CODEPROBER. However, we wanted to include these numbers to show that it is not a factor that gives any problems. As can be seen in Table 1, the longest time for a full parse (that happens initially at start up), takes 695.1 ms for the largest project (FOP, 97 kLOC), so less than 1 second.

Overall, property probes have shown to be very helpful in exploring how an analysis works, and for implementing and fixing features. The approach fits very well for analysis tools that use on-demand evaluation for individual properties, like the JastAdd-based tools we have tried it on. However, we think the approach can be very useful also for tools that do up-front evaluation, as long as the results can be tied to an AST with source text locations. In this case, properties can be explored interactively, although the user will of course have to wait for a possibly lengthy reanalysis if the underlying source text is edited.

9 Related Work

CODEPROBER allows interactive exploration of properties based on the source code. Earlier tools for debugging and exploring attribute grammars include, for example, Noosa [Slo99], DrAST [LTH16], Aki [Ike+00], and EvDebugger [RCHS14]. They all have ways of showing the syntax tree and attribute values, but none of them have any concept of probes that are updated after changes to the source text.

Noosa is a special-purpose interactive debugger for compilers implemented in the Eli attribute grammar system. It supports, e.g., visualization of the AST, display of attributes of the AST, linking between source text and AST, monitoring the stream of abstract events during data-driven attribute evaluation, and setting breakpoints relating to such events.

DrAST is an interactive tool for visualizing JastAdd ASTs and inspecting AST node properties. It introduces a filtering language to collapse subtrees in order to reduce the visual complexity of the AST, and to specify which attributes to show directly in the tree for certain node types, possibly conditionally. Individual attributes can also be inspected.

Aki is a visual debugger for attribute grammars, supporting algorithmic and

slice-based debugging of attributes. Based on attribute dependencies, it can systematically query the user about correctness of attribute values, in order to pinpoint the source of an error. Aki can present the syntax tree for a source program and its attribute values. Aki’s debugging techniques are developed for traditional attribute grammars where attributes are evaluated in a data-driven manner. It would be interesting future work to develop similar techniques for RAGs and demand-driven evaluation and integrate them into CODEPROBER.

EvDebugger supports creating and debugging attribute grammar implementations. The main goal of the tool is to support students in learning attribute grammars, and in particular to illustrate the attribute evaluation process. EvDebugger has support for showing a syntax tree, and stepping through an attribute evaluation process to understand in what order attributes are evaluated. It is based on traditional attribute grammars where evaluation is data-driven, according to static dependencies in the grammar.

While CODEPROBER is not primarily aimed at teaching attribute grammar evaluation algorithms, it would be interesting to add support for tracing and stepping through a demand-driven evaluation, to help students understand how such an evaluation works, when memoization happens, etc. Additionally, it would be very interesting future work to add support for showing dynamic dependencies between properties, and letting the user navigate this graph to explore property values. This would be very related to the algorithmic/slicing debugging methods mentioned above, allowing a user to explore why a certain property has a certain value.

The Language Server Protocol (LSP) [Mic] is a widely supported protocol for interaction between editors and language implementations. It supports features like code completion, refactoring, validation, and more. Some of CODEPROBER’s features could be implemented as a language server. For example, evaluating properties that produce diagnostics (“squiggly lines”) and presenting them over LSP is possible. However, many features are not possible to replicate within LSP. This includes custom UI such as the probe windows and arrows rendered on top of the code. In addition, node locators depend on the user’s input actions to adjust TAL steps, and that information isn’t easily accessible over LSP. There is interesting future work in trying to port as much functionality as possible over to LSP.

The concept of node locators has relations to *origin tracking* [VDKT93]. This is a set of techniques for identifying where a node came from after tree rewrites. This is useful, for example, when generating error messages for transformed trees. Origin tracking has also been integrated with attribute grammars with higher-order attributes [WW14] (HOAs), which might be useful for improving locations for HOAs used in our CODEPROBER.

Node locators also have connections to *edit scripts*. Edit scripts describe differences between two versions of a source file, either in textual or AST form. This can be used to generate detailed program diffs, or track nodes across multiple versions of a source file. However, existing techniques, like GumTree [Fal+14],

IJM [Fri+18], MTDIFF [DP16], and TrueDiff [ESP21], are focused on detecting differences between two files, without any knowledge of the actual input sequence that transformed one file to another. Our node locators require that we have input information available while editing. This is a limitation, but it also makes the algorithm much simpler. It might be possible to derive input information using edit scripts, and thus make it easier to integrate property probes with, for example, LSP.

Property probes in CODEPROBER can be viewed as being on liveness level 3 out of 6 according to Tanimoto [Tan13]. Probes are automatically updated when either the input source file or the analysis tool have been changed, but do not predict user actions.

Property probes can also be compared to *watch* expressions found in many debuggers. Watch expressions typically only run while a debugging session is running, and the expressions are evaluated in the context of the current debugging session state. Property probes, on the other hand, are always active, and are evaluated without any state (except the source file contents that were used to initially construct the AST).

Beller, Spruit, Spinellis and Zaidman found that “developers spend surprisingly little time in the debugger” [Bel+18], citing complexity of debuggers as a potential reason. Many developers they surveyed preferred using “print debugging” instead, despite its limitations. This indicates to us that there is a need to develop new ways of exploring/debugging programs. It might be easier to develop debugging tools for particular use cases, like for exploring partial program analysis results. This paper represents one such tool.

Erdweg et al. define *language workbenches* as “tools that support the efficient definition, reuse and composition of languages and their IDEs” [Erd+13]. They define a feature model according to which a language workbench must support notation, semantics, and an editor for the defined language. Optional features include testing and debugging of the language definition. For the editor, optional features include support for semantic services like reference resolution (i.e., name binding), semantic completion, error markings, live translation, etc. In relation to language workbenches, CODEPROBER supports many of these optional features since it can be used for debugging and exploratory testing of the language specification, and it can be used for prototyping semantic language services. So while CODEPROBER is not a language workbench by itself, it could be used as a component of one.

10 Conclusion

We have presented the concept of property probes, an interactive mechanism for exploring program analyses in terms of the source code.

To support probes to be updated after edits, we introduced node locators with

three kinds of steps: `Child`, `TAL`, and `FN`, and illustrated how they are used to robustly map between source code and the nodes in the program representation, and to handle synthetic nodes that have no representation in the source code.

We have developed CODEPROBER to support property probes, and discussed its client-server architecture and implementation. To validate our work, we successfully applied CODEPROBER to a number of tools for different languages and analyses, all based on Reference Attribute Grammars. We have also used CODEPROBER in two courses, and received positive feedback from students. This initial testing has already shown the utility of the tool. We are now using the tool extensively in our own work on program analysis. We have also shown through experiments that the overhead of using probes is very small, even if the analyzed project is large, giving latencies in the interactive tool that are far below the recommended limit of 0.1 seconds.

Acknowledgements

This work was funded by ELLIIT – Excellence Center at Linköping-Lund on Information Technology and by the Swedish Foundation for Strategic Research.

We would like to thank Idriss Riouak, main developer of IntraJ, for continuously testing and giving feedback during the development of CODEPROBER. We would also like to thank Christoph Reichenbach, course leader for the program analysis course where CODEPROBER was used, for his feedback and his contributions to CODEPROBER.

References

- [Apa] *Apache(tm) FOP - a print formatter driven by XSL formatting objects (XSL-FO) and an output independent formatter.* — xmlgraphics.apache.org.
<https://xmlgraphics.apache.org/fop/>. [Accessed 28-Jun-2022].
- [Bel+18] Moritz Beller et al. “On the dichotomy of debugging behavior among programmers”. In: *Proceedings of the 40th International Conference on Software Engineering*. 2018, pp. 572–583.
- [Llv] *Clang Static Analyzer* — clang-analyzer.llvm.org.
<https://clang-analyzer.llvm.org/>. [Accessed 22-Jun-2022].
- [DP16] Georg Dotzler and Michael Philippsen. “Move-optimized source code tree differencing”. In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2016, pp. 660–671.

- [EH07a] Torbjörn Ekman and Görel Hedin. “The JastAdd system - modular extensible compiler construction”. In: *Sci. Comput. Program.* 69.1-3 (2007), pp. 14–26.
- [EH07b] Torbjörn Ekman and Görel Hedin. “The Jastadd Extensible Java Compiler”. In: *SIGPLAN Not.* 42.10 (2007), 1–18.
- [ESP21] Sebastian Erdweg, Tamás Szabó, and André Pacak. “Concise, type-safe, and efficient structural diffing”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 2021, pp. 406–419.
- [Erd+13] Sebastian Erdweg et al. “The state of the art in language workbenches: Conclusions from the language workbench challenge”. In: *Software Language Engineering: 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings 6.* Springer. 2013, pp. 197–217.
- [Fal+14] Jean-Rémy Falleri et al. “Fine-grained and accurate source code differencing”. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering.* 2014, pp. 313–324.
- [FH15] Niklas Fors and Görel Hedin. “A JastAdd implementation of Oberon-0”. In: *Sci. Comput. Program.* 114 (2015), pp. 74–84.
- [FH16] Niklas Fors and Görel Hedin. “Bloqqi: Modular Feature-Based Block Diagram Programming”. In: *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* Onward! 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, 57–73.
- [Fri+18] Veit Frick et al. “Generating accurate and compact edit scripts using tree differencing”. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE. 2018, pp. 264–274.
- [Hed00] Görel Hedin. “Reference attributed grammars”. In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.
- [Ike+00] Yohei Ikezoe et al. “Systematic Debugging of Attribute Grammars”. In: *Proceedings of the Fourth International Workshop on Automated Debugging, AADEBUG 2000, Munich, Germany, August 28-30th, 2000.* Ed. by Mireille Ducassé. 2000.
- [Ext] *JastAdd API Docs — extendj.org.*
<http://extendj.org/doc/>. [Accessed 28-Jun-2022].

- [LTH16] Joel Lindholm, Johan Thorsberg, and Görel Hedin. “DrAST: an inspection tool for attributed syntax trees (tool demo)”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*. Ed. by Tijs van der Storm, Emilie Balland, and Dániel Varró. ACM, 2016, pp. 176–180.
- [MF11] Alexey Melnikov and Ian Fette. *The WebSocket Protocol*. RFC 6455. Dec. 2011.
- [Mic] *Monaco Editor* — microsoft.github.io. <https://microsoft.github.io/monaco-editor/>. [Accessed 22-Jun-2022].
- [Nie93] Jakob Nielsen. “Response times: the three important limits”. In: *Usability Engineering* (1993).
- [Pmd] *PMD*. <https://pmd.github.io/>. [Accessed 2-Oct-2023].
- [PSH19] Rohan Padhye, Koushik Sen, and Paul N Hilfinger. “Chocopy: A programming language for compilers courses”. In: *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. 2019, pp. 41–45.
- [Rio+21] Idriss Riouak et al. “A Precise Framework for Source-Level Control-Flow Analysis”. In: *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2021, pp. 1–11.
- [RA+22] Anton Risberg Alaküla et al. “Property Probes: Source Code Based Exploration of Program Analysis Results”. In: *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*. 2022, pp. 148–160.
- [RCHS14] Daniel Rodríguez-Cerezo, Pedro Rangel Henriques, and José-Luis Sierra. “Attribute grammars made easier: EvDebugger a visual debugger for attribute grammars”. In: *2014 International Symposium on Computers in Education (SIIE)*. 2014, pp. 23–28.
- [Slo99] Anthony M. Sloane. “Debugging Eli-Generated Compilers With Noosa”. In: *Compiler Construction, 8th International Conference, CC’99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS’99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*. Ed. by Stefan Jähnichen. Vol. 1575. Lecture Notes in Computer Science. Springer, 1999, pp. 17–31.
- [Spo] *SpotBugs*. <https://spotbugs.github.io/>. [Accessed 2-Oct-2023].

- [Tan13] Steven L. Tanimoto. “A perspective on the evolution of live programming”. In: *Proceedings of the 1st International Workshop on Live Programming, LIVE 2013, San Francisco, California, USA, May 19, 2013*. Ed. by Brian Burg, Adrian Kuhn, and Chris Parnin. IEEE Computer Society, 2013, pp. 31–34.
- [VDKT93] Arie Van Deursen, Paul Klint, and Frank Tip. “Origin tracking”. In: *Journal of Symbolic Computation* 15.5-6 (1993), pp. 523–545.
- [VSK89b] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. “Higher-Order Attribute Grammars”. In: *Proceedings of the ACM SIGPLAN’89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*. Ed. by Richard L. Wexelblat. ACM, 1989, pp. 131–145.
- [Wal] WALA. <https://sourceforge.net/projects/wala/>. [Accessed 2-Oct-2023].
- [WW14] Kevin Williams and Eric Van Wyk. “Origin Tracking in Attribute Grammars”. In: *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*. Ed. by Benoît Combemale et al. Vol. 8706. Lecture Notes in Computer Science. Springer, 2014, pp. 282–301.

STUDY OF THE USE OF PROPERTY PROBES IN AN EDUCATIONAL SETTING

Abstract

Context: Developing compilers and static analysis tools (“language tools”) is a difficult and time-consuming task. We have previously presented *property probes*, a technique to help the language tool developer build understanding of their tool. A probe presents a live view into the internals of the compiler, enabling the developer to see all the intermediate steps of a compilation or analysis rather than just the final output. This technique has been realized in a tool called CODEPROBER.

Inquiry: CODEPROBER has been in active use in both research and education for over two years, but its practical use has not been well studied. CODEPROBER combines liveness, AST exploration and presenting program analysis results on top of source code. While there are other tools that specifically target language tool developers, we are not aware of any that has the same design as CODEPROBER, much less any such tool with an extensive user study. We therefore claim there is a lack of knowledge how property probes (and by extension CODEPROBER) are used in practice.

Approach: We present the results from a mixed-method study of use of CODEPROBER in an educational setting, with the goal to discover if and how property probes help, and how they compare to more traditional techniques such as test cases, print debugging, etc. In the study, we analyzed data from 11 in-person interviews with students using CODEPROBER as part of a course on program analysis. We also analyzed CODEPROBER event logs from 24 students in the same course, and 51 anonymized survey responses across two courses where CODEPROBER was used.

Knowledge: Our findings show that the students find CODEPROBER to be useful, and they make continuous use of it during the course labs. We further find that the students in our study seem to partially or fully use CODEPROBER instead of other development tools and techniques, e.g. breakpoint/step-debugging, test cases and print debugging.

Grounding: Our claims are supported by three different data sources: 11 in-person interviews, log analysis from 24 students, and surveys with 51 responses.

Importance: We hope our findings inspire others to consider live exploration to help language tool developers build understanding of their tool.

1 Introduction

Language tooling, like compilers and static analyzers, can easily become complex to develop. A professional compiler takes many person-years to develop and typically has to comply with complex semantic specifications. For example, the specification of Java version 8¹ is 788 pages long, and its reference implementation² is over 5 million lines of code.³ The language community has developed numerous tools over the years to assist with this complex activity. We have seen advances in areas such as language tool generation [HM03; SH11] all the way to full language workbenches [KV10; DS11]. There has been a lot of progress in the development of language tool chains to enable faster development of language tools, but we still have more work to do.

One activity in language tool development worthy of more attention is program comprehension, which underpins tool understanding, feature development, maintenance, and debugging. As we increase the level of abstraction and introduce more code generation into the workflow, the distance to the running code

¹<https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf>. Accessed September 2024.

²<https://github.com/openjdk/jdk8u-ri>. Accessed September 2024.

³This includes the sources of the JVM, the standard library and other tools necessary to fully support the language. The `langtools` directory, which contains sources for `javac` is 120k lines of code.

increases. Declarative specifications can be great as a way to separate the ‘what’ from the ‘how’, but when the specification is not doing what it should, it can be tricky to get insights into how to improve it. We hypothesize that the nature of a declarative approach may introduce additional hidden dependencies [Gre89] that may decrease usability when things break down.

One approach to shed light on the “hidden” inner functionality of a language tool is to utilize so-called property probes [Ala+24]. Property probes, which have been realized in the tool CODEPROBER⁴, provides a live, exploratory view into the functionality of a language tool. The aim with CODEPROBER is to assist during the development of language tools in a way that complements existing tools, with the goal to help explore and build understanding of language tooling. CODEPROBER is meant to be used both by students learning about building compilers and program analyzers, and used by practitioners in industrial language tool development. CODEPROBER has been in active use in education and research for over two years now. However, we lack an understanding of how property probes are used in practice and what the users’ experience of using them are.

In this paper, we present a mixed-method study on the use and user experience of CODEPROBER in an educational setting, focusing on students learning about compilers and program analyzers. We combine the results from 11 interviews, logs analysis from 24 students, and surveys results from 51 responses to find an answer to the following research questions:

- RQ1** What is the user experience of using CODEPROBER in an educational setting?
- RQ2** How is CODEPROBER used during the development of compilers and static analysis tools in an educational setting?
- RQ3** How does the use of CODEPROBER compare to other tools used by students during the development process (e.g. debuggers, test cases, print-statements, AI, etc.)?

We find that the students find CODEPROBER to be useful, and they make continuous use of it during the course labs. We further find that the students in our study seem to partially or fully use CODEPROBER instead of other development tools and techniques, e.g. breakpoint/step-debugging, test cases and print debugging.

The rest of the paper is structured as follows. We start by giving some background into how language tooling is built, specifically focusing on Reference Attribute Grammars which CODEPROBER works with (Section 2), before we give an introduction into CODEPROBER’s features (Section 3). We then introduce the overall design of the study (Section 4), followed by the method and results for the interview part (Section 5), log file analysis part (Section 6), and survey part (Section 7). Finally, we discuss the results in light of our research questions (Section 8), before we discuss related work (Section 9) and conclude (Section 10).

⁴<https://github.com/lu-cs-sde/codeprober>. Accessed September 2024.

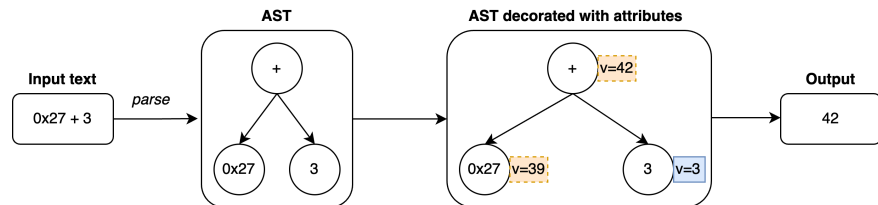


Figure 1: Overview of how a RAG-based compiler works. The original code is parsed, and the AST is decorated with attributes. Finally, the attribute v of the root node is accessed, which computes the other attributes on demand. Solid blue color indicates an intrinsic attribute, i.e., something that is already known in the original AST. Dashed orange color indicates something that is computed.

2 Language Tools with On-Demand Evaluation

In this paper, we refer to compilers and static analysis tools as “language tools”. They have similar overall goals: transform source code into some desired output. The output can be machine code, a list of diagnostic messages, the result of running test cases, etc. The implementation of a language tool can be pass-based, where values needed internally in the tool are computed in passes, e.g. computing a symbol table before type resolution. Alternatively the evaluation of semantic values can be on-demand, e.g., computing names needed to resolve a type.

In this section, we provide a high level overview of building language tools with Reference Attribute Grammars (RAGs) [Hed00], an approach for building language tools that compute semantic values on-demand. RAGs are an extension of Attribute Grammars [Knu68a]. RAG-based tools start by parsing source code into an abstract syntax tree (AST). They then associate functionality (“attributes”) with the AST nodes. Attributes may depend on other attributes, and can compute values on-demand. This lets the developer specify a full language tool as a set of smaller attributes that depend on each other. In the end, RAGs can compute the same information as a traditional pass-based compiler, but the evaluation can be seen as happening in reverse. Evaluation starts by accessing the desired output, and then gradually works backwards through the intermediate steps.

Figure 1 provides an illustration of a RAG-based language tool in the form of simple calculator language. The language supports additions and two kinds of integers: base-10 and base-16. The desired output for this language is the base-10 value of the program. For example, for the input $0x27 + 3$, the output should be 42. The computation is defined as an attribute v that has different definitions depending on the node type. For example, the definition for addition nodes accesses v on its children and adds them together. The program output is then computed by accessing the attribute v on the root node (which might trigger more attributes to be computed).

An interesting aspect of RAGs is that all functionality is always accessible on the AST, but nothing is computed by default. As long as only a subset of the functionality is accessed, it can usually be computed very quickly. For example, the performance evaluation by Alaküla et al. [Ala+24] shows that some selected attributes finish evaluating in 1 – 30 milliseconds. The exact time it takes depends on the complexity of the chosen attribute, but in general it is quick enough to appear instant to the user [Nie93]. This enables quick, interactive exploration of the inner workings of the language tool, which can be of great help during the development process. For example, assume that the code that transforms from base-16 to base-10 integers computes the wrong value. In Figure 1 it is possible to directly access the v attribute of `0x27` and evaluate it in isolation.

2.1 Debugging RAGs

In order to debug and build understanding of a RAG-based tool, it is beneficial to be able to explore each attribute individually. Accessing and evaluating these internal properties is possible to do with traditional debugging tools and techniques, but we believe it can be inconvenient to do so. In this section, we give examples of the problems that may arise, especially when working with language tools for non-trivial languages.

Assume a language tool developer wants to inspect a specific attribute like v in Figure 1 using print debugging. They would have to: 1) Add a print statement to the language tool specification, 2) Rebuild the language tool, 3) Run the tool with an example input file, and 4) Filter the output to find the line corresponding to the node of interest. Steps 1 and 2 need to be done for every new information the developer wants to extract, and build times can be prohibitive for larger language tools. Additionally, step 4 can be a significant hurdle when exploring larger input files, as the same attribute may be invoked many times for different AST nodes. Step 4 can be mitigated by making the print statement conditional. However, this might require some non-trivial conditional expression. Similar issues arise with traditional breakpoint/step-debuggers, where defining a conditional breakpoint for a particular node might be non-trivial.

Print debugging and breakpoint/step-debugging are viable options for exploring attributes on an AST. However, we believe it is more convenient to freely explore attributes without rebuilding the language tool and to use the input text to find nodes of interest. This has been realized in the tool CODEPROBER [Ala+24] that supports property probes, which we will describe in the next section.

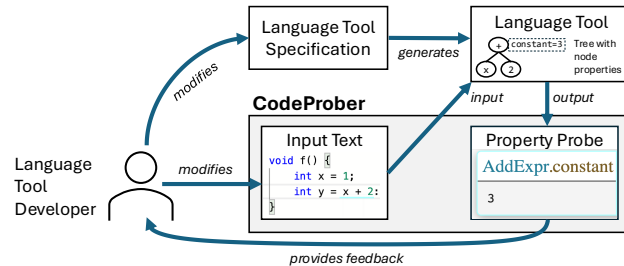


Figure 2: Overview of CODEPROBER. The language tool developer has created a property probe of the expression $x + 2$ in the input text. The property constant computes the compile-time constant of that expression. The developer can either change the input text or the language tool, and the probed result is automatically updated.

3 CODEPROBER

This section gives a brief introduction to CODEPROBER [Ala+24] and its features⁵. CODEPROBER allows for an interactive exploration of intermediate results in language tools. The results are live and automatically updated when the input text (typically code) or the language tool changes. An overview of CODEPROBER is shown in Figure 2.

CODEPROBER does not compute any values on its own. Rather, it simplifies accessing and exploring values that another language tool computes. CODEPROBER requires that the language tool parses input text into a tree with node properties that can be accessed. The language tool developer can then create *probes* via the input text in CODEPROBER to access these properties. The requirement of node properties maps very well to attributes described in the previous section. However, property probes are a general technique that can be applied to other kinds of language tools as long as they fulfill the requirements, i.e., they associate properties with nodes in a tree.

3.1 Basic Usage

As an example, the language developer starts by writing the input text they wish to explore into the CODEPROBER editor, and then right clicks to create a probe. Figure 3 shows the process of creating a probe in the Java compiler ExtendJ [EH07b]. In it, the developer started by writing a small Java program. Then, they create a probe for the compile-time constant value of an addition expression. From this point, if the developer makes any change to the Java program, or if they update

⁵A video demonstration of CODEPROBER: <https://youtube.com/watch?v=1kTJ4VL0xtY>.

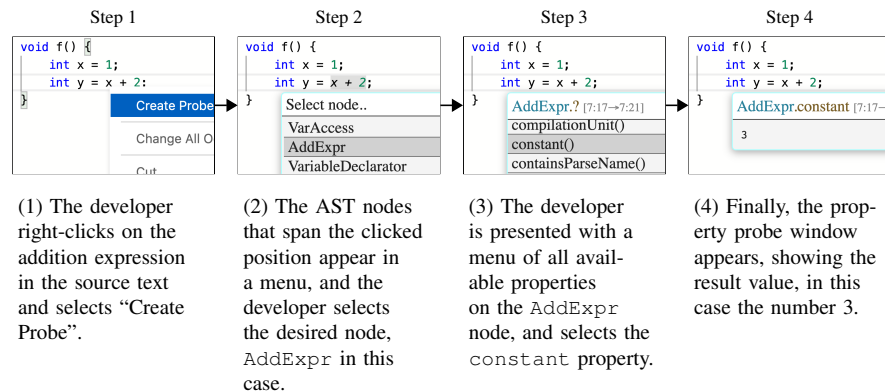


Figure 3: Steps to create a probe for the `constant` property of the addition expression `x + 2`.

the underlying Java compiler, then the probe will display updated values. The developer can then continue to create probes for other properties, for example, byte code generation, without requiring recompilation. This is different from e.g. print debugging, where the developer has to modify source code, compile and run for each new piece of information they wish to extract.

This ability to explore properties, together with the liveness features, is central to the experience of using CODEPROBER. Normal probes, as shown in Figure 3, is a core way of interacting in CODEPROBER. However, several more ways of interaction are supported. Some of these are illustrated in the following list using a teaching language called TEAL. See also Section 4.2 for more information about TEAL.

- *Diagnostic contributions* allow probes to contribute diagnostics to the text editor, in the form of “squiggly lines” (Figure 4a) and arrows (Figure 4b). This can be used for example to present semantic issues or render a control-flow graph.
- *AST probes* display part of the AST in graphical form. This can be seen in Figure 5.
- *Search probes* support finding a set of nodes that pass some predicate, and evaluate an attribute for all of them simultaneously. A search probe can be seen in Figure 6.

Currently, we have implemented support for language tools specified with the meta-compilation system JastAdd [EH07a]. JastAdd combines reference attribute grammars with object- and aspect-orientation. For an introduction to JASTADD,

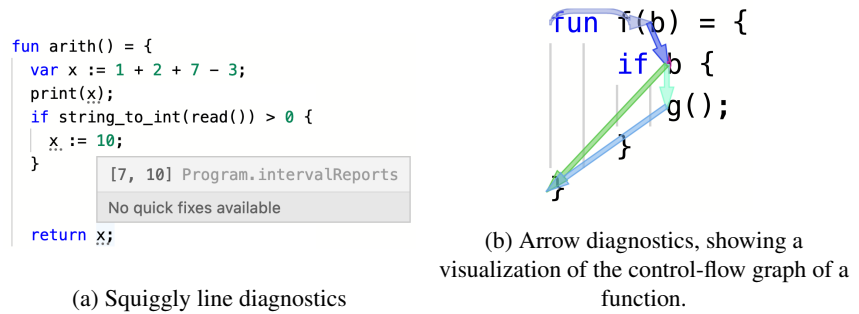


Figure 4: Screenshots of diagnostic contributions in CODEPROBER. The most common kind of diagnostic is “squiggly lines”, and they are shown in three locations in Figure 4a. The user is hovering the last location (x variable on the last line), and the popup shows the variable’s interval value at that point in the program. In this case, the value of x on the last line must be within the interval of $[7, 10]$.

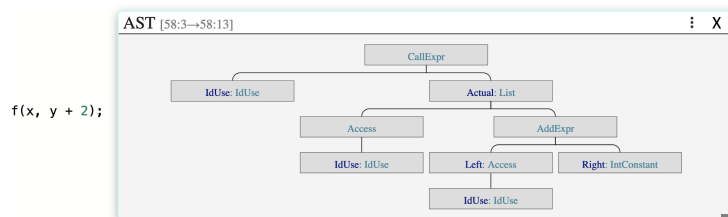


Figure 5: AST probe showing the AST of a function call. More probes can be created by clicking on individual nodes in the AST.

see [Hed09]. However, there is an interface that other kinds of language tools can implement to integrate with CODEPROBER.

4 Study Overview

In this section, we provide an overview of our study along with a description of the context in which the data for the study was collected. With the objective of gaining a deeper understanding of how property probes are used in practice via the CODEPROBER tool and the user experience of using CODEPROBER in an educational setting with students learning about compilers and program analysis, we designed a mixed-method study to address the research questions defined in Section 1.

```

3
4
5
6
7
8 fun arith() = {
9   var x := 1 + 2 + 7 - 3;
10  print(x);
11  if string_to_int(read()) > 0 {
12    x := 10;
13  }
14  return x;
15 }
16
17
18
19
20
21

```

The screenshot shows a search probe window titled 'FunDecl.*.interval?isAccess [8:1→15:1]'. It lists three nodes found:

- Access [10:9→10:9] with interval [10] and result [7, 7]
- Access [12:5→12:5] (highlighted) with interval [12] and result [7, 7]
- Access [14:10→14:10] with interval [14] and result [7, 10]

Figure 6: Search probe that finds all nodes in the function (FunDecl) where property `isAccess` is true, and opens a nested probe for property `interval` on them. The user is hovering the middle Access node, which causes the corresponding span in the text editor to be highlighted. (Note that the middle result of `[7, 7]` is correct, it displays the interval value prior to the assignment).

4.1 Study Objective and Design

The study is composed of three parts; interviews with students, analysis of logs from students using CODEPROBER, and a survey sent to students after they used CODEPROBER as part of course labs. An overview of when the different parts took place is seen in Figure 7.

Each part helps address one or more of the research questions. The interview questions are designed to help answer each research question (**RQ1**, **RQ2**, and **RQ3**). The log analysis tells us how CODEPROBER is used, which helps answer **RQ2**. The course survey contains a question about the “effectiveness” of CODEPROBER, which helps in answering **RQ1**. Triangulation, in the context of user

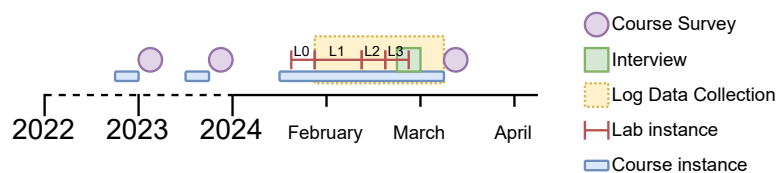


Figure 7: Overview of when the different parts of the study took place. The four red lines labeled “L0” to “L3” represent the planned durations of Lab 0 to Lab 3 in the program analysis course. Late lab submissions were allowed, which is why the log data collection proceeded after the end of Lab 3.

studies, refers to investigating a phenomenon from at least two different perspectives. Perspectives can mean making observations at different points in time, using different techniques, observing different groups of people, etc. The main idea is that by observing the same phenomenon from multiple perspectives, it is possible to draw conclusions with more confidence. We hope that our study design, combining three parts with different methods, will help to answer our **RQs** with better confidence.

The details of each of the study parts, along with their results, are presented in Section 5, 6, and 7, respectively. For the rest of this section, we will focus on describing the context of the study.

4.2 Educational Setting

The data collection for this study is carried out in connection to two university courses taught to engineering students typically specializing in computer science; a course on program analysis and a course on compilers. Both of these courses have integrated the use of CODEPROBER in the practical work in the course. Our data collection is focused on the Spring 2024 instance of the program analysis course, but we also include survey responses gathered for the Fall 2023 instance of the compilers course and also the earlier Fall 2022 instance of the program analysis course.

It should be noted that students who take the program analysis course have often taken the compilers course prior to it (or another compilers course) and at least two other programming courses. In the Spring 2024 program analysis course instance, 27 of the 31 students that finished the course had previously taken the compilers course at our university (note that 3 of those students had taken the compilers course before CODEPROBER was introduced in the course). Half of the remaining 4 students had taken a compilers course at another university, and 2 had not taken any compilers course.

In the below subsections we describe the student population of the program analysis, as well as the two courses in more detail.

Student Population

The average student in the program analysis course is in their 4th year of a 5-year computer science and engineering program⁶ at Lund University. There are a few cases where PhD students or students from a different engineering program take the course as well. In the 2024 instance of the course, there were 31 students, and 27 of these were from the computer science and engineering program.

The first three years at the computer science and engineering program contains 7 mandatory programming-related courses, totaling 44.5 ECTS credits. These

⁶Full name: “Master of Science in Engineering, Computer Science and Engineering”

courses cover topics like object-oriented programming (Scala & Java), concurrency, agile software development (including test-driven development), functional programming and more. There are also several labs that contain the use of traditional debuggers.

The students in the program analysis course develop language tools in an educational setting, making them target users for our user study, especially since CODEPROBER has been incorporated in the course. They further have experience from debugging in previous courses, which gives them a good foundation for reasoning about CODEPROBER and comparing it with alternatives.

As can be seen in Figure 7, we interview the students at the end of the last lab in the course. This is so that they have time to build as much experience with CODEPROBER as possible, and they still have that experience fresh in their heads when we interview them.

The Compilers Course

In the compilers course (7.5 ECTS credits), students learn how to create a compiler from scratch. Over a series of 6 labs they create a basic C-like compiler. They do this incrementally, starting with scanning and parsing, and eventually adding code generation.

CODEPROBER is introduced to the students at the end of lab 3, at the same time as they start writing their own RAG attributes. Then for the remainder of the labs, they are told that they should implement test cases for everything they do, but they are also welcome to use CODEPROBER if they want.

We have not collected logs-based data on how many students make use of CODEPROBER in this course. However, at the end of the lab sessions we usually ask how they approached solving the lab, and we estimate that roughly two thirds of the students mention using CODEPROBER. This estimate is also reflected in the course evaluation for the compilers course (Figure 9c), where just over two thirds of the students (15 of 22) “agree” or “fully agree” that CODEPROBER is “effective”.

The Program Analysis Course

In the program analysis course (7.5 ECTS credits), the students implement several different kinds of program analyses on top of an existing compiler in a language called TEAL (“Typed Easily Analysable Language”). TEAL is a gradually typed imperative language. An example of TEAL code from the labs is included in Figure 4a. At the start of each lab, the students are provided with a working compiler and instructions for what to add. The provided compilers are all implemented using JASTADD. In the 2024 course instance, the following labs were included:

Lab 0: Introduction to JASTADD and CODEPROBER. A relatively short lab, specifically designed to assist students with no prior experience with

JASTADD. In this lab, they implement simple type checking. No log data was collected from this lab.

Lab 1: Type Inference. In this lab, the students implement monomorphic type inference, based on collecting and solving constraints.

Lab 2: Dead-Assignment Analysis. In this lab, the students mainly implement dead-assignment analysis, i.e., given a declaration or assignment, compute whether the assignment is unnecessary.

Lab 3: Interval Analysis. In the final lab, the students implement interval analysis, i.e., for all integer variables, compute its possible range of values. An example from this lab is included in Figure 4a.

The size of the labs vary significantly, both in terms of how much code is handed out and how much code is required to solve them. The handout code varies from 3000 to 4000 lines of code (not counting tests). The solutions to the labs vary from 100 to 600 lines of code. A snapshot of the handout code for all labs is available on Zenodo ⁷.

Each lab contains a small set of example files that can be opened in CODEPROBER. The version of CODEPROBER used in the labs is preconfigured to extract some diagnostic information from the students tools. This diagnostic information is usually presented as hoverable squiggly lines or dots, such as the dots seen in Figure 4a.

The students are encouraged to write tests, and use of CODEPROBER is in theory optional. However, examples are often given in terms of CODEPROBER, and the TAs often ask to see functionality via CODEPROBER, so in practice all students actively use it to some degree.

5 Interviews

This section presents the method and the results of the semi-structured interviews carried out with students taking the Spring 2024 instance of the program analysis course described in Section 4.

5.1 Method

Below, we describe how we designed and executed the interview study, i.e., how we collected data in the study and how we analyzed that data.

⁷<https://doi.org/10.5281/zenodo.13380279>

Data Collection

As a first step of the data collection for this interview study, we designed the interview protocol in connection to the research objective. After testing the interview protocol with a pilot interview and refining the protocol, we continued to recruit participants and to carry out the interviews. See below for further details.

Designing the Interview Protocol The interview protocol includes the following parts:

1. **Warmup.** We ask background questions about the participants experience and skill, and basic questions about CODEPROBER. Some of these questions do not directly relate to a research question, but help us get to know the participant, and helps the participant to start thinking more about CODEPROBER.

For the questions about skill, Peitek et al. [Pei+22] found that one of the best indicators of “programming efficacy” is to ask the programmer to rate themselves compared with their peers. Therefore, in the background form we asked the participants to rate their programming skills in comparison to their classmates on a scale of $[-2, 2]$. Value -2 =“Much Worse”, -1 =“Worse”, 0 =“Identical”, 1 =“Better” and 2 =“Much Better”.

2. **Workflow.** We ask a set of questions relating to how the students approach the labs. What editor do they use, do they write test cases, when is CODEPROBER used, etc. This section relates to **RQ2** and **RQ3**.
3. **Scenarios and Tools.** We ask how much the student uses different development tools during different scenarios of working in a code base. This discussion is driven by them filling in a table of how much they use each tool when developing language tools (mostly in the program analysis course). One axis of the table contains development tools, and the other axis contains development scenarios. The included tools are “Print debugging”, “Breakpoint/step debugging”, “Test cases”, CODEPROBER and “AI” (Copilot/ChatGPT/etc.). The scenarios are “Developing a new feature”, “Developing understanding of a codebase” and “Fixing a bug”. There is some overlap between the scenarios, but they roughly map to the middle, beginning and end of the lab, respectively. Finally, we ask them how much they *like* using each development tooling. This section relates to **RQ1** and **RQ3**.
4. **Likes and Dislikes.** Finally, we ask the participants what they like most and least about CODEPROBER. We also ask if there are any features they would like to add. This ends up being somewhat a repetition of what was said in the previous two sections, but it gives the interviewee a chance to highlight which things matter most. This section relates to **RQ1** and **RQ2**.



Figure 8: Room used during the interviews.

After designing the main interview protocol we performed a pilot study. Afterwards a few minor things were adjusted. The end result is an in-person interview that takes about 50 minutes to perform. The interview is a mix of open questions and some forms for the interviewee to fill in. The full interview design is available in Appendix A, B, C and D.

Recruiting Participants We attended one of the course lectures and presented our intention to perform this study. An outline of the interview was shown, as well as their rights and expected reward (a small take-home gift). We also stressed that the interview would not measure or benchmark the students in any way. We hoped this would make students more eager to apply, and it seemed to have worked as 9 of them applied (out of 31). In addition, we interviewed 2 teaching assistants that either are or were involved in the course, but are not part of the team working on CODEPROBER. The pilot study was carried out with one of the TAs. In total, we interviewed 11 people.

Execution of the Interviews The first and second author of this paper met each interview subject in turn. We sat around a table that had a laptop running CODEPROBER, to be used as demo/reference if necessary. One led the interview and the other took notes. A picture of the room used during all interviews is visible in Figure 8. The interviews were recorded after informed consent from participants. The laptop had active screen- and voice-recording throughout the interview. In addition, a phone was used to record voice for redundancy.

Data Analysis

The interviews provided two kinds of data: the forms and audio recordings. The form data was relatively small, and was manually entered into a spreadsheet. The audio required more processing. First, we transcribed all recorded interviews using OpenAI Whisper [Rad+23]. Then, we did a manual inspection over all the transcripts and fixed any errors from the model. This resulted in 1770 lines of text

being added and 2434 lines being removed. The corrected transcriptions are in total 11607 lines of text. Finally, we performed coding.

Coding the Transcripts A coding scheme was developed to help with extracting information from the interview transcripts. The goal of this coding scheme was to extract common themes, with regard to the use and experience of using CODEPROBER and other development tools, as formulated in the research questions. The first and second author of this paper independently extracted a coding scheme from the same transcript. Then they met, discussed the result and merged the two coding schemes into one. This process repeated three additional times, until the coding scheme was relatively stable. Then, the scheme was applied to all transcripts. The final coding was reviewed by the third author. The full coding scheme can be found in Appendix E.

5.2 Results

During the interviews, our participants filled in three forms focused on their experience and skill, feature usage, and development techniques and their use in different scenarios. We will present the results from these three forms before we move on to present the themes constructed from the analysis of the remaining interview data. The data from the forms is split into two groups: *students* and *TAs*. The student group has 9 participants, and TAs has 2. This split is done since the TAs background and relation to the course differs significantly from the average student.

Participants Experience and Skill

The median student we interviewed had 6 years programming experience and was in their 4th year of university studies. All but one of the students had previously taken the compiler course at our university. One interviewee was a PhD student, and the rest were students in the computer science and engineering program (see Section 4.2).

For the self-assessed skill, we got responses of 0 and 1, with the average response being 0.45. With little variation in the result for the skill self-assessment, we decided to not split results based on skill.

CODEPROBER Feature Usage

The feature usage form contains a list of features inside CODEPROBER. The interviewees were asked to fill in how much they use each feature, on a scale of [0, 5]. They could also answer with “x” if they did not know said feature existed. The results are presented in Table 1.

Table 1: Average usages of CODEPROBER features. Feature names are shortened to fit the page, see Appendix C for full names. Usage rate is reported on a scale of $[0, 5]$, where 0=Never, 1=Very Rarely, 2=Rarely, 3=Sometimes, 4=Often, 5=Very Often. In case the interviewee had not heard about the feature, they could also answer “x”, which is shown as a separate column.

Feature	Students		TAs	
	Use	#x	Use	#x
Creating probes from text (e.g. Figure 3)	4.67	0	4.50	0
Creating probes from node references	3.88	1	5.00	1
Inspecting probe outputs	4.56	0	4.50	0
Hovering AST node references	4.00	0	2.00	0
Inspecting the AST (e.g. Figure 5)	2.39	0	1.50	0
Creating nested probes (‘▼’)	3.75	1	4.00	0
Using Minimized probes	1.17	3	2.00	1
Using arrows, showing e.g. the control-flow graph	2.61	0	2.00	0
Looking at/conveying information in squiggly lines	4.44	0	3.50	0
Liveness from text updates	3.78	0	4.00	0
Liveness from rebuilding the compiler	4.56	0	4.50	0
Using the “Stop” button for long-running probes	2.00	2	1.00	1
Search probes (e.g. Figure 6)	2.00	8	1.00	1
Tracing	1.20	4	1.50	0

The table shows a few clear winners in terms of features. Creating probes, looking at their outputs, and performing live updates are all very common. These features also happen to be the ones that are described in detail in the lab instructions. Some other features are less used or less well known, such as search probes and tracing (showing what attributes, with their intermediate values, an attribute depends on).

Techniques and Scenarios

The technique and scenario form consists of two parts. First, some questions of how much different development techniques are used for different scenarios. The context here is developing language tools, and most students ($N=5$) answer it solely based on the experiences in the courses. Second, the interviewees are asked to rate how much they *like* using each technique. The results are shown in Tables 2 and 3.

Interview Themes

Here we present the main themes constructed from analyzing the interview transcripts. In cases where individual quotes are used, the participants name is presented as P X , where X is an integer in the range of $[0, 10]$. P0 and P10 are teaching assistants, the rest are students in the program analysis course. Any quotes that

Table 2: Average usage of tools for different development scenarios. Scale is [0, 5], where 0=Never, 1=Very Rarely, 2=Rarely, 3=Sometimes, 4=Often, 5=Very Often. Some names have been shortened to fit the table, full table is available in Appendix D. Note that these numbers are in the context of developing language tooling; they do not apply for software development in general.

Tool	New feature		Understanding code		Bugfixing	
	Students	TAs	Students	TAs	Students	TAs
Print debugging	1.44	3.50	1.11	0.5	2.33	3.50
Breakpoint/step	0.33	1.50	0.11	3.5	1.22	3.00
Test cases	2.67	4.50	1.33	0.5	2.50	4.00
CODEPROBER	4.44	2.50	4.44	0.0	5.00	3.25
AI	0.78	1.75	0.22	0.0	0.44	0.50

Table 3: Average responses to how much the interviewee likes using each development technique. Scale is [1, 5] where 1=Strongly dislike, 2=Dislike, 3=Neutral, 4=Like and 5=Strongly like. Tool names have been shortened to fit the table, full table is available in Appendix D.

Tool	Students	TAs
Print debugging	3.67	2.00
Breakpoint/step	3.89	3.75
Test cases	3.11	3.50
CODEPROBER	4.78	4.25
AI	2.33	2.50

were originally in Swedish have been translated to English.

We believe the interviews achieved a degree of data saturation [GBJ06]. The later interviews mostly repeated themes that earlier interviews had brought up, which makes us believe that our sample size is good enough to make some meaningful observations. We use the syntax “(N=NUM)” below to indicate how many (NUM) of the 11 participants said a given theme.

Theme: Liveness All (N=11) participants mentioned that the *liveness* is a positive aspect of CODEPROBER. Liveness comes in two forms; changing the input text inside CODEPROBER, and updating the language tool being explored. Some interviewees (N=5) mentioned that they rely more on the second kind of liveness, because they know of a specific example that produces incorrect behavior, so there is no need to keep changing the text. They instead keep working on their compiler until the probes display the expected output. This way of working has some similarities to test-driven development. Some related quotes:

P3: “Because it’s often that you are trying to debug something. You have a piece of code that should produce an error, but there is no

error. So you change your own code and recompile and see, is there an error? No, not now either.”

P4: *“Often you have an example you create. And then you use hot reloading to see, to make it work.”*

P5: *“[talking about changing the text] ...not that often actually, it feels like I often have a specific example I want to look at. Rebuilding the compiler and seeing the probes update live, yes quite often.”*

P6: *“Most of the time I write my example and then run CODEPROBER”*

Still, everybody relies on the first kind of liveness as well, which can be seen in Table 1.

Theme: Exploration All (N=11) participants mentioned some form of exploration when talking about CODEPROBER. Liveness, mentioned earlier, is one form of exploration, in that it enables the developer to explore which combinations of input leads to what output. There is also exploration in terms of listing which AST nodes exist, which properties are available to use, and how the different properties link the AST nodes together. Some related quotes:

P2: *“many times we have searched how to jump through the AST”*

P5: *“CODEPROBER is nice because you like do not have to set up anything, you just click around.”*

P7: *“[when asked about what they like most about CODEPROBER] it is how you can step through if you want to find.. like, for example that you can explore a bit. [...] If you did not use CODEPROBER it would be very difficult to find what methods to use.”*

P8: *“It [CODEPROBER] is very good in how you can visualize things. And easily understand, if you have a node here, then I can see all functions that can be used”*

Theme: Freezes and Crashes By far the most common negative feedback relating to CODEPROBER are about freezes and crashes, with all but one mentioning having issues (N=10). Most of the mentioned issues relate to a specific lab in the course. In this lab, the students implement interval analysis, i.e. they should find out what interval a variable’s value may have a given point in the program. To handle loops, their analyses must run iteratively until the analysis values converge. To make sure that the analysis always finishes in a reasonable time they must additionally implement *widening*, i.e. overapproximate the interval values after a certain number of iterations.

During development most students had some bugs in their implementation, such as not implementing widening correctly, and this could cause their analysis

to get stuck in an infinite loop. In some scenarios, this was not presented clearly in the CODEPROBER UI, causing some (N=4) to state that they can't always trust the values in the CODEPROBER UI, because they might be stuck. CODEPROBER tries to recover from this stuck state, but did so incorrectly, which meant that even when CODEPROBER became responsive again, it could be that the values shown in the UI aren't accurate. Some related quotes:

P4: *"it's maybe your compiler that is wrong, but sometimes it's that CODEPROBER get stuck and you have to restart it."*

P5: *"I have no idea why it crashed, because later on it didn't crash at all"*

P8: *"I don't know if the error is because there is an error in my code, or that something froze"*

Theme: Reduced Use of Other Tools While CODEPROBER isn't a direct replacement for any other tool, it does seem to lead to reduced use of the other tools. A majority of the participants (N=9) mention that CODEPROBER has partially or fully replaced testing. Some related quotes:

P2: *"sometimes I am content with seeing that it works as I want in CODEPROBER."*

P4: *"if I have CODEPROBER then I write fewer tests because in a way I have verified it by hand"*

P7: *"when you are finished with something and are going to write a lot of test cases when you already know that it works, thats annoying"*

This reduction in testing is not positive, as CODEPROBER does not help prevent regressions. All but one of the participants (N=10) mention that one of the positive aspects of test cases is that it helps prevent regressions. A majority (N=6) of interviewees also mention that they dislike the process of writing test cases. We believe it is important that the students learn to work with tests more, but also understand some of their rationale for not writing tests here. It is "just" code for a lab after all, and perhaps they would be more open to writing tests for a long-term project.

A majority of participants (6), mentioned using print debugging more outside the course. This is in part due to the nature of the lab assignments. In the labs, the student code could run iteratively until values converged. If print statements were added there, there could be thousands of log entries, reducing the efficiency of printing. Similarly, some (N=4) mentioned using debuggers more outside the course. This is in part because in the course the students work with JASTADD, and debugging support is not very strong for it. JASTADD generates Java code, and this can be stepped through with any standard Java debugger. However, the

developer would have to step through some internal evaluation code that JAST-ADD generates, which negatively impacts the experience. CODEPROBER seems to partially substitute both printing and stepping for the students. Some related quotes:

P3: “CODEPROBER has kind of replaced print debugging in these courses. Because you can, without writing a lot yourself, just open and look at it. [‘it’ being an attribute value that would otherwise be printed]”

P4: “Breakpoint debugging is easier if you are in a project that doesn’t have a lot of JASTADD caching code and such, but is more straight forward. There I use it more. And I use print-f debugging more then as well.”

P8: “CODEPROBER [...] I compare it a lot to a breakpoint/step-debugger. Just because it makes it a lot easier to understand.”

Theme: Reasons for Using AST View Some participants (N=4) mentioned making use of the AST view (shown in Figure 5). The reasons for doing this include trying to understand the structure of the AST, seeing which children belong to which node, and more. Some related quotes:

P1: “I needed to find what one of the children nodes was called.”

P2: “It feels like I did it a lot in the beginning, but maybe less and less.. [...] when you have an understanding of how it is built, then you do not need to look at it.”

P5: “AST View, it is mostly when I’ve forgotten which attributes exists and I don’t feel like looking at the generated Java code, or the pre-written source code. [...] You can see them [the AST nodes] and it is easy to see the methods”

Even among the participants using the AST view, usage was quite low. In Table 1, the average usage of the AST view is 2.39 (\approx “Rarely”). Most (3 of the 4) participants that mentioned using the AST view also mentioned using alternative methods. For example, to understand the structure of the AST, it can be convenient to look at the definition of the AST structure (abstract grammar) instead. Only one person said that they use the AST view “Often”, and this is the only participant that did not previously take the compilers course.

We initially added the AST view in response to students in the compilers course requesting it. They requested it after having seen a similar feature in another tool called DrAST [LTH16], and mentioned that the graphical view helps to build understanding of the structure of the AST. However, once that understanding is achieved, then the AST view might not be as helpful anymore. This matches what

some participants said. A possible takeaway from this is that the usefulness of an AST view is inversely proportional to the users experience level. Therefore, when building a tool meant to be used by students, or a tool to help introduce people to a larger codebase, consider adding an AST view. When building a tool for more experienced users, then it might not be as important to add.

Theme: Continuous use of CODEPROBER When asked about their workflow throughout the labs, a majority of participants (N=8) talked about an iterative process where they switch between writing code and checking the result inside CODEPROBER. Some (N=4) also said their usage of CODEPROBER goes up or down depending on where in the labs they are. Some related quotes:

P4: “[..] when you get into the middle [of the lab] the usage goes down, and then towards the end and beginning you use it [CODEPROBER] a lot.”

P7: “more in the end [of the lab] you do smallfixes and such, then you go into CODEPROBER to see if things actually changed or not. So more coding in the beginning, and then more CODEPROBER in the end.”

P8: “I would say that in the middle [of the labs] you spend more time looking at code. Because I have [..] identified the parts I need to work on in CODEPROBER. [..] And towards the end you go back [to CODEPROBER] to verify if that works.”

P9: “Towards the end it is a lot more CODEPROBER. [..] It is about trying to find why things go wrong. I go into the code, make a small change, recompile and check inside CODEPROBER again.”

We asked the participants for a rough estimate of how much of the total lab time they spend inside CODEPROBER. Both the average and median response of the 7 answers was 30% inside CODEPROBER. The lowest individual answer was 5% from one of the TAs. They said that the reason for this is that they started using JASTADD and building compilers before CODEPROBER existed, so they have gotten used to working with test cases and print debugging instead.

6 Log File Analysis

In this section, we present the method and the results from the log file analysis part of the study.

6.1 Method

For the 2024 instance of the program analysis course, we modified CODEPROBER to log user interactions into a file on the users machine. Here, we describe the

design of these log files, how we collected the data, and how the data was analyzed.

Data Collection

The log files generated by CODEPROBER contain lists of JSON objects on the following form:

```
{ "s":SESSION, "t":TIME, "d":{ "t":TYPE } }
```

SESSION is an ID that is randomly generated every time CODEPROBER is started. TIME is a timestamp. TYPE is the type of event. Depending on the type of event, the object containing TYPE can contain more fields.

The log files are generated on the machine where CODEPROBER is run. After a session is completed, the students using the tool upload the log files with `git` to their repositories. We ask students for informed consent to inspect the log files.

Having the students manually upload their log files has the upside that it makes it very clear to them what data we are collecting. However, there is a risk that some log files get missed due to the extra work of uploading via `git`.

Self-reported Time The course responsible collected anonymous feedback from the students on how much time they spent on the labs. This was done in order to help adjust the labs for future instances of the course. However, this information also fits well with the log analysis, and is therefore used as a supplementary data source.

Data Analysis

Once the course was done, we cloned all the student repositories to a local machine. Then we created a script that traverses all repositories (that had given consent) and processes the log files. The script extracts values from the logs and writes them to csv files that we could import into a spreadsheet. Most extracted pieces of data are quite simple, such as counting how many instances of a certain event type occur. One of the nontrivial points of data being extracted relates to figuring out how much CODEPROBER is used during the labs.

Estimating Use of CODEPROBER Based on our observations, CODEPROBER is often left idle in the background while the developer works on their language tool. When they have updated their tool, or figure out something new they wish to investigate, they bring CODEPROBER back into the foreground and interact with it. Since CODEPROBER is mostly left idle, the time difference between the first and last event of a session cannot be used to measure how much the tool is used. Instead, we can detect periods of use by grouping events together if they happen close to each other. We define a mini session as a sequence of events e_1, e_2, \dots, e_n , where their time differences $|T_{e_{i+1}} - T_{e_i}| \leq \Delta$ for some

Table 4: The number of collected events per lab, and the number of repositories those events were collected from. Lab 1 was performed in groups of two, so “12” represents 24 people. Labs 2 and 3 were done individually.

Lab	#repos	#events
1	12	223277
2	21	90600
3	21	262290
All	54	576167

Table 5: Approximations of number times CODEPROBER was used in the program analysis course labs. Computed by grouping events together based on a certain threshold.

Lab	Average usage counts based on grouping length (Δ)							
	1sec	5sec	10sec	1min	5min	15min	30min	1hour
1	1405.7	648.1	448.3	180.3	48.2	19.8	13.1	10.0
2	477.1	234.7	166.7	75.0	27.0	12.0	8.8	7.3
3	1195.3	565.6	408.8	188.5	60.4	26.8	17.7	12.4
All	962.8	455.3	323.4	142.5	44.8	19.5	13.2	9.9

grouping length Δ . Summarizing the durations of all mini sessions gives us a better approximation of how much CODEPROBER was used.

6.2 Results

In total, 576167 log events were uploaded by the students. The number of events and repositories is presented in Table 4.

Amount of CODEPROBER Use

Table 5 shows the average number of mini sessions. Table 6 shows the combined duration of these mini sessions. Both tables show computed values for several grouping lengths (Δ). Note that the log data only contains active interactions within CODEPROBER. This means that the time spent looking at probe outputs and thinking about what to do is not captured. We believe the data in Table 6 is still quite accurate, but the real numbers may be slightly higher, especially in the columns with lower Δ .

The log data also tells us the number of times the students rebuild their compiler per lab: 103 times on average, and a median of 62. Comparing this with the corresponding number in Table 5 for a grouping of 10 seconds (323.4), we can deduce that about one third of the mini sessions are due to the tools being recompiled. In other words, for each time they rebuild their tool, they perform on average two

Table 6: Approximations of the duration CODEPROBER was used in the program analysis course labs. Computed by grouping events together based on a certain threshold.

Lab	Average usage duration (hours) based on grouping length (Δ)							
	1sec	5sec	10sec	1min	5min	15min	30min	1hour
1	0.19	0.69	1.08	3.00	7.74	11.59	13.87	16.20
2	0.05	0.21	0.34	0.98	2.77	4.82	5.91	7.01
3	0.13	0.55	0.85	2.43	7.27	12.03	15.09	18.66
All	0.11	0.45	0.70	2.00	5.62	9.13	11.25	13.58

Table 7: Self-reported durations of labs.

Lab	Self reported time (hours)		# Responses
	Average	Median	
1	18.8	18	13
2	11.1	10	11
3	38	30	8
All	21	16.5	32

actions inside CODEPROBER (creating probes, changing text, etc.). This matches Table 1, which lists liveness from rebuilding the compiler as one of the most used features, with only “Creating probes from text” having a higher usage rate.

If CODEPROBER is used continuously throughout the labs, then we should be able to estimate how much time students spent on the labs based on how much CODEPROBER is used. Looking at Table 6 with grouping length set to e.g 1 hour, we can approximate that the average lab takes 13.58 hours to complete. The students self-reported the time they spent on the labs, and this is presented in Table 7. The average self-reported lab duration is 21 hours. Since the self-reporting is anonymous, we do not know if the measured times and self-reported times are from the same individuals. That said, we believe that the numbers are similar enough to indicate that CODEPROBER is used continuously throughout the labs.

Spread of CODEPROBER Use

Table 8 shows details about the distribution, quantity and duration of the log events that were collected. The three inner tables represent lab 1, 2 and 3 respectively. Each row is a single student or student group for the corresponding lab. The ten center columns show the distribution of the captured log events, normalized across the span of those events. For example, the table shows that students S22,S10 produced 23.8% of their lab 1 events in the first 10% of the event span.

The table shows a few patterns in the data. First, across all labs the students

Table 8: Distribution of events per 10-percentile of normalized time during the labs (first table is Lab 1 etc). Columns 1-10, 11-20, etc are the 10-percentiles of time. For example, during Lab 1, the students S22,S10 produced 23.8% of the events in the first 10% of the time. The tables also include number of events and number of days between first and last event.

Students	1-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80	81-90	91-100	Events	Days
S22,S10	23.8	6.0	0.9	0.0	0.0	16.2	49.9	0.0	0.0	3.3	4500	17.4
S23,S24	4.5	7.5	0.2	0.0	0.9	0.9	12.6	10.0	30.3	33.3	469	0.1
S14,S8	11.5	5.3	2.2	42.6	0.0	7.9	0.0	4.9	0.0	25.5	19263	31.4
S17,S2	0.1	8.4	4.1	36.2	0.0	11.9	15.7	17.2	0.0	6.5	22789	4.9
S25,S15	0.7	4.6	20.5	17.4	17.2	0.0	0.0	0.0	0.0	39.6	12574	24.2
S19,S6	7.7	0.0	0.0	0.0	8.8	15.2	0.0	0.0	0.0	68.3	58284	15.3
S4,S12	0.0	0.0	0.0	7.1	20.5	0.0	0.0	18.8	0.0	53.5	14229	7.1
S5,S20	40.4	0.0	1.7	0.0	28.0	0.0	1.3	27.6	0.0	1.1	41336	14.0
S21,S7	0.2	0.0	0.0	11.9	26.5	0.0	0.0	0.0	0.0	61.4	6037	11.7
S11,S9	0.1	0.0	17.3	27.7	3.4	19.0	32.0	0.0	0.0	0.5	12731	21.0
S13,S3	28.1	0.0	0.0	0.2	2.5	5.4	1.0	0.0	20.0	42.8	24340	7.1
S18,S16	10.2	0.0	0.0	0.0	0.0	0.0	16.8	1.9	0.0	71.2	6725	6.5
Average	10.6	2.6	3.9	11.9	9.0	6.4	10.8	6.7	4.2	33.9	18606	13.4

Students	1-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80	81-90	91-100	Events	Days
S1	0.7	0.0	0.0	0.0	0.0	0.0	25.3	38.6	7.4	28.1	4027	1.3
S2	3.2	0.0	0.0	0.0	44.5	0.0	0.0	0.4	0.2	51.7	6920	8.1
S3	41.9	56.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.5	8946	7.9
S4	82.1	0.0	10.9	0.0	0.0	0.0	0.0	0.0	0.0	6.9	3162	4.0
S5	1.0	0.0	0.0	0.0	0.0	0.0	0.0	6.3	9.0	83.6	5144	11.7
S6	4.1	0.0	0.0	11.8	0.0	2.9	73.5	0.0	0.0	7.8	6998	5.0
S7	24.3	32.7	8.2	0.0	0.0	0.0	0.0	0.0	0.0	34.8	4907	16.0
S8	7.0	10.3	6.2	5.1	24.6	0.0	3.6	3.7	0.1	39.4	8736	3.3
S9	6.4	0.0	0.0	0.0	15.2	0.0	0.0	0.0	67.7	10.7	3522	8.2
S10	61.9	34.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.7	4168	12.0
S11	20.2	0.0	0.0	0.0	0.0	1.1	0.0	17.7	0.0	61.0	1953	6.7
S12	1.9	30.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	67.8	4904	1.4
S13	1.7	0.0	19.8	8.8	17.5	5.8	0.0	10.5	13.3	22.5	2445	3.7
S14	82.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	17.4	6540	18.9
S15	3.6	32.1	55.4	0.0	4.4	1.7	0.0	0.0	0.0	2.8	4717	23.0
S16	24.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	36.4	39.6	984	1.2
S17	22.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	11.1	66.7	9	0.0
S18	5.0	0.9	0.0	2.1	39.2	1.9	1.5	0.0	24.7	24.7	778	0.1
S19	17.8	32.9	0.0	0.0	13.4	12.1	0.0	0.0	0.0	23.9	4461	2.3
S20	19.8	0.6	0.0	3.6	11.0	8.6	0.0	0.0	0.0	56.3	6217	1.3
S21	13.3	5.0	0.0	15.9	13.3	25.0	5.9	0.0	4.3	17.2	1062	0.2
Average	21.2	11.2	4.8	2.3	8.0	3.5	5.2	3.7	8.3	31.8	4314	6.5

Students	1-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80	81-90	91-100	Events	Days
S1	2.8	7.1	21.0	0.0	3.3	11.6	12.1	5.1	21.6	15.5	12025	9.8
S2	30.8	0.0	0.0	17.5	0.0	0.0	0.0	5.3	0.0	46.4	2825	3.1
S3	40.1	5.1	0.0	0.0	0.0	15.1	5.9	5.3	26.8	1.6	12353	7.8
S4	0.4	0.0	0.4	9.7	64.1	24.7	0.0	0.0	0.6	0.3	5772	27.3
S5	8.9	0.9	0.0	0.0	8.8	19.8	1.5	0.8	27.5	31.6	1413	7.0
S6	0.3	0.5	29.8	31.2	0.9	11.1	0.0	0.6	1.0	24.7	25166	10.6
S7	0.3	2.8	0.0	39.9	0.0	0.0	0.0	15.3	23.3	18.3	8874	7.9
S8	5.8	3.1	15.9	2.8	0.0	0.0	13.1	17.8	20.5	20.9	10349	9.9
S9	51.4	3.8	7.8	0.0	0.0	8.6	9.4	0.0	0.0	18.9	3367	9.2
S10	1.8	5.3	0.2	2.9	56.4	0.0	19.9	0.0	0.0	13.4	12490	18.9
S11	17.9	0.0	0.0	0.0	0.0	0.0	73.3	0.0	0.0	8.8	7422	11.8
S12	49.7	14.5	0.0	0.0	0.0	0.0	0.0	23.6	0.0	12.1	3557	3.3
S13	1.7	0.0	0.0	0.0	5.2	5.2	8.8	13.1	31.3	34.7	7714	10.2
S14	0.9	25.0	0.0	0.0	0.0	0.8	30.1	0.0	27.1	16.9	15544	8.2
S15	0.2	0.0	0.0	0.6	0.0	0.0	35.9	1.1	13.3	48.0	20814	12.6
S25	8.6	0.0	0.0	0.7	22.8	0.0	15.1	21.0	0.0	31.9	3002	3.2
S16	6.8	6.8	0.0	0.0	0.0	0.0	36.8	3.2	5.7	40.6	2032	1.2
S26	21.2	10.1	0.0	0.0	0.0	1.0	41.7	2.4	17.8	5.8	70567	15.1
S19	6.6	3.6	39.0	0.0	0.0	11.8	4.1	0.2	19.0	15.5	3481	9.2
S20	73.7	1.2	0.0	0.0	0.0	1.1	0.8	0.1	1.8	21.2	31940	6.1
S21	36.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	63.7	1583	7.2
Average	17.5	4.3	5.4	5.0	7.7	5.3	14.7	5.5	11.3	23.4	12490	9.5

produce the largest portion of events in the last 10% of the time. Second, there is some tendency to use CODEPROBER more in the beginning of the lab, especially for lab 2 and 3.

We believe the distribution of the log events is due to a combination of the following two things:

1. The students work a moderate amount of time once they get access to the labs. After that they work in quite short bursts, until reaching the deadline of the lab, at which point they put in the largest amount of work.
2. The amount that CODEPROBER is used is different depending on how far the students have progressed in the labs.

The second explanation is supported by the responses to the question about their workflow in the labs (Section 5.2, theme: Continuous use of CODEPROBER).

We believe the events are more spread out for lab 1 because CODEPROBER is more useful later in the lab. The lab involves collecting and solving type constraints. Until the students have implemented the base classes required for representing and solving constraints, there is quite little to inspect in CODEPROBER. For labs 2-3, the data shows some tendency to use CODEPROBER more in the beginning and at the end of the lab. We interpret this as some students use CODEPROBER first to build an understanding of the code they were handed. Once that understanding is in place, the students enter into a phase of mostly developing new features and briefly coming back to CODEPROBER to check that the most recently developed feature works as expected. Towards the end of the lab, they need to make sure everything is working correctly in order to pass the labs.

7 Survey

In this section, we described the method and results of the survey part of the study.

7.1 Method

At the end of each course in Lund University, the students are invited to participate in an anonymous survey about the course. Large parts of the survey is standardized by our faculty, but the course responsible is able to insert up to 4 custom questions for their course. These questions come in the form of statements that the student should respond to on a scale of $[-100, +100]$, where -100 means “fully disagree” and 100 means “fully agree”.

At the end of the two courses where CODEPROBER is used, one or more custom questions relating to CODEPROBER was added. There is a limit of four questions per course instance, and there are other non-CODEPROBER concerns that need to be surveyed as well. Therefore, we only managed to get one near identical across the two courses:

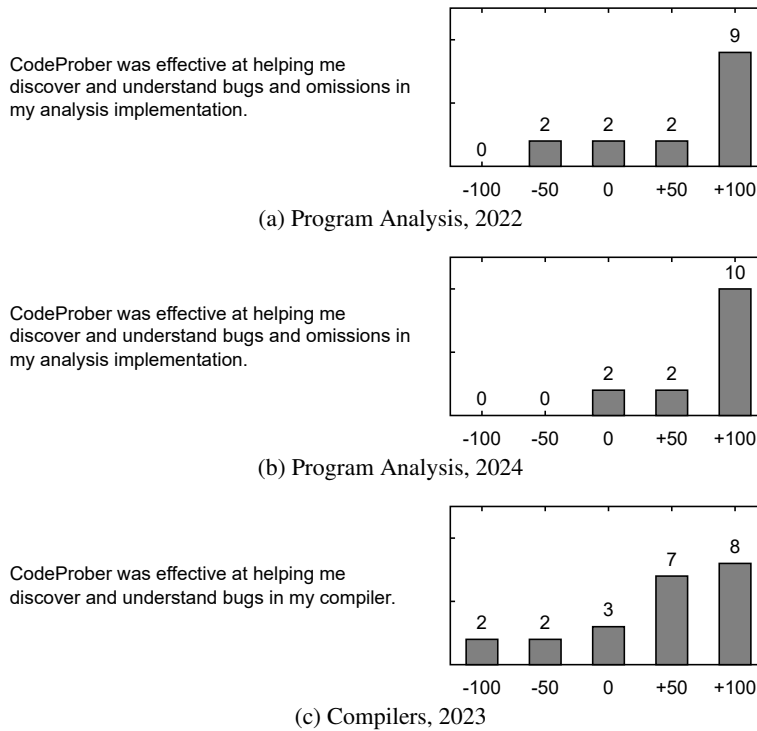


Figure 9: Results from course evaluations in three course instances. X-axis goes from -100 , meaning “fully disagree”, up to $+100$, meaning “fully agree”. Y-axis is the number of individual responses.

Program Analysis Course Survey: *CODEPROBER was effective at helping me discover and understand bugs and omissions in my analysis implementation.*

Compilers Course Survey: *CODEPROBER was effective at helping me discover and understand bugs in my compiler.*

7.2 Results

In total, we have survey responses from three course instances; the program analysis course instance in 2022 and 2024, the compilers course in 2023.

Figure 9 shows the responses from the three course evaluations that included a question about CODEPROBER. In the program analysis course, 23 of the 29 responses “agree” or “fully agree” with the statement that CODEPROBER is “effective”. In the compilers course, the corresponding numbers are 15 out of 22.

The more positive response in the program analysis course can be explained

by a few factors. For one, CODEPROBER is introduced from the very first lab, so it becomes a more integral part of the students' workflow early on. In the compilers course, the students complete ~ 2.5 labs before seeing CODEPROBER, so by that point they have gotten used to working with test cases.

Another factor to explain the difference may be the nature of the lab assignments. In the compilers course the labs vary significantly. The tasks include parsing, analyzing, interpreting and generating code from source code. We have anecdotal evidence from talking with students that CODEPROBER is most used and most appreciated by the students in the analysis lab. The program analysis course labs exclusively consist of analysis-related tasks, and CODEPROBER may therefore be an overall better fit for that course. Another difference is that the students are given an existing compiler in the program analysis course that they extend, rather than building it from scratch (like in the compiler course). Thus, the need to build an understanding of a given codebase is bigger in the program analysis course.

8 Summary of Results

In this study, we have three sources of data, whose background and individual results are presented in Sections 5, 6 and 7 respectively. In this section, we combine all the individual results to answer our research questions.

8.1 RQ1 What is the user experience of using CODEPROBER in an educational setting?

RQ1 can be answered by the interviews and course evaluations. Both paint a positive picture of CODEPROBER. The students found the liveness and exploratory nature of CODEPROBER useful. Being able to explore their language tool without having to add print statements or set a breakpoint enables quick and convenient usage. The course evaluations also confirm the usefulness of CODEPROBER. While the course evaluations are not as extensive as the interviews, they are anonymous, allowing students to voice their opinions without social pressure. Since the survey answers (despite their limited scope) are positive, we are more confident in the interview results.

The most common point of negative feedback was related to freezes and crashes. The way CODEPROBER was used in the 2024 instance of the program analysis course highlighted a few bugs we had not seen before. We plan on improving this for future course instances.

In the interviews, we asked the students questions regarding the usefulness of CODEPROBER and how much they “like” using it. We believe both aspects are important to the overall user experience. For example, while most (N=10) interviewees agree that test cases are useful, a majority (N=6) of interviewees also

mention that they dislike writing test cases, which can lead to less automated testing overall. The score in Table 3 shows that interviewees, on average, “strongly like” using CODEPROBER.

In summary, we consider the answer to **RQ1** to be that the students find CODEPROBER to be a useful tool that is enjoyable to use, despite some technical issues.

8.2 RQ2 How is CODEPROBER used during the development of compilers and static analysis tools in an educational setting?

RQ2 can be answered by the results from the interviews and analysis of log files. From the interviews, we find that CODEPROBER is used to varying degrees throughout the entire labs. This is confirmed by the log files, as there are a number of log events throughout the entire lab series. Almost half of the events occur in the first and last 10% of the labs. We interpret this as some students use CODEPROBER first to build an understanding of the handout code and later on to verify that everything works as expected.

In terms of features, students mainly focused on standard probes, squiggly lines and liveness, as shown in Table 1. The other features are less used, either due to lack of need or because they did not know those features existed. Whether this means that the other features are worthwhile is hard to say, as more experienced developers may have different usage patterns. One thing we can say however is that feature discoverability within CODEPROBER may need some improvement. For example, a majority (N=7) of interviewees remarked that *search probes* seem useful after we demonstrated it to them, but they had no idea that the feature existed.

In summary, we consider the answer to **RQ2** to be that the students made continuous use of CODEPROBER, and they mainly rely on standard probes, squiggly lines and liveness.

8.3 RQ3 How does the use of CODEPROBER compare to other tools used by students during the development process (e.g. debuggers, test cases, print-statements, AI, etc.)?

RQ3 can be answered by results from the interviews. Table 2 shows that students use CODEPROBER more than any other tool in the labs. Test cases and print debugging are quite close to each other in a shared second place, and breakpoint/step debugging and AI assistants are last.

The reason for not using those other tools can in part be explained by the developer experience of working with JASTADD/RAGs. Some attributes in RAGs can have their results cached, and others may evaluate themselves multiple times

until a fixpoint is achieved. This means that when an attribute with a print statement is invoked, the print may not execute at all, or it might execute many times, depending on what has been cached in the AST so far. This hurts the usefulness of print statements. Similarly, a breakpoint/step debugger would have to step through the code that handles caching and fixpoint iteration, which is likely not what the developer is interested in. So while we found in the interviews that CODEPROBER is used a lot more than those other techniques, it is not necessarily because the experience is so much better. It can be because the experience of print debugging and breakpoint/step-debugging is in general worse when working with RAGs, and CODEPROBER is able to fill that role instead.

Note that test cases are different: we believe that they are generally as useful for RAGs as in general software development. Here, we believe the problem is that it is too quick and convenient to verify that something works in CODEPROBER, so students do not want to spend the extra time in creating a proper test case. CODEPROBER does not currently support any form of regression testing, so the reduced use of test cases is not a desired outcome. To help combat this, we plan on adding test support to CODEPROBER, i.e. the ability to save a probe as a test case that can later be run in for example in JUNIT. That would reduce the barrier of creating a test case to a few clicks. In addition, we believe this problem will likely naturally disappear in larger projects, where manually verifying all functionality simply is not practical. There the developers will have a stronger motivation to write tests.

In summary, we consider the answer to **RQ3** to be that the students in our study used CODEPROBER to partially replace print debugging, breakpoint/step-debuggers and test cases. We hypothesize that this is in part because the challenges of debugging RAGs are not handled as naturally by those traditional tools. We cannot say much about AI because none of our survey participants used it very much, neither in the course nor in general software development.

8.4 Threats to Validity

Internal Validity There is risk of bias in the findings from the interviews due to participant responder bias [De1+12]. We have interacted with a majority of these students before in some capacity, either as teaching assistants or supervisors in various courses. The main author of this paper was a teaching assistant in the program analysis course itself. The students know that we are developing CODEPROBER, and therefore they may want to be “kind” and mostly say positive in the interviews. We tried to mitigate this by explicitly asking for feedback on things that do not work well, and by having the person they knew the least well lead the interviews. This risk of bias is also why we spent time on analyzing log files and course evaluations, as these are anonymized and can help verify whether the interviews were overly positive or not.

When designing surveys, acquiescence bias needs to be considered, that is, the tendency for respondents to agree with statements as they are presented. The

course survey in our study only contained a single statement relating to CODEPROBER, due to the size restrictions mentioned in Section 7. The statement is phrased positively, which may positively skew the responses we got. A common method to prevent this bias is to add both positive and negative versions of the same statement [Nun78; SÁ+18]. However, due to the limited total number of statements that could be added to the survey (4), and the fact that there were several aspects of the courses that should be surveyed (not just about CODEPROBER), we could only add one common statement across both courses. Despite the potential for bias, we still believe the survey is valuable due to its anonymity.

The number of students in the interviews is quite low (9), and is a potential threat to the interview findings. However, we did notice data saturation in the interview responses, and we believe that additional interviews would not significantly affect the results.

External Validity This study was conducted on a limited set of students in one specific educational setting. This limits the generalizability of the results. Students from different universities, educational settings or cultural contexts may respond differently to a tool like CODEPROBER. CODEPROBER is mainly created to help develop JASTADD-based tools, as was the case in this study. This means that the results may not be generalizable to other students unless they also use JASTADD, or a similar RAG-based language tool stack. Still, we believe that it would be possible to reproduce the results of the study in a different educational setting, provided that the student group has a similar educational background and use a similar tool stack.

Due to the focus on an educational setting, the results may not be generalizable to other contexts, e.g., industry, hobbyists, etc. However, we still believe that CODEPROBER can be useful for other language tool developers who use a similar tool stack.

9 Related Work

In this section we present related work for four different aspects of this study: 1) Liveness in development tools, 2) User studies, 3) Language tool development, 4) Use of debugging tools.

9.1 Liveness

Many live development tools have similar overall designs. They have a text area where the developer can input code. When changes are made, the tool immediately runs the code, extracts runtime information and displays the information back to the developer [Ler20; Krä+14; Dub+16; McD13; HWX23]. A significant

difference between CODEPROBER and these other tools is that the target audience for CODEPROBER is language tool developers, whereas the other tools target programmers in general. The users of CODEPROBER are usually interested in the behavior of the language tool for a particular input code rather than the behavior of the input code. Like other live development tools, the values displayed in CODEPROBER are computed by the underlying language tool for a given input code. One difference is that the values in CODEPROBER are usually static information about the input code (e.g., the type of an expression) rather than its dynamic behavior. However, the underlying language tool can provide properties that compute the input code's dynamic behavior and display it in CODEPROBER. It would be interesting future work to improve the visualization of such dynamic information, for example, by integrating *projection boxes* (see below) in CODEPROBER. CODEPROBER supports changing the underlying language tool, and this also changes which language is supported in the text area and what properties that are available. In contrast, other tools generally only support one language. We will discuss some of the other tools here in more detail, and how they compare to CODEPROBER.

Lerner [Ler20] presented VERSABOX, a Python editor that displays runtime values of variables in floating windows (*projection boxes*) next to the code. The boxes move around so that the information for the currently focused line is displayed most prominently. In CODEPROBER, the probes are used to select entry points into the underlying language tool. This lets the developer pick which subset of possible information they want to see at any given time. In VERSABOX, the full program is always executed, and in the paper they mention that information overload is a potential problem. We believe CODEPROBER's design scales better when developing larger language tools, both in terms of usability and performance. It would be interesting future work to explore integration of projection boxes into CODEPROBER. The probes could be used as entry points for collecting runtime information of the underlying language tool.

McDermid [McD13] presented YINYANG, a code editor and language that supports live “probing” of expressions. The language supports prefixing any expression with an @-sign. This has no impact on the semantics of the expression, but it causes the runtime value of the expression to be rendered in a small box (“probe”) inside the code editor. The ability to select which expression to probe via the @-sign is similar to how CODEPROBER only displays information once a probe has been created. This helps reduce information overload. However, we believe YINYANG has similar scaling issues as VERSABOX when developing non-trivial language tools. For example, the developer may want to inspect the type of a specific expression inside a large input text. With YINYANG, they would have to 1) parse the input text, 2) programmatically locate the node representing the expression inside the parsed tree, and 3) invoke a “type” function (and possibly its dependencies) with an @ decorator. In CODEPROBER, the developer would right-click the expression and create a probe for “type”.

9.2 User Studies

There are several user studies that investigate the effect of liveness in development tools. A common theme [GTS24; Wil+97; HB07; McC+22; Krä+14; Ler20] is to perform controlled experiments with beginners (e.g., students), and tasks that should be solved during the experiment, with or without live feedback. In contrast, the students in our study use CODEPROBER for several weeks, and work in a codebase of several thousand lines of code. We believe this makes it possible to make interesting observations of how CODEPROBER is used to develop larger, real-world-like codebases. Nonetheless, it could be interesting as future work to perform a controlled experiment with CODEPROBER too. In this section we discuss a few of the related studies in more detail.

Hundhausen et al. [HB07] studied the effect using development environments with three levels of feedback: 1) No feedback is given, 2) Feedback is given after a button is pressed, and 3) Live feedback is always given. They found that any form of feedback is better than none. However, they found no significant difference between continuous live feedback and feedback that was given after a button press. They suggest that continuous live feedback may sometimes be a distraction, and the button allows the developer to wait until they are ready to take advantage of the feedback. CODEPROBER combines liveness and letting the developer choose when to receive feedback. By default, CODEPROBER gives no feedback to the user. Only when the user has created a probe will some form of feedback be given. In addition, changing the specification for the underlying language tool has no immediate impact, the developer must compile the underlying language tool for changes to appear inside CODEPROBER. Creating probes and compiling the underlying language tool can be compared to the button in Hundhausen et al.’s study. They enable the developer to get live feedback, but only when they are ready for it. Once a probe is created, then the feedback will be given continuously when the input text is changed, like scenario 3 in Hundhausen et al.’s study.

Kramer et al. [Krä+14] extended the JavaScript IDE Brackets⁸ with a view that displays runtime values of the code to the side, similar to the work on projection boxes discussed above. They also performed a user study where they compared the performance of developers based on if they received live feedback inside the editor or not. Interestingly, they did notice some significant workflow differences. Developers without their extension (without live feedback) tended to solve tasks in two phases: first they implemented most of the functionality, and then they spent time on making sure everything works. The developers with live feedback instead tended to continuously fix issues as they wrote the code. In our study, we did not have a control group that worked without CODEPROBER, so we can not observe this difference directly. However, some interviewees did hint towards this iterative develop-fix workflow (Section 5.2, theme: Continuous use of CODEPROBER).

Rein et al. [Rei+24] extended a Squeak/Smalltalk environment with a “cross-

⁸<https://brackets.io/>. Accessed September 2024.

cutting perspective” to help the developer filter and visualize execution traces. This perspective was evaluated in an exploratory user study with 7 students. The interviews were each 2.5 hours long. First, the interview subjects worked to solve some tasks using the new perspective for 1.5 hours. Then for the remaining time they were asked about their experiences. Live task solving like this could have let us make some more detailed observations about exactly how CODEPROBER is used.

9.3 Language Tooling Development

Language workbenches are tools that “supports the efficient definition, reuse and composition of languages and their IDEs” [Erd+13]. Language workbenches are often able to provide liveness features, in part because they control and tightly integrate parsing, semantic specification, testing, and more. For example, Gabriël et al. added incremental compilation of the grammar in Spoofox [KEV16]. This enabled grammar changes to immediately show updated parse trees and/or syntax errors within Spoofox. Dubroy et al. [Dub+16] created Ohm, a workbench that lets the developer specify a grammar, examples (\simeq test cases) and semantic actions. The actions behave similarly to synthesized attributes in Reference Attribute Grammars [Hed00], except they are specified on the concrete syntax tree rather than the abstract one. Whenever grammar, examples or actions are modified, live feedback is given to the user. The tight integration inside a workbench enables some features they provide, but it can also sometimes be a limitation. For example, if one wanted to use Spoofox or Ohm with a custom parser, then this would likely require forking and modifying the respective tool’s source code. JASTADD helps the language tool developer with specifying semantics, but does not have any opinions regarding parsing or debugging. The goal of CODEPROBER is to provide debugging functionality for JASTADD and tools similar to it. By itself, CODEPROBER is not a language workbench. However, by combining a parser generator, JASTADD and CODEPROBER, it is possible to get an experience similar to one provided by language workbenches.

There exists several other tools that enable some form of AST exploration. Some examples include Noosa [Slo99], DrAST [LTH16], Aki [Ike+00], and EvDebugger [RCHS14]. None of these tools have the concept of probes that are updated after changes to the source text. They also focus much of their interaction on the AST. CODEPROBER on the other hand handles most user interactions in terms of source code, which we believe scales better for more complex language tools and when exploring larger input texts. For example, we think that displaying and navigating an AST view for 500 lines of Java code is non-trivial, while rendering and navigating 500 lines of text is simpler. These tools are discussed in more detail in [Ala+24].

9.4 Debugging

CODEPROBER can aid in the process of debugging, but it is not a traditional debugger by itself. Part of the goal of this study is to investigate if, and to what degree, the need for debugging can be met by CODEPROBER. This section presents studies on the usage (or lack thereof) of traditional debuggers.

Ko et al. [Ko+23] performed a survey to investigate which barriers and factors prevent students from making more use of debuggers. They did this with an online survey where 73 students participated. They found that the lack of focus on debugger usage in academic courses is one of the main reasons. Also, the complexity of debuggers high initial learning curve was one reason. In our interviews a majority (N=8) of interviewees also mentioned that a downside of debuggers is that they can be quite difficult to set up and run.

Beller et al. [Bel+18] performed a mixed-method study to determine “how developers debug software problems in the real world”. They performed an online survey where 458 developers participated, and performed automated data collection from IDEs of 108 people. They found that “developers spend surprisingly little time in the debugger; only 13% of their total development time on average”. Many developers preferred using print debugging instead. The reasons they found for this behavior include the complexity of modern debuggers. Another reason is experience – they found that more experienced developers tended to use debuggers slightly more. In the context of language tools, we believe that the design of CODEPROBER solves some problems of complexity found with traditional debuggers, as discussed in Section 2.1. That is, finding the AST node of interest during debugging is non-trivial with traditional debuggers, but is one of the primary features of CODEPROBER. The fact that our students seem to use CODEPROBER more than debuggers, test cases and print debugging strengthens this claim. But even with CODEPROBER, we believe that debuggers are still very useful, especially for more complex bugs.

10 Conclusions

We have presented a mixed method study of the experience of using CODEPROBER in an educational setting. Our findings show that the students find CODEPROBER to be useful, and they make continuous use of it during the course labs. One of the most used features is its liveness, e.g. the ability to nearly instantly respond to changes. We found that CODEPROBER to some extent replaces existing tools and techniques like test cases, breakpoint/step-debugger, etc. We hypothesize that this is in part because the nature of RAGs does not work so well with these more general tools, so CODEPROBER is able to better meet this demand. The reduced use of test cases can be seen as a negative outcome, as CODEPROBER does not support automatic regression testing. For this reason, we are interested in adding support for testing inside CODEPROBER in the future.

Further research is needed to explore how CODEPROBER or a tool like it would be perceived by people outside an educational setting, for example industry professionals. There is also interesting future research in trying to apply it to completely different domains. We have used CODEPROBER exclusively for language tooling, but in theory any program that performs computations on a tree structure is a possible target. For example, many models for building UI are tree structures. Scene graphs (often used in games) and the Document Object Model (“DOM”, used in browsers) fit into this category, and are therefore possible future targets for CODEPROBER.

Acknowledgements

This work was funded by ELLIIT – Excellence Center at Linköping-Lund on Information Technology.

We want to thank all the participants in this study, Christoph Reichenbach for integrating CODEPROBER into the program analysis course and Görel Hedin for integrating CODEPROBER into the compilers course.

References

- [Ala+24] Anton Risberg Alaküla et al. “Property probes: Live exploration of program analysis results”. In: *Journal of Systems and Software* 211 (2024), p. 111980.
- [Bel+18] Moritz Beller et al. “On the dichotomy of debugging behavior among programmers”. In: *Proceedings of the 40th International Conference on Software Engineering*. 2018, pp. 572–583.
- [Del+12] Nicola Dell et al. ““Yours is better!”: participant response bias in HCI”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’12. Austin, Texas, USA: Association for Computing Machinery, 2012, 1321–1330.
- [DS11] Tijds van Der Storm. “The Rascal language workbench”. In: *CWI. Software Engineering [SEN]* 13 (2011), p. 14.
- [Dub+16] Patrick Dubroy et al. “Language hacking in a live programming environment”. In: *Proceedings of the LIVE Workshop co-located with ECOOP 2016*. 2016.
- [EH07a] Torbjörn Ekman and Görel Hedin. “The JastAdd system - modular extensible compiler construction”. In: *Sci. Comput. Program.* 69.1-3 (2007), pp. 14–26.
- [EH07b] Torbjörn Ekman and Görel Hedin. “The Jastadd Extensible Java Compiler”. In: *SIGPLAN Not.* 42.10 (2007), 1–18.

- [Erd+13] Sebastian Erdweg et al. “The state of the art in language workbenches: Conclusions from the language workbench challenge”. In: *Software Language Engineering: 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings 6*. Springer. 2013, pp. 197–217.
- [GTS24] Oliver Graf, Sverrir Thorgeirsson, and Zhendong Su. “Assessing Live Programming for Program Comprehension”. In: *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1, ITiCSE 2024, Milan, Italy, July 8-10, 2024*. Ed. by Mattia Monga et al. ACM, 2024.
- [Gre89] Thomas RG Green. “Cognitive dimensions of notations”. In: *People and computers V* (1989), pp. 443–460.
- [GBJ06] Greg Guest, Arwen Bunce, and Laura Johnson. “How many interviews are enough? An experiment with data saturation and variability”. In: *Field methods* 18.1 (2006), pp. 59–82.
- [HWX23] Devamardeep Hayatpur, Daniel Wigdor, and Haijun Xia. “CrossCode: Multi-level Visualization of Program Execution”. In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, CHI 2023, Hamburg, Germany, April 23-28, 2023*. Ed. by Albrecht Schmidt et al. ACM, 2023, 593:1–593:13.
- [Hed00] Görel Hedin. “Reference attributed grammars”. In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.
- [Hed09] Görel Hedin. “An Introductory Tutorial on JastAdd Attribute Grammars”. In: *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*. Ed. by João M. Fernandes et al. Vol. 6491. Lecture Notes in Computer Science. Springer, 2009, pp. 166–200.
- [HM03] Görel Hedin and Eva Magnusson. “JastAdd—an aspect-oriented compiler construction system”. In: *Science of Computer Programming* 47.1 (2003), pp. 37–58.
- [HB07] Christopher D. Hundhausen and Jonathan Lee Brown. “An experimental study of the impact of visual semantic feedback on novice programming”. In: *J. Vis. Lang. Comput.* 18.6 (2007), pp. 537–559.
- [Ike+00] Yohei Ikezoe et al. “Systematic Debugging of Attribute Grammars”. In: *Proceedings of the Fourth International Workshop on Automated Debugging, AADEBUG 2000, Munich, Germany, August 28-30th, 2000*. Ed. by Mireille Ducassé. 2000.

- [KV10] Lennart CL Kats and Eelco Visser. “The Spoofox language workbench: rules for declarative specification of languages and IDEs”. In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 2010, pp. 444–463.
- [Knu68a] Donald E. Knuth. “Semantics of Context-Free Languages”. In: *Math. Syst. Theory* 2.2 (1968), pp. 127–145.
- [Ko+23] Minhyuk Ko et al. “Exploring the Barriers and Factors that Influence Debugger Usage for Students”. In: *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2023, Washington, DC, USA, October 3-6, 2023*. IEEE, 2023, pp. 168–172.
- [KEV16] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. “Towards Live Language Development”. In: *Proceedings of the LIVE Workshop co-located with ECOOP 2016*. 2016.
- [Krä+14] Jan-Peter Krämer et al. “How live coding affects developers’ coding behavior”. In: *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2014, Melbourne, VIC, Australia, July 28 - August 1, 2014*. Ed. by Scott D. Fleming, Andrew Fish, and Christopher Scaffidi. IEEE Computer Society, 2014, pp. 5–8.
- [Ler20] Sorin Lerner. “Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming”. In: *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*. Ed. by Regina Bernhaupt et al. ACM, 2020, pp. 1–7.
- [LTH16] Joel Lindholm, Johan Thorsberg, and Görel Hedín. “DrAST: an inspection tool for attributed syntax trees (tool demo)”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*. Ed. by Tijs van der Storm, Emilie Balland, and Dániel Varró. ACM, 2016, pp. 176–180.
- [McC+22] Alan T McCabe et al. “Visual Cues in Compiler Conversations.” In: *PPIG*. 2022, pp. 25–38.
- [McD13] Sean McDirmid. “Usable live programming”. In: *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*. Ed. by Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld. ACM, 2013, pp. 53–62.
- [Nie93] Jakob Nielsen. “Response times: the three important limits”. In: *Usability Engineering* (1993).

- [Nun78] Jum C Nunnally. “An overview of psychological measurement”. In: *Clinical diagnosis of mental disorders: A handbook* (1978), pp. 97–146.
- [Pei+22] Norman Peitek et al. “Correlates of programmer efficacy and their link to experience: a combined EEG and eye-tracking study”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore: Association for Computing Machinery, 2022, 120–131.
- [Rad+23] Alec Radford et al. “Robust Speech Recognition via Large-Scale Weak Supervision”. In: *Proceedings of the 40th International Conference on Machine Learning*. Ed. by Andreas Krause et al. Vol. 202. Proceedings of Machine Learning Research. PMLR, 2023, pp. 28492–28518.
- [Rei+24] Patrick Rein et al. “Broadening the View of Live Programmers: Integrating a Cross-Cutting Perspective on Run-Time Behavior into a Live Programming Environment”. In: *Art Sci. Eng. Program*. 8.3 (2024).
- [RCHS14] Daniel Rodríguez-Cerezo, Pedro Rangel Henriques, and José-Luis Sierra. “Attribute grammars made easier: EvDebugger a visual debugger for attribute grammars”. In: *2014 International Symposium on Computers in Education (SIIE)*. 2014, pp. 23–28.
- [Slo99] Anthony M. Sloane. “Debugging Eli-Generated Compilers With Noosa”. In: *Compiler Construction, 8th International Conference, CC’99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS’99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*. Ed. by Stefan Jähnichen. Vol. 1575. Lecture Notes in Computer Science. Springer, 1999, pp. 17–31.
- [SH11] Emma Söderberg and Görel Hedin. “Building semantic editors using JastAdd: tool demonstration”. In: *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*. 2011, pp. 1–6.
- [SÁ+18] Javier Suárez Álvarez et al. “Using reversed items in Likert scales: A questionable practice”. In: *Psicothema*, 30 (2018).
- [Wil+97] E. M. Wilcox et al. “Does Continuous Visual Feedback Aid Debugging in Direct-Manipulation Programming Systems?” In: *Human Factors in Computing Systems, CHI ’97 Conference Proceedings, Atlanta, Georgia, USA, March 22-27, 1997*. Ed. by Steven Pemberton. ACM/Addison-Wesley, 1997, pp. 258–265.

Appendices

A Interview Design

The script used during interviews.

Interview structure

0. Physical setup

- The interview happens in Niklas' room
- One "demo computer" with VS Code and CodeProber running, in case we/the interviewee wants to show something
 - This computer will also run audio recording the whole time
- Process for getting the computer ready:
 - Open quicktime, prepare for starting audio recording (File->New Audio Recording, don't press start yet)
 - Open the directory `edap15-2024-exercise-2` in Visual Studio Code
 - Note: this is cloned from <https://git.cs.th.se/crskchen/edap15-2024-exercise-2> at revision `8b2501b748ab784c3936842207e8588ae9dfc52a`, with a small modification to make at least 1 "report" show up in CodeProber.
 - Run `./gradlew clean jar`
 - Run `./code-prober.sh examples/hw2-task-3-4.html`
 - Click the link <https://localhost:3888/> to open in browser of choice, preferably Chrome or Firefox (there may be issues in Safari)
 - Close all other windows to avoid distraction (only keep Quicktime, VS Code and browser).
- One phone, running audio recording as well, laying on the table
- One note-taking computer, to be used by the interviewer that isn't talking.
- Some cans of drinks (soda, water, ...) and cookies
- A set of printed papers for each test subject:
 - Informed consent form (2 copies, one to sign and one to take home)
 - The tables to be filled in (Table 1, 2 and 3 below)
- Pens

1. Introduction

1.1. Background

- Offer a drink and cookies
- Hello, welcome. Introduce Niklas and Anton. Niklas will ask questions and Anton will take notes.
- Go over overview and purpose of study:
 - CodeProber is an active research project
 - We are curious of how you use CodeProber
 - We try different solutions
 - We would like to know what work and what doesn't work
 - We are happy to get honest feedback, including flaws
 - This helps us better understand the tool and how it can be improved
- Research questions:
 - How is CodeProber used during the development of compilers and static analysis tools?
 - What is the user perception of CodeProber? Things that you like/don't like.
 - How does CodeProber compare to other tools during the development process (e.g debuggers, test cases, print-statements)?
- Interview information:
 - We will record for transcription purposes. These transcriptions will be stored locally on our devices (accessed by Niklas/Anton).
 - Anonymized results will be discussed in the research team for this study (Anton, Niklas, Emma Soderberg).
 - Anonymized results from interviews may be included in a publication
 - You can withdraw from the study within 1 month of the interview
- Ask participant to sign [Informed Consent Form](#)
- Start audio recording.

1.2. Warmup

Hand them this form on a printed piece of paper: [Table 1: Background Information](#). Ask them to fill the form. Note: 1 question require a longer answer, so it should be answered verbally. Also ask:

- Experience of compiler development/program analysis beyond the compiler and program analysis course?

Briefly describe the rest of the interview:

- We'll begin by talking about CodeProber specifically
- Then talk about other development tools, and how they all fit together

2. Main

2.1. Intro to CodeProber

Showing CodeProber

- Present one of our laptops. It has VS Code and CodeProber running on it.
- Exercise 2 - dataflow analysis. Null-pointer- and dead assignment analysis
- Please show us and think aloud:
 - Please open a probe showing `Program.reports`
 - Niklas makes changes to remove the null pointer bug (illustrating liveness)
 - Please open an AST view / AST probe for the function called `f`

Features

- Hand them this table on a printed piece of paper (2 sided): [Table 2: CodeProber Features](#)
- CodeProber supports a number of different features.
- For each one of the following, please fill in #how much you use it.
- Please think aloud
- (If they don't recognise a feature, show it on the laptop)

2.2. Workflow

Lets pretend that you are working on one of the labs in the program analysis course. On a high level, what are you spending your time on? Where do you look for information, what editor do you write code in, how do you debug problems, etc. In other words, please describe your workflow. You can describe it just using words, or use the laptop as a reference.

- (Wait a while while they answer, then add the questions below)
- Can you estimate how large portion of your time you spend interacting with the things you mentioned? For example, X% writing code, Y% in CodeProber, Z% reading lecture slides, etc.
- Do you ever look at the code generated by JastAdd?
- If/when you work with test cases, do you prefer writing them early (e.g "Test-Driven Development", TDD), late, or a mix of both?
- Is there any difference in the workflow based on how far along you are in the lab? For example:
 - in the beginning, in the middle, and in the end?
- If you took the compiler course, was there any difference in your workflow for those labs?

2.3. Comparison to other development tools

When working in a codebase, you almost inevitably run into problems. When you do, there are several different tools or techniques you can use to overcome the problem. We have chosen to focus on 5.

Hand them this table on a printed piece of paper:

- [Table III: Development Tools](#)
- First table is about how much you use a tool/technique
- Second table is about how you like using the tool/technique.
- Please fill in the tables and think aloud

For each technique/tool also discuss:

- How much they are used when you develop software in general
- Its main strengths and weaknesses

Do you feel there is a scenario or tool we have missed above? If so, please add it! (they might mention code review or pair programming here perhaps)

2.4. Perception of CodeProber

- Back to CodeProber
- What do you like most with CodeProber? Why?
- What do you like least with CodeProber? Why?
- What feature(s) would you want to add to CodeProber?

If you took the compiler course, did you notice any differences between CodeProber in the compiler vs program analysis course?

- If yes, any thoughts on the difference?
(Main difference from their perspective is probably the predefined minimised probes)

2.5. Thoughts on labs

In the program analysis course labs you have had access to CodeProber every lab so far.

- What do you think about the labs?
- Was your workflow different in the different labs? For example, did you use test cases more or less in certain labs, etc.
 - In case they need reminder of the different labs, they are:
 1. JastAdd intro
 2. Type Inference
 3. Dead assignment analysis
 4. Interval & array bounds analysis

If you took the compiler course, then the similar question as before.

- What do you think about the labs?
- Did you use CodeProber more or less in any of the labs?
 - In case they need reminder of the different labs, they are:
 1. Scanning & Parsing (LL1)
 2. Scanning & Parsing (LR)
 3. Visitor pattern, basic properties
 4. Semantic analysis (names & types)
 5. Interpretation
 6. Code generation

3. Cooldown

- Anything you would like to add to this interview?
 - Topics we have missed/not spent enough time on
- Do you have any questions to us?

4. Closing

- Thank you for participating, here is your Coffee mug

B Background Information Form

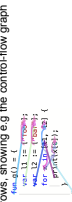

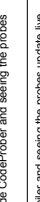



The first form filled in by the interviewees.



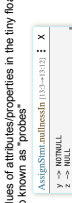




Table I: Background Information

Which program are you in? (D, C, ..)					
Which year are you in at LTH?					
How many years have you been programming? Free time counts!					
How would you compare your programming skills to your classmates?	Much Worse	Worse	Identical	Better	Much Better
Did you take the compiler course?					
What made you apply to the program analysis course (and compiler course, if applicable)	<i>(answer verbally)</i>				

C CODEPROBER Feature Form

The second form filled in by the interviewees, two pages.

Feature	How much you use it x = Haven't heard of it before 0 = Never 1 = Very Rarely 2 = Rarely 3 = Sometimes 4 = Often 5 = Very Often
Looking at/drawing arrows, showing e.g. the control-flow graph 	
Looking at/conveying information in squiggly lines 	
Changing the text inside CodeProber and seeing the probes update live 	
Re-building your compiler and seeing the probes update live Stopping a long-running probe via the stop button in the UI 	
Search probes like "--nullnessValue" 	
Tracing 	
Other feature (if there is something you feel is missing above)	

Feature	How much you use it x = Haven't heard of it before 0 = Never 1 = Very Rarely 2 = Rarely 3 = Sometimes 4 = Often 5 = Very Often
Creating probes by right clicking in the text 	
Creating probes by clicking on node references in the output of a probe 	
Inspecting values of fields/properties in the tiny floating windows, also known as "probes" 	
Hovering AST node references in a probe to see where they are in the text. 	
Inspecting the AST (the graphical view) 	
Nested probes, i.e. clicking the downwards-facing triangle (▼) next to an AST node reference to create a probe inside a probe. 	
Minimizing probes to- and opening probes from- the minimized area at the top of the screen 	

D Tool/Scenario Form

The third form filled in by the interviewees.

Table III: Development Tools/Techniques
During the development of compilers or static analyzers

How much do you **use** each tool/technique

Scenario	Printf debugging	Breakpoint/step debugging	Test cases	CodeProber	AI (Copilot / ChatGPT / similar)
Developing a new feature					
Building understanding of a codebase <i>Could be code you wrote yourself some time ago, or somebody else's code</i>					
Fixing a bug					

How much do you **like** using each tool/technique

Printf debugging	Breakpoint/step debugging	Test cases	CodeProber	AI (Copilot / ChatGPT / similar)

Legend:

x = Haven't heard of it before
 0 = Never
 1 = Very Rarely
 2 = Rarely
 3 = Sometimes
 4 = Often
 5 = Very Often

Legend:

x = Haven't heard of it before
 1 = Strongly dislike
 2 = Dislike
 3 = Neutral
 4 = Like
 5 = Strongly like

E Coding Scheme

The coding scheme extracted from the interview transcripts.

Category 1	Category 2	Category 3	Category 4	C Code	Quotes
Background	Reason for selecting course			E1	
		Found complex course fun/interesting		E1.1	[1] [2] [3]
		Analysis course seemed fun/interesting		E1.1.2	[4] [5] [6]
		Wanted a more challenging course		E1.1.3	[7]
		It part of the specialisation		E1.1.3	[8] [9] [10]
		Interested in statically analysing programs		E1.1.4	[11]
		Simple dev. experience outside course		E1.2	
		Yes	As hobby	E1.2.1	
			As work	E1.2.2	[12] [13]
			Course (project course / master's thesis)	E1.2.3	[14]
Before Using CodeProver		Yes	E1.2.2	[15] [16] [17] [18]	
	Was created normal probe		E2		
		Yes	E2.1.1	[19] [20] [21] [22] [23] [24] [25] [26]	
	Was created AST view probe		E2.1.2		
		Yes	E2.2	[27]	
CodeProver features		Yes, with some help	E2.2.1	[28] [29]	
		Yes	E2.2.2	[30] [31]	
		Problem with accuracy	E2.2.3	[32]	
			E2.2.4	[33]	
	Exploration		E3		
		Iteratively following node references across the AST	E3.1.1	[34] [35]	
		Checking that attribute reference goes to parent node	E3.1.2	[36]	
		Editing available attributes/properties	E3.1.3	[37] [38] [39]	
		Explore nodes that aren't visible in source code	E3.1.4	[40]	
	Revisions		E3.2	[41]	
	Changing how to CodeProver to cause updates	E3.2.1	[42] [43] [44] [45] [46] [47] [48]		
	Rebuilding compiler to cause updates	E3.2.2	[49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60]		
	More often than changing the test, due to working on one example	E3.2.3	[61] [62] [63] [64]		
Revising		E3.3	[65] [66] [67] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77] [78] [79] [80]		
	Differentiating multiple probes	E3.3.1	[81] [82]		
	Used when confused about where the probe is	E3.3.2	[83]		
Using probes from node references		E3.4	[84] [85] [86] [87] [88]		
	Did not know clicking on node name would open a new window (only used nested probes)	E3.4.1	[89] [90] [91]		
	Didn't know you could create nested probes	E3.4.2	[92]		
AST View		E3.5	[93] [94] [95] [96] [97] [98] [99] [100]		
	When it's used	E3.5.1	[101]		
		E3.5.1.1	[102]		
		E3.5.1.2	[103]		
	Why it's used	E3.5.2	[104]		
		E3.5.2.1	[105] [106] [107] [108] [109] [110]		
		E3.5.2.2	[111] [112]		
		E3.5.2.3	[113] [114]		
		E3.5.2.4	[115] [116] [117] [118] [119] [120]		
	Observations	E3.5.3	[121]		
		E3.5.3.1	[122] [123] [124] [125]		
		E3.5.3.2	[126] [127]		
		E3.5.3.3	[128] [129] [130] [131] [132]		
	Include about its use	E3.5.4	[133]		
	Used more in compiler course	E3.5.5	[134]		
	Had to build AST yourself there. In analysis course, the AST is already correctly built	E3.5.5.1	[135]		
Missing probes		E3.6	[136] [137] [138] [139] [140] [141] [142] [143] [144] [145] [146] [147] [148] [149] [150]		
Arrows		E3.7	[151]		
	Confusing L/R/W	E3.7.1	[152]		
	Mainly used in one tab	E3.7.2	[153] [154] [155] [156] [157] [158] [159] [160]		
	Pointing to the terminal/output	E3.7.3	[161] [162] [163] [164] [165] [166] [167] [168] [169] [170]		
	File checkout	E3.7.4	[171]		
	Zooming in to look at the arrows more clearly	E3.7.5	[172]		
	Highlight underlining	E3.7.6	[173]		
	Just empty lines to see arrows more clearly	E3.7.7	[174]		
	More useful in tabs 2 and 3	E3.8	[175] [176] [177] [178] [179] [180] [181] [182] [183] [184] [185] [186] [187] [188] [189] [190]		
Sticky lines		E3.8.1	[191]		
	Highlighting an error to check if it appears in error probe	E3.8.2	[192]		
	Use check that we implement the right things	E3.8.3	[193] [194] [195]		
	Better than probe windows.	E3.8.4	[196]		
	Increase often you are interested in a specific node/place, not a lot of nodes/lines	E3.8.4.1	[197]		
Stopping long running		E3.9	[198] [199] [200] [201] [202] [203] [204] [205] [206] [207] [208] [209] [210] [211] [212] [213] [214] [215] [216] [217] [218] [219] [220] [221] [222] [223] [224] [225] [226] [227] [228] [229] [230] [231] [232] [233] [234] [235] [236] [237] [238] [239] [240] [241] [242] [243] [244] [245] [246] [247] [248] [249] [250] [251] [252] [253] [254] [255] [256] [257] [258] [259] [260] [261] [262] [263] [264] [265] [266] [267] [268] [269] [270] [271] [272] [273] [274] [275] [276] [277] [278] [279] [280] [281] [282] [283] [284] [285] [286] [287] [288] [289] [290] [291] [292] [293] [294] [295] [296] [297] [298] [299] [300] [301] [302] [303] [304] [305] [306] [307] [308] [309] [310] [311] [312] [313] [314] [315] [316] [317] [318] [319] [320] [321] [322] [323] [324] [325] [326] [327] [328] [329] [330] [331] [332] [333] [334] [335] [336] [337] [338] [339] [340] [341] [342] [343] [344] [345] [346] [347] [348] [349] [350] [351] [352] [353] [354] [355] [356] [357] [358] [359] [360] [361] [362] [363] [364] [365] [366] [367] [368] [369] [370] [371] [372] [373] [374] [375] [376] [377] [378] [379] [380] [381] [382] [383] [384] [385] [386] [387] [388] [389] [390] [391] [392] [393] [394] [395] [396] [397] [398] [399] [400] [401] [402] [403] [404] [405] [406] [407] [408] [409] [410] [411] [412] [413] [414] [415] [416] [417] [418] [419] [420] [421] [422] [423] [424] [425] [426] [427] [428] [429] [430] [431] [432] [433] [434] [435] [436] [437] [438] [439] [440] [441] [442] [443] [444] [445] [446] [447] [448] [449] [450] [451] [452] [453] [454] [455] [456] [457] [458] [459] [460] [461] [462] [463] [464] [465] [466] [467] [468] [469] [470] [471] [472] [473] [474] [475] [476] [477] [478] [479] [480] [481] [482] [483] [484] [485] [486] [487] [488] [489] [490] [491] [492] [493] [494] [495] [496] [497] [498] [499] [500]		
	Restart CodeProver instead in terminal	E3.9.1	[501] [502]		
	It would run with circular attributes (infinite loop tendency)	E3.9.2	[503]		
	Never had issues with long running probes	E3.9.3	[504]		
	Check for correctness in terminal	E3.9.4	[505]		
	It doesn't really do anything	E3.9.5	[506]		
Search probe		E3.10	[507] [508] [509] [510] [511] [512] [513] [514] [515] [516] [517] [518] [519] [520] [521] [522] [523] [524] [525] [526] [527] [528] [529] [530] [531] [532] [533] [534] [535] [536] [537] [538] [539] [540] [541] [542] [543] [544] [545] [546] [547] [548] [549] [550] [551] [552] [553] [554] [555] [556] [557] [558] [559] [560] [561] [562] [563] [564] [565] [566] [567] [568] [569] [570] [571] [572] [573] [574] [575] [576] [577] [578] [579] [580] [581] [582] [583] [584] [585] [586] [587] [588] [589] [590] [591] [592] [593] [594] [595] [596] [597] [598] [599] [600] [601] [602] [603] [604] [605] [606] [607] [608] [609] [610] [611] [612] [613] [614] [615] [616] [617] [618] [619] [620] [621] [622] [623] [624] [625] [626] [627] [628] [629] [630] [631] [632] [633] [634] [635] [636] [637] [638] [639] [640] [641] [642] [643] [644] [645] [646] [647] [648] [649] [650] [651] [652] [653] [654] [655] [656] [657] [658] [659] [660] [661] [662] [663] [664] [665] [666] [667] [668] [669] [670] [671] [672] [673] [674] [675] [676] [677] [678] [679] [680] [681] [682] [683] [684] [685] [686] [687] [688] [689] [690] [691] [692] [693] [694] [695] [696] [697] [698] [699] [700] [701] [702] [703] [704] [705] [706] [707] [708] [709] [710] [711] [712] [713] [714] [715] [716] [717] [718] [719] [720] [721] [722] [723] [724] [725] [726] [727] [728] [729] [730] [731] [732] [733] [734] [735] [736] [737] [738] [739] [740] [741] [742] [743] [744] [745] [746] [747] [748] [749] [750] [751] [752] [753] [754] [755] [756] [757] [758] [759] [760] [761] [762] [763] [764] [765] [766] [767] [768] [769] [770] [771] [772] [773] [774] [775] [776] [777] [778] [779] [780] [781] [782] [783] [784] [785] [786] [787] [788] [789] [790] [791] [792] [793] [794] [795] [796] [797] [798] [799] [800] [801] [802] [803] [804] [805] [806] [807] [808] [809] [810] [811] [812] [813] [814] [815] [816] [817] [818] [819] [820] [821] [822] [823] [824] [825] [826] [827] [828] [829] [830] [831] [832] [833] [834] [835] [836] [837] [838] [839] [840] [841] [842] [843] [844] [845] [846] [847] [848] [849] [850] [851] [852] [853] [854] [855] [856] <		

		Backend of the information in CodeProber is up-to-date	CS2.2.14	
		Backend of it in the compiler or CodeProber has to be happy	CS2.2.15	Yes No
		Hard to know in what probe the guest will be	CS2.2.16	
		It won't show what happens internally as problems	CS2.2.17	Yes No Both
		Too much information, hard to find what you are looking for	CS2.2.18	Yes
		Some properties not visible in UI (did not check "show all properties")	CS2.2.19	Yes
		Missing cases	CS2.2.20	Yes No
		Not obvious which elements in the UI are interactive	CS2.2.21	Yes
	Technical issues		CS2.2	Yes No Both
		Freeze/crashes	CS2.2.22	Yes No Both Other
		Problems with WSL	CS2.2.23	Yes
		Can't start with getting it to backup files in files	CS2.2.24	Yes
		IDE's parsing error in terminal	CS2.2.4	Yes
		The concept of probe doesn't transfer to other parts of compiler (e.g. host)	CS2.3	Yes
	Suggested improvements		CS3	Yes
		Need a "back button" to undo incorrect operations in the UI	CS3.1	Yes No
		Being able to see what version of the compiler is used	CS3.2	
		Better support for resizing windows	CS3.3	Yes
		Refresh on file button	CS3.4	Yes
		Not all needed options in some windows, not per probe	CS3.5	Yes
		Not very debugger support	CS3.6	Yes No Both
		Show attribute kind in UI (e.g. /anh /enc)	CS3.7	Yes
		Connect probe to implementation (source file inside the comment)	CS3.8	Yes
		File system integration	CS3.9	Yes
		Be able to copy stack trace of long running stack evaluation	CS3.10	Yes
		Transitory messages in terminal	CS3.11	Yes No
		Handle Ctrl+click, don't show where clicking	CS3.12	Yes
		Just improve documentation	CS3.13	Yes
		Show attribute definition	CS3.14	Yes
	Thought and ideas went		CS4	Yes No Both Other
	Workshop and ideas went		CS5	Yes No Both Other
	Workshop and ideas went		CS6	Yes No Both Other
	Questions about the labs in program analysis course		CS1	Yes No
		File it	CS1.1	Yes No
		How to understand how an IDE works	CS1.2	Yes No Both Other
		Extra file	CS1.1.2	Yes
		Helping each other	CS1.2	Yes No
		Difficult	CS1.3	Yes No Both Other
		Time-consuming	CS1.3.1	Yes No Both Other
		Frustating	CS1.3.2	Yes
		Hard if you haven't taken the compiler course	CS1.3.3	Yes
		Show to get started	CS1.3.4	Yes No Both Other
		Especially lab 3	CS1.3.5	Yes
		Good continuation of the compiler course	CS1.4	Yes
		Many ways to solve the labs	CS1.5	Yes No
		A lot of debugging	CS1.6	Yes
		Not that much advanced IntelliJ features	CS1.7	Yes No
		Good that they contain instructions for CodeProber in the beginning	CS1.8	Yes No
		Not sure what you learn from them	CS1.9	Yes
		Lab 2 and 3 were too similar	CS1.10	Yes
	Questions about the labs in compiler course		CS2	Yes No Both Other
		File it	CS2.1	Yes No Both Other
		More straightforward guidance about what to do	CS2.2	Yes No Both
		Difficult	CS2.3	Yes No Both
		Frustating	CS2.3.1	Yes No Both
		Not easier than analysis course	CS2.3.2	Yes
	Questions about the labs in both courses		CS3	Yes No
		UI isn't done "best ever kind" so much, can be frustrating/difficult	CS3.1	Yes No
		Not much IDE support / developer tools for IntelliJ	CS3.2	Yes