

Debugging Static Analyses in Sturdy

Tomislav Pree

August 17, 2020

Johannes-Gutenberg University Mainz

Programming Languages Research Group
Institute of Computer Science

Bachelor Thesis

Debugging Static Analyses in Sturdy

Tomislav Pree

- | | |
|--------------------|--|
| <i>1. Reviewer</i> | Sebastian Erdweg, Univ.-Prof. Dr.
Institute of Computer Science
Johannes-Gutenberg University Mainz |
| <i>2. Reviewer</i> | André Brinkmann, Univ.-Prof. Dr.
Institute of Computer Science
Johannes-Gutenberg University Mainz |
| <i>Supervisors</i> | Sven Keidel, M.Sc. |

August 17, 2020

Tomislav Pree

Debugging Static Analyses in Sturdy

Bachelor Thesis, August 17, 2020

Reviewers: Sebastian Erdweg, Univ.-Prof. Dr. and André Brinkmann, Univ.-Prof. Dr.

Supervisors: Sven Keidel, M.Sc.

Johannes-Gutenberg University Mainz

Institute of Computer Science

Programming Languages Research Group

Staudingerweg 9

55128 and Mainz

Abstract

Static analyses represent an indispensable part of the IT sector. Either as a tool, that helps developers to detect errors and preserve a good code quality, or as an essential component of compilers. But the development of static analysis is a complex task and prone to errors. Thus, the analysis developers have to deal with a high number of errors while developing. Unfortunately, the debugging support for the development of static analysis is practically non-existing. Hence, this work puts focus on improving the debugging support for static analyses. This goal is accompanied by the research question:

Which information is relevant for the debugging of static analyses and how to present the information to the analysis developer efficiently?

To resolve this question, we worked together with static analysis developers. Our work leads to the results, that the stepwise and intuitive presentation of data, that either emerges from the execution of a static analysis or the analyzed behavior itself, improves the debugging of static analyses. To illustrate our results, we implemented a debugging tool for static analysis based on Sturdy, a framework for creating sound static analysis in Haskell. Our work was evaluated by a qualitative assessment with a Sturdy developer and made clear, that our approach lead to an improvement of the debugging process of static analyses.

Abstract (German)

Statische Analysen sind aus dem heutigen IT Sektor nicht mehr wegzudenken. Sei es als Werkzeug, welches Entwicklern dabei hilft, Fehler zu vermeiden und eine gute Code Qualität beizubehalten, oder als fester Hauptbestandteil von Compilern. Jedoch ist die Implementierung von statischen Analysen eine komplexe Aufgabe, die anfällig für Fehler ist. Dementsprechend müssen sich die Entwickler von statischen Analysen häufig mit Fehlern auseinandersetzen, die während dem Implementieren auftauchen. Unglücklicherweise gibt es nur wenige Werkzeuge, die dafür ausgelegt sind, Fehler in statischen Analysen zu finden und zu beheben. Daher möchten wir den Fokus dieser Arbeit darauf legen, die Fehlersuche und Behebung in statischen Analysen zu verbessern. Das lässt die folgende Forschungsfrage aufkommen: Welche Informationen sind wichtig, für das finden und beheben von Fehlern in statischen Analysen und wie sollen diese Informationen präsentiert werden? Um diese Frage zu beantworten, haben wir mit statische Analyse Entwicklern zusammen gearbeitet. Unsere Arbeit führte zu dem Ergebnis, dass eine schrittweise und intuitive Präsentation, von Informationen, die entweder die Ausführung der statischen Analyse verdeutlichen, oder Daten des Analyseziels an sich, das Finden und Beheben von Fehlern unterstützt. Um unsere Ergebnisse zu verdeutlichen, haben wir eine Debugging-Erweiterung für Sturdy, ein Haskell Framework für das Erstellen von statischen Analysen, implementiert. Um die Ergebnisse dieser Arbeit zu bewerten, haben wir eine qualitative Bewertung mit einem Sturdy Entwickler durchgeführt. Diese hat zu dem Ergebnis geführt, dass unser Ansatz zu einer verbesserten Fehlersuche und Fehlerbehebung bei statischen Analysen führt.

Contents

1	Introduction	1
2	Debug Information	7
3	Implementation	15
3.1	Server-Client Communication	16
3.2	Integration of the Debugger into Sturdy	18
3.3	Implementation of the User Interface	23
3.4	Transmitted Debug Information	25
4	Language and Analysis Independence	29
5	Evaluation	31
6	Related Work	35
7	Conclusion	37
	Bibliography	39

Introduction

Software is steadily growing in size and complexity. Modern software is an integral part of our lives and has a diverse set of applications like predicting extreme weather events, organizing entire companies or even saving lives in medical application. But the development of beneficial and correct software without mistakes is challenging. Many factors have to interact and often big teams of developers work together to implement the software. This leads to an increasing number of possible sources for mistakes, while developing software. Mistakes are frustrating and costly as they claim a large part of a software developers workload. Ideally, bugs are detected in the implementation or testing phase of the software engineering process, before the software is deployed in production. In this case, the mistake only leads to time expenditure. But a bug in production can cause serious financial damage. Therefore applies, the sooner a mistake is detected, the better. An indispensable tool for the early detection of bugs is the static analysis. Static analyses process source code without running it actually. So errors are shown to the developers, before they run their programs. Almost every IDE uses some kind of static analyses to support software developers and improve their work.

Unfortunately, with more complex software and programming languages, the development of static analyses is getting more complex too. Programming languages, code constructs and frameworks are steadily developed further, what leads to more aspects to consider for analysis developers. Another reason for the high complexity of developing static analyses are the two source codes, the analysis developers have to focus on. On the one hand, there is the source code of the analysis itself and on the other hand there is the source code of the analyzed program. Because of these facts, the development of static analyses is prone to errors. So the analysis developers have to deal with a high number of bugs, while debugging tools for static analyses are barely available. Unfortunately, existing debugging approaches are often unsuitable for troubleshooting two corresponding source codes.

Methods like debugging print statements are inappropriate, due to the mostly large output of static analyses. For example, we instrumented a type and control-flow analysis for Scheme with debugging print statements and got 415 lines of debugging trace, while analyzing a program, that contained only 6 lines of code. Such a high amount of debugging information at once makes it hard to pin down the source of

a bug. On the other hand, reducing the amount of printed debugging information, to find the source of bugs more easily, does not help either. The reduced amount of debugging information reduces the probability, that the bug can be seen from the printed debugging information. Another conventional debugging approach is to use an existing debugger, for a particular programming language. In the case of developing static analyses, this approach intends to use a debugger for the programming language, the analysis is written in.

But the problem here is, that the debugging information would be presented on the wrong level of abstraction. So the presented information is not provided to be read by developers, but to be utilized by an executed program. Figure 1.1 on page 3 shows two data structures, which contain relevant information for debugging, on the two levels of abstraction. The abstraction level of the analysis contains only the in-memory representation of the control-flow graph. To have use for the analysis developers, the in-memory representation of the control-flow graph has to be processed to an actual graph. The graphical representation of the graph is more intuitive for the developers and thus brings more benefits for troubleshooting. So it is with the analysis store. The low-level representation of the data structure is recursive, and the analysis developers would need to process the entire output, to understand the store. The representation on a higher abstraction level is already processed and dereferenced, so the developers understand the data structure on the first glance. Altogether the available debugging tools for static analyses are limited and conventional debugging approaches are unsuitable. It is not clear which information is relevant for troubleshooting and how to display the debugging information.

To tackle the described problems, this work presents the implementation of a debugging extension for Sturdy, a Haskell framework, that facilitates the creation of sound static analyses. Our debugger allows the analysis developers to set breakpoints on the analyzed code and execute the analysis stepwise. Every time a breakpoint is reached, debugging information is sent to the developers. This ensures, that the analysis developers get only the debugging information, they need at a particular moment. The user interface contains a graphical representation of the control-flow graph, this helps to comprehend the execution order of the analyzed code. The displayed stack trace contains all function calls of the analyzed code in chronological order. The store consists of a mapping of addresses and values, that are used by the analysis. Our dereferenced representation presents the stack in an intuitive way. The last debug information in the user interface is the environment data structure. The environment maps variables to addresses and contributes to the total understanding of the analysis.

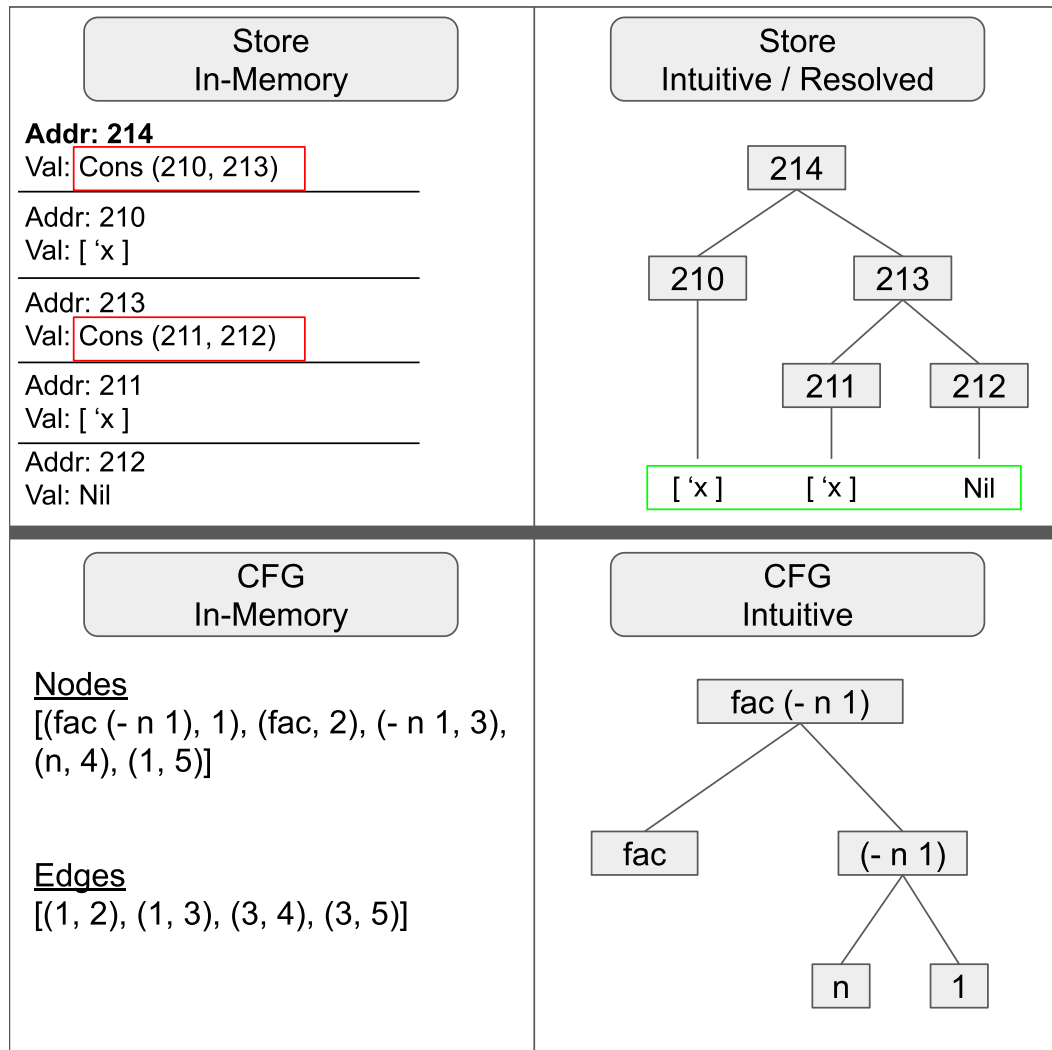


Fig. 1.1: Store element and control-flow graph representations. In-memory representation of the left side, intuitive representation on the right side.

Our implementation provides debugging support for static analyses by abstract interpretation, for the programming language Scheme. The server side of the debugger was directly integrated into the Sturdy framework, because this is a simple way to control the execution of the analysis and gather debug information. Due to the amount and complexity of the debug information a graphical user interface was required, which is represented by a web application. To determine the data structures, the debugger should show, we worked together with Sturdy developers and implemented the data structures, which seemed to provide the most relevant information for troubleshooting. The used data structures and their meaning for debugging static analyses will be explained further in Section 2. Due to the fixpoint-algorithm, Sturdy is based on, the debugging component was implemented as a combinator. Some components of the implementation can even be used language independent, as discussed in in Section 4. The domain specific debugger was presented to a Sturdy developer, to get a suitable feedback on our solution, for the previously described problems. We carried out a qualitative assessment with the analysis developer by letting the developer debug faulty static analyses, with and without the usage of the debugger.

Summarized, we are making the following contributions:

- We present an approach, that simplifies debugging of static analyses in the Sturdy framework
- We demonstrate, which information is required for efficient troubleshooting, while developing static analysis, and how this information has to be presented to the user.
- We implemented the debugger as a fixpoint combinator, that interrupts the execution of the fixpoint algorithm to offer debug information to the user.

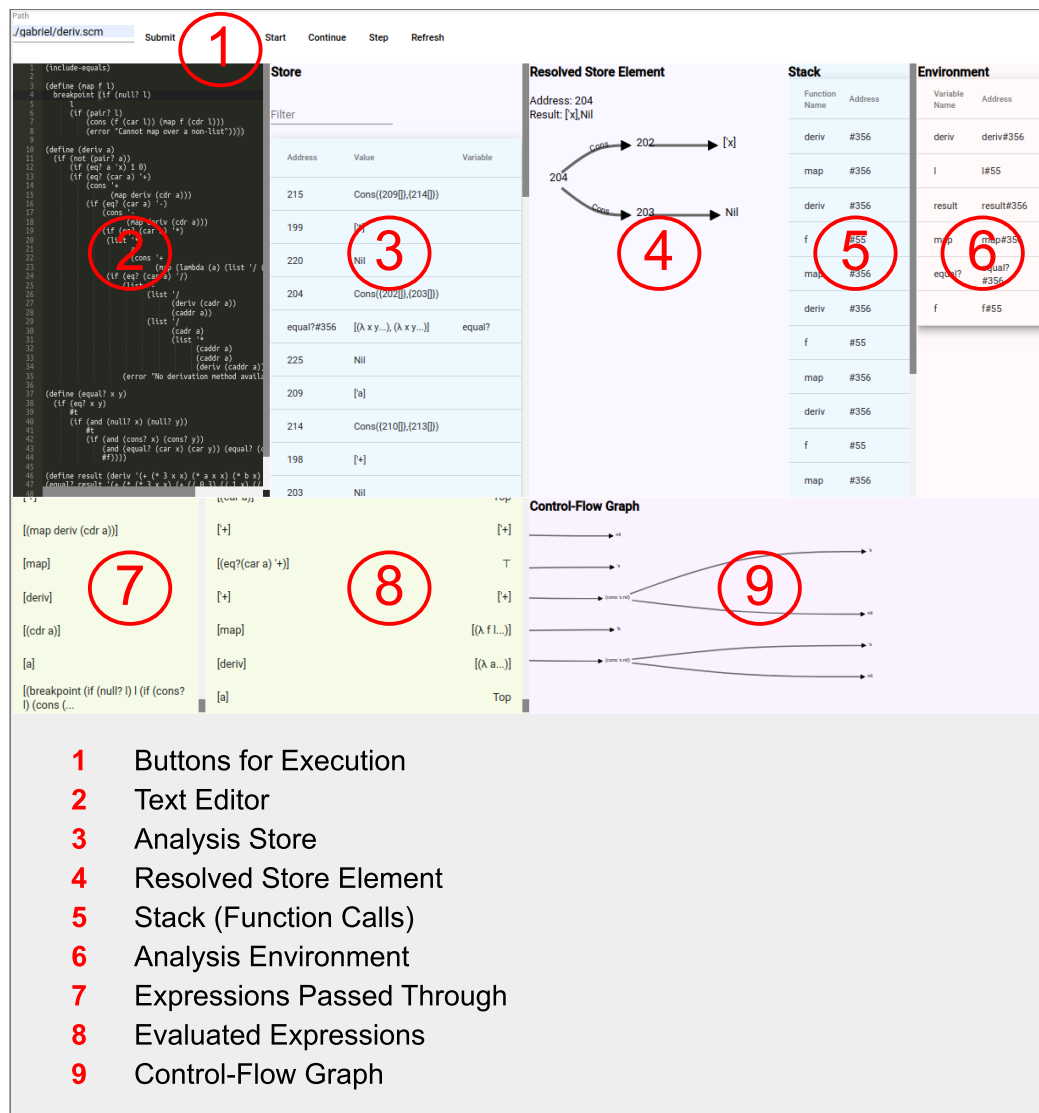


Fig. 1.2: Entire user interface of the sturdy debugging tool (screenshot)

Debug Information

In the introduction we already talked about the debug information, that will be presented to the analysis developers while debugging. This section will take a closer look to the debug information, that will be displayed while debugging static analyses of Scheme programs by abstract interpretation and explain the individual components. Although we implemented our debugging tool as a domain specific debugger for the Sturdy framework, the approaches we discuss in this section are generally applicable for debugging static analyses. In Sturdy, every single component of the debug information is gathered from a particular data structure. These data structures are primarily used to store information while a static analysis is executed. The size of the single data structures grows proportional with the progress of the analysis. The debug information, we implemented in this work, consists of four data structures, which we determined in consultation with sturdy developers. But the provided debug information is not settled and can easily be customized for current needs. Because Sturdy is based on a fixpoint algorithm, that consists of different combinators, debug information can be added and removed with small effort. Each of these combinators contributes a particular functionality to the static analysis. For instance, if the `recordControlFlowGraph` combinator is integrated into the fixpoint algorithm, the control-flow graph gets recorded during the execution of the static analysis. So to provide more debugging information to the client, the corresponding combinator has to be integrated into the fixpoint algorithm and the data structure, which contains the required information, has to be sent to the client, every time a breakpoint is reached. With an adaptation of the user interface, the new information is completely integrated into the debugger and gets displayed properly to the analysis developers.

So one of our main problems, concerning the debug information, was to determine which information should be displayed to the user. While the static analyses get executed, several data structures are involved, what leads to a large amount of possible debug information to display. But not each of the data structures provides useful information for troubleshooting, so we have to filter out the data structures with a use for debugging. Thus, to determine the most important debug information, we worked together with Sturdy developers. Because of their experience with the implementation of static analyses, Sturdy developers can evaluate better, if the information of a data structure is helpful for debugging. While questioning the

Sturdy developers about useful debug information, two different kinds of information have turned out as useful. The first kind of information provides a better insight of the static analysis and its execution. The analysis developers tried to print out this kind of information, while debugging static analyses with conventional tools. Data structures like the environment or the store are examples for this kind of information. The second kind of debug information refers to the particular purpose of a static analysis. So while executing a static control-flow analysis, it is beneficial to see the current state of the control-flow graph itself at every breakpoint. So this kind of information is able to show exactly, which code construct of the analyzed code leads to an error.

So now we have a better idea of the information, which is important for debugging static analyses. But that leads directly to our second problem, namely the suitable presentation of the debug information. The previously discussed data structures contain complex information and can grow big in size. So if the analysis developers just add print statements to the analysis code, the entire debug information gets output at once. So the developers have to process a high amount of debug information at the end of the analysis. Thus, it takes a lot effort to find the right section of the output, that reveals the error. Another challenge is to find a suitable way to process the raw debug information, so the information gets presented in an intuitive form. Because the debug information is gathered from the corresponding data structures, the unprocessed information is only available as an in-memory representation of the data. Moreover some of the data structures are complex, so the in-memory representation of the data is not readable for humans. For example, the in-memory representation of the control-flow graph consists of a list of nodes, where each node has a label and an identifier. The label shows the corresponding expression and the identifier provides uniqueness. Additionally, there is a list of edges, where each edge contains two node identifiers. So to take any advantage out of the in-memory representation of the graph, the developer has to assign each node to the corresponding edges. From a certain size of the graph, this is impossible, without taking notes or even drawing the graph. This takes a lot of time and nerves from the analysis developers. But even if there is a suitable way to process the information of the data structures, some information just gets too big. As already discussed, the data structures, that contain the debug information, grow steadily while the static analysis gets executed. If the analyzed program reaches a certain size, the data structures get too big to just process the data and display the information. To cover these cases, we have to find a way, to deal with big data structures. To put together the problems in an example, we executed a static Scheme analysis by abstract interpretation and enabled the debug tracing. Figure 2.1 on page 9 shows what happens, if the debug information, that is derived from big and complex data structures, gets displayed at once and without proper processing. Even if the analyzed code only contained 6 lines of code, the output was 415 lines long.


```
(define (fac n)
  (if (= n 1)
      1
      (* n (fac (- n 1)))))

(fac 10)
```

```
CALL
EXPR: ((if (= n 1) 1 (* n (fac (- n 1)))))
ENV: [fac -> fac#16[], n -> n#12[]]
STORE: [n#12[] -> Int, fac#16[] -> [(λ n...)]]

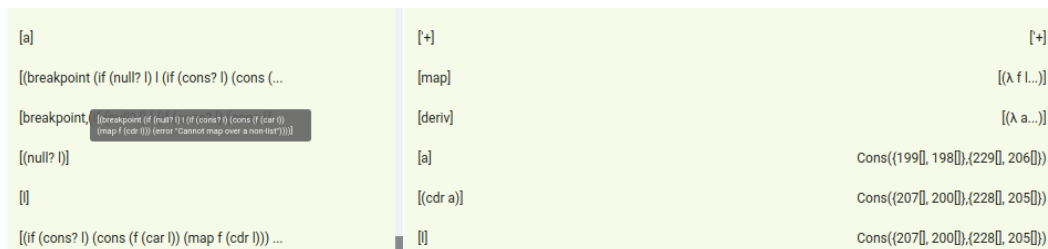
RETURN
EXPR: ((if (= n 1) 1 (* n (fac (- n 1)))))
ENV: [fac -> fac#16[], n -> n#12[]]
STORE: [n#12[] -> Int, fac#16[] -> [(λ n...)]]
RET: NonTerminating
STORE: [n#12[] -> Int, fac#16[] -> [(λ n...)]]
ERRORS:[]

RETURN
EXPR: (fac (- n 1))
ENV: [fac -> fac#16[], n -> n#12[]]
STORE: [n#12[] -> Int, fac#16[] -> [(λ n...)]]
RET: NonTerminating
STORE: [n#12[] -> Int, fac#16[] -> [(λ n...)]]
ERRORS:[]
```

[illegible]

9

To improve the sudden presentation of the debug information, we implemented a possibility, to execute the static analysis stepwise. The debug information gets gathered from the current state of the data structures at each step. Due to the stepwise execution of the analysis, the debug information will not get presented at once, but bit by bit. To determine the steps of the execution, the analysis developers have to set breakpoints in the analyzed code. Because we extended the parser by breakpoints, the debug combinator is able to stop the execution of the analysis, if a breakpoint gets detected. At each breakpoint, the current state of the data structures gets processed and displayed to the analysis developers. Another functionality that provides a bit by bit presentation of the debug information, is the *Step* function. The *Step* function allows the analysis developers to continue the execution of the analysis, until one more expression gets evaluated. So an even finer grade of the bit by bit presentation of the information can be achieved. To check the correctness of the executed analysis, we included a list of the currently processed expressions into the user interface. Next to the list of current expressions, we present the list of already evaluated expressions and their assigned value as shown in Figure 2.2. These lists provide a possibility for the analysis developers, to check each evaluated expression for correctness and thus localize the error more accurate. Because the debug information now gets presented bit by bit, the analysis developers only get confronted with information, they need for troubleshooting. Future states of the data structures will not be displayed, until the analysis gets executed far enough. This leads to a better understanding of the execution of the analysis and a faster localization of bugs. To deal with the size and complexity of the data structures,



[a]	[+]	[+]
[(breakpoint (if (null? l) l (if (cons? l) (cons (...	[map]	[(λ f l...)]
[(breakpoint (if (null? l) l (if (cons? l) (cons (f (car l)) (map f (cdr l))) (error "Cannot map over a non-list")))]	[deriv]	[(λ a...)]
[(null? l)]	[a]	Cons({199[], 198[], {229[], 206[]})
[l]	[(cdr a)]	Cons({207[], 200[], {228[], 205[]})
[(if (cons? l) (cons (f (car l)) (map f (cdr l))) ...	[l]	Cons({207[], 200[], {228[], 205[]})

Fig. 2.2: Processed and evaluated expressions (screenshot from user interface)

more specific solutions are required. Every data structure is individual and needs a special processing, to get useful debug information. The data structures can be too big, too complex or both, to display them without processing. Therefore, we will discuss the four data structures, we implemented, separately. Each data structure will get explained in detail, and our method to deal with size and complexity will be described.

Stack: The analysis stack stores the function calls, while the program gets analyzed. So every stack element represents one function call. Unfortunately, the analysis stack does not store the names of the functions, but the function body, because not

every function has to be assigned to a variable in Scheme. To get the function name, we have to compare the function body to every store element. The store contains mappings of addresses and values, so if there is an address, that maps to the required function body, we can extract the name of the function. We are able to reduce the complexity of the data structure with this method. But the store can grow big in size, so we have to find way to deal with a high amount of stack elements. Figure 2.3 shows how a scrollable table is able to fix the problem. With help of the scroll functionality, the analysis developers will not get overwhelmed with too many store elements. But they are still able to retrace the entire stack.



Stack	
Function Name	Address
deriv	#356
map	#356
deriv	#356
f	#55
map	#356
deriv	#356
f	#55
map	#356
deriv	#356
f	#55
map	#356

Fig. 2.3: List of function calls (analysis stack) (screenshot from user interface)

Store: The analysis store represents a mapping of addresses to their values. Due to the high usage of lists in Scheme, the values of the store elements are mainly lists. Unfortunately, lists in Scheme are represented through pairs. A pair only contains two values, so multiple, interlocked pairs are required, to form a list with more than two elements [5]. This leads to a recursive data structure. The in-memory representation of the store is not processed and accordingly the values are not resolved. So the analysis developers have to observe the entire store, to resolve the value of the address they need. This requires a lot of time. Because the store grows fastly, the task gets even more complicated, with a big analyzed program. To tackle this problem, we decided to represent the store in a table. One row of the table consists of a clickable store element. If the analysis developer clicks on

a store element, the value of the selected element gets resolved and displayed. To retrace the resolution of the recursive value, a graph gets displayed, that shows the procedure of the resolution. Every passed store value is represented as a node and edges show the structure of the list. Beside the high complexity of the store data structure, it can grow big in size. If the store consists of a high number of elements and the analysis developers have to find a specific address, they have to scroll trough the entire store. This problem was solved trough a filter function of the store table, so the analysis developers can enter the required address into a form field, and the table will filter for the required address, as shown in Figure 2.4.

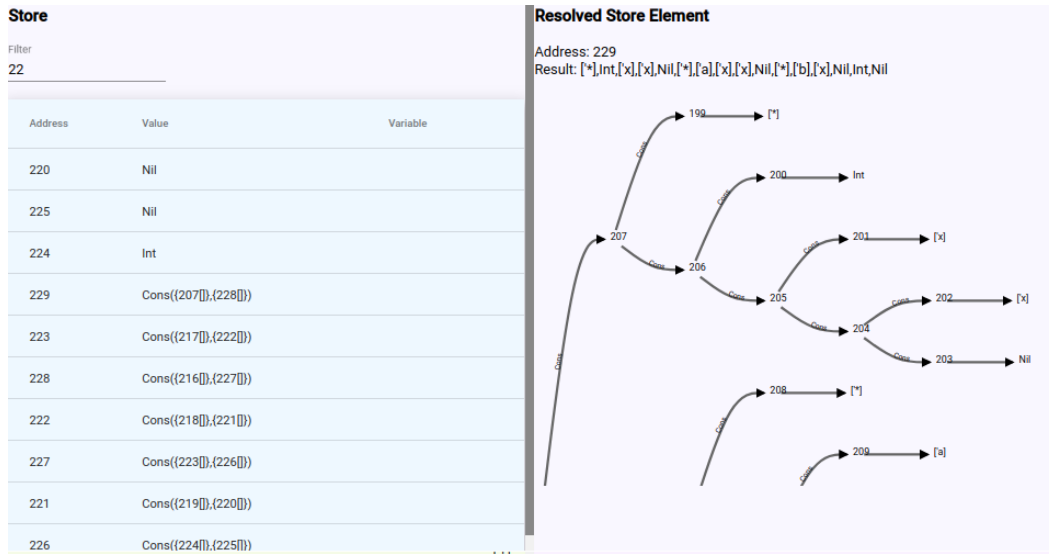


Fig. 2.4: Analysis store as a table with filter function. On the right side is a resolved store element with the value and the graph (screenshot from user interface)

Environment: The environment consists of mappings of variables to addresses. In contrast to the store data structure, the environment is more intuitive, even the in-memory representation of it. Compared to the other data structures, the environment stays small. Even when the static analysis gets executed, the environment does not grow. So to display the environment properly, a scrollable table is required as shown in Figure 2.5 on page 13. The scrollable table will even cover the rare cases of a high number of elements in the environment data structure. The values of the variable name and the address are not recursive and can be displayed the way they are.

Control-Flow Graph: The last data structure, we gathered debug information from in this work, is the control-flow graph. The control-flow graph is a directed graph, that describes the execution order of a program [6]. The in-memory representation of this data structure is unintuitive too. Thus, the raw information has to be processed, before it gets presented to the analysis developers. The raw information consists of a list of nodes and the corresponding list of edges. We carried out the processing of the information with help of a graph library. So the nodes and edges just have to get parsed in the right way, to get displayed intuitively, as seen in Figure 2.6.

Environment	
Variable Name	Address
deriv	deriv#356
l	l#55
result	result#356
map	map#356
equal?	equal?#356
f	f#55

Fig. 2.5: Analysis environment as a list (screenshot from user interface)

Here is the size of the data structure a problem too. With a progressive analysis, the graph grows quickly in size. To still guarantee a proper representation, the graph is displayed in a draggable and zoomable window. Thus, the analysis developers can analyze every component of the graph, no matter how big it is.

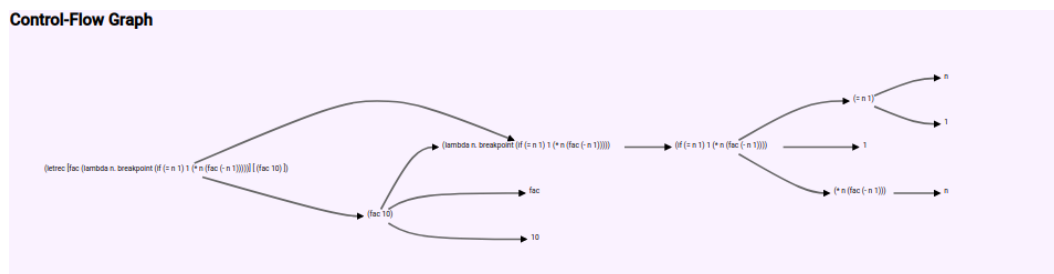


Fig. 2.6: Zoomable and draggable control-flow graph (screenshot from user interface)

Implementation

We implemented a domain specific debugger to provide debugging support for Sturdy. With our concrete implementation we get a better insight of the research question, namely which information is relevant for debugging static analyses and how this information has to be presented. Additionally, we are able to answer the research question more precisely and evaluate our results more accurate. So in particular, we implemented a debugger for static analyses by abstract interpretation of Scheme programs. The debugging server and the implemented debugging process are language and analysis independent. Only the processed and displayed data structures are more tied to the language of the analysed code and the analysis type. But even if the displayed debugging information is language and analysis specific, the principles, that have to be obeyed while gathering the debug information, are similar in general, as described more precisely in in Section 4. So the debugging server was integrated into the Sturdy framework. This seems to be most sensible way, because the execution of the analysis can be controlled and the data structures can be read out. As previously discussed, the data structures are complex and grow big in size, thus the information has to be processed graphically, to provide an intuitive presentation.

So to deal with a large amount of graphical data, we need to implement the client side as a graphical user interface. Unfortunately, there are no existing debugging interfaces, we can use. Most of the existing interfaces are designed for a specific domain, so the presentation of our debug information will not be optimal. Therefore, we implemented our own user interface. A Haskell user interface, directly connected to the Sturdy framework, seems like a good approach, but the existing packages for graphical user interfaces in Haskell are limited. So we decided to implement the user interface as a web application. A web application provides various libraries for all kind of graphical data and the high usage of web applications, makes it easy to maintain and extend the code. This is because there is a high number of developers, which are able to program web applications and a high amount of documentation is available online.

So because we use a web-application, to represent the client side, we have to establish the communication between our web-application and the Sturdy framework. The supply of existing debugging protocols we can use is limited, because the

debugging protocols are mostly designed domain specifically. So in the most cases, the protocols are too complicated for our use case. This is because conventional debuggers are primarily designed to debug an executable program or a script. So the information transmitted with the protocol is closer to the machine-level than we need it to be. So to avoid an unnecessarily complicated protocol, we decided to design our own debugging protocol. The protocol consists of a small number of uncomplicated messages and even provides customizable components, to improve language independence. The following subsections will describe the implementation of the single components in detail. Additionally, we will talk about the implementation of the processing and the presentation of our Scheme and analysis specific data structures.

3.1 Server-Client Communication

Because we implemented our user interface as a web-application, we can not just access the required information directly, but we have to communicate with the server over an interface. To find a suitable way, to implement the communication type and a corresponding interface, we have to take a look at our specific use case. Roughly, we can break down our debug application to an user-interface, that tells the server, it is connected to, what program has to be debugged and when the execution of the debugged analysis has to be stopped. While the analysis gets executed, our server has to send information to the user interface, until a breakpoint is reached. So our use case tells us: The protocol we need for the communication does not require a high amount of message types, but the messages should be able to contain complex objects. Unfortunately, existing protocols for debugging are designed for troubleshooting executable programs or scripts of a lower level. So they mostly contain a high number of different message types, which are able to transport only simple messages. Thus there is no existing protocol matching our use case, we decided to design our own protocol. This decision allows us to choose the way of communication freely, what lead us to a web socket based communication. The benefits of the communication over a web socket will get clear, after a of the itself. A web socket can be considered as a bi-directional tunnel between the server and the client [4]. The tunnel gets created, as soon as the client connects to the server. The messages, which are exchanged in the web socket protocol, are in the JSON format. This allows us to send complex debug information. We can too easily exchange our messages and only have to adapt the corresponding handlers, not considering the web socket.

With help of the web socket protocol, we can easily automate the process of sending messages and processing received messages. This works, because the protocol we

designed contains a tag, that tells either server or client, what kind of message arrived. As soon as the analysis developer decides to take an action, the client sends a message to the server, which gets automatically processed and answered by the corresponding response. Now the server sends a message, which will again get recognized by the tag, but this time on the client side. So the client processed the information and displays them with the corresponding handler. But to ensure, that the message handlers do not produce errors, we have to send messages, which are designed in a certain way. How the messages have to look like, is settled by the debugging protocol we designed, which now will get explained in detail. The definitions of the messages in our Haskell implementation is shown in Figure 3.1.

```
data DebugMessage
= LoadSourceCodeRequest {path :: String}
| LoadSourceCodeResponse {code :: FilePath}
| StartDebuggerRequest {code :: String}
| ContinueRequest {}
| StepRequest {}
| BreakpointResponse
    {stack          :: [[(Text, Text)], [Text]],
     cfgNodes       :: [(Int, Text)],
     cfgEdges       :: [(Int, Int)],
     latestStore    :: [(Text, Text)],
     latestEnv      :: [(Text, Text)]
| RefreshRequest {}
| RefreshResponse {success :: Bool}
| CurrentExpressionResponse {expr :: Text}
| EvaluatedExpressionResponse {expr :: Text, val :: Text}
| ExceptionResponse {exception :: Text}
```

Fig. 3.1: Definition of the web socket messages in a Haskell file.

The analysis developers have to set the breakpoints for debugging on the user interface. This means, the code of the analyzed program has also to be inserted into the user interface. To make sure, the analysis developer does not have to type or copy and paste the analyzed code, we implemented a functionality, that loads the code from the server into the user interface. To make this feature possible, designed the `LoadSourceCode` message pair. The `LoadSourceCodeRequest` is sent from the client to the server and contains the path of the file to load as a string. Once the server receives the request, it reads the file and uses the information, to create the `LoadSourceCodeResponse`. The response contains the source code of the requested file and as soon as the response arrives at the client side, the code gets

displayed in the text editor.

Now that the analysis developers have the code of the analyzed program, they can insert breakpoints on the critical parts of the program. After the analyzed code is provided with breakpoints and the analysis developers press the *Start* button, the client sends a `StartDebuggerRequest`. This request contains the entire code of the program, with the breakpoints included. Afterward, the server starts to execute the analysis of the sent source code. The `StartDebuggerRequest` has no specific antagonist, although several messages get sent, after the debugging process was started. While the analysis gets executed, web socket messages regarding the processed expressions get sent continuously. For every expression, that is evaluated, a `CurrentExpressionResponse` is sent to the client, that only contains the expression. As soon as the evaluation of an expression has finished, an `EvaluatedExpressionResponse` will be sent off from the server. This response contains a pair of the evaluated expression and the corresponding value. After a breakpoint is detected, the server starts to gather and process debug information. This debug information will be packed up into the `BreakpointResponse`. To `BreakpointResponse` is supposed to vary, depending analysis type and language of the analyzed program. Depending on which information is gathered and processed by the server, the message has to look different.

To control the execution of the analysis, there are the `ContinueRequest` and the `StepRequest`. Both messages lead to a continuation of the analysis. While the `ContinueRequest` tells the server, to execute the analysis until another breakpoint is detected, the `StepRequest` only continues the execution for the next expression. Both messages consist only of a tag. The user interface expects the same responses, as if the debugging process was started (`CurrentExpressionResponse`, `EvaluatedExpressionResponse`, `BreakpointResponse`).

To bring the user interface back to it's initial state, we included the `Refresh` message pair into our protocol. The `RefreshRequest` gets sent by the client, after the user presses the *Refresh* button and only contains a tag. The client expects a `RefreshResponse` as answer, which contains a boolean variable, that indicates the success of the *Refresh* process.

If an inadmissible message arrives at the server, the user interface will get an `ExceptionResponse`. This message type only contains the text of the exception and the tag. Figure 3.2 on page 19 shows the sequence diagram of a possible exchange of messages.

3.2 Integration of the Debugger into Sturdy

The server side of our debugging tool was integrated into the Sturdy Framework, thus the corresponding code is written in Haskell. One of the main components

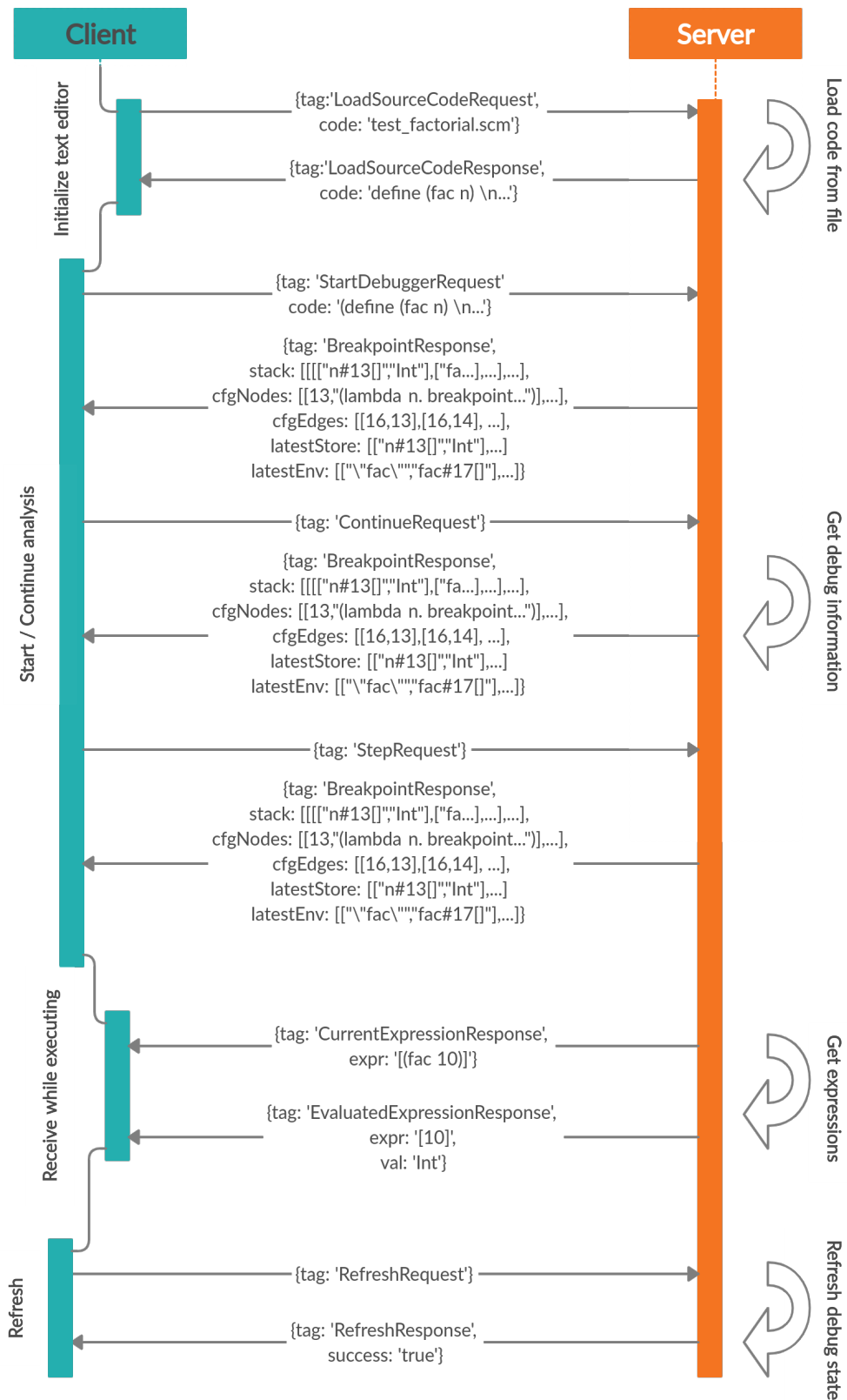


Fig. 3.2: Illustration of possible exchange of messages between client and server

on the server side is the web socket server, that is able to handle all messages, the client side could send. The second component we implemented is the debugging combinator, that can be included into each type of analysis for each programming language, that Sturdy supports. To implement the debug combinator in a proper way, a debug arrow was required, that we will explain more accurate later, after we talked about the implementation of the server.

Because the server is supposed to answer each request, the client side could possibly send, every possible message needs its own handler. The handlers we implemented are pictured in Figure 3.3 on page 21. Our server consists of an outer loop, that is listening to incoming messages, if the execution of the analysis was not started yet. The inner loop is implemented into the debugging combinator and starts listening, if a `ContinueRequest` is required, to resume the execution of the analysis. At first, the handlers of the outer loop will get explained. For each received message, the server checks the tag, the message contains and calls the right handler for the message type. If the server receives a `LoadSourceCodeRequest` from the client, it starts to read out the stated path to get the source code. After this process has finished, the read data get packed up in a `LoadSourceCodeResponse`. This object gets encoded into a `Text` data type and finally sent over the web socket to the client. If the analysis developer wants to start the debugging process, the source code and the corresponding breakpoint are sent over the web socket, in form of a `StartDebuggerRequest`. As soon as the server has recognized the message, the transmitted source code, that contains the breakpoints, gets parsed. The Sturdy framework already supported a similar function, that parses the analyzed Scheme code of a given path. We had to modify the existing function, to insert the analyzed code directly as an argument. The Scheme parser had also to be modified. In particular, we extended it, so it can parse breakpoints. If the parser detects a breakpoint, it will just add it in front of the following expression, so we are able to stop the execution of the analysis at the right point. After the transmitted code was parsed successfully, the execution of the analysis is started with the list of expressions, the parser returned. Because the fixpoint combinator of the started analysis contains the debug combinator, the debug combinator will be executed for every expression. If the combinator detects a breakpoint, the available debug information will be sent to the client and the inner listening loop of the server is called, that will be explained soon. If the server receives a request, for the continuation of the analysis, but the analysis was not started yet, the outer listening loop of the server sends an `ExceptionResponse` to the client. An incoming `RefreshRequest` will let the server restore the used `DebugState` to its default values and send a `RefreshResponse`, that contains an indicator for the success of the restoration. With the previously explained message handlers, the outer loop of our web socket server is able to answer every possible message properly.

```

-- |Listening loop, evaluates received messages
listen :: DebugState -> P.IO ()
listen debugState = do
  msg <- WS.receiveData (conn debugState)
  case msg of

    -- |Respond with source code of required file
    LoadSourceCodeRequest { path = p } -> do
      contents <- Parser.loadSourceCode p
      let responseObject = encodeDebugMessage (LoadSourceCodeResponse (contents))
      sendResponse debugState responseObject

    -- |Start debugging with received code
    StartDebuggerRequest {code = code } -> do
      expressions <- Parser.loadSchemeFileWithCode code
      evalDebug ([expressions]) -- |Start analysis -> inner loop

    -- |Respond with exception
    ContinueRequest {} -> do
      let responseObject = encodeDebugMessage (ExceptionResponse "Deb...")
      sendResponse debugState responseObject

    -- |Respond with exception
    StepRequest {} -> do
      let responseObject = encodeDebugMessage (ExceptionResponse "Deb...")
      sendResponse debugState responseObject

    -- |Refresh debugState
    RefreshRequest -> do
      let responseObject = encodeDebugMessage (RefreshResponse True)
      sendResponse debugState responseObject
      Exception.finally
        (listen debugState)
        (disconnectClient (clientId debugState) (stateRef debugState))

    -- |Loop
  listen debugState

```

Fig. 3.3: Outer listening loop of websocket server (pseudo-code)

To understand the inner listening loop of our server, we have to explain the debug combinator more accurate. As the static analyses in Sturdy are based on fixpoint algorithms, we implemented the debugging functionality as a fixpoint combinator, so it just has to be added to an analysis and the debug information will be gathered (see fig. 3.4, page 22). The single fixpoint combinators, the analyses consist of, are called for every expression, that gets processed by the analysis. Accordingly, the debug combinator is executed for every processed expression too. The debug combinator has access to several arrows [7] and a `DebugState` where information of the debugging process and the web socket connection are saved. As soon as the debugging combinator is called, the current expression gets wrapped into a `CurrentExpressionResponse` and is sent to the client. If an expression was

evaluated successfully, the evaluated expression is sent to the client too, in form of an `EvaluatedExpressionResponse`.



Fig. 3.4: Debug combinator and illustration of the relation between combinator and fixpoint algorithm

Because the evaluation of an expression is only completed, if the evaluation of all subexpressions was completed. Thus, the current expression and the evaluated expression are asynchronous, so we need two different messages for them. Further in the execution, the debug combinator checks, if the execution of the analysis has

to be stopped. The execution has to be stopped either if the current expression is a breakpoint, or if a *Step* was initialized by the analysis developer. To check if the analysis developer executed a *Step*, the debug combinator checks if the boolean `step` variable is set to `True`. If so, the same procedure is executed, as if a breakpoint was detected. If neither a breakpoint was detected, nor the client side initialized a *Step* command, the execution of the analysis gets continued. But if a breakpoint was detected or a *Step* command requested, the debug combinator calls the inner listening loop of our web socket server. At the beginning of the inner loop, the boolean `step` variable in the `DebugState` is set on false, for the case a *Step* was requested. After this, the language and analysis dependent debug information is gathered and processed. The processed debug information is used to create the `BreakpointResponse` object. This object is sent to the client side, with help of the `sendMessage` function, the `DebugArrow` provides. After the debug information was sent, the debug combinator starts listening for either a `ContinueRequest` or a `StepRequest`. Every other received message gets ignored and the server continues listening. As soon as either a `ContinueRequest` or a `StepRequest` is received, the execution of the analysis continues. In the case of a received `ContinueRequest`, the execution gets executed further, until a breakpoint is detected. In the case of a `StepRequest`, the `step` variable in the `DebugState` is set to `True`, so the analysis will be stopped again, as soon as the next expression gets processed.

3.3 Implementation of the User Interface

We implemented the user interface as a web-application. The timeframe of this work required a fast prototype. And through the different analysis types and programming languages that Sturdy supports, the web-application has to be able to show the right components depending on analysis type and language. So we decided to implement the user interface with Angular, a Typescript framework. With a high number of available libraries, the fast prototype was guaranteed. The modularization, Angular provides, supports the further development of the user interface. This leaves the possibility open, to implement components, that display debug information, for every kind of language and analysis type. Surely, the existing components can be reused for similar data structures. This subsection will focus on the general debugging procedure, while Section 3.4 will explain the language and analysis dependent debug information, we implemented on the user interface.

The screenshot in Figure 3.5 shows the components of the user interface, that are responsible for the execution of the analysis or provide a better understanding of the execution. The header contains different buttons to start and control the execution

of the analysis. Below you can see the text-editor, that contains the analyzed code and the breakpoints, the analysis developers have to debug. Under the text editor, there are two lists, that show expression. The left list shows the current expression, while the right list shows the already evaluated expressions and their corresponding value. This helps the analysis developers to comprehend the execution of the analysis better and thus helps localizing the errors. Every button sends a particular request message to the server. The corresponding response is processed by a handler function. Depending on the tag, the received message contains, the handler gets chosen and executed. The handlers process the information and display it to the analysis developers.

```

1 (include-equals)
2
3 (define (map f l)
4   breakpoint (if (null? l)
5     l
6     (begin breakpoint (if (pair? l)
7       (cons (f (car l)) (map f (cdr l)))
8       (error "Cannot map over a non-list")))))
9
10 (define (deriv a)
11   (if (not (pair? a))
12     (if (eq? a 'x) 1 0)
13     (if (eq? (car a) '+)
14       (cons '+
15         (map deriv (cdr a)))
16       (if (eq? (car a) '-')
17         (cons '-
18           (map deriv (cdr a)))
19         (if (eq? (car a) '*')
20           (list '+
21             a
22             (cons '+
23               (map (lambda (a) (list '/ (deriv a) a)) (cdr a)))
24             (if (eq? (car a) '/')
25               (list '-
26                 (list '/
27                   (deriv (caddr a))
28                   (caddr a))
29                 (list '/
30                   (cadr a)
31                   (list '*
32                     (caddr a)
33                     (caddr a)
34                     (deriv (caddr a))))))
35             (error "No derivation method available")))))
36
37 (define (equal? x y)
38   (if (eq? x y)
39     #t
40     (if (and (null? x) (null? y))
41       #t
42       (if (and (cons? x) (cons? y))
43         (and (equal? (car x) (car y)) (equal? (cdr x) (cdr y)))
44         #f))))
45
46 (define result (deriv '(+ (* 3 x x) (* a x x) (* b x) 5)))
47 (equal? result '(+ (* 3 x x) (+ (/ 0 3) (/ 1 x) (/ 1 x)))
48   (* (* a x x) (+ (/ 0 a) (/ 1 x) (/ 1 x)))
49   (* (* b x) (+ (/ 0 b) (/ 1 x))
50   0))
51

```

Fig. 3.5: Control elements for analysis execution (screenshot). A text editor for the analyzed code and buttons to control the execution on top of the text editor

The explanation of a typical debugging process will make the single components clear. To start a debugging process, the analysis developers have to insert the analyzed code into the text editor. Here we used the Ace text editor, that provides support for several languages. The analyzed code can either be directly written or copied into the editor.

Another possibility to insert the required code into the editor, is to use the *Load Source Code* functionality. Here the analysis developers only have to insert the required filename into the associated form and press the *Submit* button. This will send a `LoadSourceCodeRequest` to the server. The `LoadSourceCodeResponse` will contain the requested source code. The `loadSourceCodeResponseHandler` will insert the code into the editor and thus present it to the analysis developers. These now only have to insert the breakpoints into the code. The breakpoints are inserted by typing the string "breakpoint" in front of the expression, the analysis developers wants the analysis to stop the execution. To start the execution of the analysis, the analysis developers have to press the *Start* button. This action will lead to the sending of a `StartDebuggerRequest`, that contains the source code and the breakpoints, which are in the text editor. The *Continue* and the *Step* button, next to the *Start* button, continue the execution of the analysis after it has been stopped. The analysis will either be executed until the next breakpoint is detected (`ContinueRequest`) or until the next expression is processed (`StepRequest`). While the analysis gets executed, the client constantly receives `CurrentExpressionResponse` for every processed expression and `EvaluatedExpressionResponse` for each evaluated expression. The processed expressions will just be presented in a list. The list will always be scrolled to the bottom, so the analysis developers see the latest expression, that were processed. If the expression is longer than a particular number of chars, it will get abbreviated. By hovering over the expression, the analysis developer is able to see the entire expression. The list of evaluated expressions provides the hover and automated scroll functionalities too, but it additionally contains the resulting value of the expression. With these lists, the analysis developer is able to check every evaluated expression for correctness. If the execution of the analysis is on hold, either because a breakpoint was detected or the user requested a *Step*, the client receives a `BreakpointResponse`. Because the structure of this message is language and analysis dependent, so are the corresponding components, that process and display the information. Because the components are inserted into the user interface as a grid element, they can easily be exchanged.

3.4 Transmitted Debug Information

While the previous subsections explained how the more general components of the debugging tool were implemented, this subsection will focus on the specific components, we implemented for our prototype. In particular, we will discuss the components, that are responsible for displaying the debug information to the analysis developers. Our implementation is intended to debug static analysis by abstract interpretation of Scheme programs. Thus, we implemented the processing and the display of the store, the stack, the environment and the control-flow graph.

The store maps addresses to values, used by the abstract interpreter. But the values are not retrievable directly, but only in a recursive presentation. So the values have to be resolved, before they provide a benefit to the analysis developers. Additionally, the store can get really big and contain a high number of entries, the analysis developers have to deal with. So the processing of the store data structure begins on the server side. If the store element contains a function body, the string that shows the function body gets abbreviated. The current store is retrieved, from the latest stack element. The addresses and values get converted to a tuple of string, and finally the entire store gets converted into a list of store elements. This conversion simplifies the sending of the debug information over the web socket. As soon as the store is received by the user interface, the single store elements get displayed in a table. This table provides a filter function, to fastly find required addresses, even if the table contains a high number of elements. The single table rows are clickable. If the store element contains a recursive value, the user just has to click on the element in the table, so the value will be resolved. The resolved value and the corresponding graph, that visualizes the resolution of the value, will get displayed to the analysis developers. The resolution of the store element helps on the client side, by two different functions.

The stack contains the function calls of the analyzed program, and it's processing begins on the server side as well. To process the stack properly, the expression and the entire store are required for every stack element. The functions, that were called, are not saved by their name, but by their function body. This is because not every function is assigned to a particular name. To display the stack intuitively, we have to gain the function name with help of its body. This can be done, by filtering the store for a value, which is identical to the required function body. As mentioned previously, the store values are abbreviated, if they are function bodies. So first we have to get the raw store values for every stack element. Afterwards, we compare the function bodies of the expression with the values of the store elements, until we find a match. This has to be done for every stack element. So the processing on the server side consists of the gathering of the unprocessed store elements and the expression for every stack element. This information is sent to the client side, where the corresponding names for each function body are found. These names get displayed in a list.

The environment of the analysis maps variables to addresses. Thus, not much effort is required to process the environment. The only processing happens on the client side, where the single values of the environment are put into a table. Now the analysis developers already have an intuitive representation of the data structure.

To provide an intuitive representation of the control-flow graph, more steps are required. After the graph is gathered on the server side, it is only available in

an in-memory representation. The list of nodes and the list of edges get packed into a web socket message and are sent to the client. As soon as the information arrives at the client side, the corresponding handler passes the lists to a graph library (`ngx-graph`), that displays a draggable and zoomable graph. Thus, a high number of nodes in the graph will not be a problem, because the user can just navigate to the required section of the graph.

Language and Analysis Independence

Static analyses are used in several areas of the IT sector. They bring different benefits, depending on what kind of analysis is executed and what kind of target the analysis has. These different benefits are required by the developers of almost every existing programming language, so they can develop software faster and with fewer errors. So because different kinds of static analyses are used to analyze programs in several programming languages, a language and analysis independent debugging tool would provide obvious benefits. With a language and analysis independent debugging tool, the analysis developers only have to implement small additions to the existing debugging tool, to get debugging support for their preferred language and analysis. The adaptations to the debugging tool only need to be implemented once and shared with the coding community, to provide a debugging tool for a specific programming language or analysis.

Although our implementation only provides debugging support for the analysis of Scheme programs by abstract interpretation, we kept the language and analysis independence in mind while developing. This leads to design decisions, that prepare the debugging tool for language and analysis independence. Even if we did not implement a language and analysis independent debugging tool, we accomplished some first steps in the direction of a language and analysis independent debugger for static analyses.

More precisely, one of the main contributions we made for language and analysis independence, was to implement the main part of the debugger as a fixpoint combinator. In Sturdy, a fixpoint combinator can be inserted into any fixpoint algorithm. As a result of that every static analysis in Sturdy is based on a fixpoint algorithm, the debug combinator could be included into any kind of analysis, for any programming language, Sturdy supports. But before doing this, the language and analysis dependent functions like creating a `BreakpointResponse` and detecting a breakpoint have to be passed to the combinator as parameters. If any language and analysis dependent aspect of the existing debug combinator would be removed and passed to the combinator as a parameter, the debug combinator could be considered language and analysis independent.

Another language and analysis independent aspect of our implementation is the entire communication through the web socket server. The transmitted messages can

be used for every language and analysis. Especially the `BreakpointResponse`, that actually contains language and analysis dependent data, can be customized with a parametrized `BreakpointResponse` object. So if the `BreakpointResponse` data type looks different, depending on language and analysis, the corresponding `BreakpointResponse` will adapt properly. Of course, every kind of `BreakpointResponse` requires an own handler on the client side to display the information in the intended way. So the Server-Client Communication also provides a foundation for a language and analysis independent debugging tool.

The last design decision, that assists the language and analysis independence, is the structure of the user interface. The user interface is based on a grid list and every displayed debug information is one item of this list. So if components, that show debug information, have to be added, removed or exchanged, just a small adaptation is required. The modularization support, that Angular provides, helps too with the handling of several components, that display debug information.

So to extend the debugging tool, to be able to debug either static analyses for another programming language or a different type of static analyses, the following work steps are required: First, the debug combinator has to be modified, to a language and analysis independent combinator. This is done by passing the language and analysis dependent functionalities as arguments into the debug combinator. A `BreakpointResponse` object, that contains the language and analysis dependent debug information has to be specified. For the new `BreakpointResponse`, there has to be a fitting handler and corresponding components on the client side, that display the debug information.

In conclusion we can say, that the debugging tool is not language and analysis dependent yet. But due to design decisions, that were language and analyse independence oriented, this work contributes some first steps for a language and analysis independent debugging tool.

Evaluation

To evaluate this work, we carried out a qualitative assessment with a Sturdy developer. Since the relevant debug information and the way to present it are obeyed in our implementation, an assessment of our debug extension is a valid way to evaluate our work. First, the Sturdy developer will debug a static analysis without the support of our debugging tool. After this, the developer will be allowed, to use our debugging tool for troubleshooting. Both debugging processes will be observed and compared with each other, to evaluate our implementation. To simulate the debug process, we inserted an error into the analysis code. To ensure the same conditions on both processes, the error we inserted was shown to the developer.

At first, the Sturdy developer simulated the debugging process without our debugging tool. First the existing unit tests were executed. The unit tests perform static analyses and compare the actual result with the expected result. Trough these tests, the analysis developers are able to see, which code constructs of the analyzed code lead to a faulty analysis. Either the incorrect results of the analysis are discovered, after executing the tests, or they are discovered while testing own analyzed code. So after the analyzed code, that leads to an incorrect analysis was discovered, the analysis developer hopes, that the amount of code is small. A small amount of code is more easy to analyze, because big files lead to a high amount of debug trace output. Now the analysis developer tries to localize the error exactly. To achieve that, the developer steadily removes code or changes it in a way, that reduces the debug trace. The analysis developer tries meticulously to remove any unnecessary code, because any small reduction of code leads to a more clear debug trace. After the reduction of the code, the developer has to check, if the analyzed code still produces the error. As soon as the analyzed code is reduced completely, the analysis developer executes the analysis with an enabled debug trace. The debug trace contains each expression and the corresponding return value. Additionally, the current state of important data structures gets output with any expression. Now the developer notices for every processed expression, if the returned value corresponds to the expected return value. If the developer finds an expression, that returns a faulty value, the debug trace gets analyzed in detail. Data structures like the store are output in a unintuitive way. Other data structures, that could contain important information of the error, are not even presented to the user. Because some data structures, like the control-flow graph get really big at the end of an analysis, only a stepwise displaying makes sense. This

leads to a lot of time spent, to understand the data structures. If the more accurate analysis of the trace leads the developer to the source of error, the developer just fixes the error. In the other case, that the developer does not find the source of error, the trace gets analyzed again, or the analyzed code is reduced further, until the source of error is found.

To use the debug tool, a n analyzed code, that leads to a faulty analysis has to be found first. Similar to the debugging process without the usage of the tool, the existing test cases are primarily used for that. After the code was present to the developer, he started to localize the source of error. The developer can use a more readable list of evaluated expression to find approximately the source of error. After this, a breakpoint is inserted before the approximate source of error and the analysis is executed with the tool. After the breakpoint was reached, the developer uses the *Step* command, to find the faulty expression. If the analyzed code, that lead to a faulty analysis, contains obvious opportunities, to reduce the code and still preserve the error, the code is reduced. A reduction of the analyzed code still needs to less output, but the developer does not try anymore, to reduce the analyzed code meticulously. After the developer found the faulty expression, he started analyzing the debug information, that was directly presented in an intuitive way. If the source of error was still not found, the analysis developer would repeat this process, with other faulty analyzed expressions, until he is able to find the exact source of error.

To compare both debugging processes, we have to compare the two processes of error localization first. After this we will take a look at the two different processes, of analyzing the localized error in detail. So in both processes, the analysis developer tried to reduce the analyzed code. This is an efficient method, to take out the complexity of the analysis with small effort. Without the usage of the debugging tool, every little reduction of the code was important, to reduce the output of the debug trace. With the debugging tool, advantage was only taken of obvious opportunities, to reduce the code. To find the faulty analyzed expression without the debugging tool, the analysis developer had to work trough the debug trace of the entire analysis and compare each expression with its return value. With the debugging tool, the developer was able to navigate to the approximately to the faulty analyzed code and then observe the evaluated expressions stepwise. So the detection of the code construct in the analyzed code, that leads to a faulty analysis, is more convenient and faster with our debugging tool. After the faulty section of the analyzed code was found, the developer started to analyze the data structures. In the debug trace, the data structures were output in an in-memory representation. Thus, they were unintuitive and the developer required a lot of time, to understand them. In the debugging tool, the data structures are already processed and thus in an intuitive form. So the developer can understand the data structures faster and thus understand the source of error more efficient.

This lets us conclude, that the stepwise execution of the analysis helps the developers, to understand the analysis better and find the code construct, that leads to a faulty analysis, faster. This is because the developers do not get overwhelmed with the debug information of the analysis at once, but can see the available information step by step. The analysis of the data structures gets also more efficient with the usage of the debugging tool. The debug information is directly presented in an intuitive way, so the developers are able to understand the data structures faster.

Related Work

Since this work deals with debugging static analyses, more precisely finding out which information is relevant for troubleshooting static analysis and how this information has to be presented, this section will take a closer look at the state of the art in debugging static analyses. Even if static analyses are an important tool for developers, that helps to find errors while implementing common programs, the market for tools that support the development of static analyses is not saturated. In particular the supply for tools, that help static analysis developers troubleshooting static analyses, is low.

So to understand errors better in static analyses, the developers mostly use debugging approaches, that were usually intended to support troubleshooting in common programs. Usually, static analysis developers use debugging approaches like an existing debugger for the programming language, the analysis is written in. Print statements or self-made test cases, that compare an expected analysis result to the actual result of a static analysis, are also common approaches, the analysis developers use. There is a high amount of debugging approaches, that work for troubleshooting in common programs [1]. But these approaches are not well suited for the debugging of static analyses. Print statements mostly produce a too large output, that complicates the localization of bugs, or the output is too small, to provide enough information about the contained bugs. The usage of a regular debugger, for the programming language of the analysis code, leads to the presentation of unprocessed debugging information. And writing test cases requires a lot of work and it is impossible to cover every possible error.

Other methods, that static analysis developers often use for troubleshooting, are debugging approaches, that are better adapted to static analysis. Such a debugging approach is given by Delta Debugging [2]. Delta Debugging means, that the static analysis developers try to steadily reproduce the errors of the faulty analyses, while reduce the code of the analyzed program. The analyzed program gets smaller, while the error is preserved. Thus the possibilities for the source of the error got decreased, and the execution of the analysis produces a smaller debug trace, that is more convenient to be analyzed. This approach was also used by the questioned Sturdy developers. Even if existing debugging approaches can improve the debugging of static analyses, they are not able to provide the functionalities, that a debugging tool, specially designed for troubleshooting static analyses, is able to provide.

The only debugging tool, known to us, that is specially designed for static analysis, is VisuFlow [3]. The research team, that created VisuFlow, wanted to implement a tool for debugging static analysis. To determine the structure of the debugging tool, the team conducted a survey. 115 participants, that work with static analyses, answered questions about their work. The survey led to the decision, to create an Eclipse-based debugging environment, for data-flow analyses written on top of Soot. Soot is a framework, that provides the development of static analyses, for Java programs [8]. The debugging tool was evaluated by a user study with 20 participants. The study has shown, that the analysis developers found errors faster, if they used VisuFlow. So the developers, that use VisuFlow are able to more easily understand bugs of the static analyses and thus fix those bugs faster. To classify our work, we should compare it to VisuFlow, since this work also created a domain specific debugging tool, for static analyses. But the research team, that created VisuFlow, had a different approach. While it was clear from the beginning, that our work will be based on the Sturdy framework, the VisuFlow team conducted a survey, to decide what their debugging tool should look like. Through this survey they decided, what kind of static analyses their tool has to support and which programming language will be analyzed. So VisuFlow is focused on a particular kind of static analysis for a particular programming language. Although our implementation only supports static analysis by abstract interpretation for Scheme programs, we designed our tool in a way, that could enable debugging support for several static analyses and several programming languages in the future. This was possible, because Sturdy supports several static analyses for different programming languages, in contrast to Soot, that only enables analyses for Java.

Since the project, that was responsible for VisuFlow, had a larger scope than our work, they were able to evaluate their results more precisely. Their user study showed, that the visualization of debugging data is important for the usability of the debugger. So we tried to focus on a good visualization and an intuitive presentation of the debugging information too. While VisuFlow allows the analysis developers, to set breakpoints on the analysis code and the analyzed code, the Sturdy debugging extension only provides the breakpoint functionality for the analyzed code. Thus, the execution of the analysis can be controlled more accurately with VisuFlow.

This lets us conclude, that VisuFlow as well as our implementation are both domain specific debugging tools, for a particular framework. But since Sturdy is more general in terms of analysis type and analyzed programming language than Soot, our implementation leaves space for more generalization.

Conclusion

Because of the lack of debugging support for static analyses, this work aims at simplifying troubleshooting of static analyses. To facilitate a better debugging experience, we had to answer the following research question: Which information is relevant for the debugging of static analyses and how to present the information to the analysis developer efficiently? In cooperation with static analysis developers we elaborated the research question. To present a first implementation of our ideas, we implemented a debug extension for the Sturdy framework. Our implementation provides debugging support for static analyses by abstract interpretation for Scheme programs. The prototype also helps us to evaluate our ideas with a qualitative assessment.

Our work has shown, that relevant information for debugging static analyses can either be related to the execution of the static analysis, or contain information about the goal of the analysis itself. The presentation of the debug information should happen stepwise, so the analysis developer does not get overwhelmed by too much information at once. In addition, the debug information should be presented in an intuitive form. Since the size of the present debug information depends on the size of the analyzed code, the presentation should be also clear for a high amount of information.

Our classification for important debug information is probably applicable for static analyses in general. So it is with the presentation of the debug information. Principles for presenting debug information, that emerge from this work, are also applicable to static analyses in general. Since our prototype is implemented in the Sturdy framework, the debugging support we provide is limited by the analyses types and programming languages, that Sturdy supports. Still, the debugging extension itself can be easily expanded, to support more types of static analyses or programming languages, that can be analyzed.

So the future work, that emerges from our work, is mainly to refactor the debugging combinator to a language and analysis independent combinator. Additionally more components should be implemented, so several kinds of debug information can be presented properly to the developers.

The outlooks from this work are a faster development of static analyses in Sturdy, because the analysis developers will now understand their errors faster and thus will be able to fix them more quickly. Additionally, this work created a foundation for the implementation of debugging tools in the Sturdy framework. Future imple-

mentations of the debug support for other languages and analyses can rely on this implementation and reuse its main components.

Bibliography

- [1]P Adragna. „Software debugging techniques“. In: (2008) (cit. on p. 35).
- [2]Esben Andreasen, Anders Møller, and Benjamin Nielsen. „Systematic approaches for increasing soundness and precision of static analyzers“. In: June 2017, pp. 31–36 (cit. on p. 35).
- [3]L. N. Q. Do, S. Krüger, P. Hill, K. Ali, and E. Bodden. „Debugging Static Analysis“. In: *IEEE Transactions on Software Engineering* 46.7 (2020), pp. 697–709 (cit. on p. 36).
- [4]Ian Fette and Alexey Melnikov. „The WebSocket Protocol“. In: *RFC* 6455 (2011), pp. 1–71 (cit. on p. 16).
- [5]Robert Bruce Findler, John Clements, Cormac Flanagan, et al. „DrScheme: A Programming Environment for Scheme“. In: *J. Funct. Program.* 12.2 (Mar. 2002), pp. 159–182 (cit. on p. 11).
- [6]Robert Gold. „Control flow graphs and code coverage“. In: *Applied Mathematics and Computer Science* 20 (Dec. 2010), pp. 739–749 (cit. on p. 12).
- [7]Sven Keidel and Sebastian Erdweg. „Sound and Reusable Components for Abstract Interpretation“. In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019) (cit. on p. 21).
- [8]Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. „The Soot framework for Java program analysis: a retrospective“. In: Oct. 2011 (cit. on p. 36).

List of Figures

1.1	Store element and control-flow graph representations. In-memory representation of the left side, intuitive representation on the right side.	3
1.2	Entire user interface of the sturdy debugging tool (screenshot)	5
2.1	Debug trace of an executed static analysis of a Scheme program	9
2.2	Processed and evaluated expressions (screenshot from user interface) .	10
2.3	List of function calls (analysis stack) (screenshot from user interface) .	11
2.4	Analysis store as a table with filter function. On the right side is a resolved store element with the value and the graph (screenshot from user interface)	12
2.5	Analysis environment as a list (screenshot from user interface)	13
2.6	Zoomable and draggable control-flow graph (screenshot from user interface)	13
3.1	Definition of the web socket messages in a Haskell file.	17
3.2	Illustration of possible exchange of messages between client and server	19
3.3	Outer listening loop of websocket server (pseudo-code	21
3.4	Debug combinator and illustration of the relation between combinator and fixpoint algorithm	22
3.5	Control elements for analysis execution (screenshot). A text editor for the analyzed code and buttons to control the execution on top of the text editor	24

Colophon

This thesis was typeset with \LaTeX 2_ε. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at <http://cleanthesis.der-ric.de/>.

Declaration

I hereby declare that I have written the present thesis independently and without use of other than the indicated means. I also declare that to the best of my knowledge all passages taken from published and unpublished sources have been referenced. The paper has not been submitted for evaluation to any other examining authority nor has it been published in any form whatsoever.

Mainz, August 17, 2020

Tomislav Pree

