Abstract Slicing To Improve The Speed Of Static Program Analysis

Sarah Verbelen[1], Bram Vandenbogaerde[1], Jens Van der Plas[1], Noah Van Es[1] and Coen De Roover[1]

[1]*Vrije Universiteit Brussel, Brussels, Belgium*

Abstract

When performing a static analysis, there is a trade-off between the time it takes to run the analysis and the quality of the results of the analysis. We propose to run the analysis on an abstract slice to reduce the size of the program to be analysed without impacting the results of the analysis. We adapt an abstract slicing algorithm to produce executable abstract slices that can be analysed using abstract interpretation. Next, we implement an intraprocedural abstract slicer for Scheme programs. We evaluate the implementation using a dataset of 1050 randomly generated Scheme programs. We find that abstract slices are in general smaller than their corresponding concrete slices and that in turn they are analysed faster. Additionally, we find that there is a significant difference between the abstract slices for two different abstract domains.

Keywords

static analysis, abstract interpretation, abstract slicing, executable slicing

1. Introduction

Static program analysis is able to verify properties of the program without needing to run the program, making it useful to detect issues at compile-time. An often used technique for static analysis is abstract interpretation [1], which abstracts the semantics of a program. Part of this abstraction is the use of an abstract domain that abstracts the concrete values into the properties that are relevant for the analysis. Because abstract interpretation is very close to the semantics of the program, this technique lends itself well to mathematical proofs about the semantics and properties of interest [2]. Unfortunately, one of the main disadvantages of program analysis is that there is a trade-off between the speed of the analysis and the quality of its results. Developers do not want to wait very long for results of the analysis when they use the analysis as part of their workflow, but they also do not want to have an analysis that returns many wrong results as it impacts the usefulness of the analysis [3].

A program slice is a reduced version of a program that contains only expressions that influence a set of variables at a certain program point. The combination of these variables and the program point is called the slicing criterion. Program slicing has several use cases. It was originally introduced by Weiser [4] as a way for developers to intuitively inspect their code during debugging. Other uses include program comprehension or the automatic generation of tests. Program slicing can also be used to improve the efficiency of a static analysis [5]. For most of these use cases, it is beneficial that the computed slice is *minimal*, meaning that it contains *only* the expressions that influence the slicing criterion.

Abstract program slicing is a variant of program slicing where only expressions that influence a specific *property* of the variables in the slicing criterion remain in the slice [6]. These properties are expressed using the abstract domains used by an abstract interpreter, so that an abstract slice removes all expressions that have no influence on the abstract values of the variables in the slicing criterion. For example, when debugging, the developer might only be interested in the statements that modify the nullity of a variable because the variable was null at a point where it should not be null.

BENEVOL 2024

sarah.verbelen@vub.be (S. Verbelen); bram.vandenbogaerde@vub.be (B. Vandenbogaerde); jens.van.der.plas@vub.be (J. Van der Plas); noah.van.es@vub.be (N. Van Es); coen.de.roover@vub.be (C. De Roover)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

Abstract program slicing in general produces smaller slices than concrete program slices, as it is possible that an expression that influences the value of the variables in the slicing criterion does not influence the relevant properties. Because of this, abstract program slicing is potentially a more effective way to improve the efficiency of a static analysis compared to a concrete program slice. As an abstract slicer uses the abstract domains of an abstract interpreter, the resulting abstract slices are specific to this abstract interpretation.

In this paper, we make the following contributions:

- We adapt and implement the abstract slicing framework presented by Mastroeni and Zanardini [6] to create an intraprocedural abstract slicer for Scheme programs
- We adapt the abstract slicing algorithm to produce executable abstract slices, with the goal of being able to run an abstract interpreter on the executable abstract slices to obtain a speed up of the static analysis when compared to running the abstract interpreter on the original program or a concrete slice

The rest of the paper is organised as follows. We first briefly introduce abstract interpretation and abstract program slicing as defined by Mastroeni and Zanardini [6] in Section 2. In Section 3, we go over the modifications made to the abstract slicing algorithm to be able to slice recursive programming languages such as Scheme. In Section 4, we discuss why an abstract slice should be executable to be analysed using abstract interpretation and we adapt the abstract slicer so that it produces executable slices. In Section 5, we run benchmarks on abstract slices of randomly generated Scheme programs to measure the size of the slices when compared to concrete slices, as well as the time it takes to analyse them. In Section 6, we discuss the limitations of our work and possible future work. Finally, Section 7 concludes our work.

2. Background

In this section, we briefly introduce abstract interpretation, program slicing, and the different kinds of program slicing, particularly abstract program slicing.

2.1. Abstract Interpretation

Abstract interpretation is a static analysis technique that over-approximates the semantics of the program [1, 2]. Abstract interpretation is based on the definition of an abstract semantics as a generalisation of the concrete semantics of a program, where operations are defined over an abstract domain. These semantics describe how abstract states are transformed as the program is executed, analogous to how concrete states change during the actual program execution. The idea is that we only keep the information that is strictly necessary for our analysis, even if we need to approximate it in order to make it decidable. For example, if we want to know the sign of the variables at the end of the program, it is sufficient to work in an abstract sign domain where the only information kept is the sign of the variables. We can then run the program with an abstract interpreter, which makes use of these abstract values and abstract operations to approximate the concrete values that would be outputted by a concrete interpreter [7].

2.2. Program Slicing

Program slicing is a technique that was introduced by Weiser [4]. In his original definition, slicing is defined as “starting from a subset of a program’s behaviour, slicing reduces that program to a minimal form which still produces that behaviour". Over the years, this original definition has seen several modifications, resulting in the creation of multiple kinds of slices. Tip [8] shows an overview of these several kinds of slices and slicing methods. Binkley et al. [9, 10] define a mathematical framework to formalise the different forms of slicing in order to compare them, based on the specifics of the slicing criterion.

Program slices are used in several ways. Weiser initially proposed slices as the intuitive way for experienced programmers to reason over their programs, for example during debugging [11]. However, there are also other use cases where slicing can be used. For example, there have been suggested applications in program comprehension, software maintenance, or testing [12]. In general, we usually want slices to be as small as possible while keeping all statements that are relevant to the slicing criterion. The challenge is then to know exactly which statements are safe to be removed without modifying the slicing criterion, while keeping as few statements as possible.

Mastroeni and Zanardini propose a new form of slicing that they call abstract slicing [6]. Instead of slicing based on concrete values, they formalise a technique that allows general slicing based on properties. They argue that this form of slicing, which could in general remove more statements than concrete slicing, can help programmers discover where the error lies when one of their variables does not have a desired property (e.g. a variable is null when it should not be) by only returning the statements that actually have an effect on that property. On the other hand, a concrete slice will contain all statements that influence the value of the variable, regardless of if they impact the property of interest. They formally define this abstract slicing in a framework that makes use of abstract dependencies. Concrete slicing can be defined within this framework as well: if the abstract slicing is done with regards to the identity property (where the identity property of a value is the value itself) then the slice corresponds to a concrete slice. The properties used for the abstract slicing are based on the domains of an abstract interpreter, such that in an abstract slice in regards to a variable x for a specific property ρ, x has the same abstract value as in the original program in the state resulting from the execution of an abstract interpreter using the domain ρ.

This formal framework is an extension of the formal framework defined by Binkley et al. [9, 10]. They extend the slicing criterion into an abstract slicing criterion by adding the criterion abstraction $\mathcal{A}$, which defines the abstract domains we are interested in for each variable in $\mathcal{X}$. $\mathcal{A}$ is defined as a mapping of variables from $\mathcal{X}$ to abstract domains. In Figure 1, an example of the difference between concrete and abstract slices is given. In this example, we see the slices for slicing criterion $(d, 5)$, where the abstract slice is taken for the property ρ_{PARITY}.

```
1 (define a 1)
2 (define b 2)
3 (define c 3)
4 (define e 4)
5 (define d (- (+ (* 2 c) a
      b) a))
```

```
1 (define a 1)
2 (define b 2)
3 (define c 3)
4
5 (define d (- (+ (* 2 c) a
      b) a))
```

```
1
2 (define b 2)
3
4
5 (define d (- (+ (* 2 c) a
      b) a))
```

Figure 1: Left: the original program; middle: a concrete slice; right: an abstract slice

To compute abstract slices, Mastroeni and Zanardini [6] propose an algorithm based on two rule systems: the PP-SYSTEM and the G-SYSTEM. The PP-SYSTEM is used to infer property preservation. If an expression preserves a property of a variable, then this means that the expression could be removed in the abstract slice. The G-SYSTEM is used to propagate agreements backwards through the program, where an agreement of a program point defines which properties for which variables are relevant at that program point. At the end of the program, the relevant properties are those defined in the slicing criterion. Using the rules of the G-SYSTEM, we can then decide based on the expression and the previous agreement what properties are relevant at the previous program point. When a relevant variable is reassigned, the rules of the G-SYSTEM make use of a notion of 'abstract dependencies' to decide which properties of the variables used in the assigned expression are relevant to the property in the agreement. Finally, we can use this information to infer using the PP-SYSTEM which statements can be safely removed from the program to create the abstract slice.

3. An Abstract Slicer for Scheme

We start by adapting the abstract slicing algorithm presented by Mastroeni and Zanardini [6] to work for languages with a recursive structure, such as Scheme. In order to do this, we need to modify the rule systems for property preservation and agreement propagation as described in the original paper so that they recursively check subexpressions. These modified rule systems can be found in Appendix A and B respectively. Additionally, we modified the slicing algorithm so that the algorithm will first recursively remove subexpressions. To ensure that subexpressions without side-effects but whose value is relevant for the superexpression are not sliced, we only slice subexpressions that are not in a position where their value is potentially used.

An example is shown in Figure 2. We see a program and an abstract slice of this program where the relevant property is the sign of x at the end of the program. For every expression in the program, a postcondition is given that defines the relevant properties at this point in the program in the form of an 'agreement'. The first expression on line 2 can be sliced away, as its postcondition is the empty agreement, meaning that no variables are relevant at this point. However, the definition on line 3 has to remain. For the binding of the `let`-expression, it gets slightly more complicated as we have a nested `begin`-expression. In this `begin`-expression, we cannot slice away the `(set! z y)` expression, but we can slice away the `set!`-expression after that. Although the sign of y is relevant *before* this expression, y is not used *after* this expression so the postcondition of the expression no longer contains y and the `set!`-expression can be sliced away. The third expression in the `begin`-expression is simply `z`, which *does* preserve its postcondition, so we could argue that it could be sliced away. However, this `z` is the return value of the `begin`-expression, which is an expression in the right-hand-side of an assignment. This means that we cannot slice the expression `z`. Finally, we take a look at the body of the `let`-expression. The only expression in the body is `x`, which preserves the properties of its postcondition. Although at first sight it looks like this could be a similar situation to the `z` expression of before, as this is the return value of the `let`-expression, we can actually slice away this expression. This is because the `let`-expression is not in the right-hand-side of an assignment or a nested expression as the operator of a function application, so its return value is unused.

```
1                              ; []
2 (define z 5)                 ; []
3 (define y -6)                ; [y -> sign]
4 (let                         ; [x -> sign]
5    ((x                       ; [x -> sign]
6       (begin (set! z y)      ; [z -> sign]
7              (set! y 8)      ; [z -> sign]
8              z)))            ; [x -> sign]
9   x)                         ; [x -> sign]
```

```
1
2
3 (define y -6)
4 (let
5    ((x
6       (begin (set! z y)
7
8              z)))
9 )
```

Figure 2: Example of a sliced Scheme program

4. Executable Abstract Slicing

In order to run an abstract interpreter, it is important that the abstract slice is at least as executable as the original program so that it does not introduce new errors. However, an abstract slice is not necessarily executable.

Figure 3 depicts an example program and its abstract slice for the PARITY property of the variable x. We see that even though the original program is able to be executed without errors, the resulting abstract slice would throw an error because the variables x and y are undefined. The definitions of these variables were removed because their values are irrelevant to the properties of interest of the variable x at the end of the program. The original definition of the variable x in line 1 is irrelevant because the

```scheme
(let ((x 1)
      (y -7)
      (z 42))
   (set! x (+ y (* z 2))))
```

```scheme
(let (
      (y -7)
      )
   (set! x (+ y (* z 2)))))
```

Figure 3: Original program and its non-executable abstract slice

value is overwritten by the `set!`-expression in line 4. The variable z is used in the `set!`-expression on line 4, but its value is not relevant because the expression `(* z 2)` will always be even, so the abstract slicer deems it not necessary to keep the definition of z.

To ensure that we do not introduce extra errors, we need to know when a variable is used in an expression that remains in the slice. We can then use this information to keep `define`-expressions that would otherwise be sliced away only if necessary. However, this removes part of the advantage that an abstract slicer has over a concrete slicer. A concrete slicer always keeps these definitions, as variables are a dependency of the expressions they are used in, unrelated to whether or not their value influences the value of the expression. On the other hand, in an abstract slicer we know when the value of the variable is not relevant, meaning that we can potentially slice away a lot more expressions; especially if the definition of the variable was a complicated expression with many subexpressions. If we need to keep this entire computation when the value is not relevant, then we are knowingly keeping a lot of unnecessary computations in the abstract slice. For this reason, we can instead replace this computation by a 'dummy' value of the right type, so that we do not introduce type errors. In this way, we can still take advantage of the more aggressive slicing of an abstract slicer, while ensuring that the abstract slices remain executable.

4.1. Labelling Irrelevant Expressions

Because we do not want to keep *every* variable definition, we need to first compute what variables are used in expressions that are not sliced away. For this, we can make use of a live variables analysis. However, we need to take into account that not all expressions of the original program are in the slice, so when we check when a variable is live we need to do this for the expressions that remain in the slice. On the other hand, we do not want to run the live variables analysis on the slice itself, because the definitions that were already removed in the abstract slice could change the output of the live variable analysis. To deal with this problem, we work with an intermediate program representation where the expressions are annotated with whether or not they are sliced away, so that we can run a live variables analysis on the unsliced program while still taking the future removals into account. In Figure 4, we see an example program where the expressions that will be removed by the abstract slicer are labelled.

```scheme
(define z 5)              ; irrelevant
(define y -6)
(let
   ((x
      (begin (set! z y)
             (set! y 8)   ; irrelevant
             z)))
  x)                      ; irrelevant
```

Figure 4: Example program labelled with the irrelevant expressions

4.2. Live Variables Analysis

The next step is to perform a live variables analysis on this intermediate representation. The analysis returns a mapping of expressions to the variables that are live for that expression. Because we only consider programs without loops, the live variables analysis is a simple data flow analysis that can be done in a single iteration, starting at the end of the program with an empty list of live variables. It then traverses the program from back to front. For most kinds of expressions, the algorithm simply recurses deeper into the subexpressions. For `if`-expressions, the variables used in the condition expression are added to the list of live variables. The most interesting expressions are the assignment expressions, as these can potentially remove variables from the list of live variables.

Additionally, the live variables analysis has to take into account what expressions will be sliced away, as the variables used in these expressions are not considered live. If an expression was labelled to be sliced away and it is not a definition-expression, we do not modify the list of used variables for this expression. This is because this expression is removed, so any variable used in it is not used in the slice. However, when we have a `define`-expression that is labelled to be sliced away but the variable being defined is in the set of live variables that is being propagated during the analysis, then this means that the variable is used in later parts of the program and we are not able to slice away its definition. In this case, we need to update the set of live variables as if the `define`-expression was not labelled to be sliced. Figure 5 depicts an example program labelled with the variables that are live in the abstract slice after each expression.

```
1 (define z 5)              ; {}
2 (define y -6)             ; {y}
3 (let                      ; {}
4    ((x                    ; {x}
5       (begin (set! z y)   ; {z}
6              (set! y 8)   ; {z}
7              z)))         ; {}
8   x)                      ; {}
```

Figure 5: Example program labelled with the live variables after each expression

4.3. Dummy Values

To ensure that we do not necessarily need to keep the entire computation for the definitions of variables that are only necessary for syntactic purposes, we introduce the concept of ‘dummy values’. These dummy values are values that replace the right-hand-side of `define`-expressions or bindings of `let`-expressions. We need to make sure that they are of the same type as the original expression, so that no type errors are introduced. However, we cannot evaluate the expression to find out what kind of value it returns. The solution is to use the abstract interpreter: this returns an abstract value for the expression, which we can then convert into a concrete value using the concretisation function of the abstract domain.

In order to do this, we introduce ‘dummy’ expressions. A dummy expression evaluates to a value with the same properties as the original value of the expression e as decided by the abstract interpreter. We do this by first running the abstract interpreter and getting the abstract value, and then converting this to a random concrete value that abstracts into the same abstract value. For example, if we are in the ρ_{SIGN} domain and the abstract value is NEG, then a possible dummy value could be -1. If the abstract value is $\top$, then we can choose an arbitrary number as the dummy value. The actual value of the dummy value is not relevant, as the abstract slicer has already established that its value could be sliced away.

For example, Figure 6 depicts how the introduction of dummy values results in an executable slice. For both the variables x and y, we know from the labelling of the irrelevant expressions that their value

```
1 (let (
2       (y -7)
3       )
4    (set! x (+ y (* z 0))))
```

```
1 (let ((x (dummy 1))
2       (y -7)
3       (z (dummy 1))
4    (set! x (+ y (* z 0)))))
```

Figure 6: Example executable slice using dummy values

does not influence the properties of interest. However, during the live variables analysis they are said to be live at the expression that defines them, therefore we cannot slice away these definitions. Instead, their value is replaced by a dummy value of the right type.

5. Evaluation

For our evaluation, we formulate the following research questions:

- **RQ1:** Does our approach using abstract slicing result in smaller slices when compared to concrete slicing?
- **RQ2:** What is the impact of the abstract domain on the size of an abstract slice?
- **RQ3:** Are the abstract slices generated by our abstract slicer analysed faster than concrete slices?

We implemented our approach using Monarch[1], a Haskell framework for creating modular static analyses using abstract interpretation. **Our implementation of the abstract slicer can be found in an online source code repository[2].**

Because our abstract slicer is a prototype that only supports intraprocedural slicing using numeric domains, it is hard to find enough suitable programs to test. For this reason, we generate a dataset of random expressions to ensure that we have examples representing variable amounts of `set!`-expressions. The more `set!`-expressions a program contains, the greater the likelihood that the slicing criterion is modified, meaning that potentially more expressions are relevant to the final abstract value of the variables in the slicing criterion. We created a set of 1050 programs by generating 50 programs for every `set!`-percentage ranging from 0% to 20%. Using this dataset, we ran benchmarks on the size of the slices, which is discussed in Section 5.1, and on the time it takes to analyse them, discussed in Section 5.2.

5.1. Slice Size (RQ1 & RQ2)

We plot the sizes of the resulting abstract and concrete slices in a boxplot, depicted in Figure 7. In this plot, we see the sizes of the slices for both of the different domains, as well as the sizes of the concrete slices. The sizes are expressed as a percentage of the original program size to normalise them. In this boxplot, we clearly see that the median size of the concrete slices is larger than the median size of the abstract slices. This is especially clear when we compare the sizes of the slices for the ρ_{PARITY} domain with the sizes of the concrete slices, where we see that the concrete slices are approximately 10% larger than the abstract slices using the ρ_{PARITY} domain. Additionally, we see that the slices for the ρ_{PARITY} domain are on average approximately 5% smaller than the slices for the ρ_{SIGN} domain. This could be because there are less operations that modify the parity of a variable, so that there are more expressions that preserve the parity property and can be sliced away.

To confirm these findings, we do a statistical analysis of the data. We use a one-sided paired Wilcoxon signed-rank test [13] to compare the slice sizes. We first apply the Wilcoxon test to the sizes of concrete slices and the sizes of slices for the ρ_{SIGN} domain, so that the null hypothesis is that there is, in general, no significant difference between the sizes of concrete slices and abstract slices of the ρ_{SIGN} domain.

[1]https://github.com/softwarelanguageslab/monarch
[2]https://github.com/sarahverbelen/maf-hs

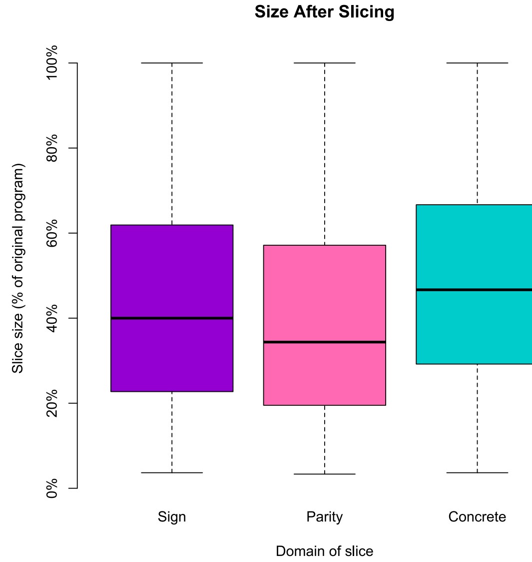

Figure 7: Boxplot showing the size of the slices as a percentage of the original program size

The alternative hypothesis is that the median size of the concrete slices is larger than the median size of the abstract slices of the ρ_{SIGN} domain. The test yields a W statistic of 4173 and a p-value that is less than 2.200×10^{-16}. Since the p-value is much less than the chosen significance level of 0.05, we reject the null hypothesis. This indicates that the median size of concrete slices is larger than the median size of abstract slices using the ρ_{SIGN} domain.

Next, we apply the Wilcoxon test to the sizes of concrete slices and the sizes of slices for the ρ_{PARITY} domain, so that the null hypothesis is that there is, in general, no significant difference between the sizes of concrete slices and abstract slices of the ρ_{PARITY} domain. The alternative hypothesis is that the median size of the concrete slices is larger than the median size of the abstract slices of the ρ_{PARITY} domain. The test yields a W statistic of 2164.5 and a p-value that is less than 2.200×10^{-16}. Since the p-value is much less than the chosen significance level of 0.05, we can again reject the null hypothesis. This indicates that the median size of concrete slices is also larger than the median size of abstract slices using the ρ_{PARITY} domain.

> *Answer RQ1.* The median slice size of the concrete slices is larger than both the median slice sizes of the abstract slices for the ρ_{PARITY} domain and of the ρ_{SIGN} domain. According to Figure 7, the concrete slices are approximately 10% larger than the slices using the ρ_{PARITY} domain and approximately 5% larger than the slices using the ρ_{SIGN} domain.

Finally, we apply a two-sided paired Wilcoxon signed-rank test to discover if there is a significant difference between the sizes of the slices for the ρ_{PARITY} domain and the ρ_{SIGN} domain. The null hypothesis is that there is no significant difference between the sizes of the abstract slices for these two domains, while the alternative hypothesis is that there *is* a significant difference. The test yields a W statistic of 26798 and a p-value that is less than 2.200×10^{-16}. As the p-value is less than 0.025, we can reject the null hypothesis and conclude that there is a significant difference between the sizes of slices for the two different domains. This indicates that the domain used for the abstract slicing is relevant.

> *Answer RQ2.* The slices using the ρ_{PARITY} domain and the slices using the ρ_{SIGN} domain have a significant difference in size. According to Figure 7, the slices using the ρ_{PARITY} domain are approximately 5% smaller than the slices using the ρ_{SIGN} domain.

5.2. Analysis Time (RQ3)

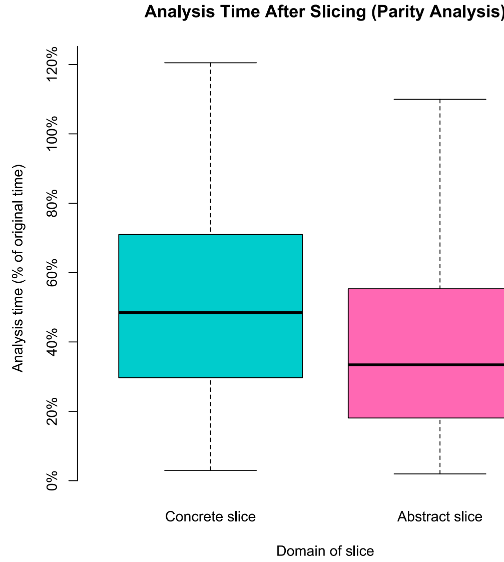

Figure 8: Boxplot showing the analysis time of the slices as a percentage of the analysis time of the original program

First, we plot the results of the benchmarks in a boxplot. For every program, we computed the relative analysis time for concrete and abstract slices by dividing them by the analysis time of the corresponding original program. This normalises the data by showing it as a percentage of the original analysis time. In the boxplot in Figure 8, we see that, in general, the abstract slices using the ρ_{PARITY} domain are analysed approximately 20% faster than the concrete slices. To confirm this, we do a statistical analysis to see the difference in the time it takes to analyse the concrete slices compared to the abstract slices. For all following statistical tests, we select a significance level of 0.05.

We use a one-sided paired Wilcoxon signed-rank test [13] to compare the analysis times of the slices, so that the null hypothesis is that there is no significant difference between the analysis time of abstract and concrete slices and the alternative hypothesis is that the median of the analysis times of the concrete slices is higher than the median of the analysis times of the abstract slices. We observe a W statistic of 482174 and a p-value of 2.200×10^{-16}. As the p-value is extremely low, it is below the chosen significance level of 0.05 and we reject the null hypothesis. We accept the alternative hypothesis and can conclude that the median analysis time of the concrete slices is greater than the median analysis time of the abstract slices for the ρ_{PARITY} domain, confirming the results of the boxplot in Figure 8.

Answer RQ3. In Figure 8, we see that the abstract slices are analysed approximately 20% faster than the concrete slices. From our statistical analysis, we can confirm that the median analysis time of the concrete slices is greater than the median analysis time of the abstract slices for the ρ_{PARITY} domain.

6. Limitations & Future Work

The first limitation of our work is that our abstract slicer is an intraprocedural slicer. In real world scenarios, programs typically consist of multiple procedures. Future work could focus on extending the abstract slicer to slice interprocedurally, which would be more applicable for real use cases. Additionally, an interprocedural abstract slicer has the potential to slice large parts of the program, as procedures

that do not influence the properties of the slicing criterion could be sliced away completely. This would likely significantly improve the speed of the static analysis that we want to run on the abstract slices.

Another limitation of our work is that our implementation is limited to numerical domains. This limited the kind of programs that we could use to evaluate our abstract slicer. Future work could investigate if the improvements that we measured on numerical domains would hold for complex domains that might be influenced by most operations, or if abstract slicing is only useful for simple domains where it is clear that many operations do not influence the property.

Finally, if we want to speed up static analysis by first creating an abstract slice for the program, we should take into account that the slicing itself also takes time. Ideally, the time it takes to slice combined with the time it takes to analyse the abstract slice should be less than the time it takes to analyse the original program. Otherwise, the overhead caused by the abstract slicing would negate the improvement in analysis time. We did not test how long it takes to slice a program using our abstract slicer, as the implementation is not optimised for efficiency. Future work could focus on creating an optimised implementation of an abstract slicer to investigate if abstract slicing is a good preprocessing step to improve the efficiency of static analyses.

7. Conclusion

In this paper, we presented an intraprocedural abstract slicer for Scheme programs by modifying an existing abstract slicing framework with the goal of using these abstract slices to improve the efficiency of static analyses by abstract interpretation. For this reason, we adapted the abstract slicer to produce executable abstract slices that can be analysed by an abstract interpreter. Next, we provided an implementation of the abstract slicer for Scheme programs in the Monarch framework. We used the implementation of abstract domains and the abstract interpreter from the framework to make the abstract slices specific to the static analyses defined in the framework. Finally, we ran benchmarks on a data set of randomly generated programs with a varied amount of `set!`-expressions. We found that in general, the abstract slices are smaller and faster to analyse than the corresponding concrete slices. Additionally, we discovered that there is a significant difference in the sizes of the slices of the ρ_{SIGN} and ρ_{PARITY} domains.

References

[1] P. Cousot, Abstract interpretation, ACM Computing Surveys (CSUR) 28 (1996) 324–328.

[2] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77), Association for Computing Machinery, 1977, p. 238–252. doi:10.1145/512950.512973.

[3] M. Christakis, C. Bird, What developers want and need from program analysis: an empirical study, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16), Association for Computing Machinery, 2016, p. 332–343. doi:10.1145/2970276.2970347.

[4] M. D. Weiser, Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method, Ph.D. thesis, 1979.

[5] N. Allen, B. Scholz, P. Krishnan, Staged points-to analysis for large code bases, in: B. Franke (Ed.), Compiler Construction, Springer Berlin Heidelberg, 2015, pp. 131–150.

[6] I. Mastroeni, D. Zanardini, Abstract program slicing: An abstract interpretation-based approach to program slicing, ACM Transactions on Computational Logic 18 (2017). doi:10.1145/3029052.

[7] P. Cousot, Abstract interpretation: From 0, 1, to ∞, Intelligent Systems Reference Library 238 (2023). doi:10.1007/978-981-19-9601-6_1.

[8] F. Tip, A survey of program slicing techniques., 1994.

[9] D. W. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Á. Kiss, B. Korel, A formalisation of the relationship between forms of program slicing, Science of Computer Programming 62 (2006) 228–252.
[10] D. W. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Á. Kiss, B. Korel, Theoretical foundations of dynamic program slicing, Theoretical Computer Science 360 (2006) 23–41.
[11] M. Weiser, Program slicing, IEEE Transactions on Software Engineering 10 (1984) 352–357. doi:10.1109/TSE.1984.5010248.
[12] J. Krinke, PROGRAM SLICING, 2005, pp. 307–332. doi:10.1142/9789812775245_0011.
[13] F. Wilcoxon, Individual Comparisons by Ranking Methods, Springer New York, New York, NY, 1992, pp. 196–202. doi:10.1007/978-1-4612-4380-9_16.

A. Property Preservation Rule System for Scheme Programs

$$\frac{}{PP^{\beta}(\mathcal{G},\ s)}\ \text{PP-SKIP}$$

$$\frac{PP^{\beta}(\mathcal{G},\ e) \qquad \forall\sigma.\ \sigma \models \beta \Rightarrow \mathcal{G}(x)(\sigma(x)) = \mathcal{G}(x)([\![e]\!]^{\rho}(\sigma))}{PP^{\beta}(\mathcal{G}, \text{(define x e)})}\ \text{PP-ASSIGN-DEFINE}$$

$$\frac{PP^{\beta}(\mathcal{G},\ e) \qquad \forall\sigma.\ \sigma \models \beta \Rightarrow \mathcal{G}(x)(\sigma(x)) = \mathcal{G}(x)([\![e]\!]^{\rho}(\sigma))}{PP^{\beta}(\mathcal{G}, \text{(set! x e)})}\ \text{PP-ASSIGN-SET}$$

$$\frac{PP^{\beta}(\mathcal{G},\ e_1) \qquad \ldots \qquad PP^{e_{n-1}(..e_1(\beta))}(\mathcal{G},\ e_n)}{PP^{\beta}(\mathcal{G}, \text{(begin } e_1 \ldots e_n\text{)})}\ \text{PP-CONCAT}$$

$$\frac{PP^{\beta\wedge b'}(\mathcal{G}_t,\ e_t) \qquad PP^{\beta\wedge\neg b'}(\mathcal{G}_f,\ e_f)}{PP^{\beta}(\mathcal{G}_t \bigsqcup \mathcal{G}_f, \text{(if } b\ e_t\ e_f\text{)})}\ \text{PP-IF}$$

$$\frac{PP^{\beta}(\mathcal{G}, \text{(define } x_1\ e_1\text{)}) \qquad \ldots \qquad PP^{e_{n-1}(..e_1(\beta))}(\mathcal{G}, \text{(define } x_n\ e_n\text{)}) \qquad PP^{\beta_{\text{let}}}(\mathcal{G},\ e_b)}{PP^{\beta}(\mathcal{G}, \text{(let ((} x_1\ e_1\text{) ... (} x_n\ e_n\text{)) } e_b\text{)})}\ \text{PP-LET}$$

$$\frac{\text{PRIMITIVE(op)} \qquad PP^{\beta}(\mathcal{G},\ e_1) \qquad \ldots \qquad PP^{e_{n-1}(..e_1(\beta))}(\mathcal{G},\ e_n)}{PP^{\beta}(\mathcal{G}, \text{(op } e_1 \ldots e_n\text{)})}\ \text{PP-APP}$$

Figure 9: The PP-SYSTEM for Scheme

B. Agreement Propagation Rule System for Scheme Programs

$$\frac{PP^{\beta}(\mathcal{G},\ e)}{\{\mathcal{G}\}^{\beta}\ e\ \{\mathcal{G}\}}\ \textsc{g-pp}$$

$$\frac{\{\mathcal{G}\}^{\beta}\ e_1\ \{\mathcal{G}_1\} \qquad \dots \qquad \{\mathcal{G}_{n-1}\}^{e_{n-1}(..e_1(\beta))}\ e_n\ \{\mathcal{G}_n\}}{\{\mathcal{G}\}^{\beta}\ (\text{begin}\ e_1\ \dots\ e_n)\ \{\mathcal{G}_n\}}\ \textsc{g-concat}$$

$$\frac{\forall y.\ \neg(y\ \overset{\mathcal{G},\ \mathcal{G}'(x)}{\rightsquigarrow}_{\text{AT}}\ e)^{\beta} \qquad \forall y \neq x.\ \mathcal{G}(y) = \mathcal{G}'(y)}{\{\mathcal{G}\}^{\beta}\ (\text{define}\ x\ e)\ \{\mathcal{G}'\}}\ \textsc{g-assign-define}$$

$$\frac{\forall y.\ \neg(y\ \overset{\mathcal{G},\ \mathcal{G}'(x)}{\rightsquigarrow}_{\text{AT}}\ e)^{\beta} \qquad \forall y \neq x.\ \mathcal{G}(y) = \mathcal{G}'(y)}{\{\mathcal{G}\}^{\beta}\ (\text{set!}\ x\ e)\ \{\mathcal{G}'\}}\ \textsc{g-assign-set}$$

$$\frac{\{\mathcal{G}\}^{\beta}\ s_t \diamond s_f\ \{\mathcal{G}'\}}{\{\mathcal{G}\}^{\beta}\ (\text{if}\ b\ e_t\ e_f)\ \{\mathcal{G}'\}}\ \textsc{g-if}1$$

$$\frac{\forall y.\ \neg(y\ \overset{\mathcal{G}_b,\ \mathcal{G}_b}{\rightsquigarrow}_{\text{AT}}\ b)^{\beta} \qquad \{\mathcal{G}_t\}^{\beta \wedge b}\ s_t\ \{\mathcal{G}'\} \qquad \{\mathcal{G}_f\}^{\beta \wedge \neg b}\ s_f\ \{\mathcal{G}'\}}{\{\mathcal{G}_b \sqcap \mathcal{G}_t \sqcap \mathcal{G}_f\}^{\beta}\ (\text{if}\ b\ e_t\ e_f)\ \{\mathcal{G}'\}}\ \textsc{g-if}2$$

$$\frac{\{\mathcal{G}\}^{\beta}\ (\text{define}\ x_1\ e_1)\ \{\mathcal{G}_1\} \qquad \dots \qquad \{\mathcal{G}_{n-1}\}^{e_{n-1}(..e_1(\beta))}\ (\text{define}\ x_n\ e_n)\ \{\mathcal{G}_n\} \qquad \{\mathcal{G}_n\}^{e_n(..e_1(\beta))}\ e_b\ \{\mathcal{G}'\}}{\{\mathcal{G}\}^{\beta}\ (\text{let}\ ((x_1\ e_1)\ \dots\ (x_n\ e_n))\ e_b)\ \{\mathcal{G}'\}}\ \textsc{g-let}$$

$$\frac{\textsc{primitive}(\text{op}) \qquad \{\mathcal{G}\}^{\beta}\ e_1\ \{\mathcal{G}_1\} \qquad \dots \qquad \{\mathcal{G}_{n-1}\}^{e_{n-1}(..e_1(\beta))}\ e_1\ \{\mathcal{G}'\}}{\{\mathcal{G}\}^{\beta}\ (\text{op}\ e_1\ \dots\ e_n)\ \{\mathcal{G}'\}}\ \textsc{g-app}$$

Figure 10: The G-SYSTEM for Scheme