Modular Specification and Compositional Soundness of Abstract Interpreters

Dissertation
submitted for the award of the title
“Doctor of Natural Science”
to the Faculty Physics, Mathematics, and Computer Science
of Johannes Gutenberg-University
in Mainz

Sven Keidel

born in Frankfurt-Höchst, Germany.
Mainz, 9.3.2021

D77

CONTENTS

Abstract iii

Acknowledgements v

1 Introduction 1
1.1 What Are Static Analyses and Why Do They Matter . . . 1
1.2 Soundness of Static Analyses . . . 2
1.3 Why Existing Analyses Are Difficult to Prove Sound . . . 4
1.4 Compositional Soundness Proofs of Static Analyses . . . 6
1.5 Dissertation Outline . . . 10

2 Background on Big-Step Abstract Interpretation and Arrows 13
2.1 Introduction to Big-Step Abstract Interpreters . . . 13
2.2 Arrows, an Abstraction for Effectful Computations . . . 15

3 Capturing the Similarities between Concrete and Abstract Interpreters 21
3.1 Introduction . . . 21
3.2 Why and How to Make Soundness Proofs Compositional . . . 23
3.3 Soundness Proposition for Arrows . . . 28
3.4 Compositional Soundness for Arrow-Based Abstract Interpreters . . . 30
3.5 Interface Design and Parametricity . . . 32
3.6 Case Studies . . . 35
3.7 Related Work . . . 38
3.8 Conclusion . . . 40

4 Sound and Reusable Components for Abstract Interpretation 41
4.1 Introduction . . . 41
4.2 Analysis Components By Example . . . 43
4.3 Analysis Components And Their Soundness . . . 47
4.4 Sound Composition Of Analysis Components . . . 50
4.5 Soundness Of Component-Based Static Analyses . . . 53
4.6 Sturdy: A Library Of Sound And Reusable Analysis Components . . . 54
4.7 Experimental Evaluation And Case Studies . . . 57
4.8 Related Work . . . 62
4.9 Conclusion . . . 63

5 Modular Fixpoint Algorithms for Big-Step Abstract Interpreters 65
5.1 Introduction . . . 65
5.2 Designing Big-Step Fixpoint Algorithms . . . 67
5.3 Modularizing the Description of Big-Step Fixpoint Algorithms . . . 73
5.4 Soundness of Modular Big-Step Fixpoint Algorithms . . . 79
5.5 Evaluation . . . 82
5.6 Related Work . . . 84
5.7 Conclusion . . . 86

6 A Systematic Approach to Abstract Interpretation of Program Transformations 87
6.1 Introduction . . . 87
6.2 Illustrating Example: Singleton Analysis . . . 88
6.3 Generic Interpreters for Program Transformations . . . 91
6.4 Sort Analysis . . . 95
6.5 Locally Ill-Sorted Sort Analysis . . . 98

6.6 Related Work 100
6.7 Conclusion 101

7 Related Work **103**
7.1 Modular Analyses Description and Compositional Soundness Proofs 103
7.2 Other Techniques for Ensuring the Soundness of Static Analyses 111
7.3 Testing Soundness 113

8 Future Work **115**
8.1 Proof Mechanization 115
8.2 Backward Analyses 115
8.3 Performance Scalability 116

9 Conclusion **117**

Bibliography **119**

ABSTRACT

Static analyses are automated tools that yield information about computer programs without running them. Static analyses are used in modern integrated development environments (IDEs), compilers, and continuous integration servers. For example, IDEs use static analyses to warn developers about common problems such as type errors, dead code, or memory leaks. Compilers use static analyses mainly to optimize the code. For example, a compiler may use a constant analysis to move variable assignments out of loops that do not change.

Depending on the use case of a static analysis, the results of the analysis need to be more or less reliable. For example, if a dead code analysis used in an IDE fails to report that a piece of code will never be executed, then the consequences will probably be not that bad. However, if a compiler optimization relies on a faulty analysis, then the optimization may change the semantics of the program. Such a changed program semantics leads to unexpected program behavior and crashes. In these use cases, the unreliable analyses are unacceptable.

One way of ensuring that a static analysis is reliable, is to prove that the analysis is *sound*. Soundness means that a static analysis never fails to report a certain class of bugs or program behavior. For example, a sound static type checker never misses a type error and hence it guarantees that a type correct program executes without crashing. Soundness is especially important for static analyses used in compiler optimizations, for security analyses, and for analyses that verify mission-critical software.

Unfortunately, formally proving that a static analysis is sound is difficult. A formal soundness proof shows *for all* programs, that the analysis result corresponds to the execution of the program. While some complexity of a soundness proof is inherent, a large part is incidental and comes from how the analysis is structured. In particular, many static analyses do not clearly separate different concerns. This coupling often makes a soundness proof infeasible. For example, the analyses in the LLVM compiler have not been proven sound and developers discovered 81 cases of unsoundness since 2010.

In this thesis, we propose a methodology for structuring static analyses such that their soundness proof becomes easier. The core principle of this methodology is to make the soundness proof *compositional*. More specifically, we show how analyses can be implemented modularly with small and reusable components. Each component can be proven sound independently and the composition of components remains sound. This means analysis developers do not need to worry about soundness, as long as they use sound components.

We evaluate our methodology by developing several control-flow and data-flow analyses for languages with different programming paradigms. We implemented these analyses by using sound and reusable components of the open-source Haskell library *Sturdy*[1], that we developed as part of this thesis. Our experiments show that compositional soundness proofs require less effort, because large parts of the analysis consist of language-independent components that have been proven sound a priori. Furthermore, compositional soundness proofs are less complicated, because components capture only a small piece of functionality, which can be easily proven sound.

[1] https://github.com/svenkeidel/sturdy

ACKNOWLEDGEMENTS

Anonymized for online publication.

1 INTRODUCTION

The goal of this thesis is to simplify the soundness proof of static analyses. To set the stage, we explain what static analyses are, how they are used, and why they matter.

1.1 What Are Static Analyses and Why Do They Matter

Static analyses are tools that analyze the source code of computer programs to provide information about their execution. They calculate this information by inspecting the source code of a program, but without running it. Static analyses have a wide variety of applications, which we discuss in this section.

Static analyses are effective tools for ensuring the security of software [Ayewah et al. 2008; Bessey et al. 2010]. For example, consider the *Heartbleed* security vulnerability in the OpenSSL library that affected millions of servers across the internet [Durumeric et al. 2014]. The vulnerability leaked sensitive information about SSL keys by reading outside the bounds of an array. Even though the OpenSSL library is widely used and well tested, the tests missed this vulnerability. To quote Dijkstra [1969], „program testing can be used to show the presence of bugs, but never to show their absence“. In contrast, a static array bounds analysis [Hovemeyer et al. 2005] could have detected the vulnerability [Wheeler 2014]. In particular, the analysis guarantees that all array accesses are within their bounds, during all executions of the program. In other words, the analysis does not miss unsafe array accesses, like the one in the Heartbleed vulnerability. In summary, static analyses are tools to ensure the security of software, because they provide stronger guarantees than testing.

Integrated development environments (IDEs) use static analyses to provide feedback to software developers about their programs. For example, Figure 1.1 illustrates the use of static analyses in a Python IDE. The highlighted area 1 lists common problems in the edited programs, such as unused variables, dead code, possible null-pointer dereferences, etc. This information is produced by several static analyses called bug finders [Ayewah et al. 2008; Calcagno et al. 2015]. Area 2 presents a list of possible code completions for the current context. To make these completions more relevant, the IDE narrows down the list of all possible completions by considering type information. This information is produced by a static analysis called a type checker [Cardelli 1996; Pierce 2002; Mitchell 1990]. Area 3 and 4 describe the class structure of the edited program. While in Python some parts of the class structure are evident by glancing at the source code, other parts are not so obvious. This is because Python allows to dynamically create or change parts of the class structure at runtime. These dynamic parts of the class structure can be determined ahead of time with another static analysis [Fromherz et al. 2018]. Lastly, static analyses enable several automated code refactorings [Mens and Tourwé 2004; Wedyan et al. 2009]. To summarize, IDEs use static analyses to enrich the development experience over plain text editors.

Compilers often use static analyses to enable code optimizations. For example, Figure 1.2 illustrates how a *loop-hoisting* compiler optimization [Aho et al. 1986] uses information of a static analysis. A loop-hoisting optimization moves code outside the body of a loop to avoid redundant recomputation. For instance, the value of the variable `c` on the left of Figure 1.2 never changes between loop iterations and hence the assignment `c=a+b` is redundant. To avoid this redundancy, the optimization *hoists* the assignment out of the loop, such that the assignment is only computed once. To detected which code is redundant, the optimization uses a *reaching definitions analysis* [Nielson et al. 1999]. This analysis computes which variable definitions reach a certain statement, without being overwritten. In particular, the variables in the expression `a+b` are defined by the function parameters and are not overwritten in between. This means the value of the expression `a+b` does not change between loop iterations and the assignment can be safely moved outside the loop body. In summary, compilers use the information of static analyses to generate efficient code.

To summarize, static analyses retrieve information about programs without running them. They can be used to ensure the security of software, they aid programmers during the develop-

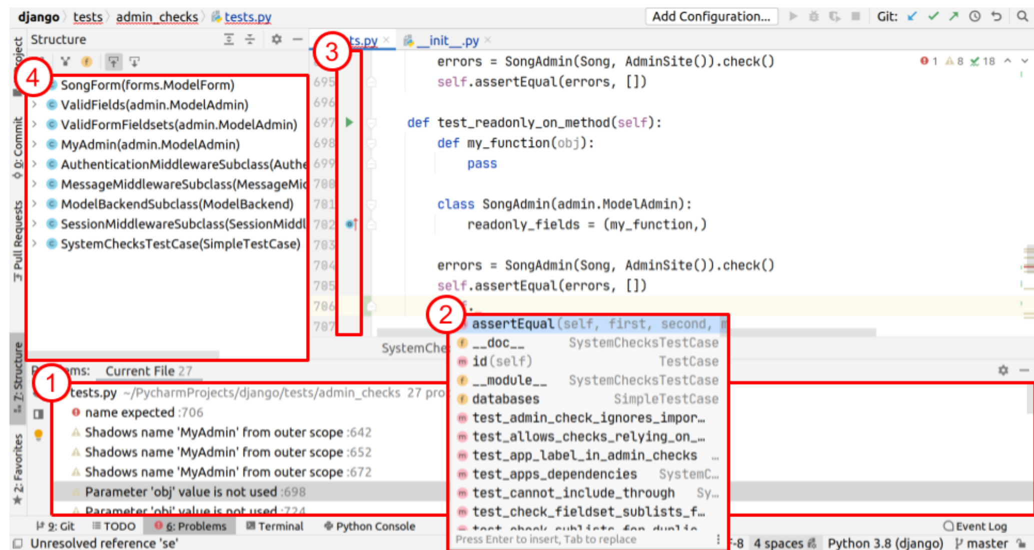


Figure 1.1: Screenshot of the Python IDE *PyCharm*. The highlighted parts contain program information produced by static analyses.

```
func(a, b) {
  for(x = y = 0; x < 100; x++) {
    c = a + b
    y = x * y + c
  }
  return y
}
```

$\Longrightarrow$

```
func(a, b) {
  c = a + b
  for(x = y = 0; x < 100; x++) {
    y = x * y + c
  }
  return y
}
```

Figure 1.2: Compiler optimization that moves constant code out of loops. The optimization requires information of a static analysis to detect which variables do not change between loop iterations.

ment process, and they allow compilers to generate efficient code. However, many static analyses lose their usefulness, if they produce false information. For example, consider a compiler optimization that uses false analysis information. The optimization may change the semantics of programs, which leads to unexpected program behavior and bugs that are hard to find. To avoid this, many analysis developers ensure that their analysis is *sound*, which we discuss in the following section.

1.2 Soundness of Static Analyses

In this section, we explain what soundness is and why it is a necessary requirement for some analysis use cases. However, to be able to understand soundness, we first need to explain why an analysis cannot always compute precise information about programs.

Consider an array bounds analysis [Hovemeyer et al. 2005] that ensures that array accesses are safely within the bounds of the array. For example, consider the following program:

```
for(i = 0; i < x.length-1; i++)
  x[i] = x[i] + x[i+1];
```

The analysis determines the range of index variable `i`. Since variable `i` is within the range $0 \leq$ `i` $<$ `x.length` $- 1$, the array accesses `x[i]` and `x[i+1]` are both safely within the bounds of array `x`.

However, guaranteeing the safety of array accesses is not always that easy. For example, consider a variation of the program above, where the index variable depends on a complicated arithmetic expression:

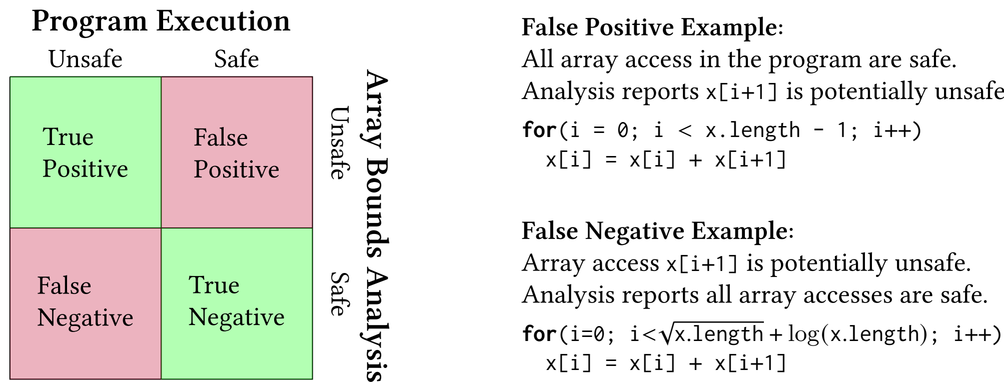


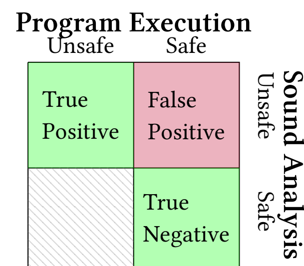
Figure 1.3: Classification of cases where an array bounds analysis correctly or incorrectly predicted the safety of array accesses.

```
for(i = 0; i < √x.length + log(x.length); i++)
  x[i] = x[i] + x[i+1]
```

To guarantee that index variable `i` does not exceed the upper bound of array `x`, the analysis needs to reason about the loop condition $i < \sqrt{\texttt{x.length}} + \log(\texttt{x.length})$. This requires knowledge about the mathematical properties of roots and logarithms. However, even if the analysis were able to reason about this program, in practice, other programs are even more complicated. The complexity of these programs makes it impossible for the analysis to give a precise answer about the safety of array accesses. To overcome this problem, the analysis may give an *imprecise* answer that an array access is *potentially* unsafe.

More generally, static analyses attempt to verify properties of the execution of programs, such as the safety of all array accesses. Unfortunately, most interesting properties about the execution of programs are undecidable due to Rice's theorem [Rice 1953]. To overcome this problem, static analyses retain decidability by giving imprecise answers. To better understand imprecise answers, we classify the cases in which an analysis correctly or incorrectly predicted the property. Figure 1.3 shows this classification at the example of the array bounds analysis. The rows of the diagram specify what the analysis predicted and the columns show what actually happens when the program is executed. The interesting cases are when the analysis was mistaken (false positive and false negative), which we discuss in the following.

An analysis answer is *false positive*, if the analysis falsely reports that a program violates a property. For example, the array bounds analysis may report that an array access is potentially unsafe, while all array accesses are safe when the program is executed. False positive answers can be annoying for software developers, because they cause them to needlessly investigate problems that a program does not have. Furthermore, software developers start to ignore the entire report of the analysis, if too many results are false positive [Sadowski et al. 2015]. For this reason, IDEs usually use static analyses with a low number of false positives answers. In contrast, for compiler optimizations false positive analysis answers are more tolerable. In such a case, the compiler may skip the optimization and leave the program unchanged.



An analysis answer is *false negative*, if the analysis falsely reports that a program satisfies a property. For example, the array bounds analysis may report that all array accesses are safe, while the program crashes due to an unsafe array access. False negative analysis answers can be more tolerable within an IDE setting. For instance, software developers care more about that reported bugs are actually bugs in the program, than about an exhaustive report of all potential bugs of the program [Sadowski et al. 2015]. However, for other analysis use cases, false negative answers are undesirable. These use cases require a special class of static analyses, which never give false negative answers. Static analyses which never give false negative answers are called *sound*. For example, it is desirable for type checkers to be sound [Pierce 2002; Rompf and Amin 2016]. A sound type

checker reports all possible type errors, before the program is executed. If a type checker is unsound, it misses some type errors, which are potential bugs in the program. Furthermore, it is important that security analyses are sound [Wassermann and Su 2007; Russo and Sabelfeld 2010]. A sound security analysis guarantees that a certain type of vulnerability cannot occur. If a security analysis is unsound, it fails to report some security vulnerabilities, which leaves programs open for attacks. Lastly, it is imperative that the static analyses of compiler optimizations are sound [Kanade et al. 2007; Lerner et al. 2003]. If these analyses were unsound, the optimizations may change the semantics of a program, which leads to unexpected program behavior and bugs that are hard to find.

To summarize, sound static analyses do not give false negative answers. Soundness is important for analysis use cases where false negative answers are undesirable. However, many analyses that attempt to be sound, are in fact unsound. This is because it is difficult to ensure that an analysis is sound.

1.3 Why Existing Analyses Are Difficult to Prove Sound

To ensure that an analysis is sound, many analysis developers prove their analysis sound formally. A formal soundness proof relates how programs are executed to the results of the analysis. However, many soundness proof attempts are difficult or even infeasible, because the proofs are *monolithic*. Monolithic means that it is difficult to split the soundness proof into smaller lemmas that are independently provable. In this section, we discuss three reasons that cause soundness proofs to become monolithic. In the following section, we discuss how this dissertation solves these problems by making soundness proofs *compositional*.

Impedance mismatch between the style of concrete and abstract semantics The first reason for monolithic soundness proofs is the impedance mismatch between the style of the concrete and abstract semantics. We illustrated this impedance mismatch with an analogy in Figure 1.4. For example, type checkers are usually described with a *big-step semantics*, whereas they are proved sound with respect to a *small-step semantics* [Pierce 2002]. The impedance mismatch between the semantics requires a *type preservation lemma*, which says that the type of an expression before and after each evaluation step are the same. Furthermore, the small-step program semantics often uses substitution to bind variables, whereas the type checker uses environments [Pierce 2002]. This impedance mismatch requires a *substitution lemma*, which says that the type before and after substitution are the same. These two lemmas are an indication of the impedance mismatch and increase the effort and complexity of the soundness proof.

For real-world programming languages, this impedance mismatch complicates the soundness proof significantly. For example, Scala's new type system (*dependent object types (Dot)*) was proposed in 2012 [Amin et al. 2012]. After its inception, it took 4 years and several attempts [Amin et al. 2014, 2016] to prove the type system sound [Rompf and Amin 2016]. Yet, the Dot core calculus only covers a small subset of the Scala type system. Despite a lot of progress [Jeffery 2019; Campos and Vasconcelos 2018; Rapoport and Lhoták 2019; Giarrusso et al. 2020], extending the Dot calculus soundly to the full Scala type system remains a challenge.

No Separation of Effects The second problem that makes soundness proofs monolithic, is that analyses often do not clearly separate the implementation of different effects of the language. For example, consider the following Java program that tries to increment an element of an array.

```
1 int safeIncrement(int arr[], int index) {
2   try {
3     arr[index] += 1; // exception may occur if the index exceeds the array bounds.
4   } catch(IndexOutOfBoundsException e) {
5     index = -1;
6   }
7   return index;
8 }
```

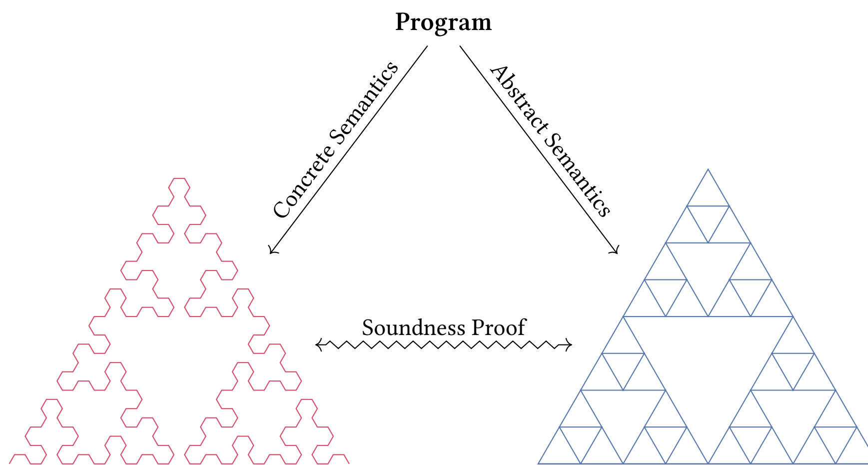


Figure 1.4: Depiction of a monolithic soundness proof. A soundness proof relates the execution of a program (concrete semantics) to the results of an analysis (abstract semantics). A problem that makes the soundness proof monolithic is the impedance mismatch between the concrete and abstract semantics. In this analogy, we depict the impedance mismatch with two different types of Sierpinski triangles [Sierpinski 1915]. Even though it is easy to see that the concrete and abstract semantics are related somehow, it is difficult to split the soundness proof into smaller lemmas, which can be independently proven.

Depending on the length of the array `arr`, line 3 may or may not cause an exception. This means the variable `index` at the `return` statement is either unchanged or `-1`. An analysis needs to consider both of these cases. This example demonstrates the interaction between the effects of mutable variables and exceptions. This interaction makes it difficult to separate these concerns in an analysis and soundness proof.

Another downside of the mixing of effects is, that it is hard to reuse existing analysis functionality. For example, it is hard to reuse functionality for analyzing exceptions, if the language also features mutable variables. This causes many analysis developers to develop and prove new analyses sound completely from scratch [Shivers 1991; Jensen et al. 2009; Jourdan et al. 2015; Rompf and Amin 2016; Al-Sibahi et al. 2018]. However, such from-scratch soundness proofs are often infeasible due to the high effort and complexity. For instance, many static analyses that aim to be sound only have partial soundness proofs or no soundness proofs [Andreasen et al. 2017; Emanuelsson and Nilsson 2008], others are known to be unsound [Wang et al. 2016; Smaragdakis and Kastrinis 2018; Livshits et al. 2015].

Monolithic Fixpoint Algorithms A central part of every static analysis is its *fixpoint algorithm*. A fixpoint algorithms calculates the analysis result for loops and recursive functions. It repeatedly reanalyzes parts of the program until the analysis result does not change anymore. However, there is often no single fixpoint algorithms that performs best for all analyzed programs. For example, consider the following Java Program:

```
1  int x[] = ...; int y[] = ...;
2  for(int i = 0; i < x.length; i++) {
3    ... // Code that is computationally expensive to analyze
4    for(int j = 0; j < y.length; j++) {
5      ... // Code that is computationally expensive to analyze
6    }
7  }
```

A fixpoint algorithm needs to repeatably reanalyze both loops. However, depending on the dimensions of the arrays `x` and `y`, it can be faster to first reanalyze the inner loop or first the outer loop, or a combination thereof. This example shows that it is hard to find a single best analysis order, that performs well for all programs. The example also shows that the development of a fixpoint algorithm for a new analysis requires fine-tuning and adaptation.

Unfortunately, many existing fixpoint algorithms are monolithic [Bourdoncle 1993; Nielson et al. 1999; Rosendahl 2013; Kim et al. 2020]. This means it is hard to split their implementation into smaller pieces, which can be independently proven sound. This makes the soundness proof more difficult as the entire fixpoint algorithm needs to be proven sound at once. Furthermore, the soundness proof of these fixpoint algorithms is also brittle, when we fine-tune or adapt them. In particular, every small change to the fixpoint algorithm invalidates the soundness proof and the proof needs to be reconstructed all over again.

To summarize, many soundness proofs are difficult because they are monolithic. Monolithic means that it is hard to decompose the soundness proof into smaller lemmas, which can be proven independently. The main reasons for monolithic soundness proofs are the impedance mismatch between the concrete and abstract semantics, the mixing of concerns, and monolithic fixpoint algorithms. In the next section, we explain how this dissertation solves these problems by making soundness proofs compositional.

1.4 Compositional Soundness Proofs of Static Analyses

In this section, we present a methodology for structuring static analyses, such that their soundness proof becomes *compositional*. Compositional means that the soundness proof consists of small independent soundness lemmas and does not require reasoning about the analysis as a whole. In particular, we claim the following hypothesis:

> Compositional soundness proofs are feasible and reduce the effort and complexity of developing sound static analyses.

Our evaluation shows our methodology simplifies soundness proofs and reduces their effort.

1.4.1 Capturing the similarities between analysis and language semantics

The first problem we need to solve is the impedance mismatch between the concrete and abstract semantics. We explained in the previous section how this impedance mismatch increases the complexity and effort of the soundness proof. We solve this by (1) describing both the concrete and abstract semantics with the same style of semantics and (2) we capture the similarities between the concrete and abstract semantics.

In particular, we capture the similarities between the language semantics and the analysis with a *generic semantics*. The generic semantics operates on expressions of the same language, but can be instantiated different ways to derive the concrete or abstract semantics. To this end, the generic interpreter is parameterized by operations that abstract over the values of the language. The interface describes the types of these operations. To recover the concrete semantics and abstract semantics, we instantiate the generic semantics with two different implementations of the operations.

This reorganization has the following benefits:

- The soundness proof takes less effort, because an instantiated generic semantics is sound, if its operations are sound. This means no reasoning about the generic semantics is necessary and it suffices to prove soundness of the operations. We prove this with a theorem once and for all generic semantics.
- The soundness proof of the operations is less complicated, because each operation captures only a small piece of functionality. This small piece of functionality is easier to reason about than the whole interpreter. Furthermore, each operation can be proven sound independently of other operations and independently of the generic semantics.
- The reorganization reduces the effort of creating new analyses for the same language. In particular, we can reuse the same generic semantics to create a variety of different analyses. We demonstrate this in Chapter 6.

We describe this technique in more detail in Chapter 3.

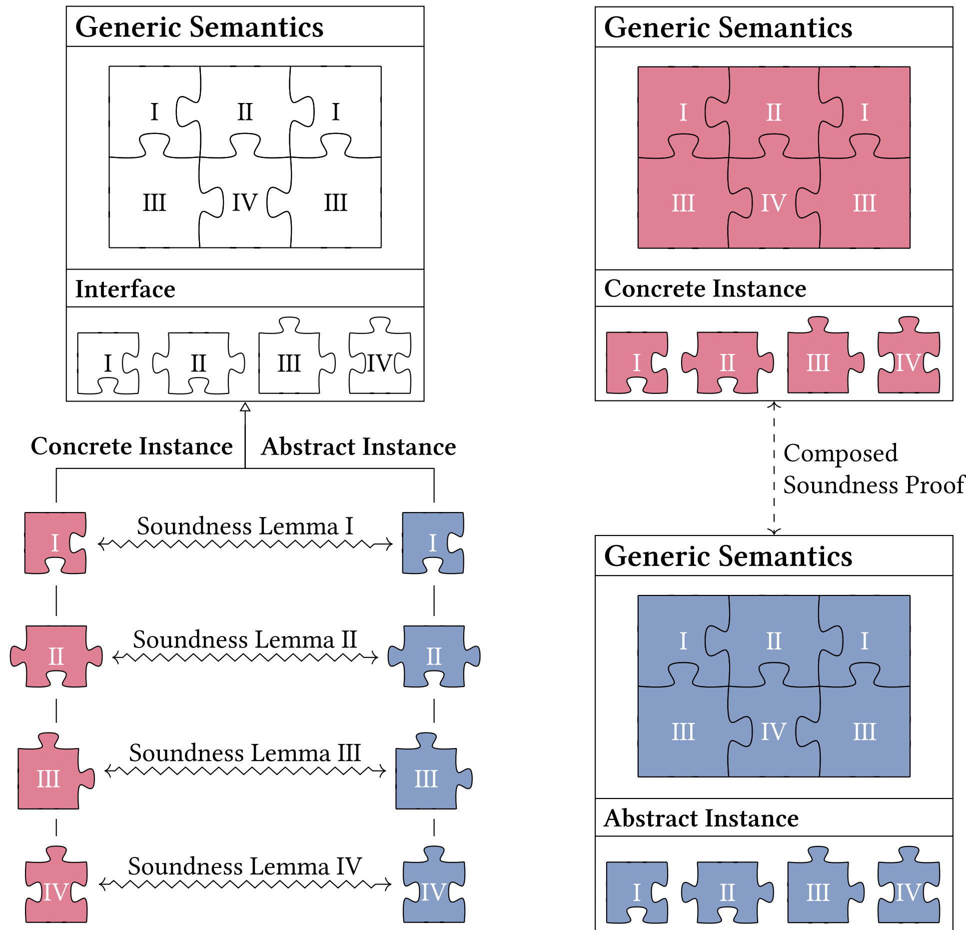


Figure 1.5: Depiction of a compositional soundness proof. In comparison to the monolithic proof (Figure 1.4), both the concrete and abstract semantics are derived from a *generic semantics* (left). The generic semantics consists of a combination of small operations depicted as puzzle pieces. Each of these operations has a specific type, which is described by an interface. Both the concrete and abstract semantics instantiate this interface in two different ways. For soundness, it suffices to prove smaller soundness lemmas for each operation (⇜⇝), each of which can be proven independently. The proof of the derived concrete and abstract semantics (right) then follows automatically by composition (⇠⇢).

1.4.2 Modularizing the analysis of effects

The second problem we address is the close coupling of effects in the analysis implementation. To solve this problem, we refine the design of the previous subsection (Figure 1.6). In comparison to Figure 1.5, we reorganized the operations into independent *analysis components*. Each analysis component captures a specific aspect of the analysis and consists of an interface for operations, a concrete and abstract instance, and a soundness lemma for each operation. For example, the store component captures the effect of variable mutation and the exception component captures the effect of exception handling. Each component consists of an interface, a concrete and abstract instance of its operations, and a soundness lemma for each operation. The components with straight lines are language-specific and cannot be easily reused. In contrast, the components with dashed lines are language-independent and reusable and can be provided as part of a library. Furthermore, the reusable components can be proven soundness from the library authors a priori, which reduces the effort of the soundness proof. Because components interact, they need to be composed with a specific composition operation (⊕). For soundness, it suffices to prove the soundness lemmas of the language-specific components (⇜⇝). The soundness lemmas of the language-independent components can be reused (⇠⇢). Lastly, soundness of the composed super

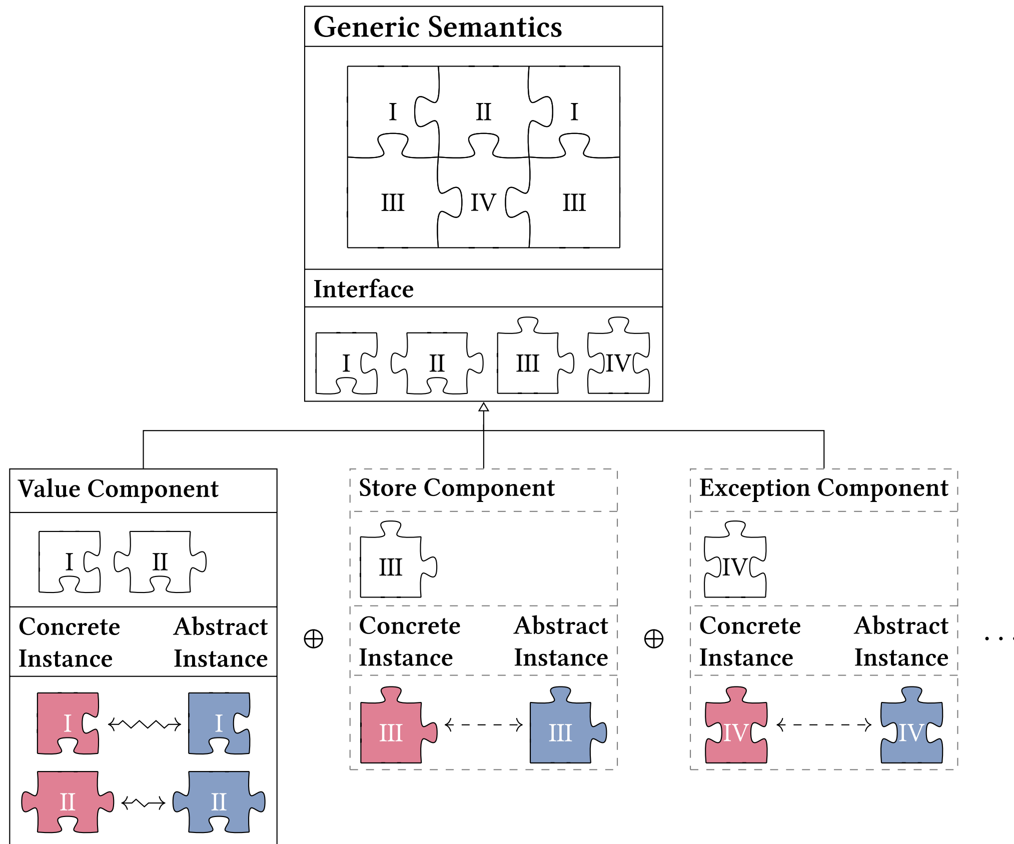


Figure 1.6: Depiction of a compositional soundness proof that modularizes the effects of the language.

component follows mostly without extra proof effort.

A challenge to make the soundness proofs compositional is the interaction between different effects, as we discussed in Section 1.3. We solve this problem by defining a specific composition operator for analysis components (⊕). If the interaction is non-trivial, as for variable mutation and exceptions, the composition requires an additional implementation that describes interaction. This implementation also needs to be proven sound, which adds extra proof work for these compositions. However, we show in our evaluation that most interactions are trivial. In these cases, the implementation of the composition and the soundness proof can be derived automatically.

This refinement has the following benefits:

- The reorganization reduces the effort of implementing new analyses. In particular, we avoid reimplement common analysis functionality by reusing existing analysis components.
- The soundness proof is less complicated, because each component only captures a small piece of functionality and can be proven sound independently.
- The soundness proof takes less effort, because it suffices to prove soundness of language-specific analysis components. In particular, the language-independent reusable components can been proven sound a priori. Furthermore, the composition of analysis components is sound, if each component is sound. We prove this with a theorem once and for all types of components.

We describe this technique in more detail in Chapter 4.

1.4.3 Modularizing fixpoint algorithms

Lastly, we make the soundness proof of fixpoint algorithms compositional by composing them from small and reusable combinators. These combinators separate different concerns of the fix-

point algorithm and have different responsibilities, such as:

- For example, some combinators detect loops and decide how many loop iterations are analyzed independently.
- Other combinators detect recursive function calls and decide how deep these function calls are analyzed.
- Other combinators pause the execution of the analysis to send debugging information to a graphical analysis debugger.
- Other combinators collect information about the analyzed program, such as its control-flow graph.

This design allows us to develop new fixpoint algorithms more easily by reusing existing language-independent combinators. Furthermore, the design allows us to fine-tune existing fixpoint algorithms more easily by adding, replacing or reordering combinators.

More specifically, the following function `fix` calculates the fixpoint of function f with combinators $\varphi_1 \ldots \varphi_n$:

$$\texttt{fix}(f) = \varphi_1(\varphi_2(\ldots \varphi_n(f(\texttt{fix}(f)))\ldots))$$

The first combinator φ_1 gets control first and may call or ignore the combinator φ_2. The second combinator φ_2 then may call combinator φ_3 and so on. This pattern continues until the last combinator φ_n calls the function f and the cycle repeats.

This reorganization has the following benefits:

- Fixpoint combinators reduce the effort of implementing new fixpoint algorithms. In particular, we avoid reimplementing common functionality by reusing existing combinators. Furthermore, we can adapt the fixpoint algorithm to a new language more easily by adding language-specific fixpoint combinators.
- The soundness proof is less complicated, because each fixpoint combinator captures only a small piece of functionality and can be proven sound independently.
- The soundness proof takes less effort, because it suffices to prove soundness of the language-specific fixpoint combinators. In particular, the language-independent reusable combinators have been proven sound a priori. Furthermore, a fixpoint algorithm composed of combinators is sound, if each combinator is sound. We prove this with a theorem once and for all modular fixpoint algorithms.

We describe this technique in more detail in Chapter 5.

1.4.4 Evaluation

The thesis of this dissertation is: *Compositional soundness proofs are feasible and reduce the effort and complexity of developing sound static analyses.* We support this statement with our evaluation. In particular, we show how our approach allows us to separate different analysis concerns by developing the *Sturdy library* with sound analysis components and fixpoint combinators. As of January 2021, the library contains 22 analysis components and 14 fixpoint combinators. The library is implemented in Haskell and is open-source.[1]

Furthermore, we demonstrate that the analysis components and fixpoint combinators of the Sturdy library are reusable by using them to develop several analyses for following languages:

- PCF [Plotkin 1977], a research language with higher-order functions and numbers,
- While, a research language with mutable state and *while* loops,
- Scheme [Abelson et al. 1998], a real-word language with dynamic typing, higher-order functions and mutable state,
- Stratego [Visser et al. 1998], a real-world domain-specific language for developing program transformations,

[1] https://github.com/svenkeidel/sturdy

For each of these languages we developed a generic semantics, that we reused to develop types of static analyses:

- Control-flow analyses for the higher-order languages PCF and Scheme,
- Data-flow analyses for PCF, Scheme, and the While language,
- Static type analyses for the dynamically-typed languages Scheme and Stratego.

These analyses appear as case studies throughout this dissertation (Chapters 3-5). The case studies have shown that our methodology reduces the development effort of static analyses, because of the reuse of generic semantics, analysis components, and fixpoint combinators. Furthermore, it reduces the effort and complexity of soundness proofs, because it constructs the soundness proofs compositionally.

In Chapter 6 we conducted a larger case study, by using our approach to develop 3 analyses for the domain-specific language Stratego. Stratego is difficult to analyze due to its unorthodox and dynamic language features, such as generic traversals. Despite these difficulties, we show how our approach allows us to systematically develop sound analyses.

1.4.5 Contributions

In summary, this thesis makes the following contributions:

- We present a methodology for structuring static analyses, that simplifies their development and simplifies their sound proof. The core principle of this methodology are compositional soundness proofs.
- We show how the similarities of the language semantics and the analysis can be captured with a *generic interpreter* (Chapter 3). We prove that the code of the generic interpreter can be disregarded in a soundness proof.
- We describe how the effects of a language can be captured with reusable analysis components (Chapter 4). We prove that an analysis is sound, as long as it is composed of sound analysis components.
- We describe how fixpoint algorithms can be implemented modularly by reusable fixpoint combinators (Chapter 5). We prove that a fixpoint algorithm is sound, as long as it consists of sound fixpoint combinators.
- We developed an open-source Haskell library with reusable analysis components and fixpoint combinators.
- We support our thesis by using our methodology to develop several analyses for different languages and report on the effort and complexity of the development and soundness proof.
- We conduct a larger case study by using our approach to develop three analyses for the domain-specific language Stratego (Chapter 6).

1.5 Dissertation Outline

In this section we outline the rest of this dissertation and indicate which chapters are based on peer-reviewed papers or drafts.

Chapter 2: Background on abstract interpretation and arrows.

Chapter 3: Capturing the Similarities between Concrete and Abstract Interpreters

This chapter is based on the following peer-reviewed article:

Compositional Soundness Proofs of Abstract Interpreters
Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg.
Proc. ACM Program. Lang. 2, ICFP (2018), 72:1–72:26.
https://doi.org/10.1145/3236767

Chapter 4: Sound and Reusable Components for Abstract Interpretation

This chapter is based on the following peer-reviewed article:

Sound and Reusable Components for Abstract Interpretation.
Sven Keidel and Sebastian Erdweg.
Proc. ACM Program. Lang. 3, OOPSLA (2019), 176:1–176:28.
https://doi.org/10.1145/3360602

Chapter 5: Modular Description and Soundness Proofs of Fixpoint Algorithms for Big-Step Abstract Interpreters

This chapter is based on the following draft:

Modular Description and Soundness Proofs of Fixpoint Algorithms for Big-Step Abstract Interpreters.
Sven Keidel, Tobias Hombücher, and Sebastian Erdweg.

Chapter 6: A Systematic Approach to Abstract Interpretation of Program Transformations

This chapter is based on the following peer-reviewed article:

A Systematic Approach to Abstract Interpretation of Program Transformations.
Sven Keidel and Sebastian Erdweg.
Lecture Notes in Computer Science, Springer, VMCAI (2020), 136–157.
https://doi.org/10.1007/978-3-030-39322-9_7

Chapter 7: Related Work

Chapter 8: Future and Ongoing Work

Chapter 9: Conclusion

BACKGROUND ON BIG-STEP ABSTRACT INTERPRETATION AND ARROWS 2

In this chapter, we introduce the necessary background for the style of static analyses that we use. Furthermore, we introduce background for the *arrow* abstraction that allows us to modularize the effects of the language.

2.1 Introduction to Big-Step Abstract Interpreters

We base our static analyses on *abstract interpretation* [Cousot and Cousot 1979]. Abstract interpretation is a methodology for rigorously proving soundness of static analyses. In this section, we provide the background on abstract interpretation necessary to understand the rest of the dissertation.

We illustrate abstract interpretation at the example of an interval analysis for a language with arithmetic and boolean expressions. Such an analysis is useful for detecting common bugs such as "divide-by-zero" errors or invalid array accesses, as it analyzes the range of numbers an expression may evaluate to. We implement the analysis in Haskell, a functional programming language. The top of Listing 2.1 defines the abstract syntax of our example language.

The function `eval` on the left of Listing 2.1 defines the concrete semantics of the language. Function `eval` is called a *big-step interpreter* [Pierce 2002], because it evaluates a program expression in one big step to a numeric or boolean value. To this end, the interpreter `eval` performs a case distinction on the top-level syntax construct. In each case, the interpreter first evaluates the subexpressions and then recombines the resulting values. For example, in case of addition, the interpreter first evaluates the subexpressions $\mathtt{e}_1$ and $\mathtt{e}_2$ to numeric values `i` and `j` and then uses the Haskell addition to combine the resulting number.

The function $\widehat{\mathtt{eval}}$ on the right of Listing 2.1 defines an interval analysis for the language. Function `eval` is called a big-step *abstract* interpreter [Darais et al. 2017], because it evaluates each program expression to an *abstract* value. In case of the interval analysis, abstract numeric values are intervals. The meaning of an interval value is that the program could evaluate to any of these numbers within the interval. Abstract boolean values can be one of three cases: $\widehat{\mathtt{True}}$ which represents the concrete boolean `True`, $\widehat{\mathtt{False}}$ which represents `False`, and $\widehat{\mathtt{Top}}$ which represents `True` or `False`.

The abstract interpreter $\widehat{\mathtt{eval}}$ works similar to the concrete interperter `eval`. It also performs a case distinction on the top-level syntax construct. In each case it first evaluates the subexpressions and then recombines the resulting values. For example, in case of addition, the abstract interpreter first evaluates the subexpressions $\mathtt{e}_1$ and $\mathtt{e}_2$ to the intervals $(\mathtt{i}_1,\mathtt{i}_2)$ and $(\mathtt{j}_1,\mathtt{j}_2)$ and then recombines them by adding the outer interval bounds.

To prove this analysis sound, we need to relate the results of the abstract interpreter $\widehat{\mathtt{eval}}$ to the results of the concrete interpreter `eval`. The relationship is described mathematically by a Galois connection [Cousot and Cousot 1977]. Each Galois connection consists of an abstraction function α that relates concrete to abstract values and a concretization function γ that relates abstract to concrete values. For example, for the numeric values the abstraction function returns the smallest interval containing a set of numbers, e.g., $\alpha(\{1,2,4\}) = [1,4]$. In contrast, the concretization function returns the set of numbers contained in the interval, $\gamma([1,5]) = \{1,2,3,4\}$. With such a Galois connection and the inclusion ordering on intervals ($\sqsubseteq$), we prove soundness by proving the following proposition:

$$\forall e \in \mathtt{Expr}.\, \alpha(\{\mathtt{eval}\ e\}) \sqsubseteq \widehat{\mathtt{eval}}\ e \qquad \text{or equivalently} \qquad \forall e \in \mathtt{Expr}.\, \{\mathtt{eval}\ e\} \subseteq \gamma(\widehat{\mathtt{eval}}\ e).$$

That is, the abstract interpreter is sound, if it overapproximates the results of the concrete interpreter, for all programs.[1]

[1] The soundness proposition is drastically simplified due to the simplicity of the analyzed language. For a more general definition see Section 3.2

```
-- Language
data Expr = NumLit Int | Add Expr Expr | BoolLit Bool | And Expr Expr
```

```
-- Concrete Interpreter
data Val = NumVal Int | BoolVal Bool

eval :: Expr → Val
eval e = case e of
  NumLit n → NumVal n
  Add e1 e2 →
    let NumVal i = eval e1
        NumVal j = eval e2
    in NumVal (i + j)
  BoolLit True  → BoolVal True
  BoolLit False → BoolVal False
  And e1 e2 →
    let BoolVal p = eval e1
        BoolVal q = eval e2
    in case (p,q) of
        (True, True) → BoolVal True
        (_   , _   ) → BoolVal False
```

```
-- Abstract Interpreter
data Val^ = NumVal^ Interval | BoolVal^ Bool^

eval^ :: Expr → Val^
eval^ e = case e of
  NumLit n → NumVal^ (n,n)
  Add e1 e2 →
    let NumVal^ (i1,i2) = eval^ e1
        NumVal^ (j1,j2) = eval^ e2
    in NumVal^ (i1 + j1, i2 + j2)
  BoolLit True  → BoolVal^ True^
  BoolLit False → BoolVal^ False^
  And e1 e2 →
    let BoolVal^ p = eval^ e1
        BoolVal^ q = eval^ e2
    in case (p,q) of
        (True^, True^) → BoolVal^ True^
        (⊤    , True^) → BoolVal^ Top^
        (True^, ⊤    ) → BoolVal^ Top^
        (_    , _    ) → BoolVal^ False^
```

Listing 2.1: Interval analysis for a small language with arithmetic and boolean expressions. We describe the analysis as a big-step abstract interpreter in Haskell. The interpreter eval on the left describes the semantics of the language we want to analyze. The interpreter $\widehat{\texttt{eval}}$ on the right describes the analysis. We use a "hat" symbol to distinguish abstract from concrete names.

In the following, we demonstrate how these proofs work by proving one case of the And construct. The proof is by structural induction over the expressions Expr of the language. In case $\widehat{\texttt{eval}}\ e_1 = \widehat{\texttt{False}}$ and $\widehat{\texttt{eval}}\ e_2 = \widehat{\texttt{Top}}$, we know by definition that $\widehat{\texttt{eval}}$ (And e_1 e_2) evaluates to $\widehat{\texttt{False}}$.[2] Furthermore, by the induction hypothesis for e_1, we know that $\{\texttt{eval}\ e_1\} \subseteq \{\texttt{False}\} = \gamma(\widehat{\texttt{False}}) = \gamma(\widehat{\texttt{eval}}\ e_1)$. Similarly, by the induction hypothesis for e_2, we know that $\{\texttt{eval}\ e_2\} \subseteq \{\texttt{True}, \texttt{False}\} = \gamma(\widehat{\texttt{Top}}) = \gamma(\widehat{\texttt{eval}}\ e_2)$. In case eval e_2 = False, then e evaluates to False. Furthermore, in case eval e_2 = True, then e also evaluates to False. Thus we conclude

$$\{\texttt{eval}\ (\texttt{And}\ e_1\ e_2)\} \subseteq \{\texttt{False}\} \subseteq \gamma(\widehat{\texttt{False}}) = \gamma(\widehat{\texttt{eval}}\ (\texttt{And}\ e_1\ e_2)).$$

Already for this tiny example the soundness proof was quite involved: We needed to reason about two concrete executions (eval e_2 = True and eval e_2 = False) and relate them to the result of an abstract executions. Proving all other cases of the And construct requires a considerable amount of effort. The example above demonstrates that soundness proofs of small analyses already may require a lot of effort.

Not only are soundness proofs a lot of effort, they are also quite intricate. In particular, it is easy to overlook mistakes in a soundness proof, which invalidates the entire proof. For example, consider the analysis of addition in Listing 2.1. Even though the analysis of addition looks sound on first sight, it is in fact *unsound*. For instance, consider the program max_int + e, where max_int is the largest possible 64-bit integer and the expression e evaluates either to 0 or 1. The concrete interpreter evaluates max_int + 0 to max_int, but overflows for $e = 1$: max_int + 1 = min_int. The abstract interpreter on the other hand evaluates the program to [max_int + 0, max_int + 1] = [max_int, min_int]. Since the right interval bound is greater than the left, the concretization function maps γ([max_int, min_int]) to $\emptyset$. However, the empty set does not overapproximate the concrete results {min_int, max_int} and hence the abstract interpreter is unsound. This example shows how subtle soundness bugs can be and how easily edge cases can be overlooked.

[2]For the sake of readability, we omit writing BoolVal.

```
run :: InterpreterArrow Stmt ()
run = proc stmt → case stmt of
  Sequence s₁ s₂ → do
    run ≺ s₁
    run ≺ s₂
  Assignment x e → do
    v ← eval ≺ e
    write ≺ (x,v)
  Throw e → do
    v ← eval ≺ e
    throw ≺ v
  ...
```

```
eval :: InterpreterArrow Expr Val
eval = ...

data Expr = ...
data Stmt = Sequence Stmt Stmt
          | Assignment Variable Expr
          | Throw Expr
          | ...
```

Listing 2.2: Interpreter for a statement-based languages that uses an arrow to abstract over the effects of the language.

To summarize, abstract interpretation is a methodology to define static analyses and rigorously prove them sound. However, with the conventional approach to abstract interpretation, soundness proofs are a lot of effort and cases of unsoundness can be easily overlooked.

2.2 Arrows, an Abstraction for Effectful Computations

In this dissertation, we use *Arrows* [Hughes 2000] to implement static analyses. In this section, we explain what arrows are and what properties they have that make them useful for our work.

2.2.1 Encoding the effects of programming languages with Arrows

Arrows are an abstraction over effectful computations. In this work, we are interested we use arrows to model the effects of programming languages and analyses with arrows. For example, consider a language with mutable variables, exceptions, and a stack of local variables. We could model these effects with the following arrow:

```
type InterpreterArrow x y = (Stack, Memory, x) → (Memory, Either Exception y)
```

The arrow `InterpreterArrow` is a function that takes a stack, a memory, and an additional argument of type `x` and returns a changed memory and either an exception or a value of type `y`. This arrow has the benefit, that we do not need to pass around the stack and memory in the interpreter implementation. Furthermore, the arrow hides the handling of exceptions from the interpreter code.

To develop an interpreter for this language (Listing 2.2), we use the pretty notation for arrows [Paterson 2001]. The `proc` keyword introduces a new arrow computation. The syntax `y ← f ≺ x` passes the input `x` to the computation `f` and binds the result to y. The syntax `f ≺ x` passes the input `x` to the computation `f`, but ignores the result. A sequence of arrow statements `y ← f ≺ x; v ← g ≺ u` first executes the computation `f` and then executes the computation `g`.

The arrow pretty notation desugars to the operators of the arrow type classes `Category`, `Arrow` and `ArrowChoice` (Listing 2.3).[3] The arrow type classes use a type variable `c` for the type of the arrow computation. This type variable `c` takes to type arguments, the input and output types of the computation. Furthermore, the arrow type classes contain the following operations:

- An identity operation `id`, that simply returns its argument to the output of the computation.
- A composition operation `f ∘ g`, that passes the output of the second computation `g` to the input of the first computation `f`.
- An operation `arr f`, that embeds a pure function `f` in an effectful arrow computation.

[3] The original definition of the arrow type classes is available at https://hackage.haskell.org/package/base/docs/Control-Arrow.html

```
class Category c where
  id :: c x x
  (∘) :: c y z → c x y → c x z

returnA = id
f ⋙ g = g ∘ f

class Category c ⇒ Arrow c where
  arr :: (x → y) → c x y
  (***) :: c x y → c u v → c (x,u) (y,v)
  (&&&) :: c x y → c x z → c x (y,z)

class ArrowChoice c where
  (+++) :: c x y →
           c u v →
           c (Either x u) (Either y v)
  (|||) :: c x z →
           c y z →
           c (Either x y) z

data Either a b = Left a | Right b
```

```
instance Category InterpreterArrow where
  id = λ(stack,mem,x) → (mem,Right x)
  f ∘ g = λ(stack,mem,x) →
    case g (stack,mem,x) of
      (mem',Right y) → f (stack,mem',Right y)
      (mem',Left exc) → (mem',Left exc)

instance Arrow InterpeterArrow where
  arr f = λ(stack,mem,x) → (mem,Right(f x))
  f *** g = λ(stack,mem,(x,y)) →
    case f (stack,mem,x) of
      (mem',Right x') →
        case g (stack,mem',y) of
          (mem'',Right y') → (mem'',Right(x',y'))
          (mem'',Left exc) → (mem'',Left exc)
      (mem',Left exc) → (mem',Left exc)

instance ArrowChoice InterpeterArrow where
  f ||| g = λ(stack,mem,e) →
    case e of
      Left x → f (stack,mem,Right x)
      Right y → g (stack,mem,Right y)
```

Listing 2.3: Arrow typeclasses (left) and excerpt of arrow instances for interpreter arrow (right).

```
class ArrowMemory var val c where
  read  :: c var val
  write :: c (var,val) ()

class ArrowExcept err c where
  throw :: c err ()
  catch :: c x y →
           c (err,x) y →
           c x y
```

```
instance ArrowMemory var val InterpreterArrow where
  read=λ(_,mem,var)→(mem,Right (lookup var mem))
  write = λ(_,mem,(var,val)) →
                (insert var val mem, Right ())

instance ArrowExcept Exception InterpreterArrow
  throw = λ(stack,mem,exc) → (mem, Left exc)
  catch f g = λ(stack,mem,x) →
    case f (stack,mem,x) of
      (mem',Right y) → (mem',Right y)
      (mem',Left exc) → g (stack,mem',(exc,y))
```

Listing 2.4: User-defined arrow operations that interact with the effects of the arrow computation.

- Two operations `f *** g` and `f &&& g`, that execute both computations `f` and `g` and collect their results into a tuple.
- Two operations `f +++ g` and `f ||| g`, that execute either the computation `f` or `g` if the input is `Left` or `Right`.

The arrow instance for our `InterpreterArrow` in Listing 2.3 is straight-forward. The instance passes the stack unchanged to the computations `f` and `g`, it threads the memory through the computations, and it pattern matches on the `Either` type in the output of `f` and `g`.

The arrow operations `write` and `throw`, are user-defined and allow interacting with effects of the arrow computation explicitly. The `write` operation updates the memory and the `throw` operation causes an exception. They are defined in custom type classes shown in Listing 2.4.

To summarize, arrows are an abstraction over effectful computations. They hide details about the effects of the computation, which makes the code easier to understand and reason about. Arrow computations are commonly described with a pretty notation that desugars to the operations of the arrow type classes.

```
type StateT s c x y = c (s,x) (s,y)

instance Arrow c ⇒ Category (StateT s c) where
  id = proc (s,x) → returnA ≺ (s,x)
  f ∘ g = proc (s,x) → do
    (s',y) ← g ≺ (s,x)
    f ≺ (s',y)

instance Arrow c ⇒ Arrow (StateT s c) where
  arr f = arr (λ(s,x) → (s,f x))
  f *** g = proc (s,(x,y)) → do
    (s',x')  ← f ≺ (s,x)
    (s'',y') ← g ≺ (s',y)
    returnA ≺ (s'', (x',y'))

instance ArrowChoice c ⇒ ArrowChoice (StateT s c) where
  f ||| g = proc (s,e) → do
    case e of
      Left x → f (s,x)
      Right y → g (s,y)
```

Listing 2.5: Arrow Transformer that adds state to an existing arrow computation.

2.2.2 Modular Effects with Arrow Transformers

In the previous section, we discussed how we can model the effects of an example language with an arrow `InterpeterArrow`. However, the arrow `InterpreterArrow` mixes the implementation of three different types of effects. This makes it difficult to reuse any of the code for another language. A solution to this problem is to construct arrows from reusable arrow transformers, which we discuss in this section.

An arrow transformer is a type that adds an effect to an existing arrow computation. For example, the state arrow transformer [Hughes 2000] in Listing 2.5 adds a piece of mutable state `s` to the input and output of an arrow computation `c`.[4] The important bit that makes this arrow transformer reusable is that it is parametric in the underlying arrow `c`. This allows us to apply this arrow transformer to any existing arrow that implements the arrow type classes. To this end, the `StateT` implements an arrow instance of the form `Arrow c ⇒ Arrow (StateT s c)`. This means, the type `StateT s c` implements an arrow instance given an arrow instance for the underlying arrow `c`. These instances are called *liftings*, because they lift the operations of the underlying arrow `c` through the arrow transformer.

With the `StateT` arrow transformer and two further transformers, we can recreate the interpreter arrow of Section 2.2.1:

```
type StateT s c x y = c (s,x) (s,y)
type ReaderT r c x y = c (r,x) y
type ExceptT err c x y = c x (Either err y)
type InterpreterArrow' x y = ExceptT Exception (ReaderT Stack (StateT Memory (→))) x y
```

We can verify that this is the case, by unfolding the type definitions of the arrow transformers from the outside in:

```
InterpreterArrow' x y
  = ExceptT Exception (ReaderT Stack (StateT Memory (→))) x y
  = ReaderT Stack (StateT Memory (→)) x (Either Exception y)
  = StateT Memory (→) (Stack,x) (Either Exception y)
  = (→) (Memory,(Stack,x)) (Memory,(Either Exception y))
  = (Memory,(Stack,x)) → (Memory,(Either Exception y))
  ≅ (Stack,Memory,x) → (Memory,Either Exception y)
  = InterpreterArrow x y
```

[4] We use the suffix “T” to distinguish arrow transformer from regular arrows.

In comparison to the original interpreter arrow of Section 2.2.1, the new type `InterpreterArrow'` is more extensible. In particular, we can add new effects of the interpreter arrow simply by adding new arrow transformer. Furthermore, we can change the semantics of arrow by reordering the arrow transformers. For example, moving the `ExceptT` transformer to the bottom of the arrow transformer stack implements a language semantics, in which exceptions reset the memory:

```
ReaderT Stack (StateT Memory (ExceptT Exception (→))) x y
  = (Memory,(Stack,x)) → Either Exception (Memory,y)
```

This arrow type resets the memory whenever an exception occurs, because the `Either` type, in the result of the function, wraps the memory.

To summarize, an arrow transformer captures a specific effect of an arrow computation. They can be used to compose complex effectful computations. This has the benefit, that we can easily change existing arrow computations by adding or rearranging existing arrow transformers.

2.2.3 Algebraic Properties of Arrows

We chose arrows as an abstraction for effects in our work, because of their algebraic properties. More specifically, arrows form an *algebra*. For example, the interpreter `run` in Listing 2.2 uses an arrow algebra, that can be described with the following BNF grammar:

```
E ::= id | E ∘ E | arr F | E &&& E | E ||| E | read | write | throw | catch E E
```

In particular, the algebra consists of the general-purpose arrow operations of the `Category`, `Arrow` and `ArrowChoice` type classes and the language-specific operations for accessing the memory and handling exceptions.

The main benefit of such an arrow algebra is that it gives us a useful reasoning principle: structural induction. For the arrow algebra above, this induction principle is defined as follows:

$$\frac{\begin{array}{c} P(\texttt{id}) \qquad P(\texttt{arr}\ f) \\ P(f) \wedge P(g) \implies P(f \circ g) \\ P(f) \wedge P(f) \implies P(f *** g) \\ P(f) \wedge P(g) \implies P(f\,|||\,g) \\ P(\texttt{read}) \qquad P(\texttt{write}) \qquad P(\texttt{throw}) \\ P(f) \wedge P(g) \implies P(\texttt{catch}\ f\ g) \end{array}}{\forall f : \texttt{E}.\ P(f)}$$

The induction principle lets us prove a property P about *all* arrow computations. More specifically, the premises above the bar are obligations we have to prove. The conclusion of the induction principle below the bar then guarantees that any arrow computation $f : E$ satisfies the property P. We use this induction principle in Chapter 3 to compose a soundness proof for an arrow-based abstract interpreter.

Another benefit of the algebra of arrows is that they allow abstract reasoning about effectful computation. In particular, arrows satisfy a number of algebraic laws [Hughes 2000] shown in Figure 2.1 These laws allow us to prove properties about arrow computations without needing to know the arrow type. These proofs are easier because they hide details of the effectful computations. Furthermore, they are more reusable, because they hold for all underlying arrows.

To summarize, arrows have algebraic properties that aid us to reason about arrow computations. The most important algebraic property for our work is the induction principle, which lets us reason about all arrow computations. The other algebraic property are the laws that arrows satisfy. These laws let us prove properties for arrow computations of an arbitrary type.

$$
\begin{aligned}
\texttt{arr id} &= \texttt{id} \\
\texttt{arr}\ (f \ggg g) &= \texttt{arr}\ f \ggg \texttt{arr}\ g \\
\texttt{first}\ (\texttt{arr}\ f) &= \texttt{arr}\ (\texttt{first}\ f) \\
\texttt{first}\ (f \ggg g) &= \texttt{first}\ f \ggg \texttt{first}\ g \\
\texttt{first}\ f \ggg \texttt{arr fst} &= \texttt{arr fst} \ggg f \\
\texttt{first}\ f \ggg \texttt{arr}\ (\texttt{id} *\!*\!* g) &= \texttt{arr}\ (\texttt{id} *\!*\!* g) \ggg \texttt{first}\ f \\
\texttt{first}\ (\texttt{first}\ f) \ggg \texttt{arr}\ \texttt{assoc}_\times &= \texttt{arr}\ \texttt{assoc}_\times \ggg \texttt{first}\ f \\
\texttt{left}\ (\texttt{arr}\ f) &= \texttt{arr}\ (\texttt{left}\ f) \\
\texttt{left}(f \ggg g) &= \texttt{left}\ f \ggg \texttt{left}\ g \\
f \ggg \texttt{arr Left} &= \texttt{arr Left} \ggg \texttt{left}\ f \\
\texttt{left}\ f \ggg \texttt{arr}\ (\texttt{id} +\!\!+\!\!+ g) &= \texttt{arr}\ (\texttt{id} +\!\!+\!\!+ g) \ggg \texttt{left}\ f \\
\texttt{left}\ (\texttt{left}\ f) \ggg \texttt{arr}\ \texttt{assoc}_+ &= \texttt{arr}\ \texttt{assoc}_+ \ggg \texttt{left}\ f
\end{aligned}
$$

where

$$
\texttt{assoc}_\times\ (a, (b, c)) = ((a, b), c) \qquad \texttt{assoc}_+\ e = \begin{cases} \texttt{Left}\ x & e = \texttt{Left}\ (\texttt{Left}\ x) \\ \texttt{Right}\ (\texttt{Left}\ y) & e = \texttt{Left}\ (\texttt{Right}\ y) \\ \texttt{Right}\ (\texttt{Right}\ z) & e = \texttt{Right}\ z \end{cases}
$$

Figure 2.1: Algebraic laws that arrows satisfy.

CAPTURING THE SIMILARITIES BETWEEN CONCRETE AND ABSTRACT INTERPRETERS 3

This chapter is based on the following peer-reviewed paper:

Compositional Soundness Proofs of Abstract Interpreters
Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg.
Proc. ACM Program. Lang. 2, ICFP (2018), 72:1–72:26.
https://doi.org/10.1145/3236767

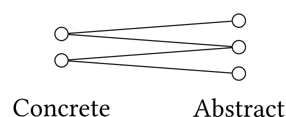
Abstract — Abstract interpretation is a technique for developing static analyses. Yet, proving abstract interpreters sound is challenging for interesting analyses, because of the high *proof complexity* and *proof effort*. To reduce complexity and effort, we propose a framework for abstract interpreters that makes their soundness proof compositional. Key to our approach is to capture the similarities between concrete and abstract interpreters in a single shared interpreter, parameterized over an arrow-based interface. In our framework, a soundness proof is reduced to proving reusable soundness lemmas over the concrete and abstract instances of this interface; the soundness of the overall interpreters follows from a generic theorem.

To further reduce proof effort, we explore the relationship between soundness and parametricity. Parametricity not only provides us with useful guidelines for how to design non-leaky interfaces for shared interpreters, but also provides us soundness of shared pure functions as *free theorems*. We implemented our framework in Haskell and developed a k-CFA analysis for PCF and a tree-shape analysis for Stratego. We were able to prove both analyses sound compositionally with manageable complexity and effort, compared to a conventional soundness proof.

3.1 Introduction

Abstract interpretation [Cousot and Cousot 2002] is an approach to static analysis with soundness at its heart: An abstract interpreter must approximate the behavior of a program as prescribed by a concrete interpreter. This soundness proposition can guide the design of abstract interpreters [Cousot 1999] and prescribes what needs to be proven about the analysis. Unfortunately, it is far less clear *how* to prove an abstract interpreter sound and, in particular, how to decompose the soundness proof into proof obligations of manageable size. Yet, compositional soundness proofs are crucial when developing verified abstract interpreters for real-world languages to reduce *proof complexity* and *proof effort*.

What makes the decomposition of the soundness proof difficult is that concrete and abstract interpreters are often misaligned, such that a case of one interpreter relates to multiple cases of the other interpreter (see figure). For example, a language construct `IfZero` that checks if a given number is zero has two outcomes in the concrete interpreter (is zero, is not zero) but three outcomes in an interval analysis (is zero, contains zero, does not contain zero). Such misalignment between concrete and abstract interpreter prevents a piece-wise decomposition of the soundness proof. Conversely, when concrete and abstract interpreter functions share the same structure we could decompose the proof along that structure.



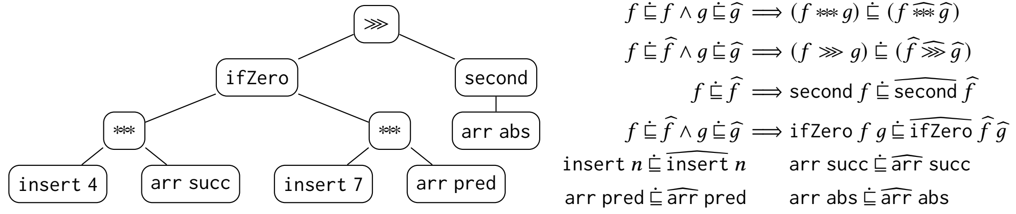
We present a novel framework for defining abstract interpreters such that their soundness proofs become compositional. Our key contributions are that (i) we can abstract from the difference between concrete and abstract interpreters such that (ii) the soundness proof for the shared parts is fully compositional and (iii) follows automatically from the soundness of the non-shared parts. Indeed, most concrete and abstract interpreter are very similar and only differ in a few places where the interpreters operate on the concrete or abstract domain (e.g., addition of numbers vs intervals). We propose to make these similarities explicit in a *generic interpreter* function, abstracting from the interpretations of primitive operations on the respective domain. We realize

this abstraction using Haskell's arrows [Hughes 2000], a generalization of monads. Instantiating the generic interpreter with arrow instances for the concrete and abstract domain fixes the respective language semantics. For an abstract interpreter factorized in this way, we obtain the following benefits when proving soundness:

1. We can decompose the soundness proof into soundness lemmas about the operations of the concrete and abstract arrow instances. Each soundness lemma is context free, i.e., independent from where the operation is used in the generic interpreter. This narrows the scope of the lemmas and makes them reusable.

2. Arrows restrict the meta-language of generic interpreters, which solely consists of arrow expressions. Because arrows are a first-order language, we can use structural induction over arrow expressions to obtain a generic soundness proof for *any* generic interpreter composed of sound arrow operations.

For example, consider the following abstract syntax tree of a shared arrow expression. On the right, we list the soundness lemmas required to prove concrete and abstract instances of the shared expression sound. We write $e \mathrel{\dot\sqsubseteq} \widehat{e}$ to mean that e is soundly approximated by $\widehat{e}$:



Functions $\ggg$, $***$, and `second` are language-independent arrow operations, function `arr` is a language-independent operation that embeds pure functions into arrow computation, and `ifZero` and `insert` are language-specific operations. The concrete and abstract arrow instances define implementations for all arrow operations; we denote abstract implementations with a *hat* $\widehat{***}$ to distinguish them from concrete definitions $***$. With that, we formulate a context-free soundness lemma for each arrow operation. For example, the lemma of $***$ is context-free in that it proves soundness of the operation *for all* sound subexpressions $f, \widehat{f}$ and $g, \widehat{g}$. This allows us to reuse the same lemma for every occurrence of $***$ in the shared expression. Soundness of the shared expression now follows by structural induction on arrow expressions: Given all leaves are sound and all intermediate nodes preserve soundness, the composed expression is sound. This way, we have effectively decomposed the soundness proof into smaller lemmas that can be proved independently and that can be composed to reason about full abstract interpreters. We assert this result as a generic meta-theorem, stating that any arrow expression is sound if the arrow operations it uses are sound.

We also show that in meta-languages with *parametricity* [Reynolds 1983], the soundness of shared code follows as a *free theorem* [Wadler 1989], given the interface does not leak details of the abstract interpreter into shared code. Based on this observation, we extract guidelines for the interface design to be used in the generic interpreter. In particular, following our guidelines, we get soundness of pure functions embedded with `arr` for free, which reduces the number of lemmas required for our example from 8 to 5. Lastly, parametricity allows us to generalize our framework to abstract interpreters that share code over interfaces other than arrows.

To evaluate our approach, we implemented a k-CFA analysis for PCF and developed a tree-shape analysis for *Stratego* [Visser et al. 1998], a dynamic language for program transformations used in practice and featuring dynamic scoping of pattern-bound variables, higher-order functions, and generic tree traversals. For both analyses, we extract a generic interpreter and prove it sound compositionally, thus demonstrating the applicability of our approach. We show that, for the k-CFA analysis, the soundness proof can be decomposed into 16 independently provable lemmas and for the tree-shape analysis into 27 lemmas. We reflect on our soundness proofs and explain why it has a reduced complexity and effort compared to conventional soundness proofs.

In summary, we make the following contributions:

- We describe a new approach for organizing abstract interpreters by sharing code with the concrete interpreter over an interface based on arrows.
- We show that the soundness proof of such abstract interpreters can be conducted compositionally, based on soundness lemmas of the arrow operations.
- We prove a generic meta-theorem showing that *any* generic interpreter is sound if it is composed of sound arrow operations. Thus, the soundness proofs of our abstract interpreters are not only compositional, but proofs about the shared parts actually follow for free.
- We apply parametricity to develop guidelines for the interface design, to obtain soundness of embedded pure functions for free, and to generalize our approach to interfaces other than arrows.
- We demonstrate the applicability of our approach through two case studies and show that our approach reduces the effort and complexity of soundness proofs.

3.2 Why and How to Make Soundness Proofs Compositional

In this section, we first discuss the *complexity* and *effort* of soundness proofs of conventional abstract interpreters. Then, we describe informally how we can make soundness proofs compositional and how this reduces proof complexity and effort.

3.2.1 Conventional Abstract Interpreters

To illustrate the difficulties of soundness proofs of conventional abstract interpreters, we construct an abstract interpreter for a small example language in Haskell. Expressions in our example language are either variables, integer literals, additions, or conditionals:

```
data Expr = Var String | Lit Int | Add Expr Expr
          | IfZero Expr Expr Expr
```

We would like to implement an abstract interpreter for this language that predicts the numbers a program evaluates to as an interval. For example, consider the following program:

```
IfZero (Var "x") (Lit 2) (Lit 5)
```

This program evaluates to `2` if `x` is bound to `0` and to `5` otherwise. In order to be sound, the abstract interpreter must approximate all possible results of this program. That is, if the interval for `x` may contain `0`, the most precise approximation of this program in the domain of intervals is $[2, 5]$.

We define a conventional concrete interpreter `eval` and a conventional abstract interpreter $\widehat{\texttt{eval}}$ for this language in Listing 3.1. The definition of the concrete interpreter is standard, hence, we only explain how the abstract interpreter differs. In case of an addition, the abstract interpreter adds the interval bounds. In case of `IfZero`, as described in the introduction, the abstract interpreter distinguishes three cases for the interval resulting from evaluating $\texttt{e}_1$: the interval contains zero only, does not contain zero, or contains zero and other values. If the interval contains zero only, we evaluate $\texttt{e}_2$; if the interval does not contain zero, we evaluate $\texttt{e}_3$. But if the interval contains zero and other values, we evaluate both $\texttt{e}_2$ and $\texttt{e}_3$ and join their results using the least upper bound operation $\sqcup$.

The abstract interpreter appears to correctly approximate the concrete interpreter's behavior. But what exactly do we have to prove to verify the soundness of $\widehat{\texttt{eval}}$? We prove the following soundness proposition for the *collecting semantics* [Cousot 1999] of `eval`:

$$\forall\, e \in \mathsf{Expr}.\, \forall\, X \subseteq \mathsf{Env}.\quad \alpha_V(\{\texttt{eval}\; e\; \rho \mid \rho \in X\}) \sqsubseteq \widehat{\texttt{eval}}\; e\; \alpha_E(X)$$

Here, α_V and α_E are abstraction functions of Galois connections [Cousot and Cousot 1979] for values and environments of the interpreters:

$$\alpha_V : \mathcal{P}(\texttt{Val}) \leftrightarrows \widehat{\texttt{Val}} : \gamma_V \qquad \alpha_E : \mathcal{P}(\texttt{Env}) \leftrightarrows \widehat{\texttt{Env}} : \gamma_E$$

$$\alpha_V(X) = (\min X, \max X) \qquad \alpha_E(X) = \bigsqcup_{\rho \in X} [\, x \mapsto \alpha_V(\rho(x)) \mid x \in \mathrm{dom}(\rho)]$$

```
type Val = Int
type Env = Map String Val

eval :: Expr → Env → Maybe Val
eval e env = case e of
  Var x → lookup x env
  Lit n → return n
  Add e1 e2 → do
    v1 ← eval e1 env
    v2 ← eval e2 env
    return (v1 + v2)
  IfZero e1 e2 e3 → do
    v ← eval e1 env
    if v == 0
    then eval e2 env
    else eval e3 env
```

```
type Val^ = (Int,Int)
type Env^ = Map String Val^

eval^ :: Expr → Env^ → Maybe^ Val^
eval^ e env = case e of
  Var x → lookup^ x env
  Lit n → return (n,n)
  Add e1 e2 → do
    (i1,j1) ← eval^ e1 env
    (i2,j2) ← eval^ e2 env
    return (i1+i2, j1+j2)
  IfZero e1 e2 e3 → do
    (i1,j1) ← eval^ e1 env
    if i1 == 0 && j1 == 0
    then eval^ e2 env
    else if j1 < 0 || 0 < i1
      then eval^ e3 env
      else eval^ e2 env ⊔ eval^ e3 env
```

Listing 3.1: Conventional design of a concrete (left) and abstract interpreter (right) for our example language.

The soundness proposition quantifies over sets of environments X, which represent properties of the program's free variables. For example, $X = \{\rho \mid \rho \in \mathsf{Env} \land \forall\ (x \mapsto v) \in \rho.\ \mathsf{even}(v)\}$ describes environments that map variables to even numbers. The soundness proposition states that, for any e, all concrete evaluations of e under environments ρ satisfying X must be predicted by a single abstract evaluation of e under the single abstract environment $\alpha_E(X)$ representing property X.

To prove this soundness proposition for our example, we proceed by structural induction over the expressions of our language. The soundness proof for `Var`, `Lit` and `Add` is easy, because the interpreters align and we only need to reason about the Galois connection α_V. The case `IfZero` $e_1\ e_2\ e_3$ is slightly more involved: We perform a case distinction on the result of $\widehat{\texttt{eval}}\ e_1\ \alpha_E(X)$, because the result prescribes which branch of `IfZero` will be analyzed.

- In case $\widehat{\texttt{eval}}\ e_1\ \alpha_E(X) = \texttt{Just}\ (0,0)$, the first branch e_2 will be analyzed. From the induction hypothesis for e_1, we learn that $\alpha_V\{\ \texttt{eval}\ e_1\ \rho \mid \rho \in X\} \sqsubseteq \widehat{\texttt{eval}}\ e_1\ \alpha_E(X) = \texttt{Just}\ (0,0)$. Since $\gamma_V(\texttt{Just}\ (0,0)) = \{0\}$, the concrete interpretation `eval` e_1 must also result in 0 and the concrete interpreter evaluates the the first branch e_2. This lets us conclude:

$$\begin{aligned}&\alpha(\{\ \texttt{eval}\ (\texttt{IfZero}\ e_1\ e_2\ e_3)\ \rho \mid \rho \in X\}) \sqsubseteq \alpha(\{\ \texttt{eval}\ e_2\ \rho \mid \rho \in X\})\\ &\quad\sqsubseteq \widehat{\texttt{eval}}\ e_2\ \alpha_E(X) \sqsubseteq \widehat{\texttt{eval}}\ (\texttt{IfZero}\ e_1\ e_2\ e_3)\ \alpha_E(X).\end{aligned}$$

- The case for intervals without 0 is analogous to the previous case.
- The last case is more involved because $\widehat{\texttt{eval}}\ e_1\ \alpha_E(X)$ contains zero and other numbers. In this case, we have to reason about multiple outcomes of behavior of the concrete interpreter. Independent of the result of e_1, the concrete interpreter will either evaluate the first or second branch of `IfZero` and hence:

$$\{\texttt{eval}\ (\texttt{IfZero}\ e_1\ e_2\ e_3)\ \rho \mid \rho \in X\}) \subseteq \{\texttt{eval}\ e_2\ \rho \mid \rho \in X\} \cup \{\texttt{eval}\ e_3\ \rho \mid \rho \in X\}$$

  This lets us conclude:

$$\begin{aligned}&\alpha_V(\{\texttt{eval}\ (\texttt{IfZero}\ e_1\ e_2\ e_3)\ \rho \mid \rho \in X\})\\ &\quad\sqsubseteq \alpha_V(\{\texttt{eval}\ e_2\ \rho \mid \rho \in X\} \cup \{\texttt{eval}\ e_3\ \rho \mid \rho \in X\})\\ &\quad\sqsubseteq \alpha_V(\{\texttt{eval}\ e_2\ \rho \mid \rho \in X\}) \sqcup \alpha_V(\{\texttt{eval}\ e_3\ \rho \mid \rho \in X\})\\ &\quad\sqsubseteq \widehat{\texttt{eval}}\ e_2\ \alpha_E(X) \sqcup \widehat{\texttt{eval}}\ e_3\ \alpha_E(X) = \widehat{\texttt{eval}}\ (\texttt{IfZero}\ e_1\ e_2\ e_3)\ \alpha_E(X).\end{aligned}$$

With this, we have proved soundness for a very simple static analysis of a very simple programming language. And already the proof was not trivial: For every case in the abstract interpreter,

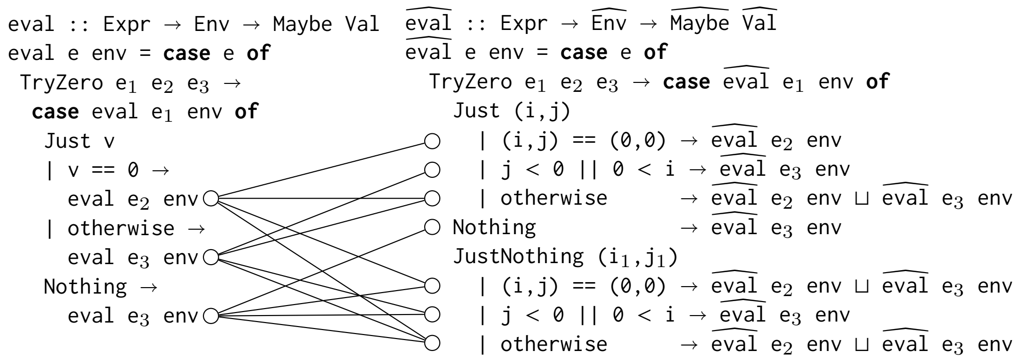


Figure 3.1: Concrete interpreter (left) and abstract interpreter (right) for `TryZero` and how their cases relate to the.

we had to establish which cases of the concrete interpreter are relevant and then establish that the abstract interpreter subsumes them all. The complexity and effort of such proofs quickly grows as language features become more complex. For example, consider another language construct `TryZero` e_1 e_2 e_3 in Figure 3.1 whose concrete semantics is like `IfZero` e_1 e_2 e_3 except the evaluation defaults to e_3 if the evaluation of e_1 fails: When defining an abstract interpreter for `TryZero`, we need to be careful about how we handle failed executions. In particular, we often do not know whether a computation definitely succeeds or fails. To be precise, we use type `Maybe` to represent potential failure (`JustNothing`) alongside definite success (`Just`) and definite failure (`Nothing`). Based on this type, we can implement `TryZero` in the abstract interpreter as shown in Figure 3.1.

In the soundness proof for `TryZero`, we have to relate 7 cases of the abstract interpreter to 3 cases of the concrete interpreter as indicated by the diagram on the right. Compared to `IfZero`, the soundness proof for `TryZero` is worse in two ways:

- We have to relate a single case of the abstract interpreter to up to 3 cases of the concrete interpreter at once. The more cases we need to relate, the higher the *proof complexity*.
- We have to prove 7 cases of the abstract interpreter sound. The more cases we need to prove, the higher the *proof effort*.

These problems are already apparent in the soundness proof for our example language. For precise abstract interpreters of real-world languages, proof complexity and proof effort quickly make a soundness proof infeasible. However, it is exactly for analyses of such languages that we need soundness proofs to ensure all corner cases are covered. Therefore, the question this paper aims to answer is: How can we make soundness proofs of abstract interpreters simpler and more systematic, such that soundness proofs of abstract interpreters for real-world languages become feasible?

3.2.2 Concrete and Abstract Interpreters using Arrows

This paper presents techniques that make soundness proofs of abstract interpreters compositional, thereby reducing proof complexity and proof effort. A key idea is to factorize the implementation of a concrete and abstract interpreter into a shared implementation based on Haskell arrows [Hughes 2000]. This factoring aligns the cases of the interpreters and exposes the structure along which a proof can be decomposed, namely the arrow operations used to define the generic interpreter. Because arrows are a first-order language and their code is not interleaved with computations of the meta-language, they induce an induction principle in the meta-language. By proving that every arrow operation preserves soundness of its arguments, the soundness of the entire generic interpreter directly follows from this induction principle. With this technique, we can decompose monolithic soundness proofs into smaller, reusable, and context free soundness lemmas about the arrow operations of the generic interpreter.

```
data Expr = Var String
          | Lit Int
          | Add Expr Expr
          | IfZero Expr Expr Expr
          | TryZero Expr Expr Expr

class ArrowFix x y c where
  fix :: (c x y → c x y) → c x y

class ArrowChoice c ⇒ IsVal v c where
  lookup :: c String v
  lit    :: c Int v
  add    :: c (v, v) v
  ifZero :: c x v → c y v → c (v,(x,y)) v
  try    :: c x y → c y v → c x v → c x v
```

```
eval' :: IsVal v c ⇒ c Expr v → c Expr v
eval' ev = proc e → case e of
  Var x → lookup ≺ x
  Lit n → lit ≺ n
  Add e1 e2 → do
    v1 ← ev ≺ e1; v2 ← ev ≺ e2
    add ≺ (v1,v2)
  IfZero e1 e2 e3 → do
    v ← ev ≺ e1
    ifZero ev ev ≺ (v,(e2,e3))
  TryZero e1 e2 e3 →
    try (proc (e1,x) → do
           v ← ev ≺ e1
          returnA ≺ (v,x))
       (ifZero ev ev)
        (proc (_,(_,e3)) → ev ≺ e3)
       ≺ (e1,(e2,e3))
```

Listing 3.2: generic interpreter based on arrows.

Note, that our technique requires to implement a concrete interpreter in the same meta-language as the abstract interpreter. This causes extra work if a reference semantics already exists and is implemented in a different meta-language. However, it simplifies the soundness proof as we do not have to conduct proofs across different meta-languages. In this subsection, we provide a brief introduction to arrows, and demonstrate how to use arrows to define a generic interpreter that corresponds to the concrete and abstract interpreters in the previous subsection. In the next subsection, we show how this generic interpreter enables a compositional soundness proof.

Arrows, like monads, support effectful computations that, for example, manipulate state, trigger exceptional control flow, or rely on non-determinism. Arrows generalize monads by internalizing the input type for a computation. For example, an arrow computation of type `(c x y)` expects a value of type `x` as input and yields a value of type `y`; `c` is the higher-order type constructor defining the arrow. In contrast, monadic computations have type `m y` and rely on meta-level bindings to implicitly provide inputs `x` through the lexical context in which the computation was defined. The pretty notation [Paterson 2001] for arrows reads similarly to `do`-notation for monads. The keyword `proc x` starts a new arrow computation with input `x`. The notation `y ←f≺ x` represents an arrow computation `f`, which receives its input from the variable `x` and binds the result to the variable `y`. This notation desugars to operations of the `Arrow`, `ArrowChoice`, and user-defined type classes with language-specific operations. This desugaring translates sequential statements `y ←f≺ x; g≺ y` into the sequential composition operator `f ⋙ g`. For arrow expressions where variables span multiple statements as in `y1 ←f≺ x1; y2 ←g≺ x2; h≺ (y1,y2)` the notation desugars into the parallel composition operator ⁂ as in `(f ⁂ g) ⋙ h`. Case expressions are translated to a pure function that destructs a sum type into an `Either` type, embedded into arrows with `arr :: (x → y) → c x y`, followed by the choice operator ||| of the `ArrowChoice` type class that encodes the bodies of each case. For example, here is an example desugaring (⇝) of an arrow expression:

```
   proc e → case e of { Var x → f ≺ x; Lit y → g ≺ y }
⇝ arr (λe → case e of { Var x → Left x; Lit y → Right y }) ⋙ (f ||| g)
```

Section 2.2.1 contains an illustrative example that the reader may find helpful for understanding how the pretty notation for arrows desugars into arrow expressions. For the full details on how arrows desugar, we refer the reader to the work of Paterson [2001].

In Listing 3.2, we use arrows to describe a generic interpreter `eval'` that generalizes both `eval` and $\widehat{\texttt{eval}}$ from the previous subsection. To do so, we extract the operations that differ between the concrete and abstract interpreter into a type class `IsVal`. Each type class member of `IsVal` in Listing 3.2 represents a language-specific operation. `lookup` defines a variable lookup operation

```
type Interp a b = Env → a → Maybe b
instance IsVal Val Interp where
  lookup = λe x → Map.lookup x e
  lit = arr id
  add = arr (λ(x,y) → x + y)

  ifZero f g = proc (v,(x,y)) →
    if v == 0
      then f ≺ x
      else g ≺ y

  try f g h = λe x → case f e x of
    Just y → g e y
    Nothing → h e x


eval :: Interp Expr Val
eval = fix eval'
```

```
type Interp̂ a b = Env̂ → a → Maybê b
 instance IsVal Val̂ Interp̂ where
  lookup̂ = λe x → toMaybe (Map.lookup e x)
  lit̂ = arr (λn → (n,n))
  add̂ = arr (λ((i₁,j₁),(i₂,j₂)) →
                (i₁+i₂,j₁+j₂))
  ifZerô f g = proc ((i,j),(x,y)) →
    if (i,j) == (0,0)
     then f ≺ x
     else if 0 ∉ (i,j) then g ≺ y
        else (f ≺ x) ⊔ (g ≺ y)
  trŷ f g h = λe x → case f e x of
    Just y → g e y
    Nothing → h e x
     JustNothing y → g e y ⊔ h e x

 evaL̂ :: Interp̂ Expr Val̂
eval̂ = fix eval'
```

Listing 3.3: Arrow instances for the concrete interpreter (left) and the abstract interpreter (right).

as an arrow from a string to the value type `v` that the type class is parameterized by. The `ifZero` operation is parameterized by two arrows as continuations and takes as argument a triple of a value and arguments `x` and `y` for the continuations. If the value in the triple is zero, the first continuation is invoked using `x`; otherwise, the second continuation is invoked using `y`. The `try` operation is parameterized by three arrows: one for computing a value (or raising an error); one for dispatching on the value resulting from invoking the first arrow if no error was raised; and one for the case where an error was raised. The `fix` operator of the type class `ArrowFix` (also Listing 3.2) computes the fixpoint of the generic interpreter. This allows concrete and abstract interpreter to employ different fixpoint strategies.

To define the concrete and abstract language semantics, we instantiate the generic interpreter with two different arrow instances. We do this in by defining two arrow types `Interp` and $\widehat{\texttt{Interp}}$ that define instances for the `Arrow`, `ArrowChoice`, `ArrowFix`, and `IsValue` type classes. In Listing 3.3, we show the arrow types, their instances for `IsValue`, and the top-level interpreters `eval` and $\widehat{\texttt{eval}}$ that instantiated the generic interpreter `eval'`. The generic interpreter completely desugars into operations of the arrow type classes implemented by `Interp` and $\widehat{\texttt{Interp}}$. Ultimately, the two instantiated interpreters have the same semantics as the interpreters of Section 3.2.1. Note that since the generic interpreter describes a parameterized semantics, we can define new alternative abstract domains by instantiating the generic interpreter with another arrow instance.

3.2.3 Compositional Soundness Proofs of Abstract Interpreters

The previous section described how to define concrete and abstract interpreters in a way that common code is shared between the two. This organization of concrete and abstract interpreter allows us to prove soundness of interpreters like `eval` and $\widehat{\texttt{eval}}$ in Listing 3.3 compositionally based on separate *soundness preservation lemmas* for each arrow operation. For our example, we prove the following soundness preservation lemmas, one for each operation of the `IsVal`, `Arrow`, `ArrowChoice`, and `ArrowFix` type classes. We use $f \mathrel{\dot\sqsubseteq} \widehat{f}$ as a compact notation for the soundness proposition.

- $\texttt{arr}\ f \mathrel{\dot\sqsubseteq} \widehat{\texttt{arr}}\ f$ for each pure function f in the generic interpreter,
- $\texttt{lit} \mathrel{\dot\sqsubseteq} \widehat{\texttt{lit}}$, $\quad \texttt{add} \mathrel{\dot\sqsubseteq} \widehat{\texttt{add}}$, $\quad \texttt{lookup} \mathrel{\dot\sqsubseteq} \widehat{\texttt{lookup}}$,
- if $f \mathrel{\dot\sqsubseteq} \widehat{f}$ and $g \mathrel{\dot\sqsubseteq} \widehat{g}$ then $\texttt{ifZero}\ f\ g \mathrel{\dot\sqsubseteq} \widehat{\texttt{ifZero}}\ \widehat{f}\ \widehat{g}$
- if $f \mathrel{\dot\sqsubseteq} \widehat{f}$ and $g \mathrel{\dot\sqsubseteq} \widehat{g}$ and $h \mathrel{\dot\sqsubseteq} \widehat{h}$ then $\texttt{try}\ f\ g\ h \mathrel{\dot\sqsubseteq} \widehat{\texttt{try}}\ \widehat{f}\ \widehat{g}\ \widehat{h}$
- if $f \mathrel{\dot\sqsubseteq} \widehat{f}$ and $g \mathrel{\dot\sqsubseteq} \widehat{g}$ then $f \ggg g \mathrel{\dot\sqsubseteq} \widehat{f} \mathbin{\widehat{\ggg}} \widehat{g}$

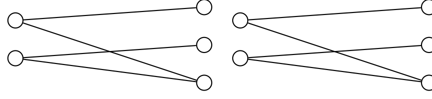


Figure 3.2: Soundness lemmas for the `try` operation (left) and for the `ifZero` operation (right).

- if $f \mathrel{\dot\sqsubseteq} \widehat{f}$ and $g \mathrel{\dot\sqsubseteq} \widehat{g}$ then $f \ast\!\ast\!\ast\ g \mathrel{\dot\sqsubseteq} \widehat{f}\ \widehat{\ast\!\ast\!\ast}\ \widehat{g}$
- if $f \mathrel{\dot\sqsubseteq} \widehat{f}$ and $g \mathrel{\dot\sqsubseteq} \widehat{g}$ then $f\ |||\ g \mathrel{\dot\sqsubseteq} \widehat{f}\ \widehat{|||}\ \widehat{g}$
- if $[\forall x, \widehat{x}. x \mathrel{\dot\sqsubseteq} \widehat{x} \Rightarrow f(x) \mathrel{\dot\sqsubseteq} f(\widehat{x})]$ then $\texttt{fix}\ f \mathrel{\dot\sqsubseteq} \widehat{\texttt{fix}}\ f$

The fixpoint combinator `fix` required a different soundness lemma, because it is the only higher-order construct compared to the otherwise first-order arrow language. To keep the rest of the shared arrow code first-order, we only allow one occurrence of `fix` at the very top-level of the interpreters.

For soundness of the interpreters $\texttt{eval} \mathrel{\dot\sqsubseteq} \widehat{\texttt{eval}}$, we first unfold the definition of `eval` and $\widehat{\texttt{eval}}$ which gives us $\texttt{fix eval}' \mathrel{\dot\sqsubseteq} \widehat{\texttt{fix}}\ \texttt{eval}'$. We then use the lemma for `fix`, which leaves us to prove $\texttt{eval}'\ x \mathrel{\dot\sqsubseteq} \texttt{eval}'\ \widehat{x}$ given $x \mathrel{\dot\sqsubseteq} \widehat{x}$. Because $\texttt{eval}'\ x$ and $\texttt{eval}'\ \widehat{x}$ refer to an arrow expression with the same structure, except for occurrences of x and $\widehat{x}$, we can use structural induction over the arrow expressions. The cases of this induction are always instances of the soundness lemmas for the arrow operations from above and the assumption $x \mathrel{\dot\sqsubseteq} \widehat{x}$. This proves that the top-level interpreters are sound.

But what impact does compositional soundness proofs have on proof complexity and proof effort? Let us compare the proof of `TryZero` to the non-compositional proof from the previous subsection. Before, we had to prove 7 cases of the abstract interpreter and relate them to up to 3 cases of the concrete interpreter. Now, `TryZero` is composed of `try` and `ifZero`. Their soundness lemmas are simpler and can be proved independently as illustrated in Figure 3.2. Moreover, the soundness lemmas are independent of their specific usage in the generic interpreter and can be reused whenever the generic interpreter makes use of `try` or `ifZero`. In particular, we reused the lemma for `ifZero` twice: Once for interpreting `IfZero` and once for interpreting `TryZero`.

To summarize, compared to conventional soundness proofs, in compositional soundness proofs we have to prove smaller lemmas that are context-free, which reduces the proof complexity. Furthermore, we have to prove less cases and lemmas are reused, which reduces the proof effort. In the next two sections we describe our framework more formally.

3.3 Soundness Proposition for Arrows

To construct compositional soundness proofs, we first need a soundness proposition $\dot\sqsubseteq$ that is applicable for all intermediate expressions of the interpreters. For example, the generic interpreter of Listing 3.2 uses the `ifZero` operator with return type `c (v,(Expr,Expr)) v`. This type is instantiated in the concrete interpreter with arrow type

$$\texttt{Interp}\ (\texttt{Val}, (\texttt{Expr}, \texttt{Expr}))\ \texttt{Val}$$

and in the abstract interpreter with arrow type

$$\widehat{\texttt{Interp}}\ (\widehat{\texttt{Val}}, (\texttt{Expr}, \texttt{Expr}))\ \widehat{\texttt{Val}}.$$

To relate values of these two types in our soundness proposition, we need to define a Galois connection [Cousot and Cousot 1979] between these arrow types. However, in general, our generic interpreter makes use of arrows of many different types, many which of which only become apparent after arrow desugaring. For example, the composition operator $\ggg$ is used by the generic interpreter with various types ranging from `Val` and `Expr` to tuples, `Maybe`, `Either`, and combinations thereof. To relate all types with a Galois connections, we require a systematic way for constructing Galois connections and, based on that, soundness propositions.

3.3.1 Systematic Way for Constructing Galois Connections

A well-known technique for constructing Galois connections is described by Nielson et al. [1999, Lemma 4.23]. A Galois connection $\alpha : \mathcal{P}A \rightleftarrows \widehat{A} : \gamma$ can be defined by an embedding function $\iota : A \to \widehat{A}$, such that the abstraction function is given by $\alpha(X) = \bigsqcup\{\iota(x) \mid x \in X\}$. Then the concretization function exists and is uniquely determined by $\gamma(\widehat{x}) = \{x \mid \alpha(x) \sqsubseteq \widehat{x}\}$. In other words, we only need to define an embedding function and we obtain the Galois connection for free.

First, we define embedding functions for abstracted base types. For example, for an interval analysis, we can define an embedding function for numeric values ι : `Int` $\to$ `Interval` by $\iota(n) = [n, n]$. Then the abstraction function sends the set $\{1, 3, 5\}$ to $\bigsqcup\{\iota(1), \iota(3), \iota(5)\} = \bigsqcup\{[1, 1], [3, 3], [5, 5]\} = [1, 5]$. Second, for compound types, we define the embedding function component-wise. For example, for products we define the embedding function $\iota_{(A,B)} : (A, B) \to (\widehat{A}, \widehat{B})$ by $\iota_{(A,B)}(a, b) = (\iota_A(a), \iota_B(b))$, given embeddings $\iota_A : A \to \widehat{A}$ and $\iota_B : B \to \widehat{B}$. This approach naturally extends to other compound data types we face in Haskell, such as lists `[a]`, `Maybe a`, `Either a b`, and so on. Note that data types in Haskell also have a coinductive interpretation, e.g., lists can be infinite. However, in this work we only consider inductive interpretations of datatypes.

However, the construction of Galois connections with embedding functions $\iota : A \to \widehat{A}$ places requirements on the concrete domain A and the abstract domain $\widehat{A}$. First, it assumes that both domains have a preorder $\sqsubset_A$ respectively $\sqsubset_{\widehat{A}}$. Second, it assumes that the abstract domain $\widehat{A}$ is finitely complete, that is, all elements x and y have a least upper bound $x \sqcup_{\widehat{A}} y$. While it is easy to define preorders for the types occurring in our interpreter, these orders often are not finitely complete. For example, type `Either Int String` has no least upper bound for `Left 5` and `Right "x"`. Fortunately, we can lift a non-completely ordered type X to a finitely complete ordered type $X^\top$. The lifting $X^\top$ adds a greatest element $\top$ to the type X, such that all incomparable elements now have a least upper bound:

$$x_1 \sqsubseteq_{X^\top} x_2 \text{ iff } x_2 = \top \vee x_1 \sqsubseteq_X x_2$$

For example, the lifted type (`Either Int String`)$^\top$ has all least upper bounds, such as (`Left 5`) $\sqcup$ (`Right` "x") $= \top$.

Based on embedding functions ι_X, partial orders $\sqsubset_X$, and the lifting $X^\top$, we can systematically construct Galois connections for all types that occurring in our interpreters. What is left, is to define the soundness proposition for arrow types `Interp` and $\widehat{\texttt{Interp}}$.

3.3.2 Soundness Proposition for Arrows

It is not possible to give a general definition of a soundness proposition for arbitrary arrows, because arrows and their soundness propositions are analysis-specific. However, we can define a soundness proposition for specific classes of arrows. In this section, we define a soundness proposition for Kleisli arrows [Hughes 2000]. Kleisli arrows are functions $A \to M(B)$ parameterized by a monad M. It is well-known that monads are expressive enough to describe a wide range of effects in programming languages [Liang et al. 1995; Wadler 1992; Moggi 1991]. For example, we can describe the two interpreter arrows of section Section 3.2.2 as Kleisli arrows:

$$\texttt{Interp}(A, B) = A \to M(B) \qquad \widehat{\texttt{Interp}}(A, B) = A \to \widehat{M}(B)$$
$$M(B) = \mathsf{Env} \to \mathsf{Maybe}\ B \qquad \widehat{M}(B) = \widehat{\mathsf{Env}} \to \widehat{\mathsf{Maybe}}\ B$$

This way, Kleisli arrows and their soundness proposition serve as a good starting point to define analysis-specific soundness propositions.

We define the soundness proposition for Kleisli arrows for the *forward collecting semantics* [Cousot and Cousot 1992a] of the concrete interpreter. The forward collecting semantics of a function $f : A \to B$ describes the strongest post-condition $\{f(x) \mid x \in X\}$ of f under a pre-condition $X \subseteq A$ over the inputs of f. For example, the strongest post-condition for $f(x) = x + x$ for the pre-condition $\mathbb{N}$ is the set of even numbers. In our scenario, we describe the forward

collecting semantics of $f : A \longrightarrow B$ as a single function $\lambda X.\{f(x) \mid x \in X\}$ of type $\mathcal{P}(A) \longrightarrow \mathcal{P}(B)$. Before we can define the soundness proposition for Kleisli arrows, we first need to define a Galois connection between the forward collecting semantics of the concrete Kleisli arrow and the $\top$-lifted abstract Kleisli arrow on the underlying function space [Nielson et al. 1999, page 253]:

$$\alpha_{A,M(B)} : (\mathcal{P}A \rightarrow \mathcal{P}(M(B))) \rightleftarrows \left(\widehat{A}^\top \rightarrow \widehat{M}(\widehat{B})^\top\right) : \gamma_{A,M(B)}$$
$$\alpha_{A,M(B)}(f) = \alpha_{M(B)} \circ f \circ \gamma_{\widehat{A}} \qquad \gamma_{A,M(B)}(\widehat{f}) = \gamma_{\widehat{M}(\widehat{B})} \circ \widehat{f} \circ \alpha_A$$

The Galois connection for Kleisli arrows uses Galois connections $\alpha_A : \mathcal{P}A \leftrightarrows \widehat{A}^\top : \gamma_{\widehat{A}}$ and $\alpha_{M(B)} : \mathcal{P}(M(B)) \leftrightarrows \widehat{M}(\widehat{B})^\top : \gamma_{\widehat{M}(\widehat{B})}$ constructed with the techniques described in Section 3.3.1. With the Galois connection between the concrete and abstract Kleisli arrows, we are ready to state soundness proposition for Kleisli arrows.

Definition 3.3.1 (Soundness proposition for Kleisli arrows). Let `Interp` and $\widehat{\texttt{Interp}}$ be Kleisli arrows. Then, a computation $f \in \texttt{Interp}(A, B)$ is sound with respect to a computation $\widehat{f} \in \widehat{\texttt{Interp}}(A, B)$

$$f \mathrel{\dot{\sqsubseteq}} \widehat{f} \quad \text{iff} \quad \alpha_{A,M(B)}(\lambda X.\, \{f(x) \mid x \in X\}) \sqsubseteq \widehat{f}^\top$$

In this definition, $\widehat{f}^\top$ is the $\top$-lifting of function $\widehat{f}$:

$$\widehat{f}^\top(x) = \begin{cases} \top, & x = \top \\ \widehat{f}(x), & x \neq \top \end{cases}$$

This definition is well-defined for Kleisli arrows over any types A and B for which Galois connections α_A and $\alpha_{M(B)}$ exist. Given these Galois connections, we can use this soundness proposition for all parts of the interpreters, making it a key ingredient for constructing compositional soundness proofs.

3.4 Compositional Soundness for Arrow-Based Abstract Interpreters

In this section, we present how our framework enables compositional soundness proofs and we prove that the composition always succeeds. Our framework is language-agnostic and can be used for any abstract interpreter satisfying the following two requirements:

- The concrete interpreter and abstract interpreter must share their implementation. That is, `eval` = `fix eval′` and $\widehat{\texttt{eval}}$ = $\widehat{\texttt{fix}}$ `eval′` for some `eval′`.
- The generic interpreter `eval′` must be an arrow computation.

The first requirement enables compositional soundness proofs, because the proof can be decomposed along the structure of the shared code. The second requirement ensures that the recomposition of subproofs must succeed. Together, they provide a powerful framework where all shared code is sound by construction and users only have to prove soundness for the differing code: the concrete and abstract implementations of arrow operations.

Arrows induce an induction principle because arrow notation [Paterson 2001] (used throughout the examples in this paper) fully desugars to operations of the arrow type classes and the residual code does not contain any non-arrow constructs of the meta-language anymore. Furthermore, the arrow type classes can be described by an endofunctor F [Hamana and Fiore 2011] and the arrow instances as algebras of this endofunctor. The initial F-algebra induces the desired induction principle. For example, the initial F-algebra for the generic interpreter of Listing 3.2 is described by the following generalized algebraic datatype (GADT) that enumerates all arrow expressions that can be described over the `IsVal` type class:

```
data AExp :: C → C → Set where
  Lit :: AExp Int v
  Add :: AExp (v,v) v
  Lookup :: AExp String v
```

```
IfZero :: AExp x v → AExp y v → AExp (v,(x,y)) v
Try :: AExp x y → AExp y v → AExp x v → AExp x v
(⋙) :: AExp x y → AExp y z → AExp x z
(***) :: AExp x y → AExp u v → AExp (x,u) (y,v)
(|||) :: AExp x z → AExp y z → AExp (Either x y) z
Arr₁ :: AExp A₁ B₁     ...    Arrₙ :: AExp Aₙ Bₙ
```

The datatype contains one constructor for each operation of the `IsVal` type class and its superclasses `Arrow` and `ArrowChoice`. It does not contain an operation for the fixpoint combinator, which requires special treatment as we discuss later. Besides these arrow operations, the desugaring arrow computations also generates pure functions that are embedded into arrow computations using the `arr` operation. To avoid a higher-order constructor `Arr :: (a → b) → AExp a b`, we enumerate each of the pure functions as individual constructors $\texttt{Arr}_i$. The initial F-algebra `AExp` then induces the following induction principle for predicates P.

$$\frac{\begin{array}{c} P(\texttt{Lit}) \qquad P(\texttt{Add}) \qquad P(\texttt{Lookup}) \\ P(f_1) \wedge P(f_2) \implies P(\texttt{IfZero}\ f_1\ f_2) \\ P(f_1) \wedge P(f_2) \wedge P(f_3) \implies P(\texttt{Try}\ f_1\ f_2\ f_3) \\ P(f_1) \wedge P(f_2) \implies P(f_1 \ggg f_2) \\ P(f_1) \wedge P(f_2) \implies P(f_1 *** f_2) \\ P(f_1) \wedge P(f_2) \implies P(f_1 \,|||\, f_2) \\ P(\texttt{Arr}_1) \qquad \dots \qquad P(\texttt{Arr}_n) \end{array}}{\forall \mathsf{A}, \mathsf{B} \in \mathbb{C}.\ \forall e \in \mathsf{AExp\ A\ B}.\ P(e)}$$

This induction principle allows us to decompose soundness proofs because of the shared implementation. Specifically, we set

$$P(e) \ \text{ iff } \ e \mathrel{\dot\sqsubseteq} \widehat{e},$$

where e refers to the concrete instance of the arrow code and $\widehat{e}$ to the abstract instance, i.e., the respective F-algebra. With this predicate, the premises of the induction principle exactly correspond to the *soundness preservation lemmas* discussed in Section 3.2.3. For example:

$$f_1 \mathrel{\dot\sqsubseteq} \widehat{f_1} \wedge f_2 \mathrel{\dot\sqsubseteq} \widehat{f_2} \implies (f_1 \ggg f_2) \mathrel{\dot\sqsubseteq} (\widehat{f_1} \mathbin{\widehat{\ggg}} \widehat{f_2})$$

Thus, the induction principle shows that *all* shared arrow code is sound if the soundness preservation lemmas hold. This is the essence of decomposing the soundness proof of an arrow-based abstract interpreter.

However, before we can state our main soundness theorem, we need to add support for fixpoint combinators. In Section 3.2.3, we applied concrete and abstract fixpoint combinators `fix` and $\widehat{\texttt{fix}}$ to the generic interpreter. Since fixpoint combinators are higher-order functions of the form

$$\texttt{Fix} : (\mathsf{AExp\ A\ B} \longrightarrow \mathsf{AExp\ A\ B}) \longrightarrow \mathsf{AExp\ A\ B},$$

adding them to our GADT would break the induction principle, because the datatype would not be strictly positive [Coquand and Paulin 1988]. Instead, we adapt the soundness proposition for fixpoint combinators by Cousot and Cousot [1992a, Proposition 4.3]:

Definition 3.4.1. A fixpoint combinator `fix` is sound with respect to $\widehat{\texttt{fix}}$ iff $\texttt{fix}\ f \mathrel{\dot\sqsubseteq} \widehat{\texttt{fix}}\ f$ for all soundness preserving functions $f_C : C(A,B) \longrightarrow C(A,B)$, that is:

$$[\forall x, \widehat{x}.\, x \mathrel{\dot\sqsubseteq} \widehat{x} \Rightarrow f(x) \mathrel{\dot\sqsubseteq} f(\widehat{x})] \implies \texttt{fix}\ f \mathrel{\dot\sqsubseteq} \widehat{\texttt{fix}}\ f.$$

Now we are ready to state our main soundness theorem:

Theorem 3.4.1 (Soundness of Abstract Interpreters based on Arrows)*.* *For a given concrete interpreter* $\texttt{eval} : \texttt{Interp}(A,B)$ *and abstract interpreter* $\widehat{\texttt{eval}} : \widehat{\texttt{Interp}}(A,B)$ *defined by* $\texttt{eval} = \texttt{fix}\ \texttt{eval}'$ *and* $\widehat{\texttt{eval}} = \widehat{\texttt{fix}}\ \texttt{eval}'$ *with a shared implementation* $\texttt{eval}'_C : C(A,B) \longrightarrow C(A,B)$ *(natural in C [Lane 1971])*[1] *over a functor F with an initial algebra, soundness* $\texttt{eval} \mathrel{\dot\sqsubseteq} \widehat{\texttt{eval}}$ *follows from (i) soundness of the fixpoint combinators* `fix` *and* $\widehat{\texttt{fix}}$ *and (ii) the soundness preservation lemmas of F.*

[1] $\texttt{eval}'_C$ is natural in C iff for all $f : C(A,B) \longrightarrow D(A,B)$, $f \circ \texttt{eval}'_C = \texttt{eval}'_D \circ f$

Proof. From the soundness proposition of the fixpoint combinators, we know that $\texttt{eval} \mathrel{\dot\sqsubseteq} \widehat{\texttt{eval}}$ if $\texttt{eval}'(x) \mathrel{\dot\sqsubseteq} \texttt{eval}'(\widehat{x})$ for all $x \in \texttt{Interp}(A,B), \widehat{x} \in \widehat{\texttt{Interp}}(A,B)$ with $x \mathrel{\dot\sqsubseteq} \widehat{x}$. Because `eval'` is natural in the arrow type C, the arrow expressions $\texttt{eval}'(x)$ and $\texttt{eval}'(\widehat{x})$ have the same structure except for occurrences of x and $\widehat{x}$. Thus $\texttt{eval}'(x) \mathrel{\dot\sqsubseteq} \texttt{eval}'(\widehat{x})$ follows by structural induction, the soundness preservation lemmas, and the assumption $x \mathrel{\dot\sqsubseteq} \widehat{x}$. □

Thus, to prove an abstract interpreter based on arrows sound, it suffices to use a sound fixpoint combinator and to verify the soundness preservation lemmas. Since each soundness preservation lemma is concerned with a single arrow operation only, the soundness proof of the abstract interpreter decomposes into small, manageable proof obligations.

The naturality of $\texttt{eval}'_C$ in the arrow type C is crucial in this proof of Theorem 3.4.1. It ensures that the generic interpreter does not produce a structurally different arrow expression when instantiated with the concrete and abstract arrow types. Only if the structure of the interpreters is the same, we can apply the induction principle. In general, we can ensure this if the generic interpreter is parametric in the arrow type.

One shortcoming of our proof method, though, is the handling of pure functions $\texttt{Arr}_1 \ldots \texttt{Arr}_n$ that the arrow desugaring generates. Proving soundness for each pure function is tedious and usually uninteresting. In the next section, we use parametricity [Reynolds 1983], a property of parametric polymorphism, to describe interface guidelines such that *all* pure functions are sound by a free theorem of parametricity.

3.5 Interface Design and Parametricity

The main goal of this paper is to reason about soundness of the operations of the interpreters, rather than about composed code of the generic interpreter itself. The design of the interface influences how much reasoning about shared code is necessary, if any at all. In this section, we provide guidelines for how to design interfaces such that soundness of pure functions follows as a free theorem of parametricity.

To this end, let us revisit the interface for `IfZero` from Section 3.2.2:

```
ifZero :: c x v → c y v → c (v,(x,y)) v
```

Instead of providing two continuations that are called when the argument value is zero or not, we could have designed an operation `isZero`, that returns a Boolean value that represents its outcome:

```
eval' ev = proc e → case e of
  IfZero e₁ e₂ e₃ → do
    b ← isZero ⋘ ev ≺ e₁; case b of
      True → ev ≺ e₂
      False → ev ≺ e₃
      ⊤ → (ev ≺ e₂) ⊔ (ev ≺ e₃)
```

```
data Bool^ = True | False | ⊤
isZero :: c v Bool^
```

The value ⊤ is solely used by the abstract interpreter to express uncertainty about whether a value is zero. The concrete instance of `isZero` never returns ⊤ because it is always certain if the value is zero. Although this definition describes an alternative but equivalent semantics, there are two problems:

1. The generic interpreter now describes behavior that is specific to the abstract interpreter but not the concrete semantics. The interface of the generic interpreter *leaks* details of the abstract interpreter into shared code.

2. Proving soundness of the instantiated generic interpreter requires reasoning about more code than just the arrow operations it is comprised of. In particular, we have to consider the entire case expression in the shared code to prove soundness. The interface design of `isZero` does not allow us to decompose the soundness proof.

But is there a metric that helps us identify interface operations that leak details of the abstract interpreter? The answer can be found in a property called *parametricity* [Reynolds 1983], a property of parametric polymorphism. The key idea of parametricity is that types can be interpreted as relations and terms in related environments yield related results [Wadler 1989].

To set the stage, we recall the definition of Reynolds' parametricity [Reynolds 1983] due to Ghani et al. [2015]. Well-typed System F programs e are identified by the typing judgment $\Gamma, \Delta \vdash e : \tau$, where τ is a type with type variables closed under Γ and Δ is the regular typing context. Parametricity describes two parallel interpretations for System F contexts, types and terms, that work in lock-step: An object interpretation $[\![T]\!]_o : \mathrm{Set}^{|\Gamma|} \to \mathrm{Set}$ that interprets types as sets and terms as functions, and a relational interpretation $[\![T]\!]_r : \mathrm{Rel}^{|\Gamma|}(A, B) \to \mathrm{Rel}([\![T]\!]_o A, [\![T]\!]_o B)$ that interprets types as relations and terms as relation preserving functions. Each interpretation takes extra arguments based on $|\Gamma|$, the number of type variables in the context Γ.

How these two interpretations interact is described by the following main theorem of parametricity [Reynolds 1983]:

Theorem 3.5.1 (Abstraction Theorem). *Let $A, B \in \mathrm{Set}^{|\Gamma|}$, $R \in \mathrm{Rel}^{|\Gamma|}(A, B)$, $a \in [\![\Delta]\!]_o A$ and $b \in [\![\Delta]\!]_o B$. For every term $\Gamma, \Delta \vdash e : \tau$, if $(a, b) \in [\![\Delta]\!]_r R$, then $([\![e]\!]_o A\ a, [\![e]\!]_o B\ b) \in [\![\tau]\!]_r(R)$.* □

More informally, if a and b are instances of the typing context Δ and are related by R, then a term e with context Δ applied to a and b are related by R. If we choose R to be the soundness proposition for arrow types, the abstraction theorem provides an alternative way to prove soundness of abstract interpreters with a shared implementation. We prove this as a theorem below. However, since arrows are higher-order types of kind $* \to * \to *$, we in fact require the abstraction theorem for higher-order parametricity that holds for System F_ω [Atkey 2012]. The general idea of the abstraction theorem for first-order parametricity carries over to the one for higher-order parametricity. Therefore, we omit the definitions for higher-order parametricity for simplicity.

Theorem 3.5.2. *In System F_ω, soundness of abstract interpreters that share a common implementation with the concrete interpreter follows from the soundness lemmas for operations of its interface.*

Proof. First, we desugar the type class `IsValue` into a record that is passed in as a dictionary [Hall et al. 1996]. This allows us to type check `eval'` with the following judgement:

$$\{\mathtt{c} : * \to * \to *,\ \mathtt{v} : *\}, \{\mathrm{dict} : \mathtt{IsValue\ c\ v}\} \vdash \mathtt{eval'} : \mathtt{c\ Expr\ v} \to \mathtt{c\ Expr\ v}$$

We now apply the abstraction theorem for higher-order parametricity as follows. The typing variable context has type variables for the arrow type `c` and value type `v`, hence, for A and B we choose the tuples $(\mathtt{Interp}, \mathtt{Val})$ and $(\widehat{\mathtt{Interp}}, \widehat{\mathtt{Val}})$ that instantiate the respective arrow and value type. Furthermore, for the relation R we have to define relations on arrows and values. For the relation on values, we choose $v \mathrel{\dot{\sqsubseteq}_{\mathtt{Val}}} \widehat{v}$ iff $\alpha_{\mathtt{Val}}(v) \sqsubseteq \widehat{v}$, where $\alpha_{\mathtt{Val}} : \mathcal{P}(\mathtt{Val}) \to \widehat{\mathtt{Val}}$ is the abstraction function for values. Because arrows are higher-kinded types, the relation on arrows is parameterized by relations R over the domain and Q over the codomain of the arrow. For the soundness relation on arrows, we choose

$$f \mathrel{\dot{\sqsubseteq}_{\mathtt{Interp}}} \widehat{f} \quad \text{iff} \quad (a, \widehat{a}) \in R \implies (f(a), \widehat{f}(\widehat{a})) \in Q \quad \text{for all } a \in A, \widehat{a} \in \widehat{A}.$$

If we instantiate R and Q with the relation $\alpha(x) \sqsubseteq \widehat{x}$, we obtain the original soundness proposition:

$$f \mathrel{\dot{\sqsubseteq}_{\mathtt{Interp}}} \widehat{f} \quad \text{iff} \quad \alpha_A(a) \sqsubseteq \widehat{a} \implies \alpha_{\widehat{A}}(f(a)) \sqsubseteq \widehat{f}(\widehat{a}) \quad \text{for all } a \in A, \widehat{a} \in \widehat{A}.$$

If we use the abstraction theorem with these definitions, we obtain the following rule.

$$\frac{\begin{array}{c} a \in [\![\mathtt{IsValue\ c\ v}]\!]_o(\mathtt{Interp}, \mathtt{Val}) \\ b \in [\![\mathtt{IsValue\ c\ v}]\!]_o(\widehat{\mathtt{Interp}}, \widehat{\mathtt{Val}}) \\ (a, b) \in [\![\mathtt{IsValue\ c\ v}]\!]_r(\dot{\sqsubseteq}_{\mathtt{Interp}}, \dot{\sqsubseteq}_{\mathtt{Val}}) \end{array}}{\begin{array}{c} ([\![\mathtt{eval'}]\!]_o(\mathtt{Interp}, \mathtt{Val})\ a, [\![\mathtt{eval'}]\!]_o(\widehat{\mathtt{Interp}}, \widehat{\mathtt{Val}})\ b) \in \\ [\![\mathtt{c\ Expr\ v} \to \mathtt{c\ Expr\ v}]\!]_r(\dot{\sqsubseteq}_{\mathtt{Interp}}, \dot{\sqsubseteq}_{\mathtt{Val}}) \end{array}}$$

The rule says, given two instances a and b for the interface `IsValue` and a and b satisfy the soundness preservation lemmas of `IsValue`, then the generic interpreter `eval'` instantiated with the instance a is sound with respect to `eval'` instantiated with b. □

The main consequence of Theorem 3.5.2 is that we do not have to reason about soundness of shared code, since it follows as a free theorem from parametricity. In particular, this relieves us from having to prove soundness of individual pure functions in arr. Instead, we obtain a generic soundness lemma for the arr operation itself:

$$(\mathtt{arr}, \widehat{\mathtt{arr}}) \in [\![\forall \mathtt{x}, \mathtt{y}.\ (\mathtt{x} \to \mathtt{y}) \to \mathtt{c}\ \mathtt{x}\ \mathtt{y}]\!]_r(\dot{\sqsubseteq}_{\mathtt{Interp}}).$$

Because all pure functions f in the generic interpreter are shared code, this lemma guarantees $(\mathtt{arr}\ f \mathrel{\dot{\sqsubseteq}} \widehat{\mathtt{arr}}\ f)$.

Theorem 3.5.2 can also help us understand how to design the interface such that the each arrow operation is compositionally sound. When a soundness proof for an arrow operation fails, it usually fails with the approach based on parametricity as well as with the approach from Section 3.4. However, the approach based on parametricity can tell us why a proof failed. To this end, it is instructive to compare the soundness lemmas of Theorem 3.5.2 to the corresponding soundness lemmas of Theorem 3.4.1. For example, for the composition operator $\ggg$, Theorem 3.5.2 requires

$$(\ggg, \widehat{\ggg}) \in [\![\forall \mathtt{x}, \mathtt{y}, \mathtt{z}.\ \mathtt{c}\ \mathtt{x}\ \mathtt{y} \to \mathtt{c}\ \mathtt{y}\ \mathtt{z} \to \mathtt{c}\ \mathtt{x}\ \mathtt{z}]\!]_r(\dot{\sqsubseteq}_{\mathtt{Interp}})$$

whereas Theorem 3.4.1 requires

$$f_1 \mathrel{\dot{\sqsubseteq}} \widehat{f_1} \wedge f_2 \mathrel{\dot{\sqsubseteq}} \widehat{f_2} \implies (f_1 \ggg f_2) \mathrel{\dot{\sqsubseteq}} (\widehat{f_1} \mathrel{\widehat{\ggg}} \widehat{f_2}).$$

The soundness lemmas have almost the same meaning, except that the orderings used in the former lemma are fixed by the relational interpretation $[\![-]\!]_r$ rather than chosen by us. This is an important distinction because it restricts how we can design our interface, while still being able to prove soundness compositionally.

For example, let us revisit the flawed version of isZero introduced earlier in this section. Observe that we cannot prove $\mathtt{ifZero} \mathrel{\dot{\not\sqsubseteq}} \widehat{\mathtt{ifZero}}$ using parametricity either:

$$(\mathtt{isZero}, \widehat{\mathtt{isZero}}) \notin [\![\mathtt{c}\ \mathtt{v}\ \widehat{\mathtt{Bool}}]\!]_r(\dot{\sqsubseteq}_{\mathtt{Interp}}, \dot{\sqsubseteq}_{\mathtt{Val}})$$

The problem is that the ordering for $\widehat{\mathtt{Bool}}$ is determined by its relational interpretation based on the underlying sum type:

$$[\![\widehat{\mathtt{Bool}}]\!]_r = \{(\mathtt{True}, \mathtt{True}), (\mathtt{False}, \mathtt{False}), (\mathtt{Top}, \mathtt{Top})\}.$$

However, we require that $\top$ is the greatest element to be able to prove the soundness lemma for isZero, which is not the case for this ordering. The underlying problem is that we exposed the type $\widehat{\mathtt{Bool}}$ with non-standard ordering to the generic interpreter. This problem exists not only for $\widehat{\mathtt{Bool}}$, but for all types with non-standard ordering, such as values, environments, etc.

These observations lead us to the following guideline for good interface design of generic interpreters, helping us to avoid leaking interfaces:

Guideline. An interface of a generic interpreter is good if its operations do not expose types with non-standard orderings. Instead, non-standard ordered types in the abstract interpreter must be hidden from the interface by using universal quantification.

To summarize, the abstraction theorem for meta-languages with parametricity provides an alternative way to prove soundness of abstract interpreters that share code. This drastically reduces the required effort of the soundness proof, because shared code is sound by a free theorem of parametricity. Furthermore, the abstraction theorem provides us with a useful guideline for how to design our interface. Finally, nothing in the proof of Theorem 3.5.2 is specific to arrows. In particular, we are not making use of the induction principle for arrows and use the abstraction theorem instead. This should allow us to apply Theorem 3.5.2 to abstract interpreters that share code with the concrete interpreter using an interface other than arrows. We have not explored this further so far.

3.6 Case Studies

This paper presents a framework for compositional soundness proofs. In this section, we report on two case studies that we conducted to answer the following research questions:

(RQ1) Is our technique applicable to interesting languages and interesting static analyses?

(RQ2) Does our technique reduce the complexity and effort of soundness proofs?

The case studies involved constructing generic interpreters for Stratego and PCF, developing concrete and abstract arrow instances, and proving the instantiated interpreters sound. For Stratego, we developed a tree-shape analysis as abstract arrow instance; for PCF, we implemented an advanced control-flow analysis (k-CFA) as abstract arrow instance.[2]

3.6.1 Tree-Shape Analysis for Stratego

We developed a sound abstract interpreter for *Stratego* [Visser et al. 1998], a real-world language for the implementation of program transformations that operate on abstract syntax trees akin to s-expressions. Stratego is being used in various projects to define interpreters [Dolstra and Visser 2002], refactorings [de Jonge and Visser 2012], desugarings [Erdweg et al. 2011], and compilers [Avgustinov et al. 2007; Bagge and Kalleberg 2006; Economopoulos and Fischer 2011]. Furthermore, Stratego is used to compile programs of WebDSL [Visser 2007], a domain-specific web-programming language in which, for example, the website `conf.researchr.org` of ICFP and others is implemented [van Chastelet et al. 2015].

Stratego transformations operate on untyped terms using rewrite rules and strategies as illustrated by the following simple evaluator for arithmetic expressions:

```
rules
  reduce: Add(Succ(m), n) → Succ(Add(m,n))
  reduce: Add(Zero(), n)  → n

strategies
  main = downup(try(reduce))
```

The strategy `main` walks the expression tree down and up again, and tries to reduce each visited node using the rewriting rule `reduce`. Rule `reduce` consists of two alternatives `reduce: pat → gen` that try to match `pat` and, if successful, generate `gen`. Stratego has many language features that make it a challenging language to statically analyze, including dynamic scoping of pattern-bound variables, higher-order functions, and generic tree traversals.

Stratego provides a rich set of abstractions for program transformations. These abstractions desugar into a core language for Stratego with just 12 constructs [Visser et al. 1998; Bravenboer et al. 2006]. We developed a generic interpreter based on arrows for this core language. For the interface of the generic interpreter, we identified 27 operations, of which 9 operations are language-independent and 18 operations are specific to Stratego. The language-specific operations consist of 10 operations for terms, 6 for term environments, and 2 for strategy environments.

We instantiated the generic interpreter with Kleisli arrows for the concrete and abstract domain. The concrete domain uses the usual interpretation of terms and environments. In the abstract domain, we approximate terms as a set of term patterns containing wildcards $*$. For example, the abstract term $\{\texttt{Zero()}, \texttt{Add}(*,*)\}$ represents the set of concrete terms containing `Zero()` and all terms with root `Add`. This way, our abstract arrow instance realizes a tree-shape analysis [Keidel and Erdweg 2017] that Stratego developers can use to predict the shape of trees a transformation will produce when run.

For the concrete instance of `ArrowFix`, we compute the usual least fixpoint. However, since the abstract domain of sets of term patterns is infinite, the least fixpoint is not computable for the abstract domain. Therefore, for the abstract instance of `ArrowFix`, we approximate the greatest fixpoint instead. Specifically, our fixpoint combinator keeps track of the recursive depth of the interpreter and yields $\top$ for recursive calls whose depth exceeds a certain threshold. This produces a finite approximation of the infinite set of terms that can be produced by a given program transformation. The precision of the abstract interpreter increases with more iterations.

We have verified the soundness of our tree-shape analysis by proving that abstract instantiations of the generic interpreter approximates the concrete instantiation. The soundness proof

[2] All code of the case studies is open source and can be found at `https://github.com/svenkeidel/sturdy/`.

```
call :: ... ⇒ StratVar → [Strat] → [TermVar] → (Strat → c t t) → c t t
call f actualStratArgs actualTermArgs ev = proc a → do
  senv ← readStratEnv ≺ ()
  case Map.lookup f senv of
    Just (Closure formalStratArgs formalTermArgs body senv') → do
      tenv ← getTermEnv ≺ ()
      mapA bindTermArg ≺ zip actualTermArgs formalTermArgs
      let senv'' = foldl bindStratArgs (if Map.null senv' then senv else senv')
                                       (zip formalStratArgs actualStratArgs)
      b ← localStratEnv senv'' (ev body) ≺≺ a
      tenv' ← getTermEnv ≺ ()
      putTermEnv ⋘ unionTermEnvs ≺ (formalTermArgs,tenv,tenv')
      returnA ≺ b
    Nothing → fail ≺ ()
  where
    bindTermArg = proc (actual,formal) →
      lookupTerm (proc t → insertTerm ≺ (formal,t)) fail ≺≺ actual

    bindStratArgs senv (v,Call v' [] []) senv =
      case Map.lookup v' senv of
        Just s → Map.insert v s senv
        _ → error $ "unknown strategy: " ++ show v'
    bindStratArgs senv (v,s) = Map.insert v (Closure [] [] s senv) senv
```

Listing 3.4: Shared implementation of calls of strategies.

is completely compositional. We decomposed the proof into 27 soundness lemmas, one for each operation in the interface of the generic interpreter. All operations in the interface conform to the guidelines of interface design of Section 3.5, and hence soundness of all pure `arr` expressions follows as a free theorem due to the parametricity of our meta-language Haskell. The soundness of the instantiated interpreters then follows from Theorem 3.4.1 and the 27 soundness lemmas.

To reflect on the complexity and effort of the soundness proof (RQ2), we want to highlight the soundness proof of the implementation of strategy calls. We show the code of the generic interpreter in Listing 3.4. A strategy in Stratego accepts two kinds of arguments, strategy arguments and term arguments. Hence, the interpreter has to bind these two kinds of arguments in the respective environment and then invoke the interpreter recursively on the body of the called strategy.

Traditionally, proving soundness of the concrete and abstract instantiations of this code is a severe challenge: The complexity of the code would be reflected in the proof. With our technique, we can decompose the proof into 2 soundness lemmas about strategy environments (`readStratEnv`, `localStratEnv`), 6 lemmas about term environments (`lookupTerm`, `insertTerm`, `unionTermEnvs`, `getTermEnv`, `putTermEnv`), a few lemmas about language-independent arrow operations, and various lemmas about embedded pure functions. Each of these lemmas is manageable and can be proved in isolation, thus reducing the proof complexity. Our approach also reduces the proof effort. First, some lemmas can be reused in other cases of the interpreter, such as the ones for term environments, which are needed for pattern matching as well. Second, we obtain the soundness lemmas for applications of embedded pure functions `Map.lookup`, `zip`, and `foldl` as free theorems of parametricity. And third, the soundness of the generic interpreter follows for free from the induction principle of Theorem 3.4.1.

In summary, we developed an arrow-based generic interpreter for Stratego together with a concrete and an abstract arrow instance. The abstract arrow instance realizes a tree-shape analysis for Stratego. We compositionally proved this analysis sound by verifying 27 smaller and individually provable lemmas. Thus, our technique was applicable to this scenario (RQ1) and, as we argued, the resulting proof is less complex and required less effort than a traditional proof (RQ2).

```
data Val                                data Val̂ = ⊤
  = ClosureVal (Expr,Env)                 | ClosureVal̂ (Set (Expr,Env̂))
  | NumVal Int                            | NumVal̂ Interval
type Env = Map String Val               type Env̂ = Map String Addr
                                        type Storê = Map Addr Val̂
```

Listing 3.5: Concrete (left) and abstract domain (right) for k-CFA analysis of PCF.

3.6.2 Control-Flow Analysis for PCF

We implemented an abstract interpreter for an analysis that has been widely studied in the literature [Midtgaard 2012]: control-flow analysis (CFA). We implemented this analysis for PCF [Plotkin 1977], a language with first-class functions, numbers, an `ifZero` construct, and fixpoint combinator `Y`. The analysis we implemented is a k-CFA analysis [Shivers 1991] and the fixpoint algorithm we used is due to Darais et al. [2017].

We briefly summarize how our analysis works. The analysis approximates functions (closures) as sets of expression and environment pairs, while natural numbers are approximated using bounded intervals. We ensure termination by employing Darais et al.'s fixpoint algorithm for big-step semantics [2017]. Darais et al.'s fixpoint algorithm memoizes the results of all interpreter calls in a cache. When the interpreter is called with the same expression and environment recursively or repeatedly, it returns the cached result instead of recursing. This guarantees termination since there are only finitely many environments to consider and the interpreter repeats itself eventually. To finitely approximate environments, we adopt a common approach for (k-)CFA [Shivers 1991; Horn and Might 2010]: We allocate the values of an environment in an abstract store that has only finitely many addresses available. There are only finitely many stores if all abstract values are finite domains. For closures this is the case, because there only finitely many expressions that can be evaluated for a given program. And our abstract domain for numbers is finite, because we restrict the maximum bounds of intervals. If an interval exceeds theses bounds, it is approximated with infinity. We summarize the concrete and abstract domain of the k-CFA interpreter in Listing 3.5.

Listing 3.6 shows the generic PCF interpreter and the interface that we developed for it. The interface has a total of 16 operations: 4 value operations (class `IsVal`), 2 closure operations (`IsClosure`), 4 environment operations (`ArrowEnv`), a fixpoint operation (`ArrowFix` from Section 3.2.2), a failure operation (`ArrowFail`), and 4 language independent arrow operations (`Arrow`, `ArrowChoice`). We developed two instances of the interface: A concrete instance and a k-CFA instance. The code of these instances can be found in the artifact of our paper and its accompanying documentation.

We compositionally proved the soundness of k-CFA instantiated interpreter relative to the concrete instantiation of the interpreter. We decomposed the soundness proof into 16 lemmas, one for each operation of the arrow type classes referenced by the generic interpreter. Soundness of all pure `arr` expressions followed by parametricity of the meta-language Haskell (Theorem 3.5.2). As is common in proofs by induction, often the induction hypothesis has to be strengthened such that all cases of the induction are provable. We encountered this situation when proving soundness of the environment operations in the `ArrowEnv` type class. We had to strengthen the soundness proposition to guarantee that all environments passed in and out of the abstract arrow operation are consistent with the abstract store, i.e., all environment-bound addresses exist in the store. Note that this strengthened requirement of store consistency is not an artifact of using our techniques: It is necessary for non-compositional soundness proof as well.

To assess the complexity of our proof, we compare it to another proof of a k-CFA for a PCF-like language that can be found in the PhD thesis of Darais [2017]. The proof in Darais' PhD thesis relates in three theorems four different semantics, each proven by induction over derivations. It is not obvious how the cases of the induction can be decomposed further systematically, because of the differences between the concrete and abstract semantics. In comparison, our proof consists of 16 soundness lemmas that relate the concrete and abstract instances directly. The lemmas prove smaller pieces of functionality than the induction cases in Darais' proof. For example, the generic

```
data Expr = Var Text | Lam String Expr
  | App Expr Expr | Y Expr | Zero | Succ Expr
  | Pred Expr | IfZero Expr Expr Expr

class IsNum v c where
  succ, pred :: c v v
  zero :: c () v
  if_ :: c x z → c y z
      → c (v, (x, y)) z

class IsClosure v env c where
  closure :: c (Expr, env) v
  applyClosure :: c ((Expr,env),v) v
                → c (v, v) v

applyClosure' ev = applyClosure $
  proc ((e,env),arg) → case e of
    Lam x body → do
      env' ← extendEnv ≺ (x,arg,env)
      localEnv ev ≺ (env', body)
    Y e' → do
      fun' ← localEnv ev ≺ (env, Y e')
      applyClosure' ev ≺ (fun',arg)
    _ → fail ≺ show e
```

```
eval :: (IsNum v c, IsClosure v env c,
   ArrowChoice c, ArrowFix Expr v c,
  ArrowEnv Text v env c,ArrowFail String c)
  ⇒ c Expr v
eval = fix $ λev → proc e → case e of
  Var x → lookup' ≺ x
  Lam x e1 → do
    env ← getEnv ≺ ()
    closure ≺ (Lam x e1, env)
  App e1 e2 → do
    fun ← ev ≺ e1
    arg ← ev ≺ e2
    applyClosure' ev ≺ (fun, arg)
  Zero → zero ≺ ()
  Succ e1 → do
    v ← ev ≺ e1; succ ≺ v
  Pred e1 → do
    v ← ev ≺ e1; pred ≺ v
  IfZero e1 e2 e3 → do
    v1 ← ev ≺ e1
    if_ ev ev ≺ (v1, (e2, e3))
  Y e1 → do
    fun ← ev ≺ e1
    env ← getEnv ≺ ()
    arg ← closure ≺ (Y e1, env)
    applyClosure' ev ≺ (fun, arg)
```

Listing 3.6: Generic interpreter for PCF and its language specific interface.

interpreter in Listing 3.6 uses a helper function `apply'` to apply a closure value to an argument value. Since we had proven the soundness of the language-independent arrow operations, the soundness proof for the shared code in `apply'` decomposed into just 3 soundness lemmas about interface operations: one for `apply`, the operation that unpacks a closure; one for `extendEnv`, the operation that extends the environment with an argument value; and one for `localEnv`, the operation that interprets under the extended environment. The functionality of `apply'` requires a manual proof in Darais' thesis, but in our setting, we get soundness of `apply'` for free because it is shared code and is sound by Theorem 3.5.2. There are of course also commonalities between the proofs, most significantly, we borrow the soundness lemma for fixpoints from Darais.

In summary, we developed a k-CFA analysis for PCF in our framework. We compositionally proved this analysis sound by verifying 16 smaller and individually provable lemmas. Thus, our techniques can be used to prove soundness of k-CFA, an interesting and widely studied static analysis (RQ1). As we argued, the resulting proof is less complex and required less effort than a traditional proof (RQ2).

3.7 Related Work

Our work is a continuation of a long line of research on constructing and proving the soundness of abstract interpreters. We have already related to many relevant sources throughout this paper. Here, we discuss related work in more detail.

One of the main ideas of abstract interpretation is to systematically *derive* a sound static analysis from a concrete semantics, by using the soundness proposition and proof as the guiding principle. Cousot and Cousot [1979] pioneered the approach, which has since been extended to a wide range of domains and semantic styles [Cousot 1999]. Such derivations enable soundness proofs that follow a systematic sequence of derivation and proof steps. But these proof steps can be involved, especially for interesting languages where one case of the abstract interpreter relates to many cases of the concrete interpreter. The focus of our work is to minimize the effort and complexity involved in proving soundness. We achieve this by factoring the concrete and abstract

interpreter into a shared implementation that is parameterized over an arrow-based interface. The abstract instance of that interface can still be derived using techniques described by Cousot [1999]. However, in our experience, a soundness proof after the definition is easier because the proof goal is clear and progress can be made from either concrete and abstract side.

The idea of defining a language by implementing an interpreter in a meta-language (definitional interpreters) was famously described by Reynolds [1998]. In the context of abstract interpretation, the idea was explored even earlier by Jones and Nielson [1994], who describe an approach that translates expressions of the object language into expressions of a suitable meta-language. Constructs of the meta-language then have two different interpretations, one that recovers to the concrete semantics and one that recovers the abstract semantics of the object language. As a reasoning principle for soundness, the authors define a logical relation [Plotkin 1980] over the meta-language. The main benefit from using a logical relation is, soundness of all programs in the meta-language follows from soundness lemmas for each meta-language construct. The logical relation has to be proven when the meta-language is created and maintained when the meta-language changes. Compared to this paper, we use arrows as a meta-language and their induction principle as reasoning tool for soundness. This induction principle is very similar to a logical relation as it allows us to prove soundness of any arrow expression from soundness lemmas for each arrow operation. However, the main benefit of this induction principle is that we do not need to prove or maintain the induction principle itself. The induction principle follows for free from the fact that we use arrows, which are a first order language and can be expressed by an algebraic datatype.

The topic of definitional abstract interpreters was also recently revisited by Darais et al. [2017]. They show that an abstract definitional interpreter inherits properties of the meta-language, such as push-down control-flow precision. Similarly to our work, the concrete and abstract interpreter that Darais et al. present share code, but over a monadic interface instead of one based on arrows. Another similarity is that the abstract interpreters that we present can be regarded as definitional abstract interpreters, since we are using a meta-language to define our interpreters. The main difference between the work of Darais et al. is that we use a restricted meta-language (arrows), not necessarily as a means to inherit functional properties, but as a means to making soundness proofs compositional, which was not the focus of Darais et al.. We provide a generic theorem that ensures the soundness of an arrow-based abstract interpreter based on the soundness lemmas of the arrow operations, and we use parametricity to obtain soundness of embedded pure functions for free.

Monadic abstract interpreters by Sergey et al. [2013] show that concepts in static analysis such as context-sensitivity, poly-variance, flow-sensitivity, etc. are independent of any particular language semantics and can be captured by an appropriate monad. These results carry over to our abstract interpreters based on arrows, because every monad gives rise to a Kleisli arrow. Sergey et al. describe their semantics using a shared monadic small-step abstract machine, but do not address the question of how to prove soundness of monadic abstract interpreters. We address soundness in this paper by factoring concrete and abstract interpreter into a shared big-step interpreter, which enables compositional soundness proofs. The usage of arrows provides an induction principle, which allowed us to ensure the soundness of the abstract interpreter by construction of sound arrow operations. We expect that it is possible to define a small-step abstract machine in the style of Sergey et al., but using arrows instead of monads in a way that our generic theorems apply.

Galois transformers and modular abstract interpreters by Darais et al. [2015] represent a systematic way to construct monadic abstract interpreters. Galois transformers are monad transformers, whose monadic operations can be proven sound with respect to each other. While our technique decomposes a soundness proof along operations of an interface, Galois transformers decompose a soundness proof along a monad transformer stack. For example, the operations `get` for fetching and `put` for writing state can be proven sound with respect to the concrete and abstract state monad transformer, independent of the rest of the monad transformer stack. The technique described in our paper and Galois transformers complement each other: Galois transformers still require a way to compose the lemmas of operations to a proof of the interpreters, which we provide. And our technique can benefit from decomposing the proof of soundness lemmas even further. In the future we want to combine these two approaches by using arrow

transformers to achieve an even larger degree of proof decomposition.

Abstracting abstract machines (AAM) by Horn and Might [2010] is a technique for deriving sound abstract interpreters from concrete language semantics described as abstract machines. The concrete semantics is transformed in multiple steps to an abstract machine that is suitable to be approximated by an abstract interpreter. Each step of the transformation is systematic and preserves soundness with respect to the original concrete semantics. A consequence of this approach is that there must be a one-to-one correspondence between transitions in the concrete and abstract semantics. As we have discussed in Section 3.2, this is often not the case, for example, for `ifZero` over the interval domain. In contrast, our approach only requires a one-to-one correspondence between concrete and abstract arrow operations, but allows for a mismatch within these operations: An abstract operation can distinguish m cases even if the corresponding concrete operation distinguishes n cases.

Cousot et al. [2006] describe a different technique of soundness proof composition which is orthogonal to ours: The technique is for composing separate abstract analyses by organizing them in a hierarchy, such that analyses further down in the hierarchy can be influenced by the output from analyses further up, but not the other way around. The focus of our paper is not on composing *different analyses*, but rather on composing a soundness proof for a generic abstract interpreter from reusable lemmas about the operations of the language being abstracted.

3.8 Conclusion

We have presented a novel technique for defining concrete and abstract interpreters by sharing code over an interface based on arrows. Such interpreters can be proven sound *compositionally*: Our Theorem 3.4.1 tells us how to compose such a proof, and reduces the effort of proving soundness to the effort of proving a context-free soundness lemma for each interface operation and each embedded pure function in the generic interpreter. Our Theorem 3.5.2 applies *parametricity* to obtain the soundness of the embedded pure functions *for free*, which further reduces the proof effort. We demonstrated the applicability of our technique by implementing two case study analyses and proving them sound: a tree-shape analysis for Stratego and a k-CFA analysis for PCF. Compared to traditional soundness proofs abstract interpreters, our soundness proofs are less complex and require less effort because we were able to decompose large proof obligations into independent soundness lemmas, from which the soundness of the abstract interpreters follows by construction. In the future, we want to investigate how our technique scales to even more complicated languages and analyses.

SOUND AND REUSABLE COMPONENTS FOR ABSTRACT INTERPRETATION

4

This chapter is based on the following peer-reviewed paper:

Sound and reusable components for abstract interpretation.
Sven Keidel and Sebastian Erdweg.
Proc. ACM Program. Lang. 3, OOPSLA (2019), 176:1–176:28.
https://doi.org/10.1145/3360602

Abstract — Abstract interpretation is a methodology for defining sound static analysis. Yet, building sound static analyses for modern programming languages is difficult, because these static analyses need to combine sophisticated abstractions for values, environments, stores, etc. However, static analyses often tightly couple these abstractions in the implementation, which not only complicates the implementation, but also makes it hard to decide which parts of the analyses can be proven sound independently from each other. Furthermore, this coupling makes it hard to combine soundness lemmas for parts of the analysis to a soundness proof of the complete analysis.

To solve this problem, we propose to construct static analyses modularly from *reusable analysis components*. Each analysis component encapsulates a single analysis concern and can be proven sound independently from the analysis where it is used. We base the design of our analysis components on *arrow transformers*, which allows us to compose analysis components. This composition preserves soundness, which guarantees that a static analysis is sound, if all its analysis components are sound. This means that analysis developers do not have to worry about soundness as long as they reuse sound analysis components. To evaluate our approach, we developed a library of 13 reusable analysis components in Haskell. We use these components to define a k-CFA analysis for PCF and an interval and reaching definition analysis for a While language.

4.1 Introduction

Abstract interpretation [Cousot and Cousot 1979] is a methodology for defining sound static analysis. A static analysis is sound if it predicts, at compile time, all relevant dynamic behavior of a program. For example, if a sound static nullness analysis claims a variable is not null, then this variable may not store a null pointer in any execution of the program. Analysis soundness is important whenever a developer or optimizing compiler acts on the analysis result [Knoop and Rüthing 1999]. For example, when a developer or compiler omits a null check, only a sound nullness analysis can provide the required guarantee that this check is indeed redundant.

Building sound static analyses for modern programming languages is difficult. Analysis developers must provide abstractions for all values (e.g., integers, strings, objects) as well as for all effects (e.g., environments, stores, exceptions) supported by the analyzed language. The combination of these abstractions forms the essence of a static analysis. However, a static analysis often closely couples different abstractions, which makes it harder to replace them. This coupling also complicates a soundness proof, as it is not clear which parts of the analysis can be proven sound independently and which parts have to be proven together. Furthermore, the coupling makes it hard to establish an end-to-end soundness proof, from soundness lemmas for each part of the analysis.

In this paper, we propose *analysis components* as modular building blocks for static analyses. An analysis component is governed by an interface $\mathcal{I}$ that describes which concern of the analyzed language the component implements. For example, an analysis component for stores will enlist read and write operations in its interface $\mathcal{I}$. The crucial feature of analysis components is that they can be proven sound individually, and the soundness of the complete static analysis follows by construction. To this end, each analysis provides both the canonical concrete semantics C and an abstract semantics $\widehat{C}$ for the operations enlisted in the interface. An analysis component

is sound if for each operation in $\mathcal{I}$, the abstract semantics $\widehat{C}$ approximates the concrete semantics C. Analysis developers can use such analysis components as building blocks to construct sound static analyses.

In our approach, analysis developers define a static analysis as an interpreter against the interfaces of analysis components. We call such an interpreter a *generic interpreter* because it is not specific to the concrete or abstract semantics stipulated by the analysis components. Indeed, we can instantiate the same generic interpreter to obtain a range of alternative language semantics by selecting compatible components:

- We obtain a concrete interpreter using the canonical concrete semantics C of the components.
- We obtain an abstract interpreter using the abstract semantics $\widehat{C}$ of the components.

A key theoretical result of this work is that the instantiated abstract interpreter is guaranteed to soundly approximate the instantiated concrete interpreter if the used analysis components are sound. That is, analysis developers do not need to worry about soundness as long as they combine sound analysis components.

As consequence of our design, the same analysis component can be reused across analyses and across languages without change. For example, when researchers discover a new abstraction for stores, they can cast it as an analysis component implementing the `Store` interface and prove the component sound. Afterwards, any existing analysis that uses a `Store` component can easily be upgraded to use the new abstraction, without needing to revisit the soundness of the analysis. Moreover, many analysis components like `Store` are actually language-independent and can be reused across languages. Indeed, most language-specific behavior is captured by the generic interpreter. Thus, to target a new language, an analysis developer can reuse existing analysis components and only has to develop a generic interpreter for the new language.

We demonstrate that our design is feasible by developing an component-based analysis framework in Haskell. In our framework, we represent analysis components as a pair of arrow transformers, a generalization of monad transformers. We can compose these arrow transformers and use them to instantiate the generic interpreter, thus obtaining executable concrete and abstract interpreters. We extend the arrow-based theory on compositional soundness proofs for abstract interpreters by Keidel et al. [2018] to allow reasoning about isolated arrow transformers. This forms the basis of our new theory about horizontal and vertical composability of analysis components, and the proof obligations entailed thereby.

We evaluate our design by creating the open-source library *Sturdy* of 13 sound analysis components in Haskell. We demonstrate the applicability of our analysis components by using them to define well-known analyses: A k-CFA analysis [Shivers 1991] for PCF as well as an interval analysis [Nielson et al. 1999] for a While language. We were able to define both analyses modularly by describing generic interpreters and analysis components separately. We then changed the While analysis in two different ways to study the impact on the analysis definition and soundness proof. First, we changed the analysis to additionally compute reaching definitions [Nielson et al. 1999] rather than intervals only. Second, we changed the While language to add exception handling. In both cases, changes were confined to a single analysis component and the generic interpreter, whereas the rest of the analysis definition and soundness proofs remained stable.

In summary, we make the following contributions:

- We propose an approach for the modular construction of static analyses from reusable analysis components, which are based on arrow transformers (Section 4.2).
- We define a soundness proposition for analysis components and demonstrate how they can be shown sound in isolation (Section 4.3).
- We develop a theory that explains the horizontal and vertical composition of analysis components and when their soundness is preserved (Section 4.4).
- We prove that a static analysis based on analysis components is sound, if all its analysis components are sound (Section 4.5).
- We provide an open-source library of reusable analysis components in Haskell (Section 4.6).
- We evaluate the applicability and reusability of our components by defining a k-CFA analysis, an interval analysis, and a reaching definitions analysis (Section 4.7).

4.2 Analysis Components By Example

Static analyses mix language concerns, which convolutes their implementation and soundness proof. In this section, we first illustrate the problems that arise when analyses mix concerns, before sketching our solution of analysis components.

4.2.1 Problem Statement

A static analysis is sound if it correctly approximates the concrete semantics. Analysis soundness has been specifically well-studied for abstract interpreters, which need to approximate the concrete interpreter. Unfortunately, the soundness criteria for abstract interpreters requires reasoning about the whole interpreter definition. As we show here, such non-modular reasoning quickly becomes unwieldy, even for simple languages.

For example, consider the following concrete interpreter `run` and abstract interpreter $\widehat{\texttt{run}}$ for a simple WHILE language implemented in Haskell. We only show the case for assignments `Assign`.

```
data Expr = …
data Stmt = Assign Var Expr | If Expr [Stmt] [Stmt] | While Expr [Stmt]

run :: Map Var Addr_1 → Map Addr Val_2 → [Stmt] → Maybe_3 (Map Addr Val_2)
run env_1 store_2 (Assign var expr : rest) = case_3 eval env_1 store_2 expr of
  Just_3 val → case lookup_1 var env_1 of
    Just addr → run env_1 (insert_2 addr val store_2) rest
    Nothing → let addr = alloc env_1 var
      in run (insert_1 var addr env_1) (insert_2 addr val store_2) rest
  Nothing_3 → Nothing_3

^run :: Map Var Addr_1 → ^Map Addr ^Val_2 → Int_4 → [Stmt] → ^Maybe_3 (^Map Addr ^Val_2)
^run _    _      fuel_4  _          | fuel ≤ 0_4 = JustOrNothing_3 ⊤_2
^run env_1 store_2 fuel_4 (Assign var expr : ss) = case_3 ^eval env_1 store_2 expr of
  Just_3 val → case lookup_1 var env_1 of
    Just addr → ^run env_1 (insertWith_2 (⊔) addr val store_2) (fuel-1_4) ss
    Nothing → let addr = ^alloc env_1 store_2 var val
      in ^run (insert_1 var addr env_1) (insertWith_2 (⊔) addr val store_2) (fuel-1_4) ss
  Nothing_3 → Nothing_3
  JustOrNothing_3 val → Nothing_3 ⊔ (... same code as for Just_3 val)
```

The concrete interpreter `run` takes an environment (mapping variables to addresses) and a store (mapping addresses to values), and yields a possibly updated store if the execution does not fail. The interpreter code itself is standard, but we color-coded and enumerated the parts of the code that relate to different concerns: $\texttt{environment}_1$, $\texttt{store}_2$, and $\texttt{failure}_3$.

The abstract interpreter $\widehat{\texttt{run}}$ handles a fourth concern: $\texttt{termination}_4$. In this simple example, we use a fuel counter that we decrease on every recursive call, and we top out when no fuel is left. For the $\texttt{environment}_1$ concern, $\widehat{\texttt{run}}$ uses the same representation and operations as `run`, but addresses may now be shared between variables. For the $\texttt{store}_2$ concern, $\widehat{\texttt{run}}$ uses a representation that maps addresses to an abstract value domain $\widehat{\texttt{Val}}$, and it uses `insertWith` to join values in the store. Finally, for the $\texttt{failure}_3$ concern, $\widehat{\texttt{run}}$ uses a representation with a third alternative `JustOrNothing`.

Even though the analysis $\widehat{\texttt{run}}$ is fairly simple, it highlights two key challenges when developing sound static analyses:

Modular Implementation The code of the analysis $\widehat{\texttt{run}}$ fails to separate concerns and mixes them with language-specific code. That is, all concerns are directly addressed in the analysis code and there is high coupling. This entails the standard problems [Parnas 1972]: It becomes hard to update the code of one concern without affecting other concerns.

Ideally, we would like to implement each concern separately and independent of $\widehat{\texttt{run}}$ as a reusable component. That is, we would like to hide the implementation of each concern behind an interface and only use that interface in $\widehat{\texttt{run}}$. We could then instantiate $\widehat{\texttt{run}}$ by selecting and composing appropriate components. This would allows us, for example, to exchange

Interface
```
class ExceptOps exc e where
  throw :: e → exc e x
  catch :: exc e y → (e → y) → y
```

Soundness Proof

$\texttt{throw} \mathrel{\dot\sqsubseteq} \widehat{\texttt{throw}}$

$\texttt{catch} \mathrel{\dot\sqsubseteq} \widehat{\texttt{catch}}$

Concrete Instance
```
data Except e x =
  Success x | Fail e

instance ExceptOps Except e where
  throw e = Fail e
  catch exc h = case exc of
    Success x → x
    Fail e    → h e
```

Abstract Instance
```
data Except e x =
  Success x | Fail e | SuccessOrFail x e

instance ExceptOps Except e where
  throw e = Fail e
  catch exc h = case exc of
    Success x         → x
    Fail e            → h e
    SuccessOrFail x e → x ⊔ h e
```

Figure 4.1: Preliminary design of an analysis component for exceptions. We write $\texttt{f} \mathrel{\dot\sqsubseteq} \widehat{\texttt{f}}$ as a shorthand to say that $\widehat{\texttt{f}}$ soundly approximates `f`.

the implementation for stores without having to think about environments or failures. This would also make it easier to adapt the analysis when the analyzed language changes. Many existing analysis frameworks separate concerns to some extent (e.g., call graph construction and transfer functions), but in a way that precludes addressing the second challenge.

Modular Soundness Proof The entanglement of concerns in $\widehat{\texttt{run}}$ also complicates the soundness proof significantly. In order to show that $\widehat{\texttt{run}}$ soundly approximates `run`, we have to reason about all concerns at once. Moreover, a change to any of the concerns potentially invalidates the entire soundness proof. Essentially, the problems from the implementation are reflected in the soundness proof: It becomes hard to update the code of one concern without affecting the other ones.

Ideally, we would like to prove the soundness of each component separately and independent of $\widehat{\texttt{run}}$. That is, we would like to find a soundness proposition that we can prove separately for each component, and only use that proposition in the soundness proof of $\widehat{\texttt{run}}$. We can then obtain a provably sound analysis by instantiating $\widehat{\texttt{run}}$ with appropriate components, as long as each component satisfies the soundness proposition. One of the key questions is how such a soundness proposition may look like, and how sound components can be composed to yield sound compound components.

In the remainder of this section, we will discuss two designs of components for modularly defined and sound static analyses. The first component design is simple yet fails to address our challenges, illustrating why a good component design is difficult to come by. We resolve these issues in our second component design, which is based on arrow transformers.

4.2.2 A First Attempt to Design Analysis Components

In this subsection, we propose a *preliminary* design for analysis components that addresses parts of the two challenges of the previous subsection. In this preliminary design, an analysis component consists of four parts, as we illustrate in Figure 4.1: An interface describing the operations of the component, a concrete and an abstract instance of the interface to define the concrete and abstract semantics, and a proof that the abstract semantics soundly approximates the concrete semantics for each operation.

We illustrate this design in Figure 4.1 for a component providing exception handling. The interface is parameterized by an exception type `exc e x`, which describes a computation that throws an exception `e` or terminates successfully with `x`. The `catch` operation takes a computation `exc e y` and extracts the value `y` or handles the exception with `(e → y)`. The concrete instance of the component use data type `Except` as exception type and implements the operations in a standard way. The abstract instance uses error type $\widehat{\texttt{Except}}$ that has an extra case $\widehat{\texttt{SuccessOrFail}}$, representing a computation that succeeded or failed. For `SuccessOrFail`, the abstract $\widehat{\texttt{catch}}$ joins (⊔) the outcomes of the success and fail cases. Finally, the component contains a soundness proof for $\widehat{\texttt{throw}}$ and $\widehat{\texttt{catch}}$ (we skip the details for now). This preliminary design of analysis components addresses parts of our design challenges, but not all of them:

Modular Implementation We succeeded in encapsulating analysis concerns in components, and components can be exchanged with other components implementing the same interface. However, our components do not compose. For example, consider the composition of the exception component from above with a component for stores. The problem is that the exception component describes computations of the form `e → y` (see the type of `catch`), where the store component describes computations of type `(store,x) → (store,y)`. To add stores, we would need to change the type of the `catch` operation to thread a store:

```
catch :: (store,exc e y) → (e → (store,y)) → (store,y)
```

This is not modular because the interface for the exception component has changed in an incompatible way and we cannot reuse previous implementations. To resolve this, we need to make the exception component parametric in the shape of computations so that it can accommodate effects (like store passing) imposed by other components.

Modular Soundness Proof We succeeded in making the soundness of each analysis component separately provable. For example, we can show that $\widehat{\texttt{catch}}$ soundly approximates `catch` given a standard Galois connection between `Except` and $\widehat{\texttt{Except}}$. But as long as the composition of components requires changes to the interface or instances to accommodate new effects, previous soundness proofs become void. The question is what happens when we follow our plan of making components parametric in the shape of computations. This will require us to make the soundness proofs parametric, too, meaning we need to proof soundness independent of effects imposed by other components.

In the following subsection, we refine our first design to address both challenges.

4.2.3 Arrow-Based Analysis Components

In the previous subsection, we presented a preliminary design for analysis components that supported separation of concerns but failed to support component composition. To make analysis components composable, we abstract over the effects imposed by other components using a higher-order type parameter `c`, which we add to each interface as illustrated in Figure 4.2 for exceptions. The type parameter `c` has kind $* \to * \to *$ and describes computations, that is, `c x y` is a computation with input `x` and output `y`. In the literature, this design is known as *arrows* [Hughes 2000]

Arrows abstract over effects of computations and are a generalization of monads. For example, we can define an arrow `Arr` as `Arr x y = (Store,x) → Except e (Store,y)`, which represents a computation that threads a store and may yield an exception. But, importantly, we can define parametric arrow computations without specifying the exact arrow type. This is similar to monads, which provide `return` and `bind` operations for defining parametric monadic computations. The set of operations for arrows is somewhat larger, but in this paper we will hide the details using the `proc`-notation [Paterson 2001] that is similar to monadic `do`-notation. For example, the implementation of `catch` in the concrete instance of Figure 4.2 uses `proc x → ...` to introduce an arrow computation that binds the input to `x`. Arrow statement `exc ← f ≺ x` runs `f` on input `x` and binds the result to variable `exc`. Function `returnA` has type `(c x x)` and embeds its input as an arrow output.

Keidel et al. [2018] have previously explored the usage of arrows in the definition of abstract interpreters. They showed that it is possible to define an arrow type for the concrete domain and a separate arrow type for the abstract domain, such that a single generic interpreter can be instantiated to yield the concrete and abstract semantics, respectively. They also showed that this design allows compositional soundness proofs, where each operation of the arrows can be verified independently and the soundness of the instantiated interpreters follows by construction. However, Keidel et al. fail our goal: Their arrows only separate concrete from abstract domain but fail to separate concerns like exceptions and stores—they did not consider *analysis components*.

Inspired by their work, we use arrows in the interface of our analysis components. However, components can only be composable if their implementations permit effects from other components. To this end, we define the concrete and abstract instances of our analysis components using *arrow transformers*. An arrow transformer wraps an arrow type to impose additional effects. For

<table>
<tr>
<td>Interface
<code>class ArrowExcept e c where</code>
<code> throw :: c e x</code>
<code> catch :: c x y → c (x,e) y → c x y</code></td>
<td>Soundness Proof
$\forall \mathtt{c} \leftrightarrows \widehat{\mathtt{c}},$
$\quad \mathtt{throw}_{\mathtt{c}} \mathrel{\dot\sqsubseteq} \widehat{\mathtt{throw}}_{\widehat{\mathtt{c}}}$
$\quad \mathtt{catch}_{\mathtt{c}} \mathrel{\dot\sqsubseteq} \widehat{\mathtt{catch}}_{\widehat{\mathtt{c}}}$</td>
</tr>
<tr>
<td>Concrete Instance
<code>data Except e x =</code>
<code> Success x | Fail e</code>
<code>type ExceptT e c x y = c x (Except e y)</code>

<code>instance ArrowExcept e (ExceptT e c)</code>
<code> throw = proc e → returnA ≺ Fail e</code>
<code> catch f h = proc x → do</code>
<code>   exc ← f ≺ x</code>
<code>   case exc of</code>
<code>   Success x → returnA ≺ x</code>
<code>   Fail e   → h ≺ (x,e)</code></td>
<td>Abstract Instance
<code>data</code> $\widehat{\mathtt{Except}}$ <code>e x =</code>
$\quad\widehat{\mathtt{Success}}$ <code>x |</code> $\widehat{\mathtt{Fail}}$ <code>e |</code> $\widehat{\mathtt{SuccessOrFail}}$ <code>x e</code>
<code>type</code> $\widehat{\mathtt{ExceptT}}$ <code>e c x y = c x (</code>$\widehat{\mathtt{Except}}$ <code>e y)</code>

<code>instance ArrowExcept e (</code>$\widehat{\mathtt{ExceptT}}$ <code>e c)</code>
$\quad\widehat{\mathtt{throw}}$ <code>= proc e → returnA ≺</code> $\widehat{\mathtt{Fail}}$ <code>e</code>
$\quad\widehat{\mathtt{catch}}$ <code>f h = proc x → do</code>
<code>   exc ← f ≺ x</code>
<code>   case exc of</code>
$\qquad\widehat{\mathtt{Success}}$ <code>y → returnA ≺ y</code>
$\qquad\widehat{\mathtt{Fail}}$ <code>e → h ≺ (x,e)</code>
$\qquad\widehat{\mathtt{SuccessOrFail}}$ <code>y e → (returnA ≺ y)</code>
<code>   ⊔ (h ≺ (x,e))</code></td>
</tr>
</table>

Figure 4.2: An arrow-based analysis component for exceptions.

example, the concrete instance in Figure 4.2 uses arrow transformer `ExceptT e c`, which adds exceptions of type `e` to the output of a computation `c`. We do the same in the abstract instance using arrow transformer $\widehat{\mathtt{ExceptT}}$ `e c`. Using arrow transformers, the implementation of the concrete and abstract operations is parametric in the underlying arrow except for the locally added effect (here: the propagation and representation of exceptions). From now on, we use the notation $\langle \mathtt{ExceptT}, \widehat{\mathtt{ExceptT}} \rangle_{\mathtt{ArrowExcept}}$ to refer to analysis components.

The revised design based on arrow-transformers addresses both of our design goals, where our preliminary design fell short:

Modular Implementation Each component encapsulates a single analysis concern and is exchangeable with other components implementing the same interface. However, in contrast to our preliminary design, analysis components based on arrow transformers are composable. For example, we can compose the exception component $\langle \mathtt{ExceptT}, \widehat{\mathtt{ExceptT}} \rangle_{\mathtt{ArrowExcept}}$ with a store component $\langle \mathtt{StoreT}, \widehat{\mathtt{StoreT}} \rangle_{\mathtt{ArrowStore}}$ to obtain a component which combines both effects:

$$\begin{aligned} &\langle \mathtt{ExceptT}, \widehat{\mathtt{ExceptT}} \rangle_{\mathtt{ArrowExcept}} \circ \langle \mathtt{StoreT}, \widehat{\mathtt{StoreT}} \rangle_{\mathtt{ArrowStore}} \\ &\quad = \langle \mathtt{ExceptT} \circ \mathtt{StoreT}, \widehat{\mathtt{ExceptT}} \circ \widehat{\mathtt{StoreT}} \rangle_{\mathtt{ArrowExcept+ArrowStore}} \end{aligned}$$

Specifically, the composition of arrow transformers stacks their effects (`ExceptT e (StoreT c)`), while the composition of interfaces combines all operations in a new interface. We do not need to change the implementation of either component. Like monad transformers [Liang et al. 1995], the composition of arrow transformers requires a lifting of the inner arrow transformer to the outermost transformer. We show in our evaluation that these liftings are mostly boilerplate and can be derived automatically.

Modular Soundness Proof Like in the preliminary design, we can prove soundness of an arrow-based component by proving each operation of the interface sound. However, since we use arrow transformers, components are parametric in the effects of other components and the soundness proof must be parametric as well. That is, we must show that $\widehat{\mathtt{throw}}_{\widehat{\mathtt{c}}}$ is sound with respect to $\mathtt{throw}_{\mathtt{c}}$ for any related arrows $\widehat{\mathtt{c}}$ and `c`. We found that such generic soundness proofs are feasible and we provide a large library of provably sound analysis components in Section 4.6. Sound analysis components following our design are freely composable and their composition always remains sound. We provide the formal results in the upcoming sections.

4.2.4 Instantiating Concrete and Abstract Interpreters

Using the analysis components described in the previous subsection, we can refactor the concrete and abstract interpreter of Section 4.2.1. First, we extract a generic interpreter as proposed by Keidel et al. [2018] to capture the similarities of the concrete and abstract interpreter. In contrast to Keidel et al., we parameterize the generic interpreter using the interfaces of analysis components:

```
runGeneric :: (ArrowEnv String addr c, ArrowStore addr val c, ArrowExcept e c, ArrowFix c)
    ⇒ c [Statement] ()
runGeneric = fix $ λrun' → proc stmts → case stmts of
  Assign var expr : rest → do
    val ⟵ eval ≺ expr
    addr ⟵ lookup id alloc ≺ var
    write ≺ (addr,val)
    local run' ≺ (var,addr,rest)
  ...
```

The second step of our refactoring is to compose analysis components implementing all required interfaces of the generic interpreter:

$$\langle \texttt{EnvT}, \widehat{\texttt{EnvT}}\rangle_{\texttt{ArrowEnv}} \circ \langle \texttt{StoreT}, \widehat{\texttt{StoreT}}\rangle_{\texttt{ArrowStore}} \circ \langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}}\rangle_{\texttt{ArrowExcept}} \circ \langle \texttt{Fix}, \widehat{\texttt{Fix}}\rangle_{\texttt{ArrowFix}}$$

The order in which the analysis components are composed matters. For example, in the order above, we obtain a language semantics in which store updates in a `try` block are reset whenever an exception occurs. In contrast, if we swap the order of the store and exception component, we obtain a language semantics in which store updates persist whenever an exception occurs.

The third and final step is to obtain the original concrete and abstract interpreters from Section 4.2.1 by instantiating the generic interpreter. The concrete slice of the composed analysis component yields the concrete interpreter, the abstract slice yields the abstract interpreter. In Haskell it suffices to specify the desired interpreter type and let the implicit type-class inference select the correct component instances:

```
run :: EnvT (StoreT (ExceptT Fix)) [Stmt] ()      run^ :: EnvT^ (StoreT^ (ExceptT^ Fix^)) [Stmt] ()
run = runGeneric                                  run^ = runGeneric
```

As we show in the following sections, we obtain that an abstract interpreter $\widehat{\texttt{run}}$ soundly approximates a concrete interpreter `run` if they are instances of the same generic interpreter and the fully-composed analysis component is sound. The fully-composed analysis component is sound if all individual analysis components are sound. Thus, analysis soundness follows directly from using sound analysis components.

4.3 Analysis Components And Their Soundness

To be able to rely on the results of an analysis, the analysis has to be proven sound. In this section, we describe our analysis components formally and how to prove them sound.

To set the stage, let us first recall the definition of soundness [Cousot 1999]: A concrete function $f : A \longrightarrow B$ is sound with respect to an abstract function $\widehat{f} : \widehat{A} \longrightarrow \widehat{B}$, if all behavior of f is overapproximated by $\widehat{f}$. More formally, let $\alpha_A : \mathcal{P}A \leftrightarrows \widehat{A} : \gamma_A$ and $\alpha_B : \mathcal{P}B \leftrightarrows \widehat{B} : \gamma_B$ be Galois connections between concrete and abstract domains, then

$$f \text{ is sound w.r.t. } \widehat{f} \quad \text{iff} \quad \forall X \in \mathcal{P}A.\ \alpha_B\{f(x) \mid x \in X\} \sqsubseteq \widehat{f}(\alpha_A(X)).$$

Here the powersets $\mathcal{P}A$ and $\mathcal{P}B$ describe properties of the concrete domain. Given such a property $X \in \mathcal{P}A$ about the inputs of f, the set $\{f(x) \mid x \in X\}$ describes the strongest post-condition of f for the pre-condition X.

This soundness proposition is relative to the Galois connection (α_A, γ_A) and (α_B, γ_B). These Galois connections describe how concrete properties $\mathcal{P}A$ and $\mathcal{P}B$ correspond to abstract values $\widehat{A}$ and $\widehat{B}$. Choosing a Galois connection and an ordering of the abstract domains $\widehat{A}$ and $\widehat{B}$ is part of the analysis design and different Galois connections provide different soundness guarantees. In the following subsection, we describe how to construct Galois connections for analysis components.

4.3.1 Galois Connections between Analysis Components

Proving soundness requires a Galois connection that relates a concrete domain A to an abstract domain $\widehat{A}$. A Galois connection [Ore 1944] between two preorders A and $\widehat{A}$ consists of a pair of

monotone functions (α, γ), where $\alpha : A \to \widehat{A}$ is called the *abstraction function* and $\gamma : \widehat{A} \to A$ is called the *concretization function*, such that

$$\forall x \in A, \widehat{x} \in \widehat{A}.\ \alpha(x) \sqsubseteq_{\widehat{A}} \widehat{x} \text{ iff } x \sqsubseteq_A \gamma(\widehat{x}).$$

Our analysis components consist of operations over a pair of arrow transformers. To relate these operations in a soundness proof, we need to define a Galois connection between these arrow transformers. However, we first describe the shape of Galois connections for regular arrows, because this will guide the design of Galois connections for arrow transformers.

In the following, we use the notation $\mathcal{G}(A, \widehat{A})$ to denote the set of all Galois connections between the types A and $\widehat{A}$. An arrow C is a function that constructs the type of a computation $\mathrm{C}(A, B)$, whose input type is A and output type is B. Analogously, a Galois connection between two arrows [Keidel et al. 2018] C and $\widehat{\mathrm{C}}$ is a function that takes a Galois connection between the inputs $\mathcal{G}(\mathcal{P}A, \widehat{A})$ and outputs $\mathcal{G}(\mathcal{P}B, \widehat{B})$ and constructs Galois connection between the arrow types:

$$\forall A, \widehat{A}, B, \widehat{B}.\quad \mathcal{G}(\mathcal{P}A, \widehat{A}) \times \mathcal{G}(\mathcal{P}B, \widehat{B}) \to \mathcal{G}(\mathrm{C}(A, B), \widehat{\mathrm{C}}(\widehat{A}, \widehat{B})).$$

Since our analysis components consist of arrow transformers, we need to generalize the construction further. A Galois connection between arrow transformers T and $\widehat{\mathrm{T}}$ maps a Galois connection between the underlying arrow types C and $\widehat{\mathrm{C}}$ to a Galois connection between the transformed arrow types $\mathrm{T}(\mathrm{C})$ and $\widehat{\mathrm{T}}(\widehat{\mathrm{C}})$.

Definition 4.3.1. A Galois connection for an analysis component $\langle \mathrm{T}, \widehat{\mathrm{T}} \rangle$ is a Galois connection between the two arrow transformers T and $\widehat{\mathrm{T}}$. It has the following type:

$$\begin{aligned}\forall \mathrm{C}, \widehat{\mathrm{C}}.\ &\left[\forall A, \widehat{A}, B, \widehat{B}.\, \mathcal{G}(\mathcal{P}A, \widehat{A}) \times \mathcal{G}(\mathcal{P}B, \widehat{B}) \to \mathcal{G}(\mathrm{C}(A, B), \widehat{\mathrm{C}}(\widehat{A}, \widehat{B}))\right] \\ \to &\left[\forall A, \widehat{A}, B, \widehat{B}.\, \mathcal{G}(\mathcal{P}A, \widehat{A}) \times \mathcal{G}(\mathcal{P}B, \widehat{B}) \to \mathcal{G}(\mathrm{T}(\mathrm{C})(A, B), \widehat{\mathrm{T}}(\widehat{\mathrm{C}})(\widehat{A}, \widehat{B}))\right].\end{aligned}$$

That is, given a Galois connection between the underlying arrow types C and $\widehat{\mathrm{C}}$, the function constructs a Galois connection between the arrows $\mathrm{T}(\mathrm{C})$ and $\widehat{\mathrm{T}}(\widehat{\mathrm{C}})$.

The design of an analysis component crucially depends on the choice of Galois connection. The Galois connection dictates how the component's abstract arrow transformer needs to approximate the concrete transformer. Moreover, the Galois connection is not uniquely determined; different Galois connections provide different soundness guarantees. Therefore, the developer of an analysis component also has to specify the corresponding Galois connection in accordance with Definition 4.3.1.

Example 4.3.1. For example, a Galois connection between two exception arrow transformers $\mathtt{ExceptT}_E(\mathrm{C})(A, B) = \mathrm{C}(A, \mathtt{Error}\ E\ B)$ and $\widehat{\mathtt{ExceptT}}_{\widehat{E}}(\mathrm{C})(A, B) = \mathrm{C}(A, \widehat{\mathtt{Error}}\ \widehat{E}\ B)$ has the type

$$\begin{aligned}\forall \mathrm{C}, \widehat{\mathrm{C}}.\ &\left[\forall A, \widehat{A}, B, \widehat{B}.\, \mathcal{G}(\mathcal{P}A, \widehat{A}) \times \mathcal{G}(\mathcal{P}B, \widehat{B}) \to \mathcal{G}(\mathrm{C}(A, B), \widehat{\mathrm{C}}(\widehat{A}, \widehat{B}))\right] \\ \to &\left[\forall A, \widehat{A}, B, \widehat{B}.\, \mathcal{G}(\mathcal{P}A, \widehat{A}) \times \mathcal{G}(\mathcal{P}B, \widehat{B}) \to \mathcal{G}(\mathtt{Except}_E(\mathrm{C})(A, B), \mathtt{Except}_{\widehat{E}}(\widehat{\mathrm{C}})(\widehat{A}, \widehat{B}))\right].\end{aligned}$$

To construct this Galois connection, we extend the Galois connections for domain and codomain to include the extra data of the exception arrow transformer. From a Galois connection (α_E, γ_E) between the exception types and a Galois connection (α_B, γ_B) between the codomains, we can easily derive Galois connections $(\alpha_{\mathtt{Error}}, \gamma_{\mathtt{Error}}) \in \mathcal{G}(\mathcal{P}(\mathtt{Error}\ E\ B), \widehat{\mathtt{Error}}\ \widehat{E}\ \widehat{B})$ for the codomain of the exception transformers. Then from the Galois connection $(\alpha_{\mathrm{C}}, \gamma_{\mathrm{C}})$ between the underlying arrows and (α_A, γ_A) between the domains, we construct the Galois connection of the underlying exception transformer types:

$$\alpha_{\mathtt{ExceptT}} = \alpha_{\mathrm{C}}((\alpha_A, \gamma_A), (\alpha_{\mathtt{Error}}, \gamma_{\mathtt{Error}})) \qquad \gamma_{\mathtt{ExceptT}} = \gamma_{\mathrm{C}}((\alpha_A, \gamma_A), (\alpha_{\mathtt{Error}}, \gamma_{\mathtt{Error}}))$$

Importantly, all Galois connections of analysis components have the same type shown in Definition 4.3.1. This allows us to compose these Galois connections with regular function composition. This becomes important, when we compose analysis components, which we discuss in Section 4.4. With Galois connections between arrow transformers, we can develop the soundness proposition of analysis components.

4.3.2 Soundness of Analysis Components

In this subsection, we describe how to prove soundness of analysis components, and what soundness means exactly. In particular, we develop a theory for analysis components and their soundness proofs that allows us to express soundness of arrow operations of arbitrary arity and type. To this end, we first describe our analysis components more formally.

An analysis component consists of a type class describing the interface of the component and two instances for two arrow transformers. Type classes and their instances can be described by algebras for a functor [Hamana and Fiore 2011]. The functor describes the codomain of each operation of the type class. For arrow type classes, this functor has type $\mathrm{Set}^{\mathcal{U}\times\mathcal{U}} \to \mathrm{Set}^{\mathcal{U}\times\mathcal{U}}$, i.e., it maps arrow types to arrow types. For example, we can describe the type class `ArrowExcept` in Figure 4.2 with the functor

$$\texttt{ArrowExcept}_E(\mathrm{C})(X,Y) = [X \equiv E] + [\mathrm{C}(X,Y) \times \mathrm{C}(X \times E, Y)]\,.$$

The first operand of the coproduct describes the type of `throw` and the second operand the arguments of `catch`. An algebra over a functor F is a function of type $\forall X, Y.\, F(\mathrm{C})(X,Y) \to \mathrm{C}(X,Y)$. This function combines all operations of the type class. In case of `ArrowExcept`, the algebra combines a computation $\mathrm{C}(E,Y)$ for `throw` and a function $\mathrm{C}(X,Y)\times\mathrm{C}(X\times E,Y) \to \mathrm{C}(X,Y)$ for `catch`. In addition, the functor is parameterized by other arguments of the type class. For example, the functor $\texttt{ArrowExcept}_E$ is parameterized by the type of exceptions E.

With this theory, we define our analysis components formally as follows:

Definition 4.3.2 (Analysis Component). An analysis component $\langle f, \widehat{f}\rangle : \langle \mathrm{T}, \widehat{\mathrm{T}}\rangle_F$ is a pair of algebras $\langle f, \widehat{f}\rangle$ over a functor $F : \mathrm{Set}^{\mathcal{U}\times\mathcal{U}} \to \mathrm{Set}^{\mathcal{U}\times\mathcal{U}}$, where $\langle f, \widehat{f}\rangle$ is a pair of functions $f_{\mathrm{C}} : F(\mathrm{T}(\mathrm{C}))(X,Y) \to \mathrm{T}(\mathrm{C})(X,Y)$ and $\widehat{f}_{\widehat{\mathrm{C}}} : F(\widehat{\mathrm{T}}(\widehat{\mathrm{C}}))(X,Y) \to \widehat{\mathrm{T}}(\widehat{\mathrm{C}})(X,Y)$.

In this definition, F defines the interface of the analysis components, f and $\widehat{f}$ implement the interface for arrow transformers T and $\widehat{\mathrm{T}}$.

This formal definition of analysis components allows us to define their soundness proposition precisely:

Definition 4.3.3 (Soundness of Analysis Components). Given an analysis component $\langle f, \widehat{f}\rangle : \langle \mathrm{T}, \widehat{\mathrm{T}}\rangle_F$ and a Galois connection for the arrow transformers of this analysis component, then the analysis component $\langle \mathrm{T}, \widehat{\mathrm{T}}\rangle_F$ is sound iff all operations of F are pair-wise sound in f and $\widehat{f}$ according to the Galois connection. More formally, let $F = F_1 + \ldots + F_n$ be the functor representing a type class, $f = f_1 \ldots f_n$ be the algebra representing the concrete operations, and $\widehat{f} = \widehat{f}_1 \ldots \widehat{f}_n$ be the algebra representing the abstract operations. Then f is sound w.r.t. $\widehat{f}$ iff

$$\forall x, \widehat{x}. \quad \alpha_{F(\mathrm{T})}(x) \sqsubseteq \widehat{x} \implies \alpha_{\mathrm{T}}(f_i(x)) \sqsubseteq \widehat{f}_i(\widehat{x}) \quad \text{for all } 1 \le i \le n.$$

In other words, an analysis component is sound if each operation preserves soundness of their arguments. For example, in the case of the `ArrowExcept` component in Figure 4.2 we have to prove the following two lemmas.

$$\alpha(\texttt{throw}) \sqsubseteq \widehat{\texttt{throw}}$$
$$\forall f, \widehat{f}, g, \widehat{g}. \quad \alpha(f,g) \sqsubseteq (\widehat{f}, \widehat{g}) \implies \alpha(\texttt{catch}(f,g)) \sqsubseteq \widehat{\texttt{catch}}(\widehat{f}, \widehat{g})$$

That is, we prove that `throw` is sound w.r.t. $\widehat{\texttt{throw}}$ and $\texttt{catch}(f,g)$ w.r.t. $\widehat{\texttt{catch}}(\widehat{f},\widehat{g})$ given sound continuations $f, \widehat{f}, g, \widehat{g}$. In contrast to Keidel et al. [2018], these soundness lemmas for the `ArrowExcept` component are reusable because of the following reasons.

- The operations are defined over arrow transformers $\texttt{Except}_E(\mathrm{C})$ and $\texttt{Except}_{\widehat{E}}(\widehat{\mathrm{C}})$ and the proofs are universal in the underlying arrows C and $\widehat{\mathrm{C}}$, which allows us to swap out the underlying arrows when we compose this component.
- The proofs are universal in the exception types E and $\widehat{E}$, which allows us to use this component and proofs in languages with different exception types.

But how do we actually prove these lemmas if we do not know the underlying arrows C and $\widehat{C}$? We need to establish a base-line, which allows us to reason about generic arrows in these soundness proofs. Fortunately, arrows already provide a basic reasoning tool-kit: the algebraic arrow laws [Hughes 2000]. To illustrate how such a proof works, we include proofs in the supplementary material accompanying this paper. These proofs show that it is feasible to reason about soundness of arrow operations over arrow transformers without knowing the underlying arrows C and $\widehat{C}$. The only assumptions we had to make about the arrow operations of C and $\widehat{C}$, was they are sound, monotone and obey the arrow laws (Section 2.2.3). We demonstrate in the following section how these assumptions are preserved under composition of components.

To summarize, in this section, we developed a generic theory to prove soundness of analysis components once and for all. These soundness proofs are reusable, because they are specific to arrow transformers rather than specific to monolithic arrows. In the following section, we explain a different way to define sound analysis components from existing analysis components.

4.4 Sound Composition Of Analysis Components

In Sections 4.2.2 and 4.2.3 we showed that we need to *compose* analysis components to combine their effects and to explain how their effects interact. In this section, we describe three different ways for composing analysis components and prove them sound.

4.4.1 Horizontal Composition

The simplest way of composition occurs when the arrow transformers of a component implement multiple interfaces $\langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_F$ and $\langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_G$. For example, the transformers `ExceptT` and $\widehat{\texttt{ExceptT}}$ defined in Figure 4.2 do not support a `finally f g` operation that executes `g` no matter if `f` succeeds or fails. We capture this operation in a new interface:

```
class ArrowFinally c where
  finally :: c x y → c x () → c x y
```

We implement this operation in another component $\langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}} \rangle_{\texttt{ArrowFinally}}$ with the same arrow transformers. We horizontally compose $\langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}} \rangle_{\texttt{ArrowExcept}}$ with the component $\langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}} \rangle_{\texttt{ArrowFinally}}$ to obtain the functionality of both components in a new component $\langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}} \rangle_{\texttt{ArrowExcept+ArrowFinally}}$. More formally:

Definition 4.4.1 (Horizontal Composition). The horizontal composition $\langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_F \oplus \langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_G$ of two analysis components $\langle f, \widehat{f} \rangle : \langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_F$ and $\langle g, \widehat{g} \rangle : \langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_G$ is defined as $\langle f + g, \widehat{f} + \widehat{g} \rangle : \langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_{F+G}$.

Furthermore, this composition preserves soundness of components:

Theorem 4.4.1 (Horizontal composition preserves soundness). *Given sound analysis components $\langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_F$ and $\langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_G$, their horizontal composition $\langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_F \oplus \langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_G$ is sound.*

Proof. Follows directly by 4.3.3, because we can separately prove soundness of each operation in the interface F and in G. □

To summarize, horizontal composition allows us to compose components with the same arrow transformers that implement different interfaces.

4.4.2 Component Lifting

In general, analysis components use different arrow transformers to implement different interfaces $\langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_F$ and $\langle \mathrm{U}, \widehat{\mathrm{U}} \rangle_G$. We detail how to compose such components using *vertical composition* in the next subsection. Vertical composition means that we stack one component on top of the other, effectively wrapping the nested component. Here, we discuss an important preliminary for vertical composition, namely the lifting of operations of the nested component through the wrapping.

To compose components vertically, we need to compose their arrow transformers: $\mathrm{T} \circ \mathrm{U} = \forall \mathrm{C}.\, \mathrm{T}(\mathrm{U}(\mathrm{C}))$. Similar to *monad transformers* [Liang et al. 1995, Section 8], to make the operations of U available for the composed arrow transformers $\mathrm{T} \circ \mathrm{U}$, we need to *lift* them through T. The reason is that we cannot interact with the inner arrow transformer U directly; all interaction with U has to go through T.

For example, to make the `ArrowExcept` operations defined by `ExceptT` available in `StoreT` ∘ `ExceptT`, we need to lift `throw` and `catch` through `StoreT`. This lifting explains how store passing interacts with exception handling. To this end, we have to implement a lifting instances that explains how and when `StoreT` provides `ArrowExcept` operations:

```
instance ArrowExcept e (StoreT (ExceptT e c)) where
  throw = Store (proc (_,e) → throw ≺ e)
  catch (Store f) (Store g) = Store $ catch f (proc ((s,x),e) → g ≺ (s,(x,e)))
```

This instance allows `Store` to provide `ArrowExcept` operations whenever `Store` is applied to `Except`. The lifting then delegates to the operations of the nested `Except` transformer.

But this lifting is not reusable, because it is coupled to the composition `StoreT` ∘ `ExceptT`. If we want to replace one of the transformers or if there is another transformer in between `StoreT` and `ExceptT`, the lifting fails to work. Therefore, we generalize the lifting definition to precisely capture when `StoreT` can provide `ArrowExcept` operations:

```
instance ArrowExcept e c ⇒ ArrowExcept e (StoreT c)
```

More specifically, `StoreT` provides `ArrowExcept` operations whenever the underlying arrow `c` provides `ArrowExcept` operations. The implementation of the operations stays the same. This lifting is more reusable because it is neither coupled to the arrow transformer `Except` nor its position in the transformer stack.

Formally, a lifting of operations in $\langle \mathrm{U}, \widehat{\mathrm{U}} \rangle_F$ through the transformers $\langle T, \widehat{T} \rangle$ corresponds to a pair of functions $\langle \delta_{\mathrm{U}}, \widehat{\delta}_{\widehat{\mathrm{U}}} \rangle : \langle \mathrm{U}, \widehat{\mathrm{U}} \rangle_F \longrightarrow \langle \mathrm{T} \circ \mathrm{U}, \widehat{\mathrm{T}} \circ \widehat{\mathrm{U}} \rangle_F$, where δ_{U} lifts the concrete part of the component and $\widehat{\delta}_{\widehat{\mathrm{U}}}$ the abstract part of the component. As discussed above, to make this lifting reusable, δ_{U} needs to be parametric in U and $\widehat{\delta}_{\widehat{\mathrm{U}}}$ parametric in $\widehat{\mathrm{U}}$.

A lifting of components is sound if the functions $\langle \delta, \widehat{\delta} \rangle$ preserve soundness:

Definition 4.4.2 (Soundness of Component Liftings). A lifting $\langle \delta, \widehat{\delta} \rangle : \langle \mathrm{U}, \widehat{\mathrm{U}} \rangle_F \longrightarrow \langle \mathrm{T} \circ \mathrm{U}, \widehat{\mathrm{T}} \circ \widehat{\mathrm{U}} \rangle_F$ is sound iff the component $\langle \delta(f), \widehat{\delta}(\widehat{f}) \rangle : \langle \mathrm{T} \circ \mathrm{U}, \widehat{\mathrm{T}} \circ \widehat{\mathrm{U}} \rangle_F$ is sound for all sound components $\langle f, \widehat{f} \rangle : \langle \mathrm{U}, \widehat{\mathrm{U}} \rangle_F$.

In general, each lifting has to be shown sound separately. In particular, because liftings are not unique for a pair of arrow transformers, we cannot formulate a generic soundness theorem. However, in many cases we obtain a proof with less or no effort, which we discuss in the following.

Reusable liftings. As described above, we can make liftings reusable by abstracting over the underlying arrow and only specifying minimal requirements. We use this technique extensively to limit the number of lifting instances and soundness arguments needed.

Generic liftings. First-order operations of type `c x y` often can be lifted with a generic `lift` operation:

```
class ArrowLift t c where
  lift :: c x y → t c x y
```

For example, the `throw` operation that throws an exception can be lifted through the `Store` arrow transformer with this generic `lift` operation:

```
instance ArrowExcept e c ⇒ ArrowExcept e (StoreT c) where
  throw = lift throw
```

It suffices to show the soundness of the generic `lift` operation to ensure all its use cases are sound.

Derivable liftings. Often concrete and abstract arrow transformer are implemented with the same arrow transformer. For example, the `StoreT` and $\widehat{\texttt{StoreT}}$ arrow transformers are both implemented with the `StateT` arrow transformer:

```
newtype StateT s c x y = StateT (c (s,x) (s,y))
newtype StoreT c = StoreT (StateT Store c x y)
newtype StoreT c = StoreT (StateT Store c x y)
```

In this case, a lifting for the `StoreT` and $\widehat{\texttt{StoreT}}$ arrow transformers can be derived automatically by Haskell [Marlow 2010] from the lifting defined on the underlying `StateT` arrow transformer.

```
deriving instance ArrowExcept e (StoreT c)
deriving instance ArrowExcept e (StoreT c)
```

Furthermore, the liftings for both arrow transformers share the same implementation and all differences of concrete and abstract store type are universally quantified. Therefore, soundness of this component lifting follows as a free theorem of parametricity [Keidel et al. 2018, Theorem 5].

We evaluate how many of these liftings fall into either of these categories in Section 4.7.

To summarize, to compose two analysis components with differing arrow transformers, we need to *lift* the operations of the inner arrow transformers through the outer arrow transformers. Such a lifting is sound if it preserves soundness of the underlying component. In general, soundness of these liftings needs to be proven manually, however, often we obtain a soundness proof with less or no effort if we can reuse the same lifting operation or share the implementation of the lifting. Equipped with component lifting, we can support the vertical composition of analysis components.

4.4.3 Vertical Composition of Analysis Components

In the remainder of this section, we discuss how to combine independent analysis components $\langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_F$ and $\langle \mathrm{U}, \widehat{\mathrm{U}} \rangle_G$ using vertical composition. Our goal is to obtain a new analysis component that implements both interfaces F and G based on the functionality of all involved arrow transformers. The key idea of vertical composition is to first lift one component and then to use horizontal lifting on the result.

For example, to obtain an analysis for store passing and exception handling, we compose the components $\langle \texttt{StoreT}, \widehat{\texttt{StoreT}} \rangle_{\texttt{ArrowStore}}$ and $\langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}} \rangle_{\texttt{ArrowExcept}}$. The order in which we compose these components matters. For example, the order `StoreT` ∘ `ExceptT` determines that store updates are reset while the order `ExceptT` ∘ `StoreT` determines that store updates propagate when an exception occurs.

The composition $\langle \texttt{StoreT}, \widehat{\texttt{StoreT}} \rangle_{\texttt{ArrowStore}} \circ \langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}} \rangle_{\texttt{ArrowExcept}}$ of these two components requires a combination of techniques we presented in the previous two subsections:

1. We lift the operations of $\langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}} \rangle_{\texttt{ArrowExcept}}$ through $\langle \texttt{StoreT}, \widehat{\texttt{StoreT}} \rangle$ to obtain a component $\langle \texttt{StoreT} \circ \texttt{ExceptT}, \widehat{\texttt{StoreT}} \circ \widehat{\texttt{ExceptT}} \rangle_{\texttt{ArrowExcept}}$.
2. We specialize the generic arrow transformer types of $\langle \texttt{StoreT}, \widehat{\texttt{StoreT}} \rangle_{\texttt{ArrowStore}}$ to obtain a component $\langle \texttt{StoreT} \circ \texttt{ExceptT}, \widehat{\texttt{StoreT}} \circ \widehat{\texttt{ExceptT}} \rangle_{\texttt{ArrowStore}}$.
3. Finally, we horizontally compose both components to obtain a component with the operations of both interfaces: $\langle \texttt{StoreT} \circ \texttt{ExceptT}, \widehat{\texttt{StoreT}} \circ \widehat{\texttt{ExceptT}} \rangle_{\texttt{ArrowStore+ArrowExcept}}$.

The lifting in this composition is *glue code* which describes how the two components interact.

More formally, we define vertical composition of analysis components with glue code as follows:

Definition 4.4.3 (Vertical Composition). The vertical composition $\langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_F \circ_\Delta \langle \mathrm{U}, \widehat{\mathrm{U}} \rangle_G$ of two analysis components $\langle f, \widehat{f} \rangle : \langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_F$ and $\langle g, \widehat{g} \rangle : \langle \mathrm{U}, \widehat{\mathrm{U}} \rangle_G$ and a lifting $\Delta = \langle \delta, \widehat{\delta} \rangle : \langle \mathrm{U}, \widehat{\mathrm{U}} \rangle_G \to \langle \mathrm{T} \circ \mathrm{U}, \widehat{\mathrm{T}} \circ \widehat{\mathrm{U}} \rangle_G$ is defined as

$$\langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_F \circ_\Delta \langle \mathrm{U}, \widehat{\mathrm{U}} \rangle_G : \langle \mathrm{T} \circ \mathrm{U}, \widehat{\mathrm{T}} \circ \widehat{\mathrm{U}} \rangle_{F+G}$$
$$\langle f, \widehat{f} \rangle \circ_\Delta \langle g, \widehat{g} \rangle = \langle f, \widehat{f} \rangle \oplus (\Delta \langle g, \widehat{g} \rangle) = \langle f + \delta(g), \widehat{f} + \widehat{\delta}(\widehat{g}) \rangle.$$

That is, we lift the operations of $\langle \mathrm{U}, \widehat{\mathrm{U}} \rangle_G$ through $\langle \mathrm{T}, \widehat{\mathrm{T}} \rangle$ and horizontally compose the resulting components. This brings us to the main soundness theorem for the composition of analysis components.

Theorem 4.4.2 (Vertical composition preserves soundness). *Given sound analysis components* $\langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_F$ *and* $\langle \mathrm{U}, \widehat{\mathrm{U}} \rangle_G$ *and a sound lifting* $\Delta : \langle \mathrm{U}, \widehat{\mathrm{U}} \rangle_G \rightarrow \langle \mathrm{T} \circ \mathrm{U}, \widehat{\mathrm{T}} \circ \widehat{\mathrm{U}} \rangle_G$*, then the vertical composition* $\langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_F \circ_\Delta \langle \mathrm{U}, \widehat{\mathrm{U}} \rangle_G$ *is sound.*

Proof. The lifted component $\langle \mathrm{T} \circ \mathrm{U}, \widehat{\mathrm{T}} \circ \widehat{\mathrm{U}} \rangle_G$ is sound because the lifting Δ preserves soundness (4.4.2) and its input $\langle \mathrm{U}, \widehat{\mathrm{U}} \rangle_G$ is sound. Furthermore, the specialized component $\langle \mathrm{T} \circ \mathrm{U}, \widehat{\mathrm{T}} \circ \widehat{\mathrm{U}} \rangle_F$ is sound because $\langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_F$ is parametric in the underlying arrow. Then by Theorem 4.4.1 the horizontal composition $\langle \mathrm{T} \circ \mathrm{U}, \widehat{\mathrm{T}} \circ \widehat{\mathrm{U}} \rangle_F \oplus \langle \mathrm{T} \circ \mathrm{U}, \widehat{\mathrm{T}} \circ \widehat{\mathrm{U}} \rangle_G$ is sound and hence $\langle \mathrm{T}, \widehat{\mathrm{T}} \rangle_F \circ_\Delta \langle \mathrm{U}, \widehat{\mathrm{U}} \rangle_G$ is sound. □

To summarize, in this section we discussed how to soundly compose analysis components to obtain a complete analyses. The composition of two components with differing arrow transformers and different interfaces requires some glue code that explains how the effects of these components interact. As for the components themselves, the definition and soundness proof of glue code can be reused, facilitating the easy construction of provably sound static analyzers.

4.5 Soundness Of Component-Based Static Analyses

The focus in the paper so far has been on analysis components themselves. However, analysis components alone do not describe complete static analyses. In this section, we describe how to use analysis components to define complete static analyses. Finally, we prove that any static analysis, that is based on sound analysis components, is sound.

To use analysis components to describe a static analysis, we need to describe the semantics of the analyzed language with an arrow-based *generic interpreter* [Keidel et al. 2018] that captures the similarities of concrete and abstract semantics. For example, Listing 3.6 in Chapter 3 and Listing 4.1 in Section 4.7 show the generic interpreters for PCF and a WHILE language. A generic interpreter does not describe the concrete semantics nor a particular abstract semantics. Instead it is a template of the language semantics, that we need to instantiate with suitable arrow instances to obtain the concrete semantics or a particular abstract semantics

To instantiate a generic interpreter with an analysis component, we first compose an analysis component $\langle \mathtt{ConT}, \widehat{\mathtt{AbsT}} \rangle_{\mathtt{I}}$ which matches the interface `I` of the generic interpreter. However, we cannot instantiate the generic interpreter with $\langle \mathtt{ConT}, \widehat{\mathtt{AbsT}} \rangle$ directly because the generic interpreter expects *arrows*, where the analysis component $\langle \mathtt{ConT}, \widehat{\mathtt{AbsT}} \rangle$ consists of *arrow transformers*. To obtain a pair of arrow instances, we apply the analysis component $\langle \mathtt{ConT}, \widehat{\mathtt{AbsT}} \rangle$ to a pair of base arrows $\langle \mathcal{P}_ \rightarrow \mathcal{P}_,\ _ \rightarrow _ \rangle$. From this application we get the collecting semantics [Cousot 1999] $\mathtt{ConT}\ (\mathcal{P}_ \rightarrow \mathcal{P}_)$ of the concrete interpreter and the abstract function space $\widehat{\mathtt{AbsT}}\ (\rightarrow)$ of the abstract interpreter. More importantly, the abstract interpreter $\mathtt{run}_{\widehat{\mathtt{AbsT}}\ (\rightarrow)}$ soundly approximates the concrete collecting semantics $\mathtt{run}_{\mathtt{ConT}\ (\mathcal{P}_ \rightarrow \mathcal{P}_)}$, which we prove below.

In fact, *any* generic interpreter instantiated with *any* sound analysis component with a compatible interface is sound, which leads us to our main soundness theorem:

Theorem 4.5.1 (Soundness of component-based analyses). *Let* $\mathrm{eval}_C : \mathcal{I}(C) \Rightarrow C(X, Y)$ *be a generic interpreter with an arrow-based interface* $\mathcal{I}$*. Furthermore, let* $\langle \mathrm{ConT}, \widehat{\mathrm{AbsT}} \rangle_{\mathcal{I}}$ *be a sound analysis component. Then the abstract semantics* $\mathrm{eval}_{\widehat{\mathrm{AbsT}}(\rightarrow)}$ *is sound with respect to the concrete collecting semantics* $\mathrm{eval}_{\mathrm{ConT}(\mathcal{P}_ \rightarrow \mathcal{P}_)}$.

Proof. The soundness lemma of the analysis component $\langle \mathrm{ConT}, \widehat{\mathrm{AbsT}} \rangle_{\mathcal{I}}$ (Definition 4.3.3) guarantees that the pair of arrow instances $\mathrm{ConT}\ (\mathcal{P}_ \rightarrow \mathcal{P}_)$ and $\widehat{\mathrm{AbsT}}\ (\rightarrow)$ are sound. Furthermore, the main soundness theorem for arrow-based generic interpreters [Keidel et al. 2018, Theorem 3] guarantees that the generic interpreter $\mathrm{eval}_{\widehat{\mathrm{AbsT}}(\rightarrow)}$ is sound w.r.t. $\mathrm{eval}_{\mathrm{ConT}(\mathcal{P}_ \rightarrow \mathcal{P}_)}$. □

Note that the arrows $\mathcal{P}_ \to \mathcal{P}_$ and $(\to)$ implement the `Arrow` and `ArrowChoice` type classes. This requires one extra sound lifting of these operations through the ConT and $\widehat{\mathrm{AbsT}}$ arrow transformers. Theorem 4.5.1 can be easily extended to account for this lifting, which we omitted for a cleaner presentation.

To summarize, in this section we have shown how to define a complete static analysis based on analysis component and how to prove it sound. This requires a generic interpreter for the analyzed language, which captures the similarities of concrete and abstract interpreter. We proved that this generic interpreter instantiated with a sound analysis component is sound.

4.6 Sturdy: A Library Of Sound And Reusable Analysis Components

We developed the *Sturdy* library of 13 sound and reusable arrow-based analysis components in Haskell.[1] We use some of these components to implement two static analyses in Section 4.7, to demonstrate the reusability of these components. In this section, we briefly describe selected components and then discuss measures to counter their performance overhead.

4.6.1 Analysis Components

Single Transformer Components A good source for components are single arrow transformers that are used both for the concrete and the abstract interpreter. For example, the arrow transformer `ReaderT r c` adds data of type `r` to the input of the arrow computation `c`. It can be easily turned into an analysis component $\langle \mathtt{ReaderT}, \mathtt{ReaderT} \rangle_{\mathtt{ArrowReader}}$ that adds data to both the concrete and abstract interpreter. Furthermore, because the concrete and abstract implementation of the `ArrowReader` operations is exactly the same, only differing in the type of data `r`, and hence can be proved sound trivially with a soundness theorem for parametricity [Keidel et al. 2018, Theorem 5].

This way we defined 5 analysis components for reading and writing state, for constant data, and for continuation-passing style.

$$\langle \mathtt{ReaderT}, \mathtt{ReaderT} \rangle_{\mathtt{ArrowReader}} \quad \langle \mathtt{StateT}, \mathtt{StateT} \rangle_{\mathtt{ArrowState}}$$
$$\langle \mathtt{WriterT}, \mathtt{WriterT} \rangle_{\mathtt{ArrowWriter}} \quad \langle \mathtt{ConstT}, \mathtt{ConstT} \rangle_{\mathtt{ArrowConst}}$$
$$\langle \mathtt{ContT}, \mathtt{ContT} \rangle_{\mathtt{ArrowCont}}$$

These arrow transformers are well-known and we borrowed their definition from the arrow transformer libraries `arrows` and `atl` on Hackage.[2] Furthermore, some of these arrow transformers appeared in form of monad transformers in [Darais et al. 2015, 2017], which we took inspiration from.

Environment Components To implement environment components, we created the type class `ArrowEnv` with an operation `getEnv` to fetch an environment, `localEnv` to set a new environment in a local context, `extendEnv` to extend the given environment with a new binding and `lookup` to look up a binding in the current environment:

```
class ArrowEnv var val env c where
 getEnv :: c () env               localEnv :: c x y → c (env,x) y
 extendEnv :: c (var,val,env) env  lookup :: c (val,x) y → c x y → c (var,x) y
```

Based on this interface, we created two components for environments. The first component $\langle \mathtt{EnvT}, \widehat{\mathtt{EnvT}} \rangle_{\mathtt{ArrowEnv}}$ implements the standard abstraction for environments [Cousot 1999], i.e., a mapping from variables to abstract values.

The second component $\langle \mathtt{EnvT}, \widehat{\mathtt{BoundedEnvT}} \rangle_{\mathtt{ArrowEnv}}$ implements a finite abstraction for environments for languages with closures [Shivers 1991]. In this component abstract environments consist of a pair of mappings $(\mathtt{MapVar}\ \widehat{\mathtt{Addr}}, \mathtt{Map}\ \widehat{\mathtt{Addr}}\ \widehat{\mathtt{Val}})$ from variables to abstract addresses to values. By limiting the amount of abstract addresses, the abstract domain of environments and abstract closures becomes finite.

[1] https://gitlab.rlp.net/plmz/sturdy/
[2] http://hackage.haskell.org/

All arrow transformers of the environment components are implemented with the `ReaderT` arrow transformer. This gives us the soundness proof for `getEnv` and `localEnv` proofs for free [Keidel et al. 2018] because they are implemented with the same `ArrowReader` operations.

Store Components To implement store components, we created the type class `ArrowStore` with operations to read from and write to a store:

```
class ArrowStore var val c where
  read :: c (val,x) y → c x y → c (var,x) y
  write :: c (var,val) ()
```

Based on this interface we defined a store component $\langle \mathtt{StoreT}, \widehat{\mathtt{StoreT}} \rangle_{\mathtt{ArrowStore}}$, which implements a path-insensitive store abstraction. The abstract store is a mapping from variables to abstract values. Each binding indicates if the binding *must* be present in the store or *may* not be present in the store. When we read a *may*-binding, the operation $\widehat{\mathtt{read}}$ `f g` joins results of the success and failure continuations `f` and `g`.

Furthermore, we implemented a component $\langle \mathtt{ReachingDefsT}, \widehat{\mathtt{ReachingDefsT}} \rangle_{\mathtt{ArrowStore}}$ for tracking reaching definitions [Nielson et al. 1999], which uses the store interface. This component calculates which variable definitions reach a certain program point without being overwritten. We implemented this analysis as a lifting of the store operations, by recording the label of the current assignment alongside the value in the store in the abstract run. After the analysis has run, we read out the store at each program point from the fixpoint cache to obtain the reaching definition information.

Failure and Exception Components To analyze failure and exceptions, we created two components $\langle \mathtt{FailureT}, \widehat{\mathtt{FailureT}} \rangle_{\mathtt{ArrowFail}}$ and $\langle \mathtt{ExceptT}, \widehat{\mathtt{ExceptT}} \rangle_{\mathtt{ArrowExcept}}$ that employ two different abstractions for error.

The abstract $\widehat{\mathtt{FailureT}}$ transformer wraps the output with an error type in which errors overapproximate successful results, i.e., $\mathtt{Success}\ x \sqsubseteq \mathtt{Fail}\ e$. With this ordering, erroneous branches of computation overwrite successful branches of computation: $\mathtt{Fail}\ e \sqcup \mathtt{Success}\ x = \mathtt{Fail}\ e$. This abstraction is useful when we want to propagate information about failures in programs.

The abstract $\widehat{\mathtt{ExceptT}}$ transformer wraps the output with an error type (Figure 4.2) in which $\mathtt{Success} x \sqsubseteq \mathtt{SuccessOrFail} x e \sqsupseteq \mathtt{Fail} e$. This abstraction is more precise than the error type of `FailureT`, because the case `Success` describes computation that must succeed and cannot fail.

Fixpoint Components To implement fixpoint components, we created a type class with a `fix` operation that calculates the fixpoint over an arrow computation:

```
class ArrowFix x y c where
  fix :: (c x y → c x y) → c x y
```

Based on this interface we implemented a fixpoint component $\langle \mathtt{Fix}, \widehat{\mathtt{Fix}} \rangle_{\mathtt{ArrowFix}}$, whose concrete `fix` operation calculates the standard fixpoint `fix f = f (fix f)`. The abstract `fix` operation implements a parallel/sequential fixpoint algorithm [Darais et al. 2017]. We parameterized this fixpoint algorithm by a widening operator [Cousot and Cousot 1992b] for the codomain `y` that ensures that the fixpoint iteration terminates and a second operator on the domain `x` that joins recursive calls to avoid infinitely deep chains of recursive calls. The fixpoint component $\langle \mathtt{Fix}, \widehat{\mathtt{Fix}} \rangle$ is the only component which does *not* consist of arrow transformers and hence has to be placed at the bottom of the component stack. This ensures that the function `f` we fix over is a pure function and no side effects interfere when we iterate on `f` multiple times.

Additionally, the abstract $\widehat{\mathtt{fix}}$ operation detects computations which potentially do not terminate. Non-terminating computations are usually represented with the bottom element $\perp$ of the abstract domain for values and add a lot of boilerplate to the abstract interpreter. Instead, we capture the bottom element with a component $\langle \mathtt{TerminatingT}, \widehat{\mathtt{TerminatingT}} \rangle$ that wraps the output with a `Maybe`-like type:

```
data Terminating a = NonTerminating | Terminating a
```

This transformer allows us to remove the bottom element from the abstract domain of values and the boilerplate of propagating bottom values.

Based on the same ArrowFix interface we implemented a component $\langle \texttt{ContourT}, \widehat{\texttt{ContourT}} \rangle$ that tracks the *call context* of the abstract interpreter. This call context consists of a list of recursive calls to the abstract interpreter and is useful, for example, in a k-CFA analysis [Shivers 1991]. We implement this component as a lifting of the fix operation:

```
fix f = ContourT $ proc (δ,x) → fix (runContourT ⌊x : δ⌋_k ∘ f ∘ ContourT) ⤛ x
```

On each recursive call we push the argument x of the abstract interpreter onto the current call string δ and limit it size to at most k.

4.6.2 Reducing the Performance Overhead of Analysis Components

Every analysis component adds some overhead to the runtime of the analysis. We identified two main sources for this performance overhead: Inefficient arrow code and dynamic dispatch. In the rest of this section, we explain how we addressed these issues to reduce the performance overhead of analysis components in our library.

The first issue was an inefficient pattern of arrow code that occured frequently in the implementation of arrow transformers: The composition of a pure function f with an effectful arrow computation g as in g ∘ arr f. The problem with this pattern is that the composition operation "∘" does not know that arr f is a pure computation and hence has to consider all possible effects of both operations. For example, if the arrow type supports exceptions, the composition operation "∘" has to check if an exception occured in arr f, even though arr f cannot cause an exception. Fortunately, we can eliminate this inefficient pattern by using a type class which captures the composition of pure functions with effectful computations. The Profunctor type class[3] defines an operation dimap which pre- and post-composes two pure functions with an effectful computation:

```
dimap :: (x → x') → (y → y') → c x' y → c x y'
```

The implementation of dimap for arrows is more efficient than effectful composition, because it can exploit that the functions it composes with are pure. For example, in contrast to g ∘ arr f, the operation dimap f id g does not have to check that f cause an exceptions.

As a second source of performance overhead we identified the dynamic dispatch of type class methods. Without any optimization options the GHC Haskell compiler, converts type classes to dictonaries (records of functions) [Hall et al. 1996]. Calling functions from these dictonaries entails a dynamic dispatch, which causes a performance overhead. We counter this issue by annotating the arrow type class methods with INLINE, such that GHC retains a copy of the source code of the type class methods. Furthermore, we annotated the complete type of the generic interpreter in the file where we compose an analysis. This allows GHC to specialize the definition of the generic interpreter, inline arrow operations and eliminate any form of dynamic dispatch. Furthermore, because all arrow operations are inlined, GHC can optimize some redundent pre- and post-processing in arrow transformer liftings.

We evaluated how these two optimizations affect the performance of arrow transformers with a benchmark.[4] The benchmark instantiates an arrow-based concrete interpreter with different arrow transformers and measures the runtime for an example program. Figure 4.3 shows the runtimes for each transformer in microseconds with and without optimizations. The results show that the profunctor optimization does not significantly improve the performance of individual transformers, but it does improve the performance by 12x if multiple transformers are combined into a stack. The inlining optimization improves the performance of individual transformers and the transformer stack by several orders of magnitude. Lastly, combining the profunctor and inlining optimization does not significantly improve the performance over just the inlining optimization.

[3] http://hackage.haskell.org/package/profunctors

[4] https://gitlab.rlp.net/plmz/sturdy/blob/benchmark/lib/bench/ArrowTransformerBench.hs

Transformer	No Opts.	Profunctor		Inline		Prof. + Inline	
`ConstT`	261 μs	233 μs	(1)	5 μs	(56)	5 μs	(56)
`ReaderT`	907 μs	874 μs	(1)	8 μs	(119)	8 μs	(117)
`StateT`	435 μs	433 μs	(1)	13 μs	(33)	13 μs	(33)
`WriterT`	1506 μs	1490 μs	(1)	13 μs	(118)	13 μs	(119)
`ErrorT`	664 μs	664 μs	(1)	14 μs	(48)	14 μs	(47)
`ExceptT`	827 μs	804 μs	(1)	15 μs	(56)	15 μs	(55)
`TerminatingT`	515 μs	502 μs	(1)	14 μs	(38)	14 μs	(37)
Stack	209408 μs	18085 μs	(12)	27 μs	(7730)	26 μs	(7978)

Figure 4.3: Benchmark results for individual arrow transformers without optimizations and with the profunctor and inlining optimization. Each column shows the average runtime in microseconds and in parentheses the speed up compared to the unoptimized version. The bottom row shows benchmark results for an arrow transformer stack which combines all transformers above.

To summarize, in this section we presented a library of analysis components for different analysis concerns. Furthermore, we described two techniques that we used to reduce the performance overhead of arrow transformers. In Section 4.7, we will use some of these components to implement static analyses, to demonstrate their reusability.

4.7 Experimental Evaluation And Case Studies

In this paper, we presented an approach to reduce the effort of defining and proving soundness of static analyses with the help of sound and reusable analysis components. To evaluate our approach, we answer the following research questions:

(RQ1) Modular implementation: Are analysis components reusable and do they compose?

(RQ2) Modular soundness proofs: Are analysis components separately verifiable and do their soundness proofs compose?

(RQ3) Liftings: Is the effort of implementing liftings and proving their soundness acceptable?

To answer these research questions, we conducted two experiments. The first experiment starts with an interval analysis for the While language. We explore how analysis components support modular analysis development and soundness proofs by building a reaching definitions analysis on top of the interval analysis. We then challenge our approach by extending the While language with exceptions and observe how the interval and reaching definitions analyses change.

In our second experiment we build a control-flow analysis (k-CFA) for PCF. The analysis predicts the control flow of calls to first-class function values, which is not statically decidable. The goal of this experiment is to test if our approach is specific to some languages or analyses.

Across both experiments we were able to answer our research questions affirmatively. In particular, the implementation and soundness proofs of most liftings comes for free.

4.7.1 Experiment 1: Analyses for a While Language

We implement an interval analysis for a statically-scoped While language. This interval analysis will serve as a base-line as we extend this analysis in two different ways to study the impact of these changes. First, we add a reaching definition [Nielson et al. 1999] analysis to the interval analysis. Second, we extend the While language with exceptions.

Interval Analysis for the While Language

We start by defining an arrow-based generic interpreter for our While language (Listing 4.1). The interpreter is basically a more complete version of the one we presented in Section 4.2.4. To implement an interval analysis for this language, we need to instantiate the generic interpreter with a "super-component" that implements all required interfaces. Our first research question **RQ1** asks if we can separate concerns in the implementation of this super-component. Indeed,

```
data Expr = ...
data Statement = Assign String Expr Label | If Expr Statement Statement Label
  | While Expr Statement Label | Begin [Statement] Label

run :: (IsVal v c, ArrowAlloc (Var,v,Label) addr c, ArrowRand v c,
  ArrowEnv Var addr env c, ArrowStore addr v c, ArrowFail e c,
  ArrowFix [Statement] () c, ArrowChoice c) ⇒ c [Statement] ()
run = fix $ λrun' → proc stmts → case stmts of
  Assign x e l:ss → do
    v ← eval ≺ e
    addr ← lookup (proc (addr,_) → returnA ≺ addr) alloc ≺ (x,(x,v,l))
    write ≺ (addr,v)
    extendEnv' run' ≺ (x, addr, ss)
  If cond s1 s2 _:ss → do
    b ← eval ≺ cond
    if_ run' run' ≺ (b,([s1],[s2]))
    run' ≺ ss
  While cond body l:ss →
    run' ≺ If cond (Begin [body,While cond body l] l) (Begin [] l) l : ss
  Begin ss _:ss' → do
    run' ≺ ss; run' ≺ ss'
  [] → returnA ≺ ()
```

Listing 4.1: Generic interpreter for statements of the While language.

Concrete Stack	IsVal	ArrowAlloc	ArrowRand	ArrowEnv	ArrowStore	ArrowFail	ArrowFix	ArrowChoice	Abstract Stack
ConcreteT	[12]	[1]	↑	↑	↑	↑	↑	↑	$\widehat{\texttt{IntervalT}}$
RandomT			[1]	↑	↑	↑	↑	↑	$\widehat{\texttt{RandomT}}$
EnvT				[4]	↑	↑	↑	↑	$\widehat{\texttt{EnvT}}$
StoreT					[2]	↑	↑	↑	$\widehat{\texttt{StoreT}}$
FailureT						[1]	↑	⇝	$\widehat{\texttt{FailureT}}$
TerminatingT							↑	⇝	$\widehat{\texttt{TerminatingT}}$
Fix							[1]	[4]	$\widehat{\texttt{Fix}}$

Figure 4.4: Interval analysis of the While language: Boxes [n] represent components, straight arrows ↑ represent trivial liftings, squiggly arrows ⇝ represent non-trivial liftings.

we were able to implement each interface in a separate component and to compose the super-component from these using the techniques described in Section 4.4.

We display the involved components and their composition in Figure 4.4, which introduces a novel notation we devised for this paper. Each box [n] in the table indicates a separate analysis component $\langle Row_L, Row_R \rangle_{Col}$. The column label indicates the component's interface; the left and right row labels indicate the used concrete and abstract arrow transformers; the boxed number indicates how many operations of the interface had to be implemented. For example, box [4] in Figure 4.4 indicates an analysis component $\langle \texttt{EnvT}, \widehat{\texttt{EnvT}} \rangle_{\texttt{ArrowEnv}}$ that implements 4 operations.

Our notation in Figure 4.4 also displays how components are composed. Components that appear on the same row compose horizontally without any extra effort. Components that appear on different rows require vertical composition based on one or more liftings. We display liftings as upward arrows, yet distinguish two kinds: A straight arrow ↑ represents a lifting whose implementation and soundness proof was trivial because the lifting was (i) reusable, (ii) generic, or (iii) derivable (Section 4.4.2). In contrast, a squiggly arrow ⇝ represents a lifting that required a non-trivial implementation and soundness proof. Regarding the research questions, we conclude the following:

Concrete Stack	IsVal	ArrowAlloc	ArrowRand	ArrowEnv	ArrowStore	ArrowFail	ArrowFix	ArrowChoice	Abstract Stack
ConcreteT	12	1	↑	↑	↑	↑	↑	↑	$\widehat{\texttt{IntervalT}}$
RandomT			1	↑	↑	↑	↑	↑	$\widehat{\texttt{RandomT}}$
EnvT				4	↑	↑	↑	↑	$\widehat{\texttt{EnvT}}$
ReachingDefsT					$\rightsquigarrow$	↑	$\rightsquigarrow$	↑	$\widehat{\texttt{ReachingDefsT}}$
StoreT					2	↑	↑	↑	$\widehat{\texttt{StoreT}}$
FailureT						1	↑	$\rightsquigarrow$	$\widehat{\texttt{FailureT}}$
TerminatingT							↑	$\rightsquigarrow$	$\widehat{\texttt{TerminatingT}}$
Fix							1	4	$\widehat{\texttt{Fix}}$

Figure 4.5: Reaching definitions analysis of the While language

(RQ1) Modular implementation: We successfully separated concerns of the analysis into 7 analysis components: 6 reusable analysis components from our library and 1 language-specific analysis component ⟨`ConcreteT`, $\widehat{\texttt{IntervalT}}$⟩ for values, conditionals, and allocation.

(RQ2) Modular soundness proofs: We successfully decomposed the soundness proof into soundness lemmas about the individual components: Each soundness lemma proves a single concern of the analysis, while it is independent of other concerns. For example, the soundness proofs of the conditional `if_` in `IsVal` is independent of the store and fixpoint cache, even though these are threaded through the branches of the conditional. This is possible because the `IsVal` component is parametric in the underlying arrow `c`, which contains the store and fixpoint cache after composition.

(RQ3) Liftings: We required 22 liftings to compose all 7 analysis components. Of these liftings, 20 liftings are trivial. Only 2 liftings required an explicit implementation and soundness proof. We conclude that the effort for liftings is modest and acceptable.

Reaching Definitions Analysis for the While Language

We want to refine our previous interval analysis to also keep track of reaching definitions [Nielson et al. 1999]. A definition (here: assignment) reaches another statement if there is at least one control-flow path where the assigned variable was not reassigned in between. Since the language syntax remains unchanged, no change occurs to the generic interpreter `run` or its required interfaces. The challenge is this: Can we reuse the implementation and soundness proofs of all previously used components unchanged?

(RQ1) Modular implementation: We encapsulate the concern of reaching definitions in its own analysis component ⟨`ReachingDefsT`, $\widehat{\texttt{ReachingDefsT}}$⟩$_{\texttt{ArrowStore}}$ as described in Section 4.6. Technically, the reaching definitions component piggybacks on another component implementing the `ArrowStore` interface, but stores additional data in the abstract run. In the concrete run, reaching definitions has no effect and uses the identity transformer. Figure 4.5 shows how the new component (gray background) neatly integrates with the existing components; no changes to other components were necessary.

(RQ2) Modular soundness proofs: We only had to prove soundness of the reaching definitions component, while all other soundness lemmas remain valid. Except for the reaching definitions component, there is no additional proof effort.

(RQ3) Liftings: Since the reaching definitions was realized as a non-trivial lifting, we additionally obtain 2 such liftings in our final composition. None of the other liftings were (or could have been) influenced. Thus, we retain that the lifting effort is modest and acceptable.

Concrete Stack	IsVal	IsExc	ArrowAlloc	ArrowRand	ArrowEnv	ArrowStore	ArrowExcept	ArrowFail	ArrowFix	ArrowChoice	Abstract Stack
ConcreteT	12	2	1	↑	↑	↑	↑	↑	↑	↑	$\widehat{\mathtt{IntervalT}}$
RandomT				1	↑	↑	↑	↑	↑	↑	$\widehat{\mathtt{RandomT}}$
EnvT					4	↑	↑	↑	↑	↑	$\widehat{\mathtt{EnvT}}$
ReachingDefsT						⇝	↑	↑	⇝	↑	$\widehat{\mathtt{ReachingDefsT}}$
StoreT						2	↑	↑	↑	↑	$\widehat{\mathtt{StoreT}}$
ExceptT							3	↑	↑	⇝	$\widehat{\mathtt{ExceptT}}$
FailureT								1	↑	⇝	$\widehat{\mathtt{FailureT}}$
TerminatingT									↑	⇝	$\widehat{\mathtt{TerminatingT}}$
Fix									1	4	$\widehat{\mathtt{Fix}}$

Figure 4.6: Reaching definitions and interval analysis of the While language with exceptions.

Extending the While Language with Exceptions

Finally, we study the effort to update an analysis when a language evolves. In particular, we add exceptions to the While language and observe how this affects the interval and reaching definitions analyses.

This time we have to change the generic interpreter, because we are adding new syntax:

```
data Expr = ... | Throw ExceptName Expr
data Stmt = ... | TryCatch Stmt ExceptName String Stmt Label | Finally Stmt Stmt Label
```

The generic interpreter implements these new expressions and statements with operations of the `ArrowExcept` interface, that we now depend on. The rest of the generic interpreter stays the same. The challenge is this: Can we reuse our analysis components unchanged given that exception handling has a cross-cutting effect on the control flow of the language?

(RQ1) Modular implementation: We encapsulate the core functionality of exception handling in the language-specific interface `IsExc` and the reusable exception analysis component (Section 4.6), which provides the operations `throw`, `catch`, and `finally`. But, since `ArrowExcept` is a new interface, we also need to add liftings for its operations through the other components used, such that `throw`, `catch`, and `finally` are available after full composition. Figure 4.6 shows how the new component (row with gray background) and the new liftings (column with gray background) neatly integrate with the existing components. No other changes unrelated to exceptions were necessary.

(RQ2) Modular soundness proofs: We only had to prove soundness lemmas for the exceptions component and for the liftings of exception operations. All other soundness lemmas remain valid.

(RQ3) Liftings: Our extension requires 8 new liftings, of which only 1 lifting was non-trivial and required an explicit soundness proof. In total, we now have 34 liftings of which 29 are trivial and 5 are non-trivial.

To summarize, we extended the interval analysis with a reaching definitions analysis and added exceptions to our While language. In both cases, our design allowed us to capture the extension as a separate analysis component, while reusing all other analysis components unchanged. We were also able to reuse all previous soundness lemmas unchanged. For the reaching definitions analysis, we only needed to prove the new component sound. For exception handling, we additionally had to prove a few new liftings sound. However, the vast majority of liftings (85%) has a trivial implementation and soundness proof that is reusable, generic, or derivable.

4.7.2 Experiment 2: Control-Flow Analysis for PCF

To confirm that the succesful application of analysis components was not particular to the analysis or language of the first experiment, we define a k-CFA analysis [Shivers 1991] for PCF [Plotkin

Concrete Stack	IsNum	IsClosure	ArrowEnv	ArrowFail	ArrowFix	ArrowChoice	Abstract Stack
ConcreteT	[3]	[2]	↑	↑	↑	↑	$\widehat{\texttt{IntervalT}}$
EnvT			[4]	↑	↑	↑	$\widehat{\texttt{BoundedEnvT}}$
ContourT				↑	$\leadsto$	↑	$\widehat{\texttt{ContourT}}$
FailureT				[1]	↑	$\leadsto$	$\widehat{\texttt{FailureT}}$
TerminatingT					↑	$\leadsto$	$\widehat{\texttt{TerminatingT}}$
Fix					[1]	[4]	$\widehat{\texttt{Fix}}$

Figure 4.7: k-CFA analysis of PCF: Components [n] and liftings ↑/ $\leadsto$.

1977]. PCF is a language with first-class functions and numbers and the main purpose of the k-CFA is to approximate which function values may be called at any function application.

To implement this analysis, we first need to describe the semantics of PCF with an arrow-based generic interpreter that captures the similarities between concrete and abstract semantics. Our case study builds on an existing generic interpreter for PCF and an existing k-CFA analysis [Keidel et al. 2018]. The goal of our case study is to modularize this analysis by using analysis components. As first step, we refactor the generic interpreter to depend on individual interfaces, each encapsulating a different concern (Listing 3.6 in Chapter 3). Except for the use of arrows, the generic interpreter is fairly standard and requires no further explanation.

k-CFA imposes different challenges than those encountered in the first experiment. In particular, environments are embedded in to closure values and therefore must be abstracted to a finite domain if we want our analysis to terminate [Horn and Might 2010]. Let us revisit our research questions to see if they are affected by this.

(RQ1) Modular implementation: Again we succeeded in decomposing the analysis into several independent analysis components as shown in Figure 4.7. Each analysis component encapsulates a single concern, which simplifies its implementation. Furthermore, the composition cleanly combines the analysis components as they need to work in concert, and the liftings explain how different components interact. For example, the environment component asks the contour component for the current call context to allocate new addresses. However, the environment component is not tightly coupled to the contour component as these components communicate through an interface and are combined with component composition.

(RQ2) Modular soundness proofs: Again we were able to prove each analysis component sound independently and composition preserves soundness. We included the soundness proofs of the analysis components for the environments, exceptions, fixpoints, and values in the supplementary material accompanying this paper. Because of the separation of concerns, each soundness lemma can be verified independently, which makes it easier to prove compared to a monolithic proof. For example, when proving environment operations sound, we do not have to reason about failure or fixpoint caches. Similarly, the proof of the value operations also became easier because the operations are independent of effects in the language.

(RQ3) Liftings: The composition of analysis components for this analysis requires in total 14 liftings. Of these 14 liftings, we were able to derive the soundness proof of 11 liftings automatically using the techniques of Section 4.4.2. Only 3 liftings required an explicit implementation and soundness proof: The `Arrow`/`ArrowChoice` liftings of the failure and termination component and the `ArrowFix` lifting of the contour component.

To summarize, we modularized the implementation and soundness proof of a k-CFA analysis for PCF using analysis components. Each analysis component encapsulates a single concern, which simplifies the implementation and soundness proof and increases its reusability. Furthermore, the composition of these analysis components required 14 liftings of which 11 could be derived and proven sound automatically and only 3 required an explicit implementation and soundness proof. Analysis components appear to be applicable to a wider range of languages and analyses.

4.8 Related Work

Proving soundness of static analyzers for real-world languages is a difficult endeavor. Some dynamic language features such as Java's reflection [Smaragdakis et al. 2015] or JavaScript's dynamic evaluation [Meawad et al. 2012] complicate static analysis and its soundness proof. As a consequence, static analyzers [Flanagan et al. 2002] and bug finders [Rutar et al. 2004] often either unsoundly approximate these language features or ignore them all together [Jourdan et al. 2015]. Unsound analyses still provide valuable information about program behavior, however, this information might not be reliable. For static analyzers that have been proven sound, the proof effort is significant. For example, *Verasco* [Jourdan et al. 2015] is a static analyzer for C, whose soundness has been formally verified in the proof assistant Coq. The implementation of the abstract interpreter consists of 17k lines of Coq code, as do the proof scripts (17k LOC). This shows that a soundness proof of a static analysis for a real-world language requires significant effort and expertise. In this work, we aim to reduce the effort and complexity of soundness proofs by separating analysis concerns with analysis components. We hope that with our technique the soundness proof of static analyses for real-world languages becomes more approachable.

Sergey et al. [2013] showed that analysis aspects such as context-sensitivity, polyvariance and flow-sensitivity can be captured by monads. A *monadic abstract interpreter* has the benefit, that it allows to change these analysis aspects by changing the underlying monad, while the rest of the analysis definition stays the same. This is possible because the monadic abstract interpreter abstracts over the underlying monad with interfaces, which are similar to the interfaces of our analysis components. However, Sergey et al. did not develop a theory to prove monadic abstract interpreters sound. In this work, we demonstrate that arrows, a generalization of monads, are also capable of capturing different analysis aspects. We improve upon the work of Sergey et al. [2013] by developing a theory that simplifies and reduces the effort of proving soundness static analyses.

In this work, we describe abstract interpreters with arrows instead of monads. The benefit of using arrows is that they form an algebra and hence provide the reasoning principle of structural induction over arrow expressions [Keidel et al. 2018]. This induction principle decomposes the soundness proof of arrow-based abstract interpreters into smaller soundness lemmas of the arrow operations. We use this induction principle in Theorem 4.5.1 in Section 4.5 to prove soundness of generic interpreters instantiated with analysis components. In contrast, monadic expressions do not support an induction principle and hence a generic interpreter based on monads requires a manual soundness proof. We are not aware of any inherent disadvantages of arrows over monads, however, one important difference between arrows and monads is that arrows also capture the input of computations. In particular, higher-order arrow operations need to pass arguments to inner computations explicitly. For example, in `lookup (proc var → alloc ≺ var) ≺ var` the argument of `alloc` need to be passed in as argument to `lookup`. This explicit argument passing can be cumbersome and sometimes might not be possible if the higher-order arrow operation does not pass along the argument to the inner computation. In contrast, monads lift this restriction and arguments can be passed freely into higher-order monad operations.

As discussed in the introduction, we build on the theory of *compositional soundness proofs of abstract interpreters* by Keidel et al. [2018]. We improve upon this work by composing the arrow instances of the concrete and abstract interpreter from modular and reusable components based on *arrow transformer*. Our work simplifies the implementation and soundness proof of arrow instances, because existing analysis functionality can be reused and does not need to be reinvented. Furthermore, the implementation and soundness proof of our analysis components themselves is simpler compared to monolithic arrows, because each component captures only a single concern of an analysis instead of mixing them. Moreover, we retain the benefit of arrow-based abstract interpreters, namely, analysis creators do not need to reason about the code of the generic interpreter.

The idea of composing static analyses from modular components has also been explored by Darais et al. [2015]. The authors also propose to share code between concrete and abstract interpreter. But the code of the generic interpreter is parameterized by a monad instead of by an arrow. To recover the concrete and abstract interpreter, the generic interpreter is instantiated with two monads composed from monad transformers. These monad transformers capture reusable anal-

ysis functionality and are called *Galois Transformers*. A short-coming of this approach is that monads are missing a reasoning principles to compose a soundness proof and hence a generic interpreter based on monads still requires an explicit soundness proof. We improve upon this work by describing static analyses with analysis components based on arrow transformers. The benefit of using arrows is that arrows provide the reasoning principle of structural induction, which makes the soundness proof of static analyses compositional [Keidel et al. 2018]. This means that when we instantiate an arrow-based generic interpreter with sound analysis components, the resulting abstract interpreter is sound and no extra reasoning about the generic interpreter is required.

Defining abstract interpreters from components has been revisited recently by Darais et al. [2017]. Core of their approach is a definitional interpreter (generic interpreter) that is parameterized by a monad. Instantiating the interpreter with monads composed of monad transformers, yields the concrete and abstract semantics. The authors describe several analysis components, such as a fixpoint algorithm for big-step semantics, a trace collecting or dead code collecting semantics. We took inspiration of these components, especially of the fixpoint algorithm. We improve upon this work by describing a theory that modularizes the soundness proof of analysis components. This means we can prove analysis components sound independently and the composition of analysis components preserves soundness.

There are several techniques to compose different abstract domains to improve the precision of the analysis such as reduced products [Cousot and Cousot 1979] and cofibered products [Venet 1996]. For example, the reduced product $\mathcal{P}(\mathbb{Z}) \leftrightarrows \texttt{Interval} \times \texttt{Parity}$ combines the abstract domains of intervals to rule out interval bounds with a wrong parity. While techniques such as reduced products compose different abstractions for data (e.g. values, environments, stores), in contrast, our paper describes a technique to modularly define and modularly prove sound the semantics for different cross-cutting aspects of the analysis (e.g. values, exceptions, mutable state, fixpoint computations). In the future, we want to explore if we can combine the technique of this paper and the techniques for composing abstract domains, by creating a component that combines two value components and computes their reduced product.

Madsen and Lhoták [2018] proposed an approach that reduces the soundness proof burden of static analyses. The approach uses SMT solvers to prove soundness of operations over some abstract domains automatically. Annotations in the code aid the SMT solver in the proving process. These annotations contain mathematical properties, such as monotonicity, required to prove soundness. The authors evaluate their approach by proving soundness of value operations over abstract domains for booleans, strings and integers. However, the authors have not explored if their verification technique scales to a soundness proof of a complete static analysis. Compared to our work, we currently do not use proof automation to prove soundness of our analysis components. However, our technique guarantees that a complete static analysis is sound if all its analysis components are sound. In the future, we want to explore how we can incorporate proof automation to simplify the soundness proof of analysis components.

4.9 Conclusion

We propose a novel approach to constructing static analyses modularly from reusable analysis components. Each analysis component covers one aspect of the analyzed language, can be proven sound independently, and their composition preserves soundness. Our analysis components consist of pairs of arrow transformers, for which we develop a Galois connection and soundness proposition. We use analysis components to instantiate arrow-based generic interpreters [Keidel et al. 2018] to obtain complete sound static analyses. A key result of our work is that a static analysis based on analysis components is sound, if all their analysis components are sound. We demonstrate the applicability and usefulness of our approach by creating a library of 13 reusable analysis components that allow us to define a k-CFA analysis for PCF and an interval and reaching definition analysis for a WHILE language.

MODULAR DESCRIPTION AND SOUNDNESS PROOFS OF FIXPOINT ALGORITHMS FOR BIG-STEP ABSTRACT INTERPRETERS

5

This chapter is based on the following draft:

> Modular Description and Soundness Proofs of Fixpoint Algorithms for Big-Step Abstract Interpreters.
> **Sven Keidel**, Tobias Hombücher, and Sebastian Erdweg.

Abstract — Abstract interpretation is a methodology for defining sound static analyses. In this paper, we study fixpoint algorithms for big-step abstract interpreters. We identify three challenges regarding the termination of a fixpoint algorithm, two of which are unique to big-step abstract interpreters. To this end, we develop a novel big-step fixpoint algorithm that solves these challenges and iterates on the strongly-connected components of the call graph. However, since the algorithm is implemented as a single monolithic function, it is hard to extend, configure and adapt the algorithm for new languages, analyses and use-cases. In particular, for each new use case we would need to change the implementation of the algorithm and redo its soundness proof.

We address this issue by describing a framework for developing modular fixpoint algorithms, that describes an algorithm with small and reusable combinators. This framework allows us to fine-tune an algorithm more easily by rearranging existing combinators or by adding new language and analysis-specific combinators. Furthermore, the framework simplifies the soundness proof of fixpoint algorithms, as the proof can be composed from soundness lemmas for each combinator. This means analysis developers do not have to worry about soundness of the fixpoint algorithm, as long as they reuse sound combinators. Lastly, we show with our evaluation, that our framework describes an entire family of fixpoint algorithms for different languages and analyses, that can be easily extended and fine-tuned.

5.1 Introduction

Abstract interpretation [Cousot and Cousot 1977] is a methodology for defining sound static analyses. While in the past, many static analyses have been described as abstract interpreters in small-step style [Schmidt 1996; Sergey et al. 2013; Horn and Might 2010; Might and Shivers 2006a,b; Darais et al. 2015], more recently big-step abstract interpreters have been investigated more thoroughly [Darais et al. 2017; Keidel et al. 2018; Keidel and Erdweg 2019; Wei et al. 2019; Bodin et al. 2019]. Such big-step abstract interpreters can be simply described as recursive functions in a meta-language such as Haskell. For example, consider the following big-step abstract interpreter that calculates the interval of an arithmetic expression:

```
eval^ :: Env^ → Expr → Maybe Val^
eval^ env expr = case expr of
  Var x → lookup x env
  Num n → return (n,n)
  Add e1 e2 → do
    (i1,i2) ← eval^ env e1
    (j1,j2) ← eval^ env e2
    return (i1+j1,i2+j2)
```

```
Expr = Var String
     | Num Int
     | Add Expr Expr
     | ...

Env^ = Map String Val^
Val^ = Interval
```

This abstract interpreter looks like a regular concrete interpreter in big-step style, except that it computes intervals instead of numbers as values. Abstract interpreters in big-step style have the advantage that they are easy to understand and reason about, while they seamlessly combine data-flow and control-flow information. However, if we were to add a looping or recursive construct to the language above, the abstract interpreter may not terminate anymore. While for a

concrete interpreter non-termination is part of the expected language behavior, non-termination is undesirable for abstract interpreters. To ensure that an abstract interpreter terminates, we cannot use unbounded recursion like above. Instead, we need to take special care how we compute the *fixpoint* of the abstract interpreter.

In this work, we study fixpoint algorithms for big-step abstract interpreters (*big-step fixpoint algorithms* for short) and how to describe them modularly. Like most fixpoint algorithms, big-step fixpoint algorithms apply the abstract interpreter repeatably until the analysis result is stable. However, the recursive definition of big-step abstract interpreters poses specific termination challenges that we identified. First, a big-step fixpoint algorithm must detect recurrent recursive calls of the abstract interpreter to interrupt cyclic call chains:

$$\widehat{\texttt{eval}}[x \mapsto \texttt{Even}](x{+}1) \to \ldots \to \widehat{\texttt{eval}}[x \mapsto \texttt{Odd}](x{+}1) \to \ldots \to \widehat{\texttt{eval}}[x \mapsto \texttt{Even}](x{+}1) \to \ldots$$

Second, not all infinite call chains are cyclic, yet a big-step fixpoint algorithm must prevent acyclic diverging call chains as well:

$$\widehat{\texttt{eval}}[x \mapsto [0,0]](x{+}1) \to \ldots \to \widehat{\texttt{eval}}[x \mapsto [0,1]](x{+}1) \to \ldots \to \widehat{\texttt{eval}}[x \mapsto [0,2]](x{+}1) \to \ldots$$

Here, a big-step fixpoint algorithm must introduce a cycle after finite steps, reducing this problem to the first challenge. The third challenge is standard: the fixpoint algorithm needs to ensure the analysis result does not grow indefinitely, for example, by relying on a finite abstract domain or widening operators [Cousot and Cousot 1992b]. We prove that these three challenges are sufficient to ensure the termination of big-step fixpoint algorithms.

We survey existing solutions to the three individual termination challenges and combine them into a big-step fixpoint algorithm. This way we obtain the first fixpoint algorithm for big-step abstract interpreters based on chaotic iteration over strongly-connected components of the call graph [Bourdoncle 1993]. Even though this initial fixpoint algorithm works and is sound, it has several short-comings:

- The fixpoint algorithm is language-specific and analysis-specific. In particular, the algorithm was designed for a functional language with recursion and it references the expressions of the language and the abstract interpreter directly. This makes it difficult to reuse parts of the fixpoint algorithm to develop new algorithms for other languages and analyses. For example, we may want to reuse the strategy for eliminating acyclic diverging call chains when analyzing an imperative language with loops.
- The fixpoint algorithm mixes different concerns, which makes it difficult to experiment with variations or fine-tune the algorithm. For example, the fixpoint algorithm uses a specific iteration order, with which it iterates on the analysis result for different parts of the program. We may want to experiment with different iteration orders, to find orders that are more performant or more precise (akin to using a priority list in worklist algorithms [Hind and Pioli 1998]). However, since the iteration order is *baked* into the fixpoint algorithm, it is difficult to change the order without completely reimplementing the algorithm.
- The fixpoint algorithm is implemented as a single monolithic function. Since the soundness proof of a fixpoint algorithm follows its implementation, a monolithic implementation makes it difficult to prove soundness and to maintain an existing soundness proof. In particular, since changes to the algorithm often require a significant restructuring (see previous point), it is unclear how to reestablish soundness efficiently.

To eliminate these short-comings, we modularize the description of big-step fixpoint algorithms, which is the core contribution of this work. In particular, we propose to describe fixpoint algorithms modularly with sound and reusable *fixpoint combinators*.[1] These combinators describe, for example, the order in which the program is analyzed, how deep recursive functions are unfolded and loops unrolled, or they record auxiliary data such as a control-flow graph. For example, consider the following fixpoint algorithm for an imperative language with while loops. We compose the fixpoint algorithm from 4 fixpoint combinators (each starting with the letter φ):

[1] Note that we mean *non-standard* fixpoint combinators φ, that do not necessarily satisfy the standard fixpoint property $\varphi(f) = f(\varphi(f))$, but rather $\varphi(f) \sqsupseteq f(\varphi(f))$, which is sufficient for soundness.

$\varphi_{\texttt{filter}}$ `isWhileLoop` ($\varphi_{\texttt{unroll}}\ 10\ \varphi_{\texttt{joinLoopIterations}} \circ \varphi_{\texttt{innermost}}$)

The combinator $\varphi_{\texttt{unroll}}$ 10 unrolls the first 10 recursive iterations of the abstract interpreter. After the 10th iteration, $\varphi_{\texttt{unroll}}$ falls back to the combinator $\varphi_{\texttt{joinLoopIterations}}$, that joins the analysis results of all subsequent loop iterations. The combinator $\varphi_{\texttt{innermost}}$ describes the order with which the fixpoint algorithm iterates on nested loops, i.e., it iterates on the innermost loop first. Lastly, the combinator $\varphi_{\texttt{filter}}$ applies the other combinators only to while loops, because while loops are the only constructs in the language that can cause the analysis to diverge.

This modular description enables analysis developers to implement their own custom fixpoint algorithms more easily by reusing existing language-independent and analysis-independent fixpoint combinators. For example, we can extend the fixpoint algorithm from above for a language with loops and recursive functions as follows:

$\varphi_{\texttt{filter}}$ `isFunctionBody` ($\varphi_{\texttt{stackWiden}} \circ \varphi_{\texttt{outermost}}$) $\circ$ $\varphi_{\texttt{filter}}$ `isWhileLoop` (...)

The combinator $\varphi_{\texttt{stackWiden}}$ joins the arguments of recursive function calls to avoid infinite recursive call chains. Furthermore, analogous to $\varphi_{\texttt{innermost}}$, the combinator $\varphi_{\texttt{outermost}}$ iterates on the *outermost* strongly-connected components of the call graph of the abstract interpreter. This fixpoint algorithm seamlessly interleaves the intraprocedural analysis of loops with the interprocedural analysis of recursive function calls. Both of these aspects can be individually changed and fine-tuned by adding, replacing, and reordering fixpoint combinators.

Besides making fixpoint algorithms configurable and composable, our modular approach has one other advantage: it simplifies the soundness proof. Specifically, we develop a formal theory for fixpoint combinators that allows us to reason about the soundness of fixpoint combinators individually and to verify each combinator sound and once and for all. Furthermore, we show that a composed fixpoint algorithm is sound if all involved fixpoint combinators are sound. This allows us to reuse the soundness proof of existing fixpoint combinators, such that it becomes easy to (re)establish the soundness of fixpoint algorithms.

We demonstrate that our approach of modularizing the description of big-step fixpoint algorithms is feasible by implementing it in Haskell as part of the *Sturdy* framework [Keidel et al. 2018; Keidel and Erdweg 2019]. We developed 11 fixpoint combinators and compose them to obtain fixpoint algorithms for 4 analyses for 4 different languages. This demonstrates the analysis-independence and language-independence of the fixpoint combinators. Moreover, we demonstrate that the composed fixpoint algorithms can be easily changed by adapting them to 3 new use cases. In none of these cases did we need to change the implementation of the combinators, which demonstrates their reusability. Lastly, we evaluate the performance of our algorithms on a 0CFA analysis for Scheme, where we find that no single algorithm that performs best for all analyzed programs. We conclude that configurable fixpoint algorithms are necessary to allow analysis developers to fine-tune their analyses.

In summary, we make the following contributions:

- We identify three termination challenges for big-step fixpoint algorithms. We survey existing solutions to these challenges and combine them into a sound big-step fixpoint algorithm.
- We propose an approach to modularize the description of big-step fixpoint algorithms by the means of sound and reusable fixpoint combinators.
- We develop a formal theory for these combinators that allows us to prove their soundness separately and once and for all.
- We demonstrate that our approach is feasible and useful by implementing it as part of the Sturdy framework.

5.2 Designing Big-Step Fixpoint Algorithms

In this section, we first discuss challenges regarding the termination of big-step fixpoint algorithms. In the second half, we survey existing solutions to these challenges and combine these solutions into a big-step fixpoint algorithm.

5.2.1 Enforcing Termination of Big-Step Fixpoint Algorithms

There are three challenges that a fixpoint algorithm for big-step abstract interpreters needs to solve to guarantee termination.

To this end, we first extend the language from the introduction with recursive functions. Now consider the analysis of the factorial function implemented in this language. The following diagram shows a big-step reduction trace of an abstract interpreter with unbounded recursion, where $\rho \vdash e \Downarrow v$ evaluates an expression e under environment ρ to an abstract value v. Such a trace looks similar to the trace of a concrete big-step interpreter, except that the values are intervals.

$$\dfrac{\mathsf{n} \mapsto [0,0] \vdash 1 \Downarrow [1,1] \qquad \dfrac{\dfrac{\dfrac{\vdots}{\mathsf{n} \mapsto [0,\infty] \vdash \mathtt{if(n == 0)\ 1\ else\ fact(n-1)*n} \Downarrow ?}}{\mathsf{n} \mapsto [1,\infty] \vdash \mathtt{fact(n-1)} \Downarrow ?}}{\mathsf{n} \mapsto [1,\infty] \vdash \mathtt{fact(n-1)*n} \Downarrow ?}}{\dfrac{\mathsf{n} \mapsto [0,\infty] \vdash \mathtt{if(n == 0)\ 1\ else\ fact(n-1)*n} \Downarrow ?}{\mathsf{n} \mapsto [0,\infty] \vdash \mathtt{fact(n)} \Downarrow ?}}$$

The analysis starts at the call `fact(n)`, where `n` is bound to the interval $[0,\infty]$ in the environment. Because the interval $[0,\infty]$ contains 0 and other numbers, the abstract interpreter has to evaluate both branches of the conditional `if(n == 0)` and join the results. Whereas the analysis of the first branch terminates after only one step, the second branch diverges while recurrently calling the factorial function with the same environment over and over again (see *highlighted* calls). We write the question mark symbol to represent that the abstract interpreter diverged and did not produce a result. This leads us to our first challenge:

Challenge 1 *A big-step fixpoint algorithm has to detect recurrent recursive calls and cut off recursion to avoid non-termination.*

Detecting recurrent calls allows the fixpoint algorithm to iterate that part of the computation that spans the initial call and the recurrent call. One way of detecting recurrent recursive calls is to remember the calls of the abstract interpreter on each branch of the derivation tree. Each call consists of the inputs of the abstract interpreter, e.g., an expression and an environment. By remembering the calls, we can easily detect a diverging call, if the exact same call occured earlier, further down the derivation branch.

However, this way of detecting recurrent recursive calls is error-prone. For example, consider the analysis of the factorial function for negative arguments. Clearly, the factorial function does not terminate for negative arguments, and we expect the abstract interpreter to return an analysis result that represents non-termination. Instead, the abstract interpreter itself diverges:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \mathsf{n} \mapsto [-\infty,-2] \vdash \mathtt{if(n == 0)\ 1\ else\ fact(n-1)*n} \Downarrow ?\\ \vdots\end{array}}{\mathsf{n} \mapsto [-\infty,-1] \vdash \mathtt{if(n == 0)\ 1\ else\ fact(n-1)*n} \Downarrow ?}}{\mathsf{n} \mapsto [-\infty,-1] \vdash \mathtt{fact(n)} \Downarrow ?}$$

The abstract interpreter analyzes the factorial function with smaller and smaller intervals, because factorial decrements its argument on every recursive call. Even though the intervals become smaller, the chain of recursive calls is still infinite. Therefore, the fixpoint algorithm never encounters a recurrent recursive call. This means that a fixpoint algorithm that solves the first challenge still may not terminate. This leads us to our second challenge:

Challenge 2 *A big-step fixpoint algorithm has to ensure that all possibly infinite call chains have a recurrent call.*

In other words, all call chains are either finite or repeat themselves after finitely many calls. This ensures that a fixpoint can find a recurrent call even in infinite call chains.

While the first and second challenge concern the inputs of the abstract interpreter, the third challenge concerns its outputs. To illustrate this challenge, we have to switch to another example, namely the multiplication of Peano numbers. Consider an interval analysis of the multiplication function where we initially bind `m` to $[1,\infty]$ and `n` to $[1,1]$.

$$\dfrac{\dfrac{\mathtt{n} \mapsto [1,1] \vdash \mathtt{n} \Downarrow [1,1] \qquad \dfrac{\dfrac{\mathtt{m} \mapsto [1,\infty], \mathtt{n} \mapsto [1,1] \vdash \mathtt{if(m == 1)\ n\ else\ mult(m-1,n)+n} \Downarrow X}{\mathtt{m} \mapsto [2,\infty], \mathtt{n} \mapsto [1,1] \vdash \mathtt{mult(m-1,n)} \Downarrow X}}{\mathtt{m} \mapsto [2,\infty], \mathtt{n} \mapsto [1,1] \vdash \mathtt{mult(m-1,n)+n} \Downarrow X + [1,1]}}{\mathtt{m} \mapsto [1,\infty], \mathtt{n} \mapsto [1,1] \vdash \mathtt{if(m == 1)\ n\ else\ mult(m-1,n)+n} \Downarrow [1,1] \sqcup (X + [1,1])}}{\mathtt{m} \mapsto [1,\infty], \mathtt{n} \mapsto [1,1] \vdash \mathtt{mult(m,n)} \Downarrow ?}$$

The right branch of the derivation tree contains a recurrent call of `mult`. In this example, we represent the result of the recurrent call with a symbolic variable X. By tracing back the result to the initial call of `mult`, we end up with the recursive equation $X = [1,1] \sqcup (X + [1,1])$, which a fixpoint algorithm needs to solve. An established technique for solving such an equation is to start with the empty interval $\bot$ and then to apply iteratively the equation until the interval does not change anymore [Cousot and Cousot 1992b]. However, when we apply this technique to the equation above, we do not reach a fixpoint in a finite number of steps:

$$X_0 = \bot \quad X_1 = [1,1] \sqcup (X_0 + [1,1]) = [1,1] \quad X_2 = [1,1] \sqcup (X_1 + [1,1]) = [1,2] \quad \ldots$$

This example shows that even if a big-step fixpoint algorithm ensures and detects recurrent calls, it still might iterate on the analysis result indefinitely. This leads us to the third and final challenge:

Challenge 3 *A big-step fixpoint algorithm may only iterate on a result a finite number of times.*

The third challenge is well-known and is not unique to big-step abstract interpreters. There are standard solutions to this challenge, such as finite abstract domains or widening operators [Cousot and Cousot 1992b].

These challenges and one extra condition are sufficient to guarantee termination:

Theorem 5.2.1 (Termination). *A big-step fixpoint algorithm terminates if it solves the three termination challenges and the derivation tree has a finite branching factor.*

Proof. Challenge 1 and *2* ensure that each infinite call chain is eventually cut off at a recurrent call and hence is finite. Finite call chains and a finite branching factor guarantee that the big-step derivation tree is finite. Lastly, *Challenge 3* ensures that the fixpoint algorithm iterates on the analysis result for each node of the tree finitely many times. Therefore, the fixpoint algorithm terminates. □

We already explained how a fixpoint algorithm can solve the third challenge. Hence, in the remainder of this section, we focus on solutions for the first two challenges.

5.2.2 Detecting Recurrent Calls and Cutting off Recursion

The first challenge is to detect recurrent recursive calls of the abstract interpreter within a call chain and to cut off recursion to avoid non-termination. A solution to this challenge is described by Schmidt [1995]. In particular, the fixpoint algorithm registers recursive calls of the abstract interpreter on a stack. Whenever a fixpoint algorithm calls the abstract interpreter recursively, it first checks if a call with the same arguments already occurs on the stack. If it does, the call is recurrent and the fixpoint algorithm has to stop recursing deeper to avoid non-termination. Otherwise, the fixpoint algorithm pushes the call to the stack and continues recursing, popping the stack upon return.

The key question is what to do when a recurrent call occurs. Recall that the fixpoint algorithm may not recurse further, yet it must return an analysis result. But which analysis result should it be? Of course, the fixpoint algorithm could simply return $\top$, which is definitely a sound over-approximation. However, this would surrender precision of the underlying analysis altogether. Ideally, we would like to guess the exact analysis result of the recurrent call, but this is impossible in general. Instead, we return $\bot$ at the recurrent call and then repeatedly analyze the program between the first call and the recurrent call until we reach a fixpoint.

Below we illustrate Schmidt's solution to this challenge. In particular, we show the first three iterations of a big-step fixpoint algorithm for the interval analysis of the factorial function.

1. **Iteration** In the first iteration, the algorithm starts to analyze the program until it detects the recurrent call on the stack. At this point, the algorithm cuts off recursion by returning the empty interval $\bot$ to avoid non-termination. It then continues normally until it returns to the

original call with result $[1{,}1] \sqcup \bot = [1{,}1]$. Stopping fixpoint iteration at this point would be unsound, because the interval $[1{,}1]$ underapproximates the correct analysis result $[1{,}\infty]$.

$$\dfrac{\dfrac{\mathtt{n} \mapsto [0{,}0] \vdash \mathtt{1} \Downarrow [1{,}1] \qquad \dfrac{\dfrac{\mathtt{n} \mapsto [0{,}\infty] \vdash \mathtt{if(n == 0)\ 1\ else\ fact(n - 1) * n} \Downarrow \bot}{\mathtt{n} \mapsto [1{,}\infty] \vdash \mathtt{fact(n - 1)} \Downarrow \bot}}{\mathtt{n} \mapsto [1{,}\infty] \vdash \mathtt{fact(n - 1) * n} \Downarrow \bot}}{\mathtt{n} \mapsto [0{,}\infty] \vdash \mathtt{if(n == 0)\ 1\ else\ fact(n - 1) * n} \Downarrow [1{,}1] \sqcup \bot}}{\mathtt{n} \mapsto [0{,}\infty] \vdash \mathtt{fact(n)} \Downarrow ?}$$

2. **Iteration** In the second iteration, the fixpoint algorithm reanalyzes the factorial function. However, this time, at the recurrent call the algorithm returns the result of the previous iteration $[1{,}1]$ instead of $\bot$. It continues normally until returning to the original call, whose analysis result now becomes $[1{,}1] \sqcup [1{,}\infty] = [1{,}\infty]$. Because the new analysis result is greater compared to the previous iteration, the algorithm may not have reached a fixpoint yet and must iterate again.

$$\dfrac{\dfrac{\mathtt{n} \mapsto [0{,}0] \vdash \mathtt{1} \Downarrow [1{,}1] \qquad \dfrac{\dfrac{\mathtt{n} \mapsto [0{,}\infty] \vdash \mathtt{if(n == 0)\ 1\ else\ fact(n - 1) * n} \Downarrow [1{,}1]}{\mathtt{n} \mapsto [1{,}\infty] \vdash \mathtt{fact(n - 1)} \Downarrow [1{,}1]}}{\mathtt{n} \mapsto [1{,}\infty] \vdash \mathtt{fact(n - 1) * n} \Downarrow [1{,}\infty]}}{\mathtt{n} \mapsto [0{,}\infty] \vdash \mathtt{if(n == 0)\ 1\ else\ fact(n - 1) * n} \Downarrow [1{,}1] \sqcup [1{,}\infty]}}{\mathtt{n} \mapsto [0{,}\infty] \vdash \mathtt{fact(n)} \Downarrow ?}$$

3. **Iteration** In the third iteration, the fixpoint algorithm reanalyze the factorial function again, using the previous analysis result $[1{,}\infty]$ at the recurrent call. However, the analysis result does not grow further, which indicates that we have found a fixpoint.

To summarize, the fixpoint algorithm detects recurrent calls of the abstract interpreter by remembering calls of the abstract interpreter on a stack. When the algorithm encounters a recurrent call, it yields $\bot$ at first and then iterates the call until reaching a fixpoint.

5.2.3 Ensuring Recurrent Calls in Infinite Call Chains

The second challenge is to ensure that every infinite call chain has a recurrent call after a finite amount of steps. Unfortunately, there is no decidable technique for detecting if a call chain is finite or infinite, due to the halting problem. To solve this problem, we use a technique by Schmidt [1995], that introduces an *artificial recurrent calls* into a *possibly infinite call chain* by exploiting the monotonicity of the abstract interpreter.

For example, consider the infinite call chain from Section 5.2.1:

$$(n \mapsto [-\infty,-1], \mathtt{fact(n)}) \quad (n \mapsto [-\infty,-2], \mathtt{fact(n)}) \quad \ldots$$

We can introduce a recurrent call into this chain by replacing the second call with the first call. This is sound because the interval $[-\infty,-1]$ is greater than $[-\infty,-2]$ and monotonicity of the abstract interpreter guarantees that the analysis result for the call $[-\infty,-1]$ overapproximates the analysis result for the call $[-\infty,-2]$.

To ensure that we find such recurrent call, we define a widening operator [Cousot and Cousot 1992b] for stacks:

$$\begin{aligned}
&\nabla_{\widehat{\mathtt{Stack}}} : \widehat{\mathtt{Stack}} \times (\widehat{\mathtt{Env}} \times \mathtt{Expr}) \longrightarrow \widehat{\mathtt{Stack}} \times (\widehat{\mathtt{Env}} \times \mathtt{Expr}) \\
&s \; \nabla_{\widehat{\mathtt{Stack}}} \; (\rho, e) = \\
&\qquad \begin{cases}
(s, (\rho', e)) & \text{if } e \in \mathrm{dom}(s) \land \rho \sqsubseteq s(e), \text{let } \rho' = s(e) \\
(s[e \mapsto \rho'], (\rho', e)) & \text{if } e \in \mathrm{dom}(s) \land \rho \not\sqsubseteq s(e), \text{let } \rho' = s(e) \; \nabla_{\widehat{\mathtt{Env}}} \; (s(e) \sqcup \rho) \\
(s[e \mapsto \rho], (\rho, e)) & \text{if } e \notin \mathrm{dom}(s)
\end{cases}
\end{aligned}$$

The abstract stack ($\widehat{\mathtt{Stack}} = \mathtt{Expr} \rightharpoonup \widehat{\mathtt{Env}}$) maps an expression (the body of a function) to an environment that binds its arguments. In case the expression appeared on the stack and the environment of the call is smaller than the environment on the stack, the stack widening operator introduces a recurrent call by reusing the environment on the stack. In case the environment on the stack is not an upper bound of the environment in the call, the stack widening operator joins both environments. The least upper bound $s(e) \sqcup \rho$ ensures that the sequence of environments

passed to $\nabla_{\widehat{\mathsf{Env}}}$ is an ascending chain. Lastly, in case the expression did not occur on the stack, the operator adds the call to the stack without changing it.

The following example illustrates how the stack widening operator introduces a recurrent call into an otherwise infinite, non-repeating call chain:

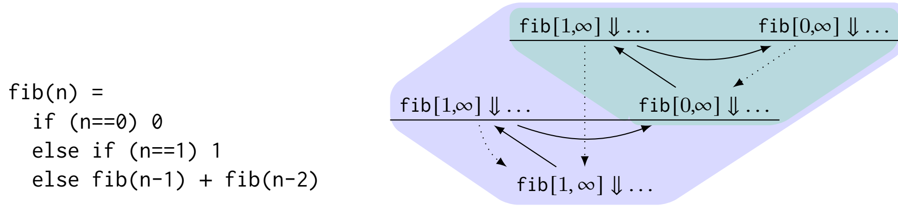
$$\dfrac{\dfrac{n \mapsto [-\infty,-2] \overset{\nabla_{\widehat{\mathsf{Stack}}}}{\rightsquigarrow} n \mapsto [-\infty,-1] \vdash \mathtt{if(n == 0)\ 1\ else\ fact(n-1) * n} \Downarrow \bot}{\vdots}}{\dfrac{n \mapsto [-\infty,-1] \vdash \mathtt{if(n == 0)\ 1\ else\ fact(n-1) * n} \Downarrow \bot}{n \mapsto [-\infty,-1] \vdash \mathtt{fact(n)} \Downarrow \bot}}$$

At the second recursive call of factorial, the operator detects that factorial has been called with a greater environment $\mathtt{n} \mapsto [-\infty,-1]$ further up the stack and replaces the current environment. This replacement allows the fixpoint algorithm to terminate with $\bot$, which represents non-termination of the factorial function.

5.2.4 A Big-Step Fixpoint Algorithm that iterates on Strongly-Connected Components of the Call Graph

In this section, we combine the solutions to the termination challenges to develop a fixpoint algorithm for big-step abstract interpreters. The fixpoint algorithm targets the simple functional language from Section 5.2, but we generalize it in Section 5.3 by making it language-independent and modular.

The fixpoint algorithm iterates on the strongly-connected components (SCCs) of the call graph of the abstract interpreter [Bourdoncle 1993]. An SCC is a set of calls from which it is possible to reach all other calls in the same set. For example, consider the following call graph for the analysis of the fibonacci function:



The graph has an *outer* SCC and an *inner* SCC indicated by the differently shaded areas. The solid arrows indicate calls in the order in which they are executed by the abstract interpreter. The dotted arrows indicate recurrent calls. We write $\mathtt{fib}[i,j]$ as a shorthand for a call $\mathtt{n} \mapsto [i,j] \vdash \mathtt{fib(n)}$.

To compute a fixpoint, the algorithm has to iterate on all calls in the body of an SCC. The order in which the fixpoint algorithm iterates over the calls does not matter for soundness [Bourdoncle 1993]. However, the order matters for performance and precision of the analysis. In this section we present an algorithm that prioritizes calls in the *innermost* SCCs, before iterating on the outer SCCs.

Since we do not know the call graph of the program a priori, our fixpoint algorithm has to discover SCCs during the analysis. To detect SCCs, our algorithm tracks recurrent calls, because some recurrent calls are the entry calls of SCCs. For example, in the call graph of the fibonacci function above the recurrent call $\mathtt{fib}[0,\infty]$ points to the entry call of the inner SCC (rightmost dotted arrow), whereas the recurrent calls of $\mathtt{fib}[1,\infty]$ point to the entry call of the outer SCC. To detect the *innermost* SCCs, the algorithm looks for the first recurrent call that it encounters upon returning.

Function $\mathtt{fix}_{\mathtt{monolithic}}$ in Listing 5.1 captures the main fixpoint algorithm. We adapt the abstract interpreter $\widehat{\mathtt{eval}}$ (not shown) to call $\mathtt{fix}_{\mathtt{monolithic}}$ in place of recursion, and $\widehat{\mathtt{fix}}_{\mathtt{monolithic}}$ calls $\widehat{\mathtt{eval}}$. This allows us to encapsulate the fixpoint logic in $\mathtt{fix}_{\mathtt{monolithic}}$, whereas $\widehat{\mathtt{eval}}$ captures the rest of the abstract language semantics. Our fixpoint algorithm uses three data structures: A stack to detect recurrent calls, a cache to remember intermediate analysis result, and a set of calls that are heads of SCCs.

```
1   Stack = Map Expr Env      Cache  = Map Call (Stable,Val)     SCC = Set Call
2   Call  = (Env,Expr)        Stable = Stable | Unstable
3
4   fix_monolithic :: Call → Stack → Cache → (Val, Cache, SCC)
5   fix_monolithic call stack cache
6     | not (isFunctionBody call)           = eval call stack cache
7     | call∈cache && isStable cache(call)   = (cache(call), cache, ∅)
8     | call∈cache && isUnstable cache(call) = (cache(call), cache, {call})
9     | call∉cache && call∈stack             = (⊥,          cache, {call})
10    | call∉cache && call∉stack             = iterate call stack cache
11
12  iterate :: Call → Stack → Cache → (Val, Cache, SCC)
13  iterate call stack cache_1 =
14    let (stack_widened, call_widened) = stack ∇_Stack call
15        (val_new, cache_2, scc)      = eval call_widened stack_widened cache_1
16    if call_widened ∈ scc then
17      let val_old     = if call_widened ∈ cache_2 then cache_2(call_widened) else ⊥
18          val_widened = val_old ∇_Val val_new
19          stable    = if val_old ⊄ val_new && scc=={call_widened} then Stable else Unstable
20          cache_3   = cache_2[call_widened ↦ (stable,val_widened)]
21      if val_old ⊏ val_widened
22      then iterate call stack cache_3
23      else (val_widened, cache_3, scc \ {call_widened})
24    else (val_new, cache_2, scc)
```

Listing 5.1: Big-step fixpoint algorithm iterating on the innermost strongly connected component. The code uses common mathematical notation for operations on maps and sets for readability. In particular, the notation `cache(call)` looks up the key `call` in the map `cache` and the notation cache[call ↦ res] updates the map entry `call` to `res`. Furthermore, {call} refers to the singleton set with the element `call`.

The algorithm first checks in line 6, if the expression is a function body and hence a potentially diverging call. In case the expression is not a function body, for instance, in case of arithmetic or boolean expressions, the algorithm simply recurses without iteration. This not only saves analysis time, but also reduces the size of the stack and cache tremendously. In case the expression is a function body, the algorithm then checks if the cache contains a stable analysis result for the call and returns this result in line 7. This avoids the redundant reanalysis of this call and is sound because stable results are guaranteed to overapproximate the least fixpoint of $\widehat{\texttt{eval}}$. In case the cache only contains an unstable or no analysis result, the algorithm checks if the call `(env,expr)` is a recurrent call by searching for it on the stack. In case of a recurrent call, the algorithm solves *Challenge 1* by either returning the unstable analysis result (line 8) or returning ⊥ (line 9). Furthermore, since the analysis result needs to be iterated on, the algorithm adds its call to the SCC set. If the call does not appear on the stack, the algorithm calls a recursive helper function `iterate` (line 10), that iterates the analysis result until it stabilizes.

Function `iterate` is responsible for iterating on calls in SCCs. The first line of `iterate` applies the stack widening operator $\nabla_{\texttt{Stack}}$ of Section 5.2.2 to the call. This ensures that all infinite non-repeating stacks eventually have a recurrent call (*Challenge 2*). In line 15, the algorithm then calls the abstract interpreter $\widehat{\texttt{eval}}$ with the widened inputs. The algorithm then iterates on the call, in case the call is a head of an SCC (line 16), or otherwise simply returns the result of $\widehat{\texttt{eval}}$ (line 24). In line 18, the algorithm uses a conventional widening operator for values $\nabla_{\widehat{\texttt{Val}}}$ to ensure that the analysis result does not grow indefinitely (*Challenge 3*). If the widened value is strictly greater than the cached value, the algorithm keeps iterating (line 21), otherwise it returns the widened value (line 23).

The following example illustrates how this algorithm works. In particular, we show the first four iterations of the $\texttt{fix}_{\texttt{monolithic}}$ fixpoint algorithm for the analysis of the fibonacci function:

$$\text{I}\;\frac{\langle \texttt{fib}[1,\infty],\sigma_1\rangle \Downarrow \langle \bot,\sigma_1,\theta_1\rangle \qquad \dfrac{\langle \texttt{fib}[1,\infty],\sigma_1\rangle \Downarrow \langle \bot,\sigma_1,\theta_1\rangle \qquad \langle \texttt{fib}[0,\infty],\sigma_1\rangle \Downarrow \langle \bot,\sigma_1,\theta_0\rangle}{\langle \texttt{fib}[0,\infty],\sigma_1\rangle \Downarrow \langle [0,1]\sqcup[1,1]\sqcup(\bot+\bot),\sigma_2,\theta_0\cup\theta_1\rangle}}{\langle \texttt{fib}[1,\infty],\sigma_1\rangle \Downarrow\, ?}$$

$$\text{II}\ \frac{\langle \texttt{fib}[1,\infty],\sigma_1\rangle \Downarrow \langle \bot,\sigma_1,\theta_1\rangle \quad \dfrac{\langle \texttt{fib}[1,\infty],\sigma_2\rangle \Downarrow \langle \bot,\sigma_2,\theta_1\rangle \quad \langle \texttt{fib}[0,\infty],\sigma_2\rangle \Downarrow \langle [0,1],\sigma_2,\theta_0\rangle}{\langle \texttt{fib}[0,\infty],\sigma_2\rangle \Downarrow \langle [0,0]\sqcup[1,1]\sqcup(\bot+[0,1]),\sigma_2,\theta_0\cup\theta_1\rangle}}{\langle \texttt{fib}[1,\infty],\sigma_1\rangle \Downarrow \langle [1,1]\sqcup(\bot+[0,1]),\sigma_3,\theta_1\rangle}$$

$$\text{III}\ \frac{\langle \texttt{fib}[1,\infty],\sigma_3\rangle \Downarrow \langle [1,1],\sigma_3,\theta_1\rangle \quad \dfrac{\langle \texttt{fib}[1,\infty],\sigma_3\rangle \Downarrow \langle [1,1],\sigma_3,\theta_1\rangle \quad \langle \texttt{fib}[0,\infty],\sigma_3\rangle \Downarrow \langle [0,1],\sigma_3,\theta_0\rangle}{\langle \texttt{fib}[0,\infty],\sigma_3\rangle \Downarrow \langle [0,0]\sqcup[1,1]\sqcup([1,1]+[0,1]),\sigma_4,\theta_0\cup\theta_1\rangle}}{\langle \texttt{fib}[1,\infty],\sigma_3\rangle \Downarrow ?}$$

$$\text{IV}\ \frac{\langle \texttt{fib}[1,\infty],\sigma_3\rangle \Downarrow \langle [1,1],\sigma_3,\theta_1\rangle \quad \dfrac{\langle \texttt{fib}[1,\infty],\sigma_4\rangle \Downarrow \langle [1,1],\sigma_4,\theta_1\rangle \quad \langle \texttt{fib}[0,\infty],\sigma_4\rangle \Downarrow \langle [0,\infty],\sigma_4,\theta_0\rangle}{\langle \texttt{fib}[0,\infty],\sigma_4\rangle \Downarrow \langle [0,0]\sqcup[1,1]\sqcup([1,1]+[0,\infty]),\sigma_4,\theta_0\cup\theta_1\rangle}}{\langle \texttt{fib}[1,\infty],\sigma_3\rangle \Downarrow \langle [1,1]\sqcup([1,1]+[0,\infty]),\sigma_5,\theta_1\rangle}$$

$$\begin{aligned}
\sigma_1 &= \varnothing & \sigma_4 &= \sigma_3[\texttt{fib}[0,\infty] \mapsto (\texttt{Unstable}, [0,\infty])] \\
\sigma_2 &= \sigma_1[\texttt{fib}[0,\infty] \mapsto (\texttt{Unstable}, [0,1])] & \sigma_5 &= \sigma_4[\texttt{fib}[1,\infty] \mapsto (\texttt{Unstable}, [1,\infty])] \\
\sigma_3 &= \sigma_2[\texttt{fib}[1,\infty] \mapsto (\texttt{Unstable}, [1,1])] & \sigma_6 &= \sigma_5[\texttt{fib}[1,\infty] \mapsto (\texttt{Stable}, [1,\infty])] \\
\theta_0 &= \{\texttt{fib}[0,\infty]\} & \theta_1 &= \{\texttt{fib}[1,\infty]\}
\end{aligned}$$

To make the internals of the fixpoint algorithm visible, we write $\langle e,\sigma\rangle \Downarrow \langle v,\sigma',\theta\rangle$ for a call e that evaluates to the interval v, where σ and σ' are the input and output cache and θ the SCC. An iteration starts and ends when the algorithm recursively calls `iterate` again (line 22). A question mark indicates that the algorithm did not return from a call within an iteration. For brevity, we omit the stack and only show the evaluation of function calls. The highlighting indicates what changed in the analysis result compared to the previous iteration.

To summarize, we combined the solutions to the termination challenges into a big-step fixpoint algorithm, that iterates on the strongly-connected component of the call graph. We prove soundness of this algorithm in Section 5.4.

5.3 Modularizing the Description of Big-Step Fixpoint Algorithms

In the previous section, we discussed a big-step fixpoint iteration that iterates on the strongly-connected components of the call graph. Even though the initial fixpoint algorithm works and is sound, it is hard to reuse parts of its implementation for another analysis, or to experiment with it and fine-tune it. We can solve these problems by *modularizing* the description of the fixpoint algorithm, which we discuss in this section. The modularization description of fixpoint algorithms is novel and is the core contribution of this paper.

5.3.1 Why Modularization Matters

Before presenting our framework for modular fixpoint algorithms, we motivate the need for modularization through series of examples.

New language or analysis Imagine we want to implement a sign-analysis for an imperative language. For example, consider the following trace of the factorial function implemented with loops and mutable variables:

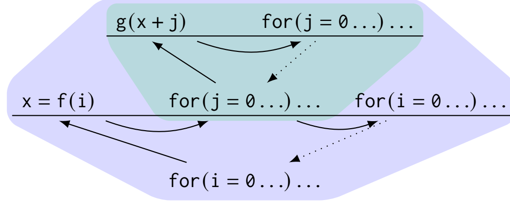
$$\frac{\mathtt{n} \mapsto \mathtt{PosZero} \vdash \mathtt{n} > 1 \Downarrow \top \quad \dfrac{\dfrac{\vdots}{\mathtt{n} \mapsto \mathtt{PosZero}, \mathtt{x} \mapsto \mathtt{Pos} \vdash \mathtt{while(n >= 1)\{x = x * n; n = n - 1\}} \Downarrow ?}}{\mathtt{n} \mapsto \mathtt{PosZero}, \mathtt{x} \mapsto \mathtt{Pos} \vdash \mathtt{x = x * n; n = n - 1; while(n >= 1)\{x = x * n; n = n - 1\}} \Downarrow ?}}{\mathtt{n} \mapsto \mathtt{PosZero}, \mathtt{x} \mapsto \mathtt{Pos} \vdash \mathtt{while(n >= 1)\{x = x * n; n = n - 1\}} \Downarrow ?}$$

The analysis calculates the sign of numeric values [Cousot and Cousot 1977], where `Pos` stands for a strictly positive number and `PosZero` for a positive number including zero. In the trace, the abstract interpreter unrolls the body of the while loop, until it hits the same call of the while loop again and diverges.

Implementing a fixpoint algorithm for this analysis requires solving the same termination challenges that we discussed in the previous section. Ideally, we would like to reuse (parts of) our initial fixpoint algorithm $\texttt{fix}_{\texttt{monolithic}}$ from the previous section. Unfortunately, the fixpoint algorithm is language-specific and analysis-specific: it refers to specific data types for environments, expressions, and values that are incompatible with our imperative language and sign domain. Moreover, the fixpoint algorithm only considers function calls as a source of non-termination and thus would ignore while loops altogether.

Fine-tuning fixpoint algorithms Fixpoint algorithms can be fine-tuned to improve performance and precision. For example, consider a numeric analysis of the following program:

```
for(i = 0; i < m; i++) {
  x = f(i);
  for(j = 0; j < n; j++) {
    g(x + j);
  }
}
```



The call graph of the abstract interpreter for this program consists of two SCCs: One SCC for the outer loop and one SCC for the inner loop. Our fixpoint algorithm $\texttt{fix}_{\texttt{monolithic}}$ iterates on the inner SCC, reanalyzing the function call `g(x + j)` until its analysis result stabilizes. However, depending on the functions `f` and `g`, it can be faster to iterate on the *outermost* SCC first. For example, if `x = f(i)` yields the analysis result $\top$ after the second iteration of the outer loop, then we would potentially waste analysis time trying to analyze `g(x + j)` precisely in the first iteration of the inner loop.

The example shows that it is difficult to find a single fixpoint algorithm that works best for all analyzed programs. Therefore, developing a fixpoint algorithm requires experimentation and careful fine-tuning. Ideally, we would like to configure and fine-tune a fixpoint algorithm without having changing its implementation. Unfortunately, it is difficult to fine-tune our initial fixpoint algorithm $\texttt{fix}_{\texttt{monolithic}}$, because it is implemented as a monolithic function. For example, changing the iteration order of $\texttt{fix}_{\texttt{monolithic}}$ requires significant and invasive changes to its implementation. This is problematic because fixpoint algorithms are, by the nature of the problem, difficult to understand and hard to get right. Moreover, by changing the implementation we risk breaking soundness.

Modular soundness Even small extensions of a fixpoint algorithm risk breaking soundness. For example, we may want to extend the fixpoint algorithm to log call traces, which is useful for debugging amongst others. However, integrating call traces into our fixpoint algorithm $\texttt{fix}_{\texttt{monolithic}}$ requires changes to its signatures and implementation:

```
Trace = [TraceElement]            TraceElement = Call Call | Return Val

fix_monolithic :: Call → Stack → Cache → Trace → (Val, Cache, SCC, Trace)
fix_monolithic call stack cache₁ trace₁ = ...
  let (val, cache₂, scc, trace₂) = eval call stack cache₁ (Call call : trace₁)
  in (val, cache₁, scc, Return val : trace₂)
  ...

iterate :: Call → Stack → Cache → Trace → (Val, Cache, SCC, Trace)
iterate call stack cache₁ trace₁ =
  ...
  let (val_new, cache₂, scc, trace₂)
           = eval call_widened stack_widened cache₁ (Call call : trace₁)
      trace₃ = Return val_new : trace₂
  ...
```

Since we changed the implementation of $\texttt{fix}_{\texttt{monolithic}}$, any existing soundness proof becomes invalid. Indeed, the above code contains a bug: $\texttt{fix}_{\texttt{monolithic}}$ returns $\texttt{cache}_1$ instead of $\texttt{cache}_2$.

A modularized fixpoint algorithm allows us to implement extensions separately from one another and separately from the core algorithm. While it is inevitable to prove soundness for the extension of the fixpoint algorithm, we want to preserve the soundness of other components. As we will show in Section 5.4, modularized fixpoint algorithms permit compositional soundness proofs.

5.3.2 A Framework for Modularized Fixpoint Algorithms

In this subsection, we introduce a framework that allows us to modularize the description of fixpoint algorithms. We illustrate this framework by refactoring the function $\texttt{fix}_{\texttt{monolithic}}$ into smaller reusable fixpoint combinators. Additionally, this framework will enable us to prove soundness of the fixpoint algorithm modularly, which we discuss in Section 5.4.

Language-Independence The problem that makes function $\texttt{fix}_{\texttt{monolithic}}$ language-dependent is that it refers to the abstract interpreter $\widehat{\texttt{eval}}$, environments, expressions, and values from the analyzed language directly. To make the algorithm language-independent, we first have to remove references to language-specific types. As first step, we replace the inputs $(\widehat{\texttt{Env}}, \texttt{Expr})$ and outputs $\widehat{\texttt{Val}}$ of the abstract interpreter with the type variables a and b.

```
Stack a = Set a          Cache a b = Map a b          SCC a = Set a
```

As second step, we remove the reference to $\widehat{\texttt{eval}}$ by turning it into an open-recursive style and passing its body as an argument to $\texttt{fix}_{\texttt{monolithic}}$. This allows us to implement $\texttt{fix}_{\texttt{monolithic}}$ independently of the analyzed language.

```
fix_monolithic :: (a ⇓ b → a ⇓ b) → a ⇓ b
eval = fix_monolithic (λev (env,expr) → case expr of ... ev ...)
```

To this end, we introduced a type (⇓) to represent the type of fixpoint computation:

```
a ⇓ b = (a, Stack a, Cache a b) → (Cache a b, SCC a, b)
```

We refrain from showing the new code of $\texttt{fix}_{\texttt{monolithic}}$ until after the second refactoring.

Reusable Fixpoint Combinators To make the fixpoint algorithm easier to adapt to a new analysis, we have to make two more changes. First, instead of implementing one single monolithic fixpoint algorithm, we split its functionality across multiple smaller fixpoint combinators $\varphi_1, \ldots \varphi_n$. These combinators are then called by a function fix in a round-robin fashion, such that each combinator has the chance to affect the fixpoint computation:

```
fix_φ :: (a ⇓ b → a ⇓ b) → a ⇓ b          φ :: a ⇓ b → a ⇓ b
fix_φ f = φ (f (fix_φ f))                  φ = λx. φ1(φ2(... φn(x) ...))
```

In particular, $\texttt{fix}_\varphi\ \widehat{\texttt{eval}}$ first invokes combinator φ_1, which then may invoke φ_2, and so on, until eventually φ_n calls $\widehat{\texttt{eval}}$ and the cycle repeats.

Even though this design of fixpoint combinators allows us to separate concerns, their type a ⇓ b is not fully extensible, as some combinators may need some extra data not present in the stack or the cache. Therefore, as second change, we generalize the type a ⇓ b to an arrow type c a b [Hughes 2000]. The arrow type reads as "some effectful computation c that takes values of type a as input and produces values of type b as output." Arrows allow us to implement fixpoint combinators without having to refer to a specific type of fixpoint computation. They are particularly useful for implementing big-step fixpoint algorithms, because they cleanly separate the inputs of an effectful computation from the outputs. Moreover, they have proven useful for modularizing other parts of the abstract interpreter [Keidel et al. 2018; Keidel and Erdweg 2019, 2020].

```
1  φ_innermost :: ∀ c a b. ... ⇒ c a b → c a b
2  φ_innermost f = proc call → do
3    (stable,result_cached) ← Cache.lookup ≺ call
4    if stable then return ≺ result_cached
5    else do
6      recurrentCall ← Stack.elem ≺ call
7      if recurrentCall then do
8        SCC.add ≺ call
9        return ≺ result_cached
10     else iterate f ≺ call
11
12 iterate :: ∀ c a b. ... ⇒ c a b → c a b
13 iterate f = proc call → do
14   (result_new) ← Stack.push f ≺ call
15   callInSCC ← SCC.elem ≺ call
16   if callInSCC then do
17     (grown, result_widened) ←
18       Cache.update ≺ (call, result_new)
19     if grown then iterate f ≺ call
20     else do
21       sizeSCC ← SCC.size ≺ ()
22       if sizeSCC == 1
23         then Cache.setStable ≺ call
24         else return ≺ ()
25       SCC.remove ≺ call
26       return ≺ result_widened
27   else return ≺ result_new
```

Listing 5.2: Fixpoint combinator that iterates on the innermost SCC of the call graph.

Refactoring the fixpoint algorithm $\texttt{fix}_{\texttt{monolithic}}$ Based on these principles, we now refactor the fixpoint algorithm $\texttt{fix}_{\texttt{monolithic}}$ into three reusable combinators $\varphi_{\texttt{chaotic}}$, $\varphi_{\texttt{filter}}$, and $\varphi_{\texttt{stackWiden}}$.

The combinator $\varphi_{\texttt{innermost}}$ (Listing 5.2) is a stripped down version of the $\texttt{fix}_{\texttt{monolithic}}$ algorithm and only solves *Challenge 1* and *3*. The combinator is parameterized by operations to access and modify the stack, cache and SCC contained in the effectful arrow computation. Furthermore, the code uses the following arrow notation: The keyword `proc x` introduces a new arrow computation that binds its argument to the variable x. The syntax `y ← f ≺ x` calls an arrow computation `f` with the argument `x` and binds the result to the variable `y`. Lastly, the keyword `return ≺ x` returns `x` as result of the arrow computation.

The combinator $\varphi_{\texttt{innermost}}$ first looks up the call in the cache. If the cached result is stable, the combinator simply returns the cached entry (line 4). Otherwise, the combinator looks up the call on the stack (line 6). In case of a recurrent call (line 7), the algorithm adds the call to the SCC and returns the cached entry. Otherwise, if the call did not appear on the stack (line 10), the algorithm calls the recursive helper function `iterate` that updates the analysis result until it does not grow anymore. The function `iterate` first calls the computation `f` while adding the current call to the stack. Afterwards it checks if the call occurred in the SCC (line 15) and hence needs to be iterated on. In case the call occurred in the SCC, function `iterate` updates the cache with the new result (line 18). The operation `Cache.update` simultaneously updates the cache, widens the new result against an existing entry and checks if the result is stable or has grown. In case the analysis result has grown (line 19) the function iterates again. Otherwise, it sets the cached result to stable, removes the call from the SCC and returns the widened result.

To solve *Challenge 2*, we implement a fixpoint combinator that applies a stack widening operator to the current call:

```
φ_stackWiden :: ∀ c a b. ... ⇒ (stack → a → (stack,a)) → c a b → c a b
φ_stackWiden ∇̂_Stack f = proc call → do
  stack ← Stack.ask ≺ ()
  let (stack_widened,call_widened) = stack_1 ∇̂_Stack call
```

```
  Stack.local f ≺ (stack_widened, call_widened)
```

Combinator $\varphi_{\texttt{stackWiden}}$ first accesses the stack contained in the arrow computation. It then applies the stack widening operator ($\nabla_{\widehat{\texttt{Stack}}}$) to this stack and current call. Afterwards it passes the widened call to the computation `f` and sets the new stack. The stack is the same stack that also combinator $\varphi_{\texttt{innermost}}$ uses to track recurrent calls.

Lastly, the higher-order fixpoint combinator $\varphi_{\texttt{filter}}$, inspired by Wei et al. [2019] `fix_select`, filters out calls not relevant to the rest of the fixpoint algorithm:

```
φ_filter :: ∀ c a b. ... ⇒ (a → Boolean) → (c a b → c a b) → (c a b → c a b)
φ_filter predicate φ f = proc call → do
  if predicate call then φ f ≺ call else f ≺ call
```

The combinator $\varphi_{\texttt{filter}}$ either calls the combinator φ whenever the predicate holds, or skips the combinator φ when the predicate does not hold.

With these three fixpoint combinators, we can recreate the fixpoint algorithm $\texttt{fix}_{\texttt{monolithic}}$ from the previous section:

$$\texttt{fix}_{\texttt{monolithic}} = \texttt{fix}_{\varphi}\ \widehat{\texttt{eval}} \qquad \varphi = \varphi_{\texttt{filter}}\ \texttt{isFunctionBody}\ (\varphi_{\texttt{stackWiden}}\ \nabla_{\widehat{\texttt{Stack}}} \circ \varphi_{\texttt{innermost}})$$

To summarize, in this section we proposed a framework for developing modular fixpoint algorithms. In particular, we describe fixpoint algorithms with reusable fixpoint combinators, where each combinator captures a certain aspect of the fixpoint algorithm.

5.3.3 Applying the Framework to Address the Modularization Challenges

In the previous subsection, we developed a framework for developing modular fixpoint algorithms by the means of reusable fixpoint combinators. In this subsection, we showcase how we can use this framework to address the modularization challenges described in Section 5.3.1.

New language or analysis We can use our framework to develop fixpoint algorithms for new languages and analyses more easily by reusing existing fixpoint combinators. We demonstrate this by developing a fixpoint algorithm for an analysis for an imperative language with loops and mutable variables. To this end, we require that the abstract interpreter is implemented with arrows:

```
Statement = Assign String Expr | While Expr [Statement] | ...
eval :: ∀ c. (Arrow c, ...) ⇒ c ([Statement], Store) Store
eval = fix_φ (λev → proc (stmts,store) → ... ev ...)
```

To develop a fixpoint algorithm for this language, we reuse existing fixpoint combinators that we developed in the previous subsection:

$$\varphi = \varphi_{\texttt{filter}}\ \texttt{isWhileLoop}\ (\varphi_{\texttt{stackWiden}}\ \nabla_{\texttt{Loop}} \circ \varphi_{\texttt{innermost}})$$

The only statements in this language that can diverge are while loops. Therefore, the combinator $\varphi_{\texttt{filter}}$ `isWhileLoop` filters out all other statements except for while loops. We can reuse combinator $\varphi_{\texttt{innermost}}$, because it is language-independent and analysis-independent. However, to reuse $\varphi_{\texttt{innermost}}$, we need to implement a new widening operator $\nabla_{\widehat{\texttt{Store}}}$ that ensures the abstract store does not grow indefinitely. We can also reuse the combinator $\varphi_{\texttt{stackWiden}}$ by implementing a new stack widening operator $\nabla_{\texttt{Loop}}$ that works for while loops and is similar to the operator in Section 5.2.3. The new stack widening operator joins subsequent iterations of the same while loop with $\nabla_{\widehat{\texttt{Store}}}$ until the store does not grow anymore. When the store does not grow anymore, the fixpoint algorithm detects a recurrent recursive call of the abstract interpreter and iterates on the SCC until the analysis result stabilizes.

To summarize, we were able to develop a new fixpoint algorithm for an imperative language, because we reused language-independent fixpoint combinators. We only needed to implement new language-specific and analysis-specific widening operators that ensure that the store and stack do not grow indefinitely.

Fine-tuning precision With our framework, we can fine-tune existing fixpoint algorithms by adding, replacing, or reordering fixpoint combinators. For example, the fixpoint algorithm we developed above is rather imprecise, because it joins subsequent iterations of the same while loop. A common technique to improve the precision of such an analysis is to unroll the first few iterations of a loop *without* joining their stores [Mauborgne and Rival 2005]. We can capture this loop unrolling with the following higher-order combinator $\varphi_{\texttt{unroll}}$:

```
φunroll :: ∀ c a b. ... ⇒ Int → (c a b → c a b) → (c a b → c a b)
φunroll limit φ f = proc call → do
  n ← getCallCount ≺ call
  if n ≤ limit then do
    incrementCallCount ≺ call
    f ≺ call
  else -- Call count exceeds limit
    φ f ≺ call
```

The operator $\varphi_{\texttt{unroll}}$ recursively calls the computation `f`, until it has seen the same call a certain number of times. When the count for a call exceeds the given limit, the operator $\varphi_{\texttt{unroll}}$ falls back to the fixpoint combinator φ.

We integrate this combinator into our fixpoint algorithm from above by preventing stack widening for the first 10 iterations of a loop:

```
φfilter isWhileLoop (φunroll 10 (φstackWiden ∇Loop) ∘ φinnermost)
```

Fine-tuning performance Next, we investigate how we can fine-tune the performance of the fixpoint algorithm above. As we explained in Section 5.3.1, for some programs, it can be faster to iterate on the outer SCCs instead of the inner SCCs. We can fine-tune our fixpoint algorithm from above for this use case by adding a fixpoint combinator $\varphi_{\texttt{outermost}}$:

```
φoutermost :: ∀ c a b. ... ⇒ c a b → c a b
φoutermost = ... iterate ...
  where iterate = proc call → do
          (resultnew) ← Stack.push f ≺ call
          callInSCC ← SCC.elem ≺ call
          sizeSCC ← SCC.size ≺ ()
          if callInSCC && sizeSCC == 1 then do
           (grown,reswidened) ← Cache.update ≺ (call,resnew)
            if grown then iterate f ≺ call
            else do
              SCC.remove ≺ call
              return ≺ resultwidened
          else return ≺ resultnew
```

The combinator $\varphi_{\texttt{outermost}}$ iterates on the outermost SCC. Its implementation is similar to $\varphi_{\texttt{innermost}}$, except that it returns to the head of the outermost component before iterating. The combinator verifies that a call is the head of the outermost component by checking the size of the calls in the component (see highlighting). To pick an iteration strategy on a case-by-case basis, we add another fixpoint combinator $\varphi_{\texttt{alternative}}$ that dispatches an incoming call based on a predicate:

```
φalternative :: ∀ c a b. ...
             ⇒ c a Boolean → (c a b→c a b) → (c a b→c a b) → (c a b→c a b)
φalternative predicate φ1 φ2 f = proc call → do
  b ← predicate ≺ call
  if b then φ1 f ≺ call else φ2 f ≺ call
```

The predicate is an effectful computation, which allows it to remember a decision for a call and all subsequent recursive calls. With these new combinators, we fine-tune the algorithm by selectively iterating on the innermost or outermost SCC:

```
φfilter isWhileLoop (φunroll 10 (φstackWiden ∇Loop) ∘ φalternative metric φinnermost φoutermost)
```

Modular soundness Now we want to extend the existing fixpoint algorithm from above such that it records a control-flow graph (CFG). A CFG describes the order in which statements of the program are evaluated. Since the control-flow of a program is encoded implicitly in the big-step abstract interpreter, all we need to do is implement a fixpoint combinator that adds an edge to the graph whenever the abstract interpreter evaluates a statement:

```
φCFG :: ∀ c a b. ... ⇒ (c a b → c a b)
φCFG f = proc call → do
  pred ← getPredecessor ≺ ()
  CFG.addEdge ≺ (pred, call)
  withPredecessor f ≺ call
```

The CFG has calls of type `a` as nodes. The combinator adds an edge between the most recently evaluated call and the current call to the CFG. Afterwards it passes control to the computation `f`. We can integrate this use-case into the fixpoint algorithm above by adding the $\varphi_{\mathtt{CFG}}$ combinator to the front:

```
φCFG ∘ φfilter isWhileLoop (φunroll 10 (φstackWiden ∇Loop) ∘
    φalternative metric φinnermost φoutermost)
```

We can control the granularity of the CFG by changing the position of the $\varphi_{\mathtt{CFG}}$ combinator. For example, if we would move $\varphi_{\mathtt{CFG}}$ inside the $\varphi_{\mathtt{filter}}$ expression, the CFG would only contain loop statements as nodes.

Note that we did not change the implementation of any of the combinators to integrate this use-case. If we had a modular soundness theory for fixpoint combinators, the soundness proofs of reused combinators would remain valid and we only would have to prove soundness for $\varphi_{\mathtt{CFG}}$. Moreover, we could establish soundness for the entire fixpoint algorithm by composition of soundness lemmas about the relevant fixpoint combinators. As it turns out, our modularized fixpoint algorithms permit a modular soundness theory, which we develop in the next section.

5.4 Soundness of Modular Big-Step Fixpoint Algorithms

In this section, we develop a formal theory to prove soundness of big-step fixpoint algorithms that consists of fixpoint combinators.

We start by introducing definitions about fixpoints found in literature about domain theory [Abramsky 1994]. Let $(D, \sqsubseteq)$ be a complete preorder, where every finite subset $X \subseteq D$ has a least upper bound $\bigsqcup X$. Then given a function $f : D \to D$, all elements $x \in D$ with $f(x) = x$ are called fixpoints of f. If f is monotone, then the *least* fixpoint exists and can be characterized by

$$\mathrm{lfp}\, f = \bigsqcup_{n \in \mathbb{N}} f^n(\bot) \;=\; \bot \sqcup f(\bot) \sqcup f(f(\bot)) \sqcup \ldots$$

A fixpoint algorithm is sound if it overapproximates the least fixpoint of the concrete interpreter. To prove soundness [Cousot and Cousot 1979] of the fixpoints of a function $\widehat{f} : \widehat{D} \to \widehat{D}$ with respect to $f : D \to D$, we first need to define a Galois connection $\alpha : D \leftrightarrows \widehat{D} : \gamma$ that relates the concrete domain D with an abstract domain $\widehat{D}$. Function $\alpha : D \to \widehat{D}$ is called abstraction function and function $\gamma : \widehat{D} \to D$ is called concretization function. With this Galois connection, we prove soundness of the least fixpoints of f and $\widehat{f}$ by showing that $\alpha(\mathrm{lfp}\, f) \sqsubseteq \mathrm{lfp}\, \widehat{f}$, i.e. that $\mathrm{lfp}\, \widehat{f}$ overapproximates $\mathrm{lfp}\, f$.

To apply this soundness proposition to big-step fixpoint algorithms, we set the concrete domain $D := A \to B$[2] and abstract domain $\widehat{D} := \widehat{A} \Downarrow \widehat{B}$, where $\widehat{A} \Downarrow \widehat{B}$ is a computation from inputs $\widehat{A}$ to outputs $\widehat{B}$ of the abstract interpreter. Therefore, for a big-step fixpoint algorithm with multiple combinators, the soundness proposition from above specializes to

$$\alpha(\mathrm{lfp}\, \mathtt{eval}) \sqsubseteq \mathrm{lfp}(\varphi_1(\varphi_2(\ldots \varphi_n(\widehat{\mathtt{eval}}) \ldots))).$$

[2] Note that $D := A \to B$ does not mean that the concrete semantics needs to be given as a definitional interpreter in the same metalanguage as the abstract interpreter. The concrete semantics might as well be described as a mathematical definition in denotational semantics.

We could attempt to prove this directly, however, the proof would be unnecessarily complicated and volatile: whenever we change the fixpoint algorithm the soundness proof would become invalid.

Instead of a volatile monolithic soundness proof, we break down the soundness proof into smaller soundness lemmas for each combinator. Let us first assume that the fixpoint algorithm is built from fixpoint combinators over the following grammar:

$$\begin{aligned} \varphi &::= \varphi \text{ atomic} && \text{(atomic combinators)} \\ &\mid \varphi \circ \varphi && \text{(combinator composition)} \\ &\mid \varphi(\varphi \ldots \varphi) && \text{(higher-order combinators)} \end{aligned}$$

This grammar allows us to formulate the following soundness lemmas for each type of combinator:

$$\begin{aligned} \text{atomic } \varphi \text{ sound} \quad &\text{iff} \quad \forall \text{ monotone } \widehat{f}.\ \varphi \circ \widehat{f} \text{ is monotone } \wedge \ \text{lfp}\,\widehat{f} \sqsubseteq \text{lfp}(\varphi \circ \widehat{f}) \\ \varphi_1 \circ \varphi_2 \text{ sound} \quad &\text{iff} \quad \varphi_1 \text{ sound} \wedge \varphi_1 \text{ sound} \\ \text{higher-order } \varphi \text{ sound} \quad &\text{iff} \quad \forall \varphi_1 \ldots \varphi_n.\ \varphi_1 \text{ sound} \wedge \ldots \varphi_n \text{ sound} \\ &\qquad \Longrightarrow \ \varphi(\varphi_1 \ldots \varphi_n) \text{ sound} \end{aligned}$$

That is, an atomic fixpoint combinator is sound if it increases the least fixpoint of a computation $\widehat{f}$. It is important that the computation $\widehat{f}$ in this lemma is monotone and universally quantified, such that the lemma is provable and can be reused in different combinator expressions. The composition of two combinators is sound if both combinators are sound. Lastly, a higher-order combinator φ is sound if for all combinators $\varphi_1 \ldots \varphi_n$ with a soundness lemma, φ applied to these combinators is sound.

These soundness lemmas allow us to prove soundness of modular fixpoint algorithms once and for all with the following main theorem:

Theorem 5.4.1 (Soundness of Modular Fixpoint Algorithms). *A modular fixpoint algorithm φ is sound, if there is a soundness lemma for each combinator, a soundness proof of the abstract interpreter $\alpha(\texttt{eval}) \sqsubseteq \widehat{\texttt{eval}}$, and the abstract interpreter is monotone.*

Proof.

$$\begin{aligned} \alpha(\text{lfp}\,\texttt{eval}) &\sqsubseteq \text{lfp}\,\widehat{\texttt{eval}} && \text{by } \alpha(\texttt{eval}) \sqsubseteq \widehat{\texttt{eval}} \text{ and [Nielson et al. 1999, Lemma 4.42]} \\ &\sqsubseteq \text{lfp}(\varphi \circ \widehat{\texttt{eval}}) && \text{by induction over the structure of } \varphi. \end{aligned}$$

□

This way of proving soundness of modular fixpoint algorithms is more flexible than a monolithic proof, because it allows us to reorder and add new combinators without invalidating the soundness proof.

5.4.1 Soundness Proof Strategies for Fixpoint Combinators

In this subsection, we discuss proof strategies for three different kind of fixpoint combinators. These strategies allow us to prove soundness lemmas of fixpoint combinators more easily.

Extensive Fixpoint Combinators For our first proof strategy, we look at the example of the stack widening combinator $\varphi_{\texttt{stackWiden}}$ of Section 5.3. The combinator $\varphi_{\texttt{stackWiden}}(\nabla_{\widehat{\texttt{Stack}}})(\widehat{f})$ calls $\widehat{f}$ with an upper bound of the current input, i.e., $\widehat{f} \sqsubseteq \varphi_{\texttt{stackWiden}}(\nabla_{\widehat{\texttt{Stack}}})(\widehat{f})$. This observation leads us to our first proof strategy:

Theorem 5.4.2 (Soundness of Extensive Combinators). *Let φ be an extensive fixpoint combinator, i.e., $\text{id} \sqsubseteq \varphi$, then φ is sound.*

Proof. For all monotone $\widehat{f}$, it holds $\text{lfp}\,\widehat{f} = \text{lfp}(\text{id} \circ \widehat{f}) \sqsubseteq \text{lfp}(\varphi \circ \widehat{f})$ because φ is extensive. □

Corollary 5.4.3. The combinators $\varphi_{\texttt{stackWiden}}$ and $\varphi_{\texttt{CFG}}$ are sound by Theorem 5.4.2. □

Interleaving Fixpoint Combinators For the second proof strategy, we look at the higher-order combinator $\varphi_{\texttt{alternative}}$. The combinator $\varphi_{\texttt{alternative}}(P, \varphi_1, \varphi_2)(\widehat{f})$ *interleaves* the fixpoint computation by either calling $\varphi_1(\widehat{f})$ or $\varphi_2(\widehat{f})$ depending on predicate P. This means, for each call, we can either prove $\varphi_1(\widehat{f}) \sqsubseteq \varphi_{\texttt{alternative}}(P, \varphi_1, \varphi_2)(\widehat{f})$ or $\varphi_2(\widehat{f}) \sqsubseteq \varphi_{\texttt{alternative}}(P, \varphi_1, \varphi_2)(\widehat{f})$. This observation leads us to our second proof strategy:

Theorem 5.4.4 (Soundness of Interleaving Fixpoint Combinators). *Let φ be a higher-order fixpoint combinator, such that for all φ_1, φ_2, it holds $\forall x. \varphi_1(x) \sqsubseteq \varphi(\varphi_1, \varphi_2)(x) \vee \varphi_2(x) \sqsubseteq \varphi(\varphi_1, \varphi_2)(x)$, then φ is sound.*

Proof. Since φ is a higher-order combinator, we can assume that φ_1 and φ_2 are two sound combinators, i.e., $\operatorname{lfp} \widehat{f} \sqsubseteq \operatorname{lfp}(\varphi_1 \circ \widehat{f})$ and $\operatorname{lfp} \widehat{f} \sqsubseteq \operatorname{lfp}(\varphi_2 \circ \widehat{f})$ for all monotone $\widehat{f}$. We instantiate the assumption for φ with $x := \widehat{f} \circ \operatorname{lfp}(\varphi(\varphi_1, \varphi_2) \circ f)$. In case $\varphi_1(x) \sqsubseteq \varphi(\varphi_1, \varphi_2)(x)$, it follows

$$
\begin{aligned}
\operatorname{lfp} \widehat{f} &\sqsubseteq \operatorname{lfp}(\varphi_1 \circ \widehat{f}) && \text{by soundness of } \varphi_1 \\
&\sqsubseteq (\varphi_1 \circ \widehat{f})(\operatorname{lfp}(\varphi(\varphi_1, \varphi_2) \circ \widehat{f}) && \varphi_1 \circ \widehat{f} \text{ is reductive at } \operatorname{lfp}(\varphi(\varphi_1, \varphi_2) \circ \widehat{f}) \\
&\sqsubseteq (\varphi(\varphi_1, \varphi_2) \circ \widehat{f})(\operatorname{lfp}(\varphi(\varphi_1, \varphi_2) \circ \widehat{f})) && \text{by assumption } \varphi_1(x) \sqsubseteq \varphi(\varphi_1, \varphi_2)(x) \\
&\sqsubseteq \operatorname{lfp}(\varphi(\varphi_1, \varphi_2) \circ \widehat{f}) && \text{by fixpoint property of } \operatorname{lfp}(\varphi(\varphi_1, \varphi_2) \circ \widehat{f}).
\end{aligned}
$$

The other case $\varphi_2(x) \sqsubseteq \varphi(\varphi_1, \varphi_2)(x)$ is analogous. We conclude φ is sound. □

Corollary 5.4.5. The combinators $\varphi_{\texttt{alternative}}$, $\varphi_{\texttt{filter}}$ and $\varphi_{\texttt{unroll}}$ are sound by Theorem 5.4.4. □

Cache-Based Fixpoint Combinators Unfortunately, neither of these strategies applies for the combinator $\varphi_{\texttt{innermost}}$. The problem is the combinator $\varphi_{\texttt{innermost}}(\widehat{f})$ is *not* extensive, because it may return an intermediate result that does not yet overapproximate $\operatorname{lfp} \widehat{f}$. Therefore, we need to find a different invariant that lets us prove soundness of $\varphi_{\texttt{innermost}}$.

The key idea of the invariant is that the *only unstable analysis results occur within SCCs* and furthermore, *all calls in an SCC appear on the stack*:

$$
\begin{aligned}
\operatorname{lfp}(\varphi \circ \widehat{f}) \text{ stabilizes iff } \forall a.\ &\left[\widehat{f}(\operatorname{lfp}(\varphi \circ \widehat{f})) \prec a \ \sqsubseteq\ \operatorname{lfp}(\varphi \circ \widehat{f}) \prec a\right] \ \vee \\
&\left[\texttt{SCC.elem}(\operatorname{lfp}(\varphi \circ \widehat{f})) \prec a \wedge \texttt{Stack.elem} \prec a\right]
\end{aligned}
$$

We use this invariant in the following soundness proof.

Theorem 5.4.6. *A cache-based fixpoint combinator φ is sound, if it stabilizes.*

Proof. Let $\widehat{f}$ be a monotone function. At the top-most call of the analysis trace, the initial stack is empty, i.e., the right conjunct $\forall a. \texttt{SCC.elem}(\operatorname{lfp}(\varphi \circ \widehat{f})) \prec a \wedge \texttt{Stack.elem} \prec a$ is false. Because $\operatorname{lfp}(\varphi \circ \widehat{f})$ stabilizes, this means that at the top-most call a the left disjunct $f(\operatorname{lfp}(\varphi \circ \widehat{f})) \prec a \ \sqsubseteq \operatorname{lfp}(\varphi \circ \widehat{f}) \prec a$ has to be true. In other words, $\widehat{f}$ is reductive at $\operatorname{lfp}(\varphi \circ \widehat{f})$ and hence by Tarski's fixpoint theorem [Tarski 1955] we conclude $\operatorname{lfp} \widehat{f} \prec a \ \sqsubseteq \ \operatorname{lfp}(\varphi \circ \widehat{f}) \prec a$. □

This theorem simplifies the soundness proof of the fixpoint combinator $\varphi_{\texttt{innermost}}$:

Theorem 5.4.7. *The fixpoint combinator $\varphi_{\texttt{innermost}}$ is sound.*

Proof. In this proof, we write φ instead of $\varphi_{\texttt{innermost}}$ for readability. Let $\widehat{f}$ be a monotone function. By Theorem 5.4.6 all that is left to prove is that $\operatorname{lfp}(\varphi \circ \widehat{f})$ stabilizes. We continue by case distinction on the results of `Cache.lookup` and `Stack.elem`.

- In case the cache entry is stable, the combinator φ returns the cached entry. This satisfies the left conjunct of the invariant, because stable cache entries are reductive at $\widehat{f}$, i.e., if $(\texttt{Stable}, b) \leftarrow \texttt{Cache.lookup} \prec a$ and $b' \leftarrow \widehat{f}(\operatorname{lfp}(\varphi \circ \widehat{f})) \prec a$, then $b' \sqsubseteq b$. This is the case because φ only marks a cache entry as stable in case the SCC is empty and hence f does not depend on unstable recursive calls. We conclude $\widehat{f}(\operatorname{lfp}(\varphi \circ \widehat{f})) \prec a \ \sqsubseteq \ \operatorname{lfp}(\varphi \circ \widehat{f}) \prec a$.

- In case the call a occurs on the stack, i.e., $\texttt{Stack.elem} \prec a$, then the combinator φ adds the call to the SCC, i.e., $\texttt{SCC.elem}(\text{lfp}(\varphi \circ \widehat{f})) \prec a$. This satisfies the right disjunct of the invariant.
- In case the cache entry is unstable, the call does not occur on the stack, the combinator φ iterates until the analysis result does not grow anymore. More specifically, it iterates until $\widehat{f}(\text{lfp}(\varphi \circ \widehat{f})) \prec a \sqsubseteq \text{lfp}(\varphi \circ \widehat{f}) \prec a$, which satisfies the left disjunct of the invariant. □

To summarize, in this section we presented a way to prove correctness of fixpoint algorithms that consist of fixpoint combinators. In particular, a fixpoint algorithm is sound, if all of its combinators are sound. This simplifies the soundness proof, as it suffices to prove each combinator sound individually. Furthermore, we presented three proof strategies that simplify the soundness proof of different types of fixpoint combinators.

5.5 Evaluation

In this section, we evaluate our framework for modular fixpoint algorithms for big-step abstract interpreters. The goal of this framework is to enable the development of fixpoint algorithms, that can be more easily extended, configured, and adapted to new languages, analyses and use cases. To this end, we examine and verify the following hypotheses (HY):

- **HY1**: Our framework is able to describe a family of fixpoint algorithms for different languages and analyses.
- **HY2**: Our fixpoint algorithms can be easily extended, configured and adapted.
- **HY3**: There is no single best performing fixpoint algorithm.

To answer these hypotheses, we implemented the fixpoint combinators of this paper in Haskell, as part of the *Sturdy* framework [Keidel et al. 2018; Keidel and Erdweg 2019].

5.5.1 HY1: Our framework is able to describe a family of fixpoint algorithms for different languages and analyses

To verify this hypothesis, we use our approach to develop fixpoint algorithms for the following analyses:

- A k-CFA [Shivers 1991] and static type analysis for Scheme [Abelson et al. 1998], a dynamically-typed real-world programming language with first-class functions and mutable state,
- a static type analysis [Keidel and Erdweg 2020] for Stratego [Visser et al. 1998], a dynamically-typed real-world language for developing program transformations,
- a k-CFA and interval analysis for PCF [Plotkin 1977], a research language with first-class functions,
- an interval analysis for a While language, a research language with mutable state, exceptions, conditionals, while loops.

This selection of programming languages covers a variety of common language features, such as higher-order functions, and uncommon language features, such as generic term traversals of Stratego. Furthermore, the analyses use popular abstractions for closures and numeric values with finite and infinite abstract domains. In all of these cases, we were able to create a fixpoint algorithm with our framework. The fixpoint algorithms all use the $\varphi_{\texttt{filter}}$ and $\varphi_{\texttt{innermost}}$ combinators, however, they vary in the combinators they use for stack widening.

> These case studies show that our fixpoint combinators are language and analysis-independent.

Furthermore, we ported an existing big-step fixpoint algorithm of Darais et al. [2017] to our framework. We discuss in more detail how this algorithm works in Section 5.6 and focus on the process of porting the algorithm in this section. We implemented the algorithm as a fixpoint combinator $\varphi_{\texttt{adi}}$. We had to make some changes to the algorithm, to make the combinator analysis-independent and allow it to interoperate with other combinators in our framework. In particular, we removed references to the abstract environment and store, as they are passed as

inputs and outputs of the abstract interpreter. Furthermore, in contrast to the original algorithm, combinator $\varphi_{\texttt{adi}}$ pushes calls on a stack, which allows us to use the combinator in conjunction with a stack widening operator. Lastly, we were able to reuse operations of the cache interface, but needed to add a new interface that operates on the caches of the current and previous fixpoint iteration.

> This case study demonstrates the generality of our framework, as we were able to integrate an existing algorithm as a reusable fixpoint combinator.

5.5.2 HY2: Our fixpoint algorithms can be easily extended, configured, and adapted

Section 5.3.3 shows how our technique allows to easily extend, configure and adapt existing fixpoint algorithms.

> In contrast to a monolithic algorithm, we were able to integrate these use cases into an existing algorithm, without needing to change any code. Instead, we added and rearranged existing combinators. This demonstrates the simplicity of fine-tuning fixpoint algorithms in our framework.

5.5.3 HY3: There is no single best performing fixpoint algorithm

The previous subsection has shown that there are many ways in which we can extend and fine-tune a fixpoint algorithm. This leads to the question if there even is a single fixpoint algorithm that performs best for all analyzed programs. In this subsection, we answer this question by measuring the performance of 4 different fixpoint algorithms: Two algorithms based on $\varphi_{\texttt{innermost}}$ and $\varphi_{\texttt{outermost}}$ that iterate on the innermost and outermost SCCs, an algorithm based on $\varphi_{\texttt{adi}}$ that we described in Section 5.5.1, and an algorithm based on $\varphi_{\texttt{parallel}}$, a variation of $\varphi_{\texttt{adi}}$ that always returns cache entries from the old iteration.

We evaluated these algorithms for a 0CFA analysis for Scheme on 8 programs[3] of the *Gabriel* benchmark suite [Gabriel 1985] and 5 programs of the Scala-AM benchmark suite [Es et al. 2019]. Figure 5.1 shows the speedup of each algorithm relative to $\varphi_{\texttt{parallel}}$.

These benchmarks show that different iteration orders have an impact on the performance of the algorithm. For example, in case of the "deriv" benchmark, the algorithm $\varphi_{\texttt{outermost}}$ is 5.1 times faster than $\varphi_{\texttt{innermost}}$. This speedup is due to an oscillation between an inner and an outer SCC, similar to the Fibonacci example above. Even though $\varphi_{\texttt{outermost}}$ does larger iterations, it requires fewer iterates to reach a fixpoint in this benchmark. Furthermore, $\varphi_{\texttt{parallel}}$ and $\varphi_{\texttt{adi}}$ are faster than $\varphi_{\texttt{innermost}}$ and $\varphi_{\texttt{outermost}}$ in benchmarks where the overhead of tracking the SCC sets outweighs the cost of larger iterations.

> These benchmarks show us that fine-tuning the iteration order of a fixpoint algorithm can improve its performance.

In the benchmarks, the algorithm $\varphi_{\texttt{outermost}}$ was always faster than $\varphi_{\texttt{innermost}}$. However, iterating on the outermost SCC does not always have to be faster. For example, consider an interval analysis of the following C program:

```
while(i < 100) { expensive(i); for(j = 0; j < 10; j++) i += 5; }
```

A fixpoint algorithm that analyzes only a single iteration of the inner loop for each iteration of the outer loop, analyzes the function `expensive` 20 times without widening. In contrast, a fixpoint algorithm that analyzes 10 iterations of the inner loop per iteration of the outer loop, analyzes the function `expensive` only 2 times. In other words, iterating on the inner SCC is faster than iterating on the outer SCC for this program.

[3] We did not benchmark all programs in the Gabriel suite, because they use language features, such as `callCC`, that our analysis does not support.

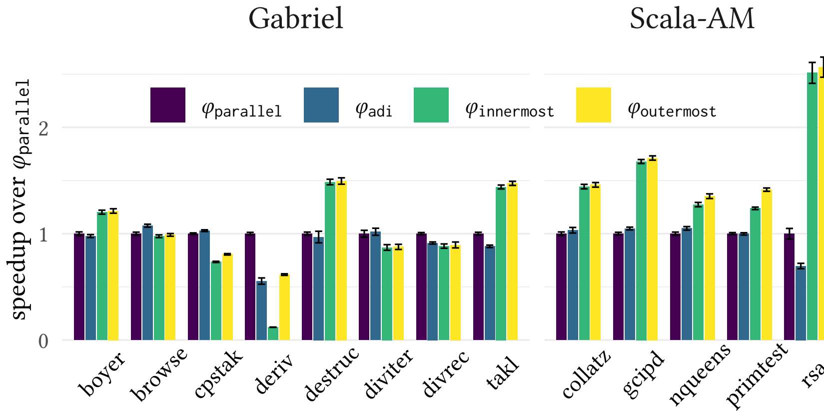


Figure 5.1: Normalized running times of 4 different fixpoint algorithms for a 0CFA analysis for Scheme. The plot shows the speedup of each algorithm over the algorithm $\varphi_{\texttt{parallel}}$ (higher is better). The error bars show the standard deviation of the ratio distribution for the normalized running time.

The example shows us that a fixpoint algorithm performs better on some programs, but worse on others. We conclude that it is hard to find a single fixpoint algorithm that performs best for all programs.

5.6 Related Work

The focus of this work is the modular description of fixpoint algorithms for big-step abstract interpreters. In this section, we discuss work related to our approach presented in this paper.

Modularizing the Definition and Soundness Proofs of Big-Step Abstract Interpreters There have been several works that modularized different parts of the definition and soundness proofs of big-step abstract interpreters. Keidel et al. [2018] describe an approach that modularizes the concrete and abstract language semantics and its soundness proof with *arrows* [Hughes 2000]. In particular, the concrete and abstract semantics is derived from the same *generic interpreter*, that is composed of a number of primitive operations over values, stores, exceptions, etc. The benefit of this approach is that it guarantees that an entire analysis is sound, as long as each operation is sound. However, [Keidel et al. 2018] do not show a fixpoint algorithm nor do they describe how a fixpoint algorithm should be implemented.

Bodin et al. [2019] describe a similar approach that derives both the concrete and abstract semantics from the same *skeletal semantics*. However, compared to arrows used by Keidel et al. [2018], they use a more liberal algebra called *skeletons*, which consists of hooks, filters, and branching operations. Yet, they provide similar soundness guarantees: an entire analysis derived from a skeletal semantics is sound, as long as all of its operations are sound. Bodin et al. [2019, Section 5.4] define the abstract semantics as the greatest fixpoint of the abstract collecting semantics. However, they do not show an algorithm that computes this fixpoint, nor do they explain how such an algorithm can be described modularly.

Keidel and Erdweg [2019] describe an approach that modularizes the *effects* of the analyzed language, such as exceptions and store mutations. More specifically, the approach captures the analysis of each effect with an *analysis component*, which consists of a concrete and abstract *arrow transformer*. This approach simplifies the analysis of languages with multiple effects, that interact with each other. Keidel and Erdweg define a single analysis for the fixpoint algorithm. However, they do not describe the fixpoint algorithm itself, nor do they describe how it can be decomposed further. In the present work, we make use of arrows and arrow transformers to modularize the description of fixpoint algorithms by the means of sound and reusable fixpoint combinators. We use arrows to describe fixpoint combinators, that are independent of the type of the fixpoint

computation. This allows us to change the type of the fixpoint computation, without needing to change the definition of the fixpoint combinators.

Darais et al. [2017] describe an approach that derives several *collecting semantics* from the same generic semantics with different combinators. These combinators, for example, collect a trace of the abstract interpreter, they collect expressions that are dead code, or they compute a fixpoint. These combinators inspired the style of fixpoint combinators we present in this paper, in that our fixpoint combinators have the same type as Darais et al. combinators. However, Darais et al. do not describe a formal theory for these combinators, which makes it hard to reason about their soundness. In this work, we developed a framework for modular fixpoint algorithms that is based on fixpoint combinators. This framework allows us to describe a family of fixpoint algorithms that can be configured and fine-tuned more easily, as we show in our evaluation. Furthermore, we developed a formal theory about these algorithms, which allows us to prove their soundness and termination compositionally.

Fixpoint Algorithms for Big-Step Abstract Interpreters The space of fixpoint algorithms for big-step abstract interpreters has not been extensively studied yet. Schmidt [1995, 1998] describes one of the first fixpoint algorithms for big-step abstract interpreters, that operates on the derivation tree. The fixpoint algorithm unfolds the abstract derivation tree until each branch either terminates or repeats itself. The algorithm detects recurrent calls of the abstract interpreter by memoizing parts of the abstract derivation tree. In case the algorithm finds a recurrent node in a branch, it cuts off recursion to avoid non-termination, which solves *Challenge 1*. Furthermore, the fixpoint algorithm sovles *Challenge 2* by joining the environments of repeating expressions with a widening operator, which ensures, that infinite recursive call chains have a recurrent call. However, the algorithm does not specify an order for iterating on the analysis results. Instead, the algorithm generates a number of recursive equations, which then can be solved with an arbitrary iteration order to calculate the fixpoint. We combine Schmidt's solutions to the termination challenges to implement our initial fixpoint algorithm $\mathtt{fix}_{\mathtt{monolithic}}$ in Section 5.2, which we later modularize. However, instead of generating recursive equations, our algorithm $\mathtt{fix}_{\mathtt{monolithic}}$ specifies an iteration order, i.e., the algorithm iterates on the innermost SCCs of the call graph [Bourdoncle 1993].

Darais et al. [2017] present another fixpoint algorithm for big-step abstract interpreters, similar to *parallel fixpoint iteration*. We implemented this algorithm in Section 5.5.1 as part of our evaluation with the combinator $\varphi_{\mathtt{adi}}$. The algorithm uses two caches to remember the analysis result of two consecutive fixpoint iterations. The algorithm then iterates over the entire program, updating the cache of the most recent iteration. If none of the caches changes anymore, the algorithm has reached a fixpoint and terminates. The algorithm solves *Challenge 1* by detecting recurrent calls if they have an existing cache entry. However, the algorithm does not solve the other two challenges, which means that it does not terminate for infinite abstract domains. In contrast, our fixpoint combinators $\varphi_{\mathtt{innermost}}$ and $\varphi_{\mathtt{outermost}}$ use only one cache. This cache remembers only the most recent analysis results and "forgets" about all prior results. Furthermore, the combinators iterate on small parts of the program, i.e., the SCCs of the call graph. Our evaluation shows that for some programs $\varphi_{\mathtt{innermost}}$ and $\varphi_{\mathtt{outermost}}$ reach a fixpoint faster, but for other programs $\varphi_{\mathtt{adi}}$ is faster due to less overhead of not needing to track the SCCs of the call graph.

In the present work, we do *not* propose a single fixpoint algorithm, that is supposed to work best for all languages, analyses and use cases. Instead, we propose an approach that modularizes the description of big-step fixpoint algorithms through sound and reusable fixpoint combinators. This approach allows analysis developers to create their own custom fixpoint algorithms more easily by combining several fixpoint combinators. Analysis developers can verify that their algorithm terminates by checking that it solves the three termination challenges. Furthermore, they can verify that their algorithm is sound, if all its fixpoint combinators are sound. This approach allows analysis developers to experiment with existing fixpoint algorithms more easily and fine-tune them.

Chaotic Fixpoint Iteration In the past and present [Bourdoncle 1993; Geser et al. 1994; Kim et al. 2020; Amato et al. 2016], chaotic fixpoint iteration has been well-studied and has become the de

facto standard for efficiently computing fixpoints of abstract interpreters. In a nutshell, chaotic iteration says that we always reach the *same* fixpoint, no matter in which order we update analysis results for different statements. However, this does not hold for algorithms with widening, as some orders yield more precise fixpoint approximations than others. Thus, the challenge is to find a precise iteration orders, that requires the least amount of updates. Bourdoncle [1993] presents a chaotic iteration order, that is based on a weak topological ordering of the control-flow graph. This iteration order improves the precision as it reduces the number of widening points to the heads of SCCs. Bourdoncle [1993] work inspired the design of the fixpoint combinators $\varphi_{\texttt{innermost}}$ and $\varphi_{\texttt{outermost}}$, that we developed in this paper, as they use the same widening points.

Fixpoint Algorithms for Small-Step Abstract Interpreters In contrast to big-step abstract interpreters, static analyses in small-step style have a longer history of research [Shivers 1991; Might and Shivers 2006a; Horn and Might 2010; Sergey et al. 2013]. Similar to big-step abstract interpreters, small-step abstract interpreters also seamlessly combine data-flow and control-flow information, however, they describe the abstract semantics as a small-step relation. A fixpoint algorithm for such interpreters explores the finite state space of the small-step relation. Unfortunately, it is unclear how small-step fixpoint algorithms apply to big-step abstract interpreters, because of differences in the style of semantics: While small-step abstract interpreters use continuations to explicitly model control of the interpreter as part of the state space, big-step abstract interpreters leverage the control of the meta-language (e.g., Haskell). This means that big-step abstract interpreters cannot ensure termination simply by making the state space finite, because their interpreter function may diverge nonetheless. To this end, big-step fixpoint algorithms must detect recurrent recursive calls and iterate on them, which is not necessary for small-step algorithms.

5.7 Conclusion

In this paper, we studied the modular description of fixpoint algorithms for big-step abstract interpreters. We identified three challenges that an algorithm needs to solve to guarantee termination. Based on these solutions, we developed a fixpoint algorithm for big-step abstract interpreters that iterates on the strongly-connected components of the call graph. However, since the algorithm consists of a single monolithic function, it is hard to extend, configure and adapt the fixpoint algorithm. To this end, we refactored the algorithm into small reusable fixpoint combinators, which allow us to change the algorithm by rearranging and adding new combinators. Furthermore, the combinators simplify the soundness proof, as each combinator can be proved sound individually once and for all. Lastly, our evaluation demonstrates that our approach describes an entire family of fixpoint algorithms for different languages and analyses that can be easily extended, adapted and configured.

A SYSTEMATIC APPROACH TO ABSTRACT INTERPRETATION OF PROGRAM TRANSFORMATIONS

6

This chapter is based on the following peer-reviewed paper:

A Systematic Approach to Abstract Interpretation of Program Transformations.
Sven Keidel and Sebastian Erdweg.
Lecture Notes in Computer Science, Springer, VMCAI (2020), 136–157.
https://doi.org/10.1007/978-3-030-39322-9_7

Abstract — Abstract interpretation is a technique to define sound static analyses. While abstract interpretation is generally well-understood, the analysis of program transformations has not seen much attention. The main challenge in developing an abstract interpreter for program transformations is designing good abstractions that capture relevant information about the generated code. However, a complete abstract interpreter must handle many other aspects of the transformation language, such as backtracking and generic traversals, as well as analysis-specific concerns, such as interproceduralty and fixpoints. This deflects attention.

We propose a systematic approach to design and implement abstract interpreters for program transformations that isolates the abstraction for generated code from other analysis aspects. Using our approach, analysis developers can focus on the design of abstractions for generated code, while the rest of the analysis definition can be reused. We show that our approach is feasible and useful by developing three novel inter-procedural analyses for the Stratego transformation language: a singleton analysis for constant propagation, a sort analysis for type checking, and a locally-illsorted sort analysis that can additionally validate type changing generic traversals.

6.1 Introduction

Abstract interpretation is a technique to define sound static analyses [Cousot and Cousot 1977]. Static analyses have proved useful in providing feedback to developers (e.g., dead code [Chen et al. 1997], type information), in finding bugs (e.g., uninitialized read [Xie et al. 2003], type errors [Pierce 2002]), and in enabling compiler optimizations (e.g., constant propagation [Callahan et al. 1986], purity analysis [Salcianu and Rinard 2005]). It is therefore no surprise that the field of abstract interpretation and static analysis has seen significant attention both in academia and industry.

Unfortunately, the analysis of program transformations has not seen much attention so far. Program transformations are a central tool in language engineering and modern software development. For example, they are used for code desugaring, macro expansion, compiler optimization, refactoring, migration scripting, or model-driven development. The development of such program transformations tends to be difficult because they act at the metalevel and should work for a large class of potential input programs. Yet, there are hardly any static analyses for program transformation languages available, and it appears to be difficult to develop such analyses. To this end, we identified the following challenges:

Domain-Specific Features Program transformation languages such as Stratego [Visser et al. 1998], Rascal [Klint et al. 2009], and Maude [Clavel et al. 2002] aim to simplify the development of program transformations. Therefore, they provide domain-specific language features such as rich pattern-matching, backtracking, and generic traversals. These domain-specific language features usually cannot be found in other general-purpose languages and the literature on static analysis provides only little guidance on how to tackle them.

Term Abstraction Programs are first-class in program transformations and are represented as terms (e.g., abstract syntax trees). Therefore, analysis developers need to find a good abstraction for terms, such as syntactic sorts or grammars [Cousot and Cousot 1995]. This term abstraction heavily influences the precision and usefulness of the analysis and most of the analysis

development effort should be spent on the design of this abstraction. We expect analysis developers to experiment with alternative term abstractions: The design of good abstract domains is inherent to the development of any abstract interpreter and cannot be avoided.

Soundness Developing an abstract interpreter that soundly predicts the generated code of program transformations is difficult. This is because real-world transformation languages have many edge cases and an abstract interpreter has to account for all of these edge cases to be sound. Furthermore, transformation languages often do not have a formal semantics, which makes it hard to verify that the abstract interpreter covered all cases.

In this paper we present a systematic approach to develop abstract interpreters for program transformation languages that addresses these challenges. It is based on the well-founded theory of *compositional soundness proofs* [Keidel et al. 2018] and *reusable analysis components* [Keidel and Erdweg 2019]. In particular, our approach captures the core semantics of a transformation language with a *generic interpreter* [Keidel et al. 2018] that does not refer to any analysis-specific details. This simplifies the analysis of the domain-specific language features. Furthermore, our approach decouples the term abstraction from the remainder of the analysis through an interface. This means that any term abstraction that implements this interface gives rise to a complete abstract interpreter. Thus, analysis developers can fully focus on developing good term abstractions. Lastly, our approach reuses language-independent functionality, such as abstractions for environments, exceptions and fixpoints, from the Sturdy standard library. This not only reduces the analysis development effort, but also simplifies its soundness proof as we can rely on the soundness proofs of the Sturdy library [Keidel and Erdweg 2019].

We demonstrate the feasibility and usefulness of our approach by developing abstract interpreters for Stratego [Visser et al. 1998]. Stratego is a complex dynamic program transformation language featuring rich pattern matching, backtracking, generic traversals, higher-order transformations, and an untyped program representation. Despite these difficulties, based on our approach we developed three novel abstract interpreters for Stratego: We developed a constant propagation analysis, a sort analysis, which checks that transformations are well-typed, and an advanced sort analysis, which can even validate type-changing generic traversals which produce ill-sorted intermediate terms. Our systematic approach was crucial in allowing us to focus on each of these abstract domains without being concerned with other aspects of the Stratego language. We implemented the analyses in Haskell in the *Sturdy* analysis framework and the code of the analyses is open-source.[1]

In summary, we make the following contributions:

- We propose a systematic approach to the development of abstract interpreters for program transformations, that lets analysis developers focus on designing the term abstraction.
- We show that many features of program transformation languages can be implemented on top of existing analysis functionality and do not require specific analysis code.
- We demonstrate the feasibility and usefulness of our approach by applying it to Stratego, for which we develop three novel abstract interpreters.

6.2 Illustrating Example: Singleton Analysis

The static analysis of program transformations can have significant merit helping developers to understand and debug their code and helping compilers to optimize the code. For example, we would like to support the following analyses: *Singleton analysis* to enable constant propagation, *purity analysis* to enable function inlining, *dead code* analysis to discover irrelevant code, *sort analysis* to prevent ill-sorted terms. While these and many other analyses would be useful, their development is complicated. In this section, we illustrate our approach by developing a singleton analysis for Stratego [Visser et al. 1998].

[1] https://gitlab.rlp.net/plmz/sturdy/tree/master/stratego

```
data Pat = Var String | As String Pat | Cons String [Pat]
  | StringLit String | NumLit Int | Explode Pat Pat

match :: (IsTerm term c, ArrowEnv String term c,
          ArrowExcept () c, …) ⇒ c (Pat,term) term
match = proc (pat,t) → case pat of
  Var "_" → returnA ≺ t
  Var x → lookup
    (proc (t',(x,t)) → do t'' ← equal ≺ (t,t');
      insert ≺ (x,t''); returnA ≺ t'')
    (proc (x,t) → insert ≺ (x,t); returnA ≺ t) ≺ (x,(x,t))
  As v p → do t' ← match ≺ (Var v,t); match ≺ (p,t')
  StringLit s → matchStringLit ≺ (s,t)
  NumLit n → matchNumLit ≺ (n,t)
  Cons c ps → matchCons (zipWith match) ≺ (c,ps,t)
  Explode c ts → matchExplode
    (proc c' →  match ≺ (c,c'))
    (proc ts' → match ≺ (ts,ts')) ≺≺ t
```

Listing 6.1: Generic abstract pattern matching for Stratego.

6.2.1 Abstract Interpreter for Program Transformations = Generic Interpreter + Term Abstraction

The development of analyses for program transformations is complicated for two reasons. First, each analysis requires a different term abstraction, with which it represents the generated code. The choice of term abstraction is crucial since it directly influences the precision, soundness, and termination of the analysis. Second, program transformation languages provide domain-specific language features such as rich pattern matching, backtracking, and generic traversals. Soundly approximating these features in an analysis is not easy, and resolving this challenge for each analysis anew is impractical.

In this paper, we propose a more systematic approach to developing static analyses for program transformations. To support static analyses for a given transformation language, we first develop a generic interpreter that implements the abstract semantics of the domain-specific language features in terms of standard language features whose abstract semantics is well-understood already. The generic interpreter is parametric in the term abstraction, such that we can derive different static analyses in a second step by providing different term abstractions. This architecture enables analysis developers to separately tackle the challenge of designing a good term abstraction.

We have developed a generic interpreter for Stratego based on the Sturdy analysis framework [Keidel et al. 2018; Keidel and Erdweg 2019] in Haskell. We explain the full details of generic interpreters and background about Sturdy in Section 6.3. Here, we only illustrate a small part of the generic interpreter, namely pattern matching.

Listing 6.1 shows the generic analysis code for pattern matching. We parameterized the pattern-matching function `match` using a type class `IsTerm` as an interface. Pattern matching interacts with the term abstraction to deconstruct terms but implements other aspects generically. In Listing 6.1, we have highlighted all calls to operations of `IsTerm`; the remaining code is generic. We provide a short notational introduction before delving deeper into the analysis code.

Our approach is based on Sturdy, which requires analyses to be written in *arrow style* [Hughes 2000]. Like monads, arrows `(c x y)` generalize pure functions `(x → y)` to support side-effects in a principled fashion. For users of our approach, this mostly means that they have to use Haskell's built-in syntax for arrows, as shown in Listing 6.1. Expression `(proc x → e)` introduces an arrow computation similar to the pure `(λx → e)`. Do notation `(do cmd*)` denotes a sequence of arrow commands, where each command takes the form `(y ← f ≺ x)` or `(f ≺ x)` [Paterson 2001]. Command `(y ← f ≺ x)` calls `f` on `x` and stores the result in `y`; `(f ≺ x)` ignores the resulting value but not the potential side-effect of `f`. For a more in-depth introduction to arrows, we refer

to Hughes's original paper [Hughes 2000] and online resources such as `https://www.haskell.org/arrows`.

The generic analysis for pattern matching in Listing 6.1 describes a computation `(c (Pat,t) t)` that is parametric in `c` and `t`, but restricts these types through type-class constraints. Type `t` must implement the term abstraction interface `IsTerm`. Type `c` is an arrow that encapsulates the side-effects of the computation and must at least support environments and exception handling. We use these side-effects to implement pattern variables and backtracking in `match`.

Computation `match` takes a pattern and the matchee (the term to match) as input and yields the possibly refined term as output. For a wildcard pattern, we yield the matchee unchanged. For pattern variables, we look up the variable in the environment and distinguish two cases. If the variable is already bound to `t'`, we require the matchee `t` to be equal to `t'`. If the variable is not bound yet, we insert a binding into the environment. For named subpatterns `(As v p)`, we invoke the code for pattern variables recursively. The remaining four cases delegate to the term abstraction, passing the function for matching subterms as needed. When a pattern match fails, it throws an exception to reset all bound pattern variables.

The generic analysis code for pattern matching captures the essence of pattern matching in Stratego and closely follows Stratego's concrete semantics. In fact, the generic code can be instantiated to retrieve a fully functional concrete interpreter for Stratego. This makes the generic interpreter relatively easy to develop: no analysis-specific code is required. All analysis-specific code resides in instances of interfaces like `ArrowExcept` and `IsTerm`. Sturdy further exploits this to support compositional soundness proofs of analyses [Keidel et al. 2018].

6.2.2 A Singleton Term Abstraction

We can derive complete Stratego analyses from the generic interpreter by instantiation. Specifically, we need to provide implementations for the type classes it is parameterized over. For standard interfaces like `ArrowExcept` and `ArrowEnv`, we provide reusable abstract semantics. However, the term abstraction `IsTerm` is language-specific and analysis-specific. Thus, this interface needs to be implemented by the analysis developer.

To illustrate the definition of term abstractions, here we develop a singleton analysis for Stratego. The analysis determines if (part of) a program transformation yields a constant output, such that the transformation can be optimized by constant propagation. Note that in this paper we are only concerned with the definition of analyses; the implementation of subsequent optimizations is outside the scope of the paper.

Each term abstraction needs to choose a term representation. For the singleton analysis, we use a simple data type $\widehat{\texttt{Term}}$ with two constructors:

$$\textbf{data}\ \widehat{\texttt{Term}} = \texttt{Single Term | Any}$$

A term `Single ct` means that the transformation produces a single concrete Stratego term `ct` of type `Term`. In contrast, `Any` means that the transformation cannot be shown to produce a single concrete term.

Based on such term representation, a term abstraction for Stratego must implement the 10 functions from the `IsTerm` interface. We show the implementation of four of these functions in Listing 6.2 that also appeared in Listing 6.1.

Function `matchString` in Listing 6.2 defines a computation that takes a string value `s` and a matchee `t` of type `Term` as input. If `t` denotes a single concrete term, `matchString` delegates to the concrete string matching semantics using `liftConcrete`. However, if the matchee is `Any`, we cannot statically determine if the pattern match should succeed or fail. Thus, we join $\sqcup$ the two potential outcomes: Either pattern matching succeeds and we return `t` unchanged, or pattern matching fails and we abort the matching by throwing an exception. Function `matchNum` is analogous to `matchString`.

Function `matchCons` distinguishes three cases. The first case checks if matchee `t` denotes a single concrete term with constructor `c` and right number of subterms. If so, we recursively match the subpatterns against the subterms, converted to singletons. Then, if all submatches yielded singleton terms again, we refine the matchee accordingly. The second case occurs when `t` denotes a singleton term but does not match the constructor pattern. In this case, we simply

```
instance (ArrowExcept () c, ArrowJoin c, …) ⇒ IsTerm Term c where
  matchString = proc (s,t) → case t of
    Single ct → liftConcrete matchString ≺ (s,ct)
    Any → (returnA ≺ t) ⊔ (throw ≺ ())

  matchNum = proc (i,t) → case t of
    Single ct → liftConcrete matchNum ≺ (i,ct)
    Any → (returnA ≺ t) ⊔ (throw ≺ ())

  matchCons matchSub = proc (c,ps,t) → case t of
    Single (Cons d ts) | c == d && eqLen ps ts → do
      ts' ← matchSub ≺ (ps,map Single ts)
      case allSingle ts' of
        Nothing → returnA ≺ Any
        Just cts → returnA ≺ Single (Cons c cts)
    Single _ → throw ≺ ()
    Any → do matchSub ≺ (ps,replicate (length ps) Any)
             (returnA ≺ t) ⊔ (throw ≺ ())
```

Listing 6.2: Parts of a singleton term abstraction for Stratego.

abort. Finally, if t is Any, we combine the two cases using a list of Any terms as subterms. Note that the recursive match on the subpatterns ps is necessary to bind pattern variables that may occur.

6.2.3 Soundness

Our approach drastically simplifies the soundness proof of the abstract interpreter. In particular, by factoring the concrete and abstract interpreter into a generic interpreter, we do not have to worry about soundness of the generic interpreter. Instead, its soundness proof follows by composing the proof of smaller soundness lemmas about its instances [Keidel et al. 2018]. Furthermore, because we instantiate the generic interpreter with sound analysis components for environments, stores and exceptions, we do not have to worry about soundness of these analysis concerns either [Keidel and Erdweg 2019]. All that is left to prove, is the soundness of the term operations.

6.2.4 Summary

Our approach to developing static analyses for program transformations consists of two steps. First, develop a generic interpreter based on standard semantic components and a parametric term abstraction. Second, define a term abstraction and instantiate the generic interpreter. While the term abstraction is language-specific and analysis-specific, the generic interpreter can be reused across analyses and only needs to be implemented once per transformation language. In the subsequent section, we explain how to develop and instantiate generic interpreters for transformation languages using standard semantic components. Sections 6.4 and 6.5.1 demonstrate the development of sophisticated term abstractions.

6.3 Generic Interpreters for Program Transformations

Creating sound static analyses is a laborious and error-prone process. While there is a rich body of literature on analyzing functional and imperative programming languages, static analysis of program transformation languages is under-explored. Most work in the area of program transformations so far focused on type checking, which considers each rewriting separately and is limited to intra-procedural analysis.

The key enabler of our approach are generic interpreters that can be instantiated with different term abstractions to obtain different analyses. In this section, we demonstrate our approach at the example of Stratego and show how to develop generic interpreters for Stratego. In particular,

```
desugar-type: PairType(t1,t2) → ⟦Pair<~t1,~t2>⟧
desugar-expr: PairExpr(e1,e2) → ⟦new Pair<>(~e1,~e2)⟧

topdown(s) = s; all(topdown(s))        try(s) = s <+ id
main = topdown(try(desugar-type + desugar-expr))
```

Listing 6.3: A generic traversal for desugaring pair notation.

we show that the features of program transformation languages do *not* require specific analysis code but can be mapped to existing language concepts whose analysis is already well-understood.

6.3.1 The Program Transformation Language Stratego

Stratego is a program transformation language featuring rich pattern matching, backtracking, and generic traversals [Visser et al. 1998]. For example, consider the following desugaring of Java extended with pairs [Erdweg et al. 2011] in Listing 6.3. The two rewrite rules of the form above use pattern matching to select pair types and expressions, respectively. They then generate representations of pair types and expressions using the `Pair` class. The `main` rewriting strategy traverses the input AST top-down and tries to apply both rewrite rules at every node, leaving a node unchanged if neither rule applies. We also added the definitions of the higher-order functions `topdown` and `try` from the standard library. The built-in primitive **`all`** takes a transformation and applies it to each direct subterm of the current term. Function `topdown` uses **`all`** to realize a generic top-down traversal over a term, applying `s` to every node. Function `try` uses left-biased choice **`<+`** to catch any failure in `s` and to resume with the identity function **`id`** instead. Furthermore, the Stratego compiler translates the rewrite rules of the form `r : p → t` to transformations `r = ?p; !t`:

```
desugar-type = ?PairType(t1,t2); !ClassType("Pair",[t1,t2])
desugar-expr = ?PairExpr(e1,e2); !NewInstance("Pair",[e1,e2])
```

The translated rule first matches the pattern `p`, binding all pattern variables to the respective subterms and then builds the term `t` using the abstract syntax of Java.

6.3.2 A Generic Interpreter for Stratego

We demonstrate how to map these language features to standard language concepts and how this enables static analysis of program transformations. To this end, we developed a generic interpreter for Stratego.[2] The generic interpreter is based on a previous Sturdy case study [Keidel et al. 2018] that was never described in detail.

We consider fully desugared Stratego code in our interpreter, ignoring Stratego's dynamic rules. This core Stratego language [Visser et al. 1998] only contains 12 constructs as defined by the data type `Strat` in Listing 6.4. We explain these constructs together with their generic semantics, shown in the same listing. The semantics is defined by a function `eval` that accepts a Stratego program and yields a computation of type `(c term term)`, meaning that a Stratego program takes a term as input and yields another term as output. That is, Stratego programs are term transformations as expected. The arrow `c` captures the side-effects of the computation, as explained in Section 2.2.

The first two core Stratego constructs deconstruct and construct terms. A `(Match pat)` transformation is based on a term pattern `pat`, which it matches against the input term `t`. Function `match` from Listing 6.1 implements the actual pattern matching, as we have discussed in Section 6.2. Recall that `match` binds pattern variables in the environment as a side-effect and throws an exception if the pattern match fails. We will see shortly how these side-effects are supported by the generic interpreter. A `(Build pat)` transformation is the dual of match: it constructs a new term according to the pattern, filling in information from the environment in place of pattern variables.

[2] https://gitlab.rlp.net/plmz/sturdy/blob/master/stratego/src/GenericInterpreter.hs

```
data Strat = Match Pat | Build Pat | Id | Seq Strat Strat
  | Fail | GuardedChoice Strat Strat Strat | Scope [String] Strat
  | Call String [Strat] [String] | Let [(String,Strategy)] Strat
  | One Strat | Some Strat | All Strat

eval :: (IsTerm term c, ArrowEnv String term c, ArrowExcept () c,ArrowFix c, …) ⇒
       Strat → c term term
eval = fix $ λev strat → case strat of
  Match pat → proc t → match ≺ (pat,t)
  Build pat → proc _ → build ≺ pat

  Id → proc x → returnA ≺ x
  Seq s1 s2 → proc t1 → do t2 ← ev s1 ≺ t1; t3 ← ev s2 ≺ t2; returnA ≺ t3
  Fail → proc _ → throw ≺ ()
  GuardedChoice s1 s2 s3 → try (ev s1) (ev s2) (ev s3)

  Scope vars s → scoped vars (ev s)
  Call f ss ts → proc t → do
    senv ← readStratEnv ≺ (); case Map.lookup f senv of
      Just (Closure s@(Strat _ ps _) senv') → do
        args ← mapA lookupOrFail ≺ ts
        scoped ps (invoke ev) ≺≺ (s, senv', ss, args, t)
      Nothing → failString ≺ "Cannot find strat"
  Let bnds body → let_ bnds body eval'

  One s → mapSubterms (one (ev s))
  Some s → mapSubterms (some (ev s))
  All s → mapSubterms (all (ev s))

scoped vars f = proc t → do
  oldEnv ← getEnv ≺ ()
  deleteEnvVars ≺ vars
  finally (proc (t,_) → f≺t) (proc (_,oldE) → restoreEnvVars vars ≺ oldE)
    ≺ (t,oldEnv)
```

Listing 6.4: Generic interpreter for Stratego.

The next four core Stratego constructs handle control-flow. The identity transformation `Id` returns the input term unchanged. A sequence (`Seq` s_1 s_2) of transformations s_1 and s_2 pipes the output of s_1 into s_2. The `Fail` transformation never succeeds and always throws an exception using `throw`, which we also used to indicate failed pattern matches. To catch such exceptions, core Stratego programs can use guarded choice, written (`s1 < s2 + s3`) in Stratego notation. Guarded choice runs s_3 if s_1 fails (throws an exception) and s_2 otherwise. We implemented guarded choice using the `try` function. Like `throw`, `try` is declared in the `ArrowExcept` interface and allows us to catch exceptions triggered by `throw`. There are two things to note here:

- The implementation of `throw` and `try` are not specific to Stratego and are provided as sound reusable analysis components [Keidel and Erdweg 2019] by the standard library of Sturdy. We are effectively mapping Stratego features to these pre-defined features of Sturdy.
- We can choose how exceptions affect the variables bound during pattern matching. For Stratego, we need exceptions to undo variable bindings in order to correctly implement backtracking. However, in other languages we may want to retain the state of a computation even after an exception was thrown.

The next three constructs handle scoping, strategy calls, and local strategy definitions. We discuss the first two of these in some detail. Stratego's scoping is somewhat unconventional, because Stratego has explicit scope declarations and environments follow store-passing style. Variables listed in a scope declaration are lexically scoped as usual, but other variables can occur in the environment and must be preserved. We use function `scoped` (at the bottom of Listing 6.4) to

```
class Arrow c ⇒ IsTerm term c where
  matchString   :: c (String,term) term
  matchNum      :: c (Int,term) term
  matchCons     :: c ([p],[term]) [term] → c (String,[p],term) term
  matchExplode  :: c term term → c term term → c term term

  buildString   :: c String term
  buildNum      :: c Int term
  buildCons     :: c (String,[term]) term
  buildExplode  :: c (term,term) term

  equal         :: c (term,term) term
  mapSubterms   :: c [term] [term] → c term term
```

Listing 6.5: An interface for operations on terms.

implement this scoping. First, we unbind the scoped variables from the current environment to allow pattern matching to bind them afresh. Second, after the scoped code finishes, we restore the bindings of scoped variables from the old environment while retaining other bindings from the current environment unchanged. Scoping also occurs when calling a strategy. To evaluate a call, we first find the strategy definition, then lookup the term arguments `ts` in the current environment, and then invoke the strategy using `scoped` for the term parameters `ps`.

The final three constructs are generic traversals that use `mapSubterms` to call `one`, `some`, or `all` on the subterms of the current input term. Function `mapSubterms` is part of the `IsTerm` interface and thus analysis-specific because depends on the term representation. Functions `one`, `some`, or `all` are part of the generic interpreter and ensure that, respectively, exactly one, at least one, or all of subterms are transformed by the given strategy `s`. This way our generic interpreter separates term-specific operations from operations that can be defined generically.

6.3.3 The Term Abstraction

At this point, all that it takes to define a Stratego analysis is to implement the `IsTerm` interface for a new term abstraction. The rest of the analysis is given by the generic interpreter and reusable functionality from the Sturdy library.

The generic interpreter described in the previous section crucially relies on the term abstraction. In particular, pattern matching, term construction, and generic traversals must inspect or manipulate terms. In Section 6.2 we have seen how `match` used term operations and how we could implement these for the singleton term abstraction. Here we show the complete interface for term abstractions.

Stratego terms are strings, numbers, or constructor terms:

```
data Term = Cons String [Term] | StringLit String | NumLit Int
```

Our interface must at least provide operations to match and construct such terms. In addition, we must support Stratego's generic traversals and explode patterns. Note that Stratego represents lists using constructors `Cons` and `Nil`:

```
Cons "Cons" [NumLit 1, Cons "Cons" [NumLit 2, Cons "Nil" []]]
```

We designed an interface for term abstractions of Stratego terms that requires only 10 operations. Listing 6.5 shows the corresponding type class. The interface contains four functions for pattern matching, four functions for term construction, one equality function, and one function to map subterms.

We have discussed the functions for pattern matching Section 6.2 already. Function `matchCons` takes a function for matching subterms against subpatterns. Function `matchExplode` takes functions for matching the constructor name and the subterms. The functions for term construction are straightforward. While function `buildCons` takes a String and a list of terms, function

buildExplode takes two terms. The first of these terms must be a string term, the second one must represent a list of terms. Finally, we require functions for checking the equality of two terms and for mapping a function over a term's subterms. This last function enables generic traversals as shown in Listing 6.4.

Our interface for term abstractions can be instantiated in various ways by defining instances of the type class. We have shown an instance for the singleton term abstraction in Listing 6.2 and will describe further term abstractions in the upcoming sections. But it is worth noting that the interface can also be instantiated for concrete Stratego terms:

```
instance … ⇒ IsTerm Term c where …
```

This concrete term instance allows us to run the generic interpreter as a concrete Stratego semantics. This is not only great for testing the generic interpreter against a reference implementation of Stratego, but also crucial for proving the soundness of term abstractions against the concrete semantics.

To summarize, we implemented the Stratego language semantics as a generic interpreter based on a few term operations only. The generic interpreter maps many aspects of Stratego language to standard language concepts such as environments and exceptions. For these language concepts, we reuse the abstract semantics found in the Sturdy standard library. In the end, to design and implement a new analysis for Stratego, all it takes is a new term abstraction. We exploit this reduction of effort in the next two sections, where we develop two novel static analyses for Stratego by defining term abstractions.

6.4 Sort Analysis

In this section, we define an inter-procedural sort analysis for Stratego. The analysis checks if a program transformation generates well-formed programs and to which sort the program belongs. That is, we implement a term abstraction where we choose to represent terms through their sort.

6.4.1 Sorts and Sort Contexts

We describe the sorts of Stratego terms by the following Haskell datatype:

```
data Sort = Lexical | Numerical | Sort String | List Sort
  | Tuple [Sort] | Option Sort | ⊥ | ⊤
```

Sort Lexical represents string values, Numerical represents numeric values. We use (Sort s) to represent named sorts such as (Sort "Exp"). We further include sorts for representing Stratego's lists, tuples, and option terms. Finally, ⊥ represents the empty set of terms and ⊤ represents all terms (also ill-formed ones). This means, we can guarantee a term is well-formed if its sort is not ⊤.

To associate terms to sorts, we parse the declaration of constructor signatures that are part of any Stratego program. Typically, these declarations are automatically derived from the grammar of the source and target language.

```
Num : Int ⟶ ArithExp
Add : ArithExp * ArithExp ⟶ ArithExp
    : ArithExp ⟶ PythonExp
```

Each line declares a constructor, the sorts of its arguments and the generated sort. We allow overloaded constructor signatures as long as they generate terms of the same sort. That is, if $c : s_1 \dots s_m \longrightarrow s \in \Gamma$ and $c : s'_1 \dots s'_n \longrightarrow s' \in \Gamma$, then $s = s'$.

The third signature declares that any term of sort ArithExp should also be considered a term of sort PythonExp. This is the result of injection production in the grammar and effectively declares a subtype relation ArithExp <: PythonExp. Dealing with subtyping correctly is one of the major challenges of developing a sort analysis. Thanks to our separation of concerns, we can fully focus on that challenge here.

We collect all constructor signatures and the subtyping relation in a sort context:

```
instance (ArrowExcept () c, ArrowJoin c, …) ⇒ IsTerm Sort c where
  buildString = proc _ → returnA ≺ Lexical

  matchString = proc (_,s) → if subtype Lexical s
    then (returnA ≺ s) ⊔ (throw ≺ ()) else throw ≺ ()

  buildCons = proc (c, ss) → returnA ≺ case (c, ss) of
    ("Nil",[]) → List ⊥
    ("Cons",[a,s]) | subtype (List ⊥) s → List a ⊔ s
    _ → ⊓ (⊤ : [t | (ss',t) ← constrSigs c, ss ⊑ ss'])

  matchCons matchSubs = proc (c,ps,s)→ case (c,ps)
    ("Nil",[]) → if subtype (List ⊥) s
      then (buildCons≺("Nil",[])) ⊔ (throw≺()) else throw≺()
    ("Cons",[hd,tl]) → if subtype (List ⊥) s
      then do let subterms = [getListElem s, s]
              ss ← matchSubs ≺ ([hd,tl],subterms)
              (buildCons ≺ ("Cons",ss)) ⊔ (throw ≺ ())
      else throw ≺ ()
    _ → lubA (proc (c',ss) → if c == c' && length ss == length ps
      then do ss' ← matchSubs ≺ (ps,ss); cons ≺ (c,ss')
      else throw ≺ ()) ≺≺ constructorsOfSort s

  mapSubterms f = proc s → do lubA (proc (c,ts) →
     do ts' ← f ≺ ts buildCons ≺ (c,ts'))
        ≺ constructorsOfSort s
```

Listing 6.6: Abstract term operations for the sort analysis.

```
type Sig = ([Sort], Sort)
data Context = Context {sorts :: Map Sort [(String,Sig)], subtypes :: SubtypeRelation}
```

Since we require the context when operating on sorts, we actually represent terms abstractly as a pair `(Sort,Context)`. However, all terms refer to the same context and the context never changes. To simplify the presentation in this paper, we assume the context is globally known and terms are represented by `Sort` alone.

6.4.2 Abstract Term Operations

In the remainder of this section, we explain how to implement the term abstraction for our sort analysis. To this end, we have to provide an instance of type class `IsTerm` as shown in Listing 6.6. We only show the code for lists and user-defined constructor and omit the other cases for tuples and optionals.

As a warm-up, consider operation `buildString` that yields sort `Lexical` independent of the string literal. When matching a string against sort `s` in `matchString`, the match can only succeed if `Lexical` terms may be part of `s` terms. Otherwise the match must fail.

Arguably the most interesting part of the term abstraction is building and matching constructor terms. Let's start with operation `buildCons`, which obtains the constructor name `c` and the list of subsorts `ss`. In Stratego, list, tuple, and optional terms use reserved constructor names. We include one case for each reserved constructor to generate the appropriate sort. For example, constructor `Nil` can be applied to an empty argument list to generate an empty list. This list has sort `(List ⊥)`. Constructor `Cons` generates a compound term that has sort list if the second argument was a list. The sort of the resulting list is the least super-sort (⊔) of the new head list and the tail. The empty constructor `""` generates tuples; `None` and `Some` generate optional terms.

The last case of `buildCons` handles user-defined constructor symbols `c`. We use `(constrSigs c)` to look up the signatures `(ss',t)` of `c` from the sort context. We only retain those signatures that can accept the constructor arguments `ss`. Finally, we collect all result sorts `t` and compute their

greatest lower bound. If none of the signatures matches, we return sort ⊤. For example, consider the call:

```
buildCons ≺ ("While",[Sort "Exp",Sort "Block"])
```

If the signature of `While` is `(Exp * Block → Stmt)`, we obtain `Sort "Stmt"` as result. If the signature is instead declared as `(Exp * Exp → Stmt)`, we obtain ⊤ because the constructed term is ill-formed (unless `Block` is a sub-sort of `Exp`).

Operation `matchCons` is quite complex, although all cases for reserved constructors follow the same pattern:

1. We check if the sort of the current term `s` is compatible with the matched constructor. For example, a match against `Nil` can only succeed if the sort is a list.
2. We retrieve the subterm sorts if any. For example, for `Cons` we have two subterms: the head element and the tail list. Auxiliary function `getListElem` carefully finds all possible list elements, taking subtyping into account.
3. We match the subterms against the subpatterns, yielding refined subterms `ss`.
4. We refine the current term by calling `buildCons` on the refined subterms and the matched constructor. Since matching may always fail, we join the result with a call to `throw`.

The last case of `matchCons` again handles user-defined constructor symbols `c`. We use the function `constructorsOfsort s` to obtain all constructors `c'` and their argument types `ss`. If the constructor has the required name and the right number of arguments, then the corresponding match might succeed. We match the subterms and refine the current term as in the other cases, but the we compute the least upper bound over all possible results. For example, when we match a constructor `Add` against sort `Exp`, we would lookup all constructors that generate sort `Exp`. For `(Add : Exp * Exp → Exp)` the match can succeed, but for `(Var : Lexical → Exp)` the match must fail. The join operator merges the results to compute a sound approximation.

Lastly, we show the code of `mapSubterms`, which needs to retrieve the current subterms as a list and pass them to `f`. However, sorts do not directly point out their subterms. Again we use `constructorsOfsort s` to retrieve the sorts of subterms indirectly by finding all constructors of the current sort and taking their parameter lists. For example, if we call `mapSubterms` with sort `"Exp"`, then computation f will be called on `[Sort "Exp", Sort "Exp"]` for constructor `Add` and on `[Lexical]` for constructor `Var`.

To summarize, in this section we defined a sort analysis for Stratego, simply by designing a sort term abstraction which implements the `IsTerm` interface. The rest of the analysis we get for free from the generic interpreter and reusable analysis code. As the reader probably noticed, the term abstraction for sorts is fairly complex in its own right. Being able to focus on the term abstraction without considering other analysis aspects was crucial.

6.4.3 Sort Analysis and Generic Traversals

In this subsection, we showcase the inter-procedurality of our sort analysis by analyzing generic traversals. A generic traversal traverses a syntax tree independent of its shape and transforms the visited nodes. Statically assigning types to a generic traversal is notoriously difficult, because the type needs to summarize all changes the traversal does to the entire tree. In this subsection, we will illustrate how our inter-procedural sort analysis can support some generic traversals, before refining our analysis further in the subsequent section.

Consider the trace of the sort analysis (Figure 6.1) of the pair desugaring from Section 6.3.1. The trace starts in the `main` function with an input term of sort `Expr`. The `main` function calls `topdown`, which calls `try(D)`, which calls the desugaring rules `desugar-type + desugar-expr`. The rule `desugar-expr` either yields a term of sort `Expr` or fails because pattern `PairExpr(...)` matches some but not all terms of sort `Expr`. Furthermore, the rule `desugar-type` definitely fails because no terms of sort `Expr` match the pattern `PairType(...)`. Even though one of the rules failed, the call `try(D)` produces a successful result by applying the input term to the identity transformation. The function `topdown` then passes the resulting term of sort `Expr` to the generic traversal **`all`**`(...)`. Since we know the sort of the current term, we enumerate all relevant constructors and the sorts of their direct subterms and recursively analyze the desugaring for them.

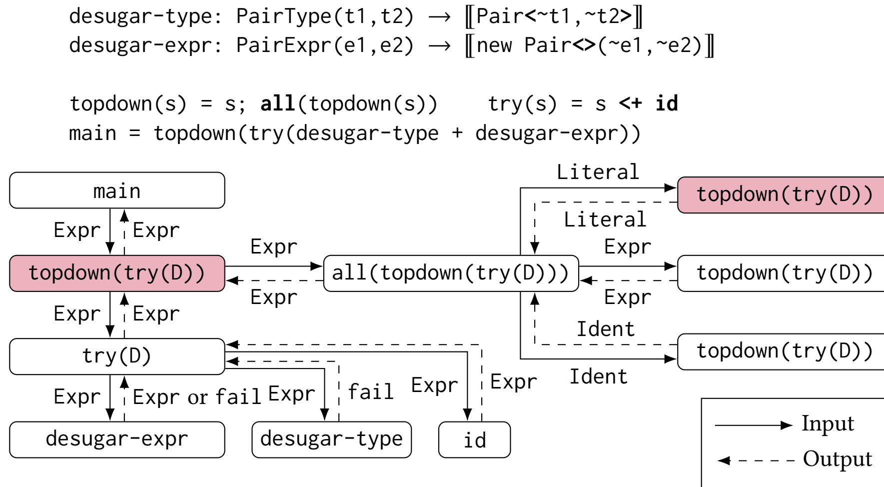


Figure 6.1: A simplified trace of the sort analysis of the pair desugaring, where we abbreviate `desugar-type + desugar-expr` with `D`.

In the example trace of Figure 6.1, we consider three subterm sorts of `Expr`. The second and third recursive call to `topdown(try(D))` resolve easily, whereas the first recursive call would end up in a cycle (shaded nodes in Figure 6.1). To this end, we use a fixpoint algorithm with widening to ensure that the analysis terminates.

The example shows why it is hard to analyze the type of a generic traversal: For different input sorts, a generic traversal might produce different output sorts. Therefore, our sort analysis reanalyzes a generic traversal for each input sort, instead of assigning a fixed type like a type checker would do.

The example is a special case of generic traversals, known as type-preserving. A generic traversal is type-preserving if the sort of the input and output term are the same at every node. However, some generic traversals change the sort of the input term. The sort analysis of this section is not capable of analyzing such type-changing generic traversals. To this end, we require a more precise analysis, which we develop in the following section.

6.5 Locally Ill-Sorted Sort Analysis

Many program transformations, like a compiler, translate terms from one sort to terms of another sort. When these program transformations use generic traversals, they produce mixed intermediate terms, which contain subterms of the input sort *and* subterms of the output sort. Because mixed intermediate terms are not well-sorted, these program transformations are challenging to type check.

For example, consider the traversal in Figure 6.2 that translates Boolean expressions into numeric expressions in a bottom-up fashion. The boolean expression `And(True(),False())` is transformed in two steps:

```
And(True(),False()) ⇝ And(1,0) ⇝ Min(1,0)
```

Even though the input term `And(True(),False())` is a valid boolean expression and the output term `Min(1,0)` a valid numeric expression, the transformation creates an intermediate term `And(1,0)`, which is ill-sorted. The sort analysis of the previous section is only able to check transformations which produce well-sorted terms and therefore cannot handle this example. To analyze this example, we need a more precise sort analysis that can represent ill-sorted terms, which we develop in the remainder of this section

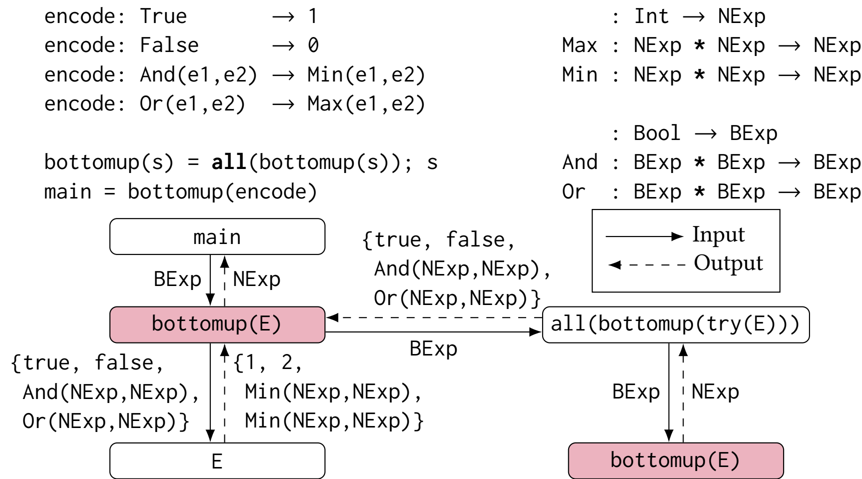


Figure 6.2: The top of the figure contains a type-changing generic traversal that translates boolean to numeric expressions. The bottom contains the analysis trace of the transformation, where we abbreviate encode with E.

6.5.1 Term Abstraction for Ill-Sorted Terms

The key idea is to use a term abstraction which can represent terms with well-sorted leafs and an possible ill-sorted prefix, such as `And(NumExp,NumExp)`. This abstract term represents all terms with `"And"` as top-level constructor and two numeric expressions as subterms. We implement this term abstraction with the following Haskell type:

```
data Term = Sorted Sort | MaybeSorted (Set (String,[Term]))
```

The case `Sorted s` represents well-sorted terms that belong to sort `s`, and the case `MaybeSorted` represents terms with an possibly ill-sorted prefix. For example, this datatype allows us to represent the ill-sorted term `And(1,0)` with the abstract term

```
MaybeSorted [("And",[Sorted "NExp",Sorted "NExp"])].
```

6.5.2 Abstract Term Operations

We develop an analysis for Stratego by implementing the term operations with the term abstraction from above. We only discuss the `matchCons` and `buildCons` operations (Listing 6.7), because the remaining functions are similar to the operations of the sort analysis.

The `matchCons` operation first matches on the term representation and in both cases calls the `matchCons'` helper function, which compares the constructors, arity and subterms. The `lookupSort'` function, similar to Listing 6.6, looks up all constructor signature for a sort, but additionally converts the signatures to abstract terms. This `matchCons` operation is more than the `matchCons` of the sort analysis, because we may know the top-level constructor of the term. This improved precision results in more pattern matches which *unconditionally* succeed or fail.

In contrast to the sort analysis, the `buildCons` operation in Listing 6.7 does not check if the constructor and its subterms belong to a valid sort. Instead, it constructs a new abstract term, which may or may not be well-sorted. The type checking of this term is then delayed until a later point.

With these definitions, the analysis would be able to check some type-changing generic traversals, however, it might not terminate because the abstract terms might grow arbitrarily large. To avoid this problem, we reduce the size of abstract terms by type checking their subterms. For example, we can type check the immediate subterms of `Or(And(1,0),1)` to obtain the abstract term `Or(⊤,NumExp)`. In the new term, the sort ⊤ indicates the type checking of `And(1,0)` failed and the term is ill-sorted. We use this technique in a widening operator [Cousot and Cousot

```
matchCons matchSub = proc (c,ps,t) → case t of
  MaybeSorted cs → matchCons' ≺(c,ps,cs)
  Sorted s → Sort.matchCons matchSub≺(c,ps,lookupSort' ctx s)
 where
  matchCons' = proc (c,ps,cs) → lubA (proc (c',ss) →
    if c == c' && length ss == length ps
      then do ss' ← mapSub ≺ (ps,ss); cons ≺ (c,ss')
      else throw ≺ ()) ≺≺ cs

buildCons = proc (c,ts) → returnA ≺ MaybeSorted [(c,ts)]

widening :: Context → Int → Term → Term → Term
widening ctx k cs1 cs2
  | k == 0 = Sorted (typecheck ctx (cs1 ⊔ cs2))
  | otherwise =
     MaybeSorted (zipSubterms (termWidening ctx (k-1)) cs1 cs2)
  where typecheck :: Context → Term → Sort
```

Listing 6.7: Abstract term operations for the locally ill-sorted sort analysis.

1992b] that ensures that the analysis terminates. The operator simply type checks all subterms deeper than a certain limit k, such that the resulting terms are not deeper than k.

6.5.3 Analyzing Type-Changing Generic Traversals

In the remainder of this section, we discuss how the analysis of this section checks type-changing generic traversals. To this end, we discuss an analysis trace of the example at the beginning of this of this section (Figure 6.2).

The trace in Figure 6.2 shows only the final fixpoint iteration (earlier iterations produce subsets of the sets shown in the trace). It starts with the analysis of the `main` function with the boolean expression sort `BExp`, which is then passed to `bottomup(E)`. In contrast to the top-down traversal, the bottom-up traversal first traverses with **`all`**`(bottomup(E))` over the subterms of boolean expressions and replaces them by numeric expressions, e.g., `And(NExp,NExp)`. The resulting set of ill-sorted terms is then passed to the rewrite rule `E`. The rule `E` then replaces each top-level boolean constructor with a numeric constructor without touching the subterms. All terms in the resulting set are now well-typed and `bottomup(E)` applies the widening operator to reduce this set to `NExp`.

In summary, we defined an advanced sort analysis, which can represent ill-sorted terms. This analysis is able to check type-changing generic traversals, which produces ill-sorted intermediate terms.

6.6 Related Work

Transformation languages like Stratego [Erdweg et al. 2014] and PLT Redex [Matthews et al. 2004] have a dynamic type checker for syntactic well-formedness. While dynamic type checking supports generic traversals, it does not help developers of transformations to understand the code. In contrast, we developed a static analysis such that program transformations can be checked before running them.

Other program transformation languages like Ott [Sewell et al. 2007], Maude [Clavel et al. 2002], Tom [Balland et al. 2007] and Rascal [Klint et al. 2009] use static type checking to ensure syntactic well-formedness. However, these languages do not support or struggle to statically check arbitrary generic traversals. Ott is a language for specifying rewrite systems and exporting them to proof assistants such as Coq or Isabelle. However, it does not support generic traversals. Maude is a language for specifying rewrite systems in membership equational logic. However, it implements generic traversals with reflection and hence cannot statically check their type. Tom and Rascal are statically typed transformation languages with support for type-preserving generic

traversals. However, they do not support type-changing generic traversals. We explained in Section 6.5 why conventional static type checkers cannot analyze type-changing generic traversals: these traversals produce intermediate terms which are ill-sorted. In this work, we aim to analyze type-preserving as well as type-changing generic traversals. We solve this problem by defining a static analysis which can represent terms with a finite ill-sorted prefix. In contrast to a conventional type checking, this term abstraction is more precise than regular types, but requires computing a fixed point.

Lämmel distinguishes two types of generic traversals [Lämmel 2003] as realized in Scrap-Your-Boilerplate [Jones and Lämmel 2003]: "type-preserving" from "type-unifying" generic traversals. A unifying generic traversal is a fold over the term that yields a value of the same "unified" type at each node. These kinds of generic traversals are easier to type statically, however, not all generic traversals fit in one of these two typing schemes. For example, a generic traversal that translates code from one language to another is neither type-preserving nor type-unifying. Rather than developing additional specialized traversal styles, our paper aims to support static analysis for arbitrary generic traversals.

Most closely related to our work, Al-Sibahi et al. present an abstract interpreter of a subset of Rascal, including generic traversals [Al-Sibahi et al. 2018]. Al-Sibahi et al. use inductive refinement types as abstract domain. The main difference of our work is that we separated analysis-independent concerns (the generic interpreter) from analysis-specific concerns (the instances). This way we can develop different analyses for program transformations with relatively little effort. Furthermore, it also simplifies the analysis definition, because most of the language complexity is captured in the generic interpreter. Lastly, our work is based on the well-founded theory of compositional soundness proofs [Keidel et al. 2018] provided by the Sturdy framework. This allows us to verify that soundness of analyses more easily, as we only need to prove that the instances are sound.

CompCert [Leroy and Others 2012] is a formally verified C compiler. The compiler guarantees that the compiled program has the same semantics as the input program. To this end, each program transformation in the compiler passes has to preserve the semantics of the transformed program. While CompCert focuses on the semantics of the transformed program, the static analyses for program transformations in this work have to satisfy a different correctness property. Soundness of these static analyses guarantees that the analyses results overapproximate which programs can be generated by a program transformation. However, soundness does not give any guarantees about the semantics of the transformed program. In the future, we aim to develop more precise analyses for program transformation languages that allow us to draw conclusion about the semantics of transformed programs.

6.7 Conclusion

To summarize, in this work, we presented a systematic approach to designing static analyses for program transformations. Key of our approach is to capture the core semantics of the program transformations with a *generic interpreter* that does not refer to any analysis-specific details. This lets the analysis developer focus on designing a good abstraction for programs. We demonstrated the usefulness of our approach by designing three analyses for the program transformation language Stratego. Our sort analyses are able to check the well-sortedness of type-preserving and even type-changing generic traversals.

7 RELATED WORK

The main contribution of this thesis is the modular specification and compositional soundness of abstract interpreters. In this chapter, we review how related works ensure soundness and compare it to our approach. The columns of Table 7.1 give an overview of how this chapter is organized. We start in Section 7.1 discussing closely related works, which modularize a certain part of the analysis definition and make its soundness proof compositional. In Section 7.2, we broaden our scope and discuss other soundness proof techniques and how they can be or have been implemented in Sturdy.

7.1 Modular Analyses Description and Compositional Soundness Proofs

Most closely related to our work are approaches that modularize some part of the analysis, such that the soundness proof of this part becomes compositional. To this end, in this section we discuss 4 dimensions along which an analysis can be modularized: language semantics (Section 7.1.1), and effects (Section 7.1.2), fixpoint algorithms (Section 7.1.3), and values (Section 7.1.4).

7.1.1 Compositional Soundness Proofs of Language Semantics

A language semantics describes how different syntax constructs are interpreted or analyzed. In this subsection, we discuss existing techniques for defining and modularizing the language semantics of abstract interpreters.

Two-Level Metalanguage Jones and Nielson [1994, Section 3] describe an approach for a modular description of a language semantics. In particular, the approach first translates the syntax of the language to algebraic operations. These algebraic operations do not carry any inherent meaning, but instead can be independently interpreted. These interpretations allow to define different semantics for the same algebraic operations like a concrete semantics or a static semantics. To prove an analysis sound, Jones and Nielson prove that it suffices to show soundness of the algebraic operations [Jones and Nielson 1994, Proposition 3.3.3]. To this end, define a logical relation [Plotkin 1980] that captures the notion of sound approximation. This proposition composes the soundness proof as no reasoning about the semantic equations is required. However, they prove this proposition for one specific algebra. This means if another analysis requires a different algebra, the proposition does not hold.

Jones and Nielson two-level metalanguage is closely related to the generic interpreters we introduced in Chapter 3. In particular, the generic interpreter maps the syntax of the language to the algebra of arrows [Hughes 2000]. Furthermore, we prove that an analysis is sound if all arrow operations of the generic interpreter are sound. This reduces the soundness proof effort and complexity, because no reasoning about the generic interpreter is necessary. In contrast, we prove a more general soundness composition theorem that holds for all arrow algebras.

Another difference is that we describe in Chapter 3 how a soundness proof can be composed with parametricity. This means that generic interpreters can be implemented with a metalanguage that enjoys parametricity, such as System-F [Reynolds 1983]. The benefit, compared to the two-level metalanguage [Jones and Nielson 1994], is that a generic interpreter based on the metalanguage System-F can capture more shared behavior. For example, imagine a language semantics that evaluates the arguments of a function in reverse order. This reversing of the argument list can be implemented as part of the generic interpreter and requires no reasoning in a soundness proof. In contrast, Jones and Nielson's two-level metalanguage needs to capture this behavior as part of an algebraic operation and hence requires more soundness proof work. A downside of relying on parametricity for soundness proof composition is that not many widely

Analysis Approaches and Frameworks	Modular Language Semantics (7.1.1)	Modular Effects (7.1.2)	Modular Fixpoints (7.1.3)	Modular Values (7.1.4)	Derive Soundness (7.2.2)	Alt. Concrete Sem. (7.2.3)	Mechanized Proof (7.2.1)
Big-Step Abstract Interpreters							
Natural-Semantics-Based AI. [Schmidt 1995] (7.1.1, 7.1.3)	○	○	○	○	○	○	○
Two-Level Metalang. [Jones and Nielson 1994] (7.1.1)	◯	○	○	○	○	○	○
Skeletal Semantics [Bodin et al. 2019] (7.1.1)	●	○	○	○	○	○	●
Cert. AI. with Pretty-Big-Step Sem. [Bodin et al. 2015] (7.1.1)	●	○	○	○	○	○	●
Galois Transformer [Darais et al. 2015] (7.1.2)	◯	●	○	○	○	○	○
Staged Abstract Interpreters [Wei et al. 2019] (7.1.2)	◯	◯	○	○	○	○	○
Abstract Definitional Interp. [Darais et al. 2017] (7.1.3)	◯	◯	◯	○	○	○	○
Astrée [Cousot et al. 2006] (7.1.4)	○	○	○	●	○	○	○
Verasco [Jourdan et al. 2015] (7.2.1)	○	○	○	●	○	○	●
Calculational Design [Cousot 1999] (7.2.2)	○	○	○	○	●	○	●
Sturdy (this work)	●	●	●	◯	●	●	◐
Small-Step Abstract Interpreters							
𝕂 framework [Rosu and Serbanuta 2010] (7.1.1)	◯	◯	○	○	○	○	○
CFA of Higher-Order Lang. [Shivers 1991]	○	○	○	○	○	●	○
Monadic Abstract Interp. [Sergey et al. 2013] (7.1.2)	○	◯	○	○	○	○	○
AAM [Horn and Might 2010] (7.2.2)	○	○	○	○	●	●	○
Scala-AM [Stiévenart et al. 2016] (7.2.2)	◯	○	○	◯	○	○	○
Datalog-based and other Logic-based Analyses							
Doop [Smaragdakis and Kastrinis 2018] (7.1.1)	○	○	○	○	○	○	○
Flix [Madsen and Lhoták 2018] (7.2.1)	○	○	○	○	○	○	●
Rhodium [Lerner et al. 2002] (7.1.4, 7.2.1)	○	○	○	◯	○	○	●
HornDroid [Calzavara et al. 2017] (7.1.1)	○	○	○	○	○	●	○
KeY [Ahrendt et al. 2016] (7.1.1)	○	○	○	○	○	●	○

Table 7.1: The table compares different analysis approaches if they modularize parts of the analysis definition (first 4 columns) and if they support certain soundness proof techniques (last 3 columns). In the first 4 columns, a full circle (●) means that a part of the analysis definition is modular and that the soundness proof for this part is compositional. A ring (◯) means that a part of the analysis definition is modular, but the soundness proof is not compositional. An empty circle (○) means that neither the analysis definition is modular nor the soundness proof is compositional. In the last 3 columns, a circle indicates if an approach supports (●), partially supports (◐), or is unknown to support (○) a particular soundness proof technique.

used programming languages enjoy parametricity. In particular, many languages were not designed with parametricity in mind and features like ad-hoc polymorphism are known to break parametricity.

Skeletal Semantics Bodin et al. follow a similar approach compared to Jones and Nielson [1994] two-level metalanguage. In particular, a *skeletal semantics* [Bodin et al. 2019] translates the syntax of the language to a specific algebra called *bones*. These algebraic operations do not carry inherent meaning, but can be interpreted to obtain a concrete semantics or a static semantics. In contrast to Jones and Nielson [1994] two-level metalanguage, the skeletal semantics algebra is more restrictive. In particular, the algebra only allows four types of operations: hooks (recursive calls to the interpreter), filters (tests which can terminate the execution), branches (parallel execution paths), and sequencing. This allows Bodin et al. to prove a more general consistency theorem that holds for all skeletal semantics. This consistency theorem can be used to prove soundness of a derived static analysis, by proving that the algebraic operations are sound. They prove the consistency theorem by induction over the structure of the algebra.

In comparison, our generic interpreters are closely related to skeletal semantics, because they map the syntax of the language to algebraic operations. In contrast, we use arrows as an algebra. The algebra of arrows is more restrictive than an arbitrary algebra used in a two-level metalanguage, but more general than the skeletal semantics algebra. More specifically, arrows allow the embedding of functions with the `arr` operation, which the skeletal semantics algebra does not. This allows us to specify more shared behavior in generic interpreters compared to a skeletal semantics, which needs to capture all behavior within algebraic operations. Furthermore, arrows are built into Haskell, which natively supports type checking of arrow expressions and provides a convenient pretty notation. In contrast, the algebra of skeletal semantics does not have built-in language support. This means that, for example, the type checking of a skeletal semantics need to be implemented explicitly for each different algebra [Bodin et al. 2019, Section 3.1]. However, the benefit of implementing type checking explicitly is that it allows to provide stronger guarantees than what the standard type system of the metalanguage can guarantee.

Another difference to our work is the use of filters in a skeletal semantics. Filters allow a skeletal semantics, for example, to check if a value has a specific type. If the value does not have the correct type, the filter fails and propagates the failure. In contrast, our case studies do not use operations equivalent to filters. Instead, the type checking of values happens within operation that combine the values, such as an addition operation. The downside of this is that it moves more behavior into algebraic operations instead of being shared in the generic interpreter, which requires more proof work. In contrast, filter operations would allow us to capture more behavior within the generic interpreter and would require less overall proof work. We believe that we could greatly benefit from using filters and want to investigate this topic in the future.

Certified Abstract Interpretation with Pretty-Big-Step Semantics Bodin et al. [2015] describe another approach that composes the soundness proof of abstract interpreters. In particular, they also derive the concrete and abstract interpreter from the same set of semantic rules. However, they use a flavor of big-step semantics called *pretty-big-step semantics* (PBS) [Charguéraud 2013]. While in standard big-step semantics each rule may evaluate arbitrarily many subterms, in PBS each rule may only evaluate one subterm. To compensate for this restriction, PBS rules accumulate an evaluation context that carries the values of previously evaluated subterms. The restricted PBS rule scheme allow Bodin et al. to describe a concrete and an abstract *meta-interpreter* for PBS rules that are independent of the object language that the rules describe. For this concrete and abstract interpreter, they prove a generic soundness theorem that holds for all object languages. This generic soundness theorem composes a soundness proof for a given set of PBS rules.

When we compare Bodin et al. approach to our approach, then, a generic interpreter is related to PBS rules for a particular language. In particular, a generic interpreter allows to derive a concrete and an abstract semantics for the same language, similar to PBS rules. Furthermore, the algebra that a generic interpreter uses is related to the rule scheme of PBS. Following this analogy, then the metalanguage of a generic interpreter would need to be related to the concrete and abstract meta-interpreter of PBS rules. However, here the analogy breaks down, as we use

the same metalanguage for the generic interpreter to derive a concrete and an abstract interpreter. This makes the proof of our generic soundness theorem easier, as it reduces to structural induction over the arrow expressions. In contrast, Bodin et al. have a more elaborate proof of the generic soundness theorem, since it needs to relate two different meta-interpreters.

Big-step Abstract Interpreters Big-step abstract interpreters [Schmidt 1995] are a specific style of static analysis. More generally, big-step abstract interpreters follow the theory of *abstract interpretation* by Cousot and Cousot [1977]. Abstract interpretation is a methodology for developing sound static analyses. In particular, an abstract interpreter is like a standard concrete interpreter, but it calculates abstract values as result. These abstract values represent properties of programs we want to analyze. To prove an abstract interpreter sound [Cousot and Cousot 1977], we need to related it to the concrete interpreter with a *galois connection* [Ore 1944]. However, abstract interpretation itself does not specify in which style the language semantics should be described in. In fact, abstract interpretation can be used to prove soundness for analyses described with several styles of language semantics.

Schmidt [1995] uses the concepts of abstract interpretation and applies it to a big-step semantics. A big-step abstract interpreter takes an expression as input and recursively evaluates the expression in one *big step* to an abstract value [Schmidt 1995]. This is in contrast to small-step abstract interpreters [Shivers 1991; Horn and Might 2010], that evaluate expressions to an abstract value in many small steps. Schmidt [1996] discusses both the small-step and big-step approach to abstract interpretation and compares them to each other. Big-step abstract interpreters have the benefit, that we can implement them as a recursive function in a metalanguage of our choice [Darais et al. 2017]. However, in comparison to small-step abstract interpreter, big-step abstract interpreters require more complicated techniques to ensure termination of the analysis, while computing a fixpoint [Schmidt 1998]. In the earliest descriptions of big-step abstract interpreters, the abstract interpreter and concrete interpreter are two separate artifacts [Schmidt 1995; Rosendahl 1995; Schmidt 1998]. To prove such abstract interpreters sound, the soundness proof has to bridge a large gap between the abstract and concrete interpreter. This causes the soundness proof to become monolithic, because the interpreters differ at various places, as we explain in Section 1.3.

In this dissertation, we describe all our analyses as big-step abstract interpreters. However, in contrast to Schmidt [1995], we modularize the language semantics of the abstract interpreter. In particular, we derive both the concrete and abstract interpreter from the same generic big-step interpreter (Chapter 3). This approach makes it easier to derive new abstract interpreters for the same language, as a part of the language semantics is already captured by the generic interpreter. But more importantly, this generic interpreter provides the necessary structure along which we compose a soundness proof.

Monotone and Distributive Analysis Frameworks Dataflow analyses are commonly implemented in the *Monotone Framework* [Nielson et al. 1999; Smits and Visser 2017] or in distributive frameworks such as *IFDS/IDE* [Reps et al. 1995; Sagiv et al. 1996]. An analysis implemented in these frameworks propagates its results along the control-flow graph of the program with the help of transfer functions. In the IFDS/IDE framework, these transfer functions are encoded in a way such that the fixpoint algorithm reduces to an efficient graph reachability problem. Using a monotone or distributive framework to implement dataflow analyses has the benefit that all analyses share the same infrastructure, fixpoint algorithm, and meta theory. This reduces the effort of implementing new analyses and reduces the likelihood of making mistakes in the analysis implementation.

Unfortunately, these frameworks do not simplify the soundness proof of their analyses. For example, Nielson et al. [1999, Section 2.2.2] show how a live variables analysis implemented in the monotone framework can be proven sound. The soundness proof relates the analysis results to a binary correctness relation over a concrete small-step semantics [Nielson et al. 1999, Figure 2.5]. This soundness proof is similar to a preservation lemma for type systems and hence it has many of the same problems as we discussed above. In particular, the soundness proof is not compositional and proof parts cannot easily been shared between different analyses in the monotone framework.

Datalog-based Analyses Many analyses are implemented in Datalog [Szabó et al. 2016, 2018; Smaragdakis and Balatsouras 2015; Grech and Smaragdakis 2017], a declarative logic programming language. Datalog is a suitable language because it has built-in support for fixpoint computations, whereas general purpose programming languages usually have no support. Furthermore, Datalog programs can be efficiently executed [Antoniadis et al. 2017] and incrementalized [Szabó et al. 2018].

Analyses in Datalog are defined with a number of logic rules. For example, the following rules define an intraprocedural points-to analysis [Smaragdakis and Balatsouras 2015]:

```
VarPointsTo(var, obj) ← AssignAlloc(var, obj)
VarPointsTo(to,  obj) ← Assign(to, from), VarPointsTo(from, obj)
```

These rules define a relation `VarPointsTo(var,obj)`, which expresses that a variable `var` may point to an object `obj`. The first rule says that a variable `x` may point to an object `obj`, if `x` appears in an allocation `x = new obj`. The second rule says that a variable `x` may point to an object `obj`, if `x` appears in an assignment `x = y` and the variable `y` may point to `obj`.

Proving a Datalog-based analysis sound requires to relate the contents of the relations to the dynamic semantics of the analyzed language [Smaragdakis and Kastrinis 2018]. For example, for the pointer analysis above we need to prove that if the evaluation of a program results in a store and this store has a binding $x \mapsto \mathrm{obj}$, then $\texttt{VarPointsTo}(x, \mathrm{obj})$. The difficulty is that one Datalog rule may relate to multiple rules of the dynamic semantics and vice versa. This requires asking if the sum of Datalog rules cover all cases of the dynamic semantics. It is unclear how such a proof can be composed of soundness lemmas for a subset of the rules. This increase the complexity of the proof and makes it hard to share parts of the proof for a different analysis for the same language.

Hoare Logic Another popular approach to implement static analysis, is to abstractly interpret a program in a Hoare logic, whose formulas are solved by an SMT solver [Schneidewind et al. 2020; Ahrendt et al. 2016; Benton 2004]. A Hoare logic is defined in terms of triples $\{\mathrm{Pre}\}$ `expr` $\{\mathrm{Post}\}$. Such a triple says that the evaluation of the expression `expr` ensures the post condition Post if the precondition Pre holds.

The soundness proof usually requires one or two steps. The first (optional) step relates that the concrete operational semantics to a concrete Hoare logic. The second step shows that the abstract Hoare logic is an overapproximation of the concrete Hoare logic. This second step tends to be easier to prove than a soundness proof between operational semantics, because the Hoare logic abstracts away some operational aspects of the language. However, the soundness proof is not compositional: Hoare triples mix different concerns of the analysis, such as the analysis of values, heap, and exceptions. This makes it hard to reuse parts of the soundness proof for a different analysis for the same language.

7.1.2 Compositional Soundness Proofs of Effect Abstractions

Many languages define effectful operations like store updates, exceptions, or file system accesses. However, implementing maintainable analyses for effectful operations is challenging, because effects are usually crosscutting concerns and interact with each other. Therefore, in this section, we discuss works that modularize the effects of the language and the analysis.

Monads Moggi [1991] describes how *monads* [Lane 1971] can be used to abstract over the effects of a programming language. A monad is a type wrapper $M(A)$, with an operation $\mathrm{inject} : A \to M(A)$ that injects a value into the monad and an operation $\mathrm{join} : M(M(A)) \to M(A)$ that joins two layers of the monad into one. This abstraction can be used to describe the effects of backtracking, continuations, exceptions, mutable state, etc [Wadler 1992]. Later, [Liang et al. 1995] showed how these effects can be described separately with *monad transformers*. Each monad transformer captures a specific type of effect and can be combined with other transformers to compose their effects. Sergey et al. [2013] showed that concepts in static analysis such as context-sensitivity, poly-variance, flow-sensitivity are analysis- and language-independent and can be captured by an appropriate monad. This allows building analyses more quickly by reusing

existing functionality. However, they do not explain how monadic abstract interpreters can be proven sound.

Arrows In this work, we use another effect abstraction called *arrows* [Hughes 2000]. Arrows are similar to monads in that they describe the type of an effectful computation. However, they are more general in that every monad is an arrow, but not every arrow is a monad. The main difference between monads and arrows is that monads only capture the outputs of a computation, whereas arrows also capture the input. This means arrows can describe both monadic as also comonadic effects [Hughes 2000]. Furthermore, arrows are described with a different algebra than monads. In particular, arrows do not have an equivalent to the join operation for monads. Instead, they are defined with operations that explain how the effects of the arrow commute with the products and sums of the language. Lastly, similar to monads, arrows have a convenient pretty notation [Paterson 2001] that makes it easier to write effectful computations.

Galois Transformer Darais et al. [2015] refine the concept of monadic abstract interpreters [Sergey et al. 2013] and describe each type of effect with a *galois transformer*. A galois transformer is a monad transformer, which commutes with respect to galois connections between two underlying monads. Galois Transformer have the benefit, that a sound abstract interpreter can be derived from a generic monadic interpreter by instantiating it with a stack of galois transformers. Furthermore, this gives the flexibility to derive alternative semantics by reordering or extending the stack of galois transformers. However, galois transformer make the assumption that the generic monadic interpreter is sound, for all sound monad instances. Yet, the paper does not explain how this can be proved. In contrast, we show in Chapter 3 how a generic interpreter can be proven sound without knowing the underlying instances.

Galois transformers are closely related to the analysis components we introduce in Chapter 4. More specifically, our analysis components consist of an arrow-based interface that describes the operations of the component, a concrete and an abstract instance, and a proof that the abstract instance soundly approximates the concrete instance. The main contribution of Chapter 4 is that analysis components can be composed into larger components while preserving soundness. Furthermore, the successive composition of analysis components gives us a super component, which can be used to soundly instantiate a generic interpreter with a compatible interface. However, in comparison, each of our analysis components consists of a *pair* of arrow transformers. This allows us to define different arrow transformers for the concrete and abstract interpreter. In contrast, a galois transformer is a *single* monad transformer. This means, the concrete and abstract interpreter need to be instantiated with the same stack of monad transformers ([Darais et al. 2015, Theorem 1]).

Staged Abstract Interpreters Wei et al. [2019] developed an approach that improves the performance of abstract interpreters following the approach of [Darais et al. 2015] and [Darais et al. 2017]. In particular, they specialize an abstract interpreter to a given program, such that it runs faster. To this end, they derive both the concrete and abstract interpreter from the same generic interpreter. This allows them to create special versions of operations that *stage* parts of the input. Furthermore, they use monads [Moggi 1991] to abstract over the effects within the generic interpreter, and they use monad transformers [Liang et al. 1995] to describe the effects modularly. This allows them to specialize the interpreter for effects such as an environment, by using versions of monads that stage that part. Lastly, they informally explain why a staged abstract interpreter remains sound if the unstaged abstract interpreter is sound. However, it is unclear if these soundness proofs are compositional.

We believe that we can make use of this technique to speed up our abstract interpreters as well. While the language of the Sturdy Framework, Haskell, does not have a library that allows staging computations, we believe that the concepts of Wei et al. work translate to Haskell as well.

$\mathbb{K}$ framework The $\mathbb{K}$ framework [Rosu and Serbanuta 2010] is an approach that allows to describe formal operational semantics for a language and to derive program verifier for that language. The $\mathbb{K}$ framework distinguishes itself from other semantic approaches in that it modular-

izes the definition of the language semantics and the effects of the language. In particular, the language semantics is defined with labeled rewrite rules in a small-step style. The rewrite rules contain labeled cells that each capture a specific effect of the language. These labeled cells allow adding additional effects to the language, without needing to change existing rules that do not interact with the new effects. For example, they allow to add a store to an existing language, without needing to change existing rules to pass around the store explicitly.

Recently, Alam et al. [2018] used the $\mathbb{K}$ framework to develop a static taint analysis. The definition is similar to other small-step abstract interpreters [Shivers 1991]. For example, to analyze an `if` statement, the analysis first analyzes the first branch. Then the analysis restores the environment to before the `if` statement and analyzes the second branch. Afterwards the analysis joins the resulting environments. While the definition of the analysis is modular, the soundness proof is not compositional. In particular, the analysis is a separate definition, which does not share any behavior with the dynamic semantics of the language. Because there is no shared behavior, there is no common structure along which a soundness proof could be composed.

In comparison, in Chapter 4 we define analysis components based on a pair of arrow transformers [Hughes 2000] that modularize the analysis definition of different effects and their soundness proof. Similar to a labeled cell of the $\mathbb{K}$ framework, each analysis components encapsulates a specific effect of the language and multiple components can be composed to combine the underlying effects. In contrast, we show that our analysis components are also a suitable abstraction for soundness proofs. In particular, each analysis component can be proven sound once and for all and the composition of multiple analysis components remains sound.

Sturdy In Chapter 4, we explain how we can modularize the analysis description of effects with analysis components. Each analysis components consists of a pair of arrow transformers [Hughes 2000] and a soundness proof. Each arrow transformer encapsulates a particular effect and can be composed with other transformers to their different effects. Since arrow transformers encapsulate the definition of effects, we can prove each analysis component sound once and for all. A challenge of this approach is to describe the interaction between different effects, e.g., the interaction between mutable variables and exceptions. We describe this interaction with liftings required for composing multiple arrow transformer of multiple analysis components. Lastly, since every monad is an arrow via the Kleisli arrow [Lane 1971], we can use existing monads described by other works [Sergey et al. 2013; Darais et al. 2017; Wei et al. 2019].

7.1.3 Compositional Soundness Proofs of Fixpoint Algorithms

Because of loops and recursion an analysis cannot compute its result in a single step. Instead, it uses a fixpoint algorithm that successively reanalyzes parts of the program until the result does not change anymore. Yet, many fixpoint algorithms are monolithic, which makes them harder to change and harder to prove them sound and terminating.

Natural-Semantics-Based Abstract Interpretation Schmidt [1995] describes one of the first fixpoint algorithms for big-step abstract interpreters, that operates on the derivation tree. The fixpoint algorithm unfolds the abstract derivation tree until each branch either terminates or repeats itself. The algorithm detects recurrent calls of the abstract interpreter by memoizing parts of the abstract derivation tree. The algorithm does not specify an order for iterating on the analysis results. Instead, the algorithm generates a number of recursive equations, which then can be solved with an arbitrary iteration order to calculate the fixpoint. We use some of the techniques of this fixpoint algorithm in Chapter 5, but modularize its definition.

Abstracting Definitional Interpreters Darais et al. [2017] propose to define fixpoint algorithms modularly from reusable fixpoint combinators. For example, Darais et al. describe fixpoint combinators that collect an execution trace, collect dead code, collect abstract garbage, and calculate the least fixpoint of a big-step abstract interpreter. The fixpoint combinators take the open-recursive type of the abstract interpreter as input, which allows to compose the combinators by nesting them. Furthermore, the fixpoint combinators are parameterized by a monad, which allows their

effects to compose. While Darais et al. [2017, Section 4.1] prove soundness and termination of the least fixpoint combinator, they do not explain how a complete modular fixpoint algorithm can be proven sound and terminating. Furthermore, Darais et al. do not show that their fixpoint combinators are language-independent and reusable.

Sturdy In Chapter 5, we introduce a notion of fixpoint combinators similar to Darais et al. [2017], which allow us to implement modular fixpoint algorithms. In particular, our combinators also take the open-recursive type of the abstract interpreter as input. However, in contrast to Darais et al., we allow higher order combinators. Furthermore, we formalize the fixpoint combinators and prove that a modular fixpoint algorithm is sound, if all its combinators are sound. This makes the soundness proof of the combinators independent of each other and reusable. Moreover, we show in our evaluation that our fixpoint combinators are language-independent and reusable across different analyses

7.1.4 Compositional Soundness Proofs of Value Abstractions

A programming language often defines multiple built-in value types such as numbers, strings, and booleans. Finding a suitable abstraction for values is one of the most important tasks of an analysis developer. However, this is difficult because there is often not a single value abstractions that works best in all cases. In particular, some value abstractions are more precise analysis for some operations, but less precise for other operations. Therefore, it can be beneficial to combine multiple abstractions into a new abstract domain that is more precise than its constituents.

Cartesian Product The simplest composition of value abstractions is the cartesian product [Cousot and Cousot 1979]. A cartesian product combines two or more value abstractions into a tuple, while allowing no exchange of information between different abstractions. Disallowing the exchange of information has the benefit that different value abstractions can be independently defined and proven sound. However, it has the drawback that the value abstractions cannot improve each other's precision.

Lerner et al. [2002] refine the approach of cartesian products. In particular, their approach allows analyses which not only produce information about the program, but can also transform the program. For example, a points-to analysis can be combined with a transformation that inlines virtual method calls, if the call target can be uniquely determined. The transformations are applied while the analysis is running. This allows different components of the cartesian product to indirectly affect each others results. This makes this approach more precise than conventional cartesian products.

Reduced Product While a cartesian product does not allow the exchange of information between its components, a *reduced product* [Cousot and Cousot 1979; Cortesi et al. 2013] lifts this restriction. In particular, reduced products define a *reduction operator* that exchanges information between components and improves their precision. For example, a reduced product of intervals and congruences improves the precision of an interval analysis of division and a congruence analysis of less-than operations. Reduced products still allow to define and prove analyses for each component independently. The only part of a reduced product that cannot be independently defined is the reduction operator. The reduction operator needs to be defined for each reduced product and proven sound explicitly.

Network of Domains Cousot et al. [2006] refine the idea of reduced products within the static analyzer *Astrée*. In particular, they define a reduced product, whose components are processed in a specific order. This means that the reduced product is not commutative anymore. The components of the reduced product can communicate information to other components with messages. These messages decouple the components from each other, in contrast to a monolithic reduction operator. This means that abstract domains of the components can be independently defined and proven sound, but they can still exchange information via messages.

Sturdy So far we have not explored modular definitions of value abstractions in Sturdy. However, we assume that they fit well into our framework. More specifically, the interfaces for value operations allow the definition of multiple value abstractions. Furthermore, the same interface allows combining multiple value abstractions with, e.g., a reduced product. However, there are also open questions. For example, it is unclear how the effects for two value abstractions can be soundly combined. For instance, imagine a case where an operation for one value abstraction causes an exception, whereas for another value abstraction it does not.

7.2 Other Techniques for Ensuring the Soundness of Static Analyses

In this section, we broaden our scope and discuss other techniques that analysis developers use to ensure that their analysis is sound.

7.2.1 Mechanized Soundness Proofs

In particular, many analysis developers prove soundness by using a mechanized proof assistant such as Agda, Coq or Idris. Such mechanized soundness proofs have the benefit that they are more trustworthy than pen-and-paper proofs. However, without modularization, mechanized soundness proofs have the same issues as monolithic proofs we discuss in Section 1.3. To this end, soundness proof mechanization is orthogonal to the soundness proof modularization we focus on in this thesis. In the remainder of this subsection, we discuss different works that successfully mechanized soundness proofs.

Verasco Jourdan et al. [2015] developed a static analyzer for the C programming language, that was formally verified with the Coq proof assistant [Coq Development Team 2019]. The analyzer is described as an abstract interpreter over the C#minor intermediate language used in the CompCert compiler. The effort of developing and proving the analysis sound is enormous, owed in part to the complexity of the C#minor language. The soundness proof consists of two main steps: The proof first relates the C#minor semantics to a specific Hoare logic and in a second step shows that the abstract interpreter infers properties of this Hoare logic. The proof consists of 17k lines of Coq code, which is approximately the same size as the implementation of the abstract interpreter itself. This means that the effort of proving the analysis sound (measured in lines of code) is approx. the same effort as developing the analysis itself.

Flix Madsen and Lhoták [2018] also mechanize soundness proofs in the Flix analysis framework, but follow a different approach. Flix uses an SMT solver to prove soundness of some parts of an analysis automatically. To this end, analysis developers need to add annotations to the analysis code, that aid the SMT solver in the soundness proof. These annotations describe mathematical properties that the analysis code satisfies, such as monotonicity. While this approach works well for proving soundness of value operations, Madsen and Lhoták [2018] have not explored if a complete analysis can be proven sound with an SMT solver.

Rhodium Similar to Flix, the analysis framework *Rhodium* [Lerner et al. 2005] mechanizes its soundness proofs with an automatic theorem prover. In particular, they manually prove that an analysis implemented in Rhodium is sound, if all its flow functions are sound. Soundness of the flow functions is then proved automatically with an SMT solver. In contrast to Flix, Rhodium does not require annotations on flow functions to aid the theorem prover. However, Lerner et al. [2005] also do not evaluate how many flow functions could be proven sound automatically.

Mechanized Proofs vs. Pen-And-Paper Proofs In the following we discuss the tradeoffs between mechanized soundness proofs and pen-and-paper proofs. Due to the high complexity of analyses and languages, conventional soundness proofs require a lot of bookkeeping. For example, this bookkeeping includes the substitution of logic variables, verifying that a lemma applies to the proof goal, and keeping track of available hypotheses. Manual bookkeeping in pen-and-paper proofs can lead to mistakes, whereas in mechanized proofs the proof assistants takes care

of the bookkeeping. This means, in contrast to pen-and-paper proofs, mechanized proofs are 100% trustworthy. The trustworthiness of the soundness proof is especially important for static analyzes that verify mission-critical software. However, it also significantly increases the effort of the soundness proof. For example, it takes more effort to convince a proof assistant that a seemingly trivial property holds, than to make a hand-wavy argument in a pen-and-paper proof. Furthermore, many other analyses uses cases do not justify the additional effort of a mechanized soundness proof. For these cases a high-level pen-and-paper proof may be more valuable than a fully-fledged mechanized proof.

Sturdy The main goal of this work is to simplify the soundness proof by decomposing it into smaller lemmas. This decomposition reduces the amount of bookkeeping and hence reduces the likelihood of making a mistake in a pen-and-paper proof. This makes pen-and-paper proofs of static analyses more approachable, as we show with our case studies. That being said, we acknowledge the benefits of mechanized soundness proofs. In particular, we investigated a formalization of our theory in the Coq proof assistant, which we documented in the master thesis of Jens de Waard [de Waard 2020].

7.2.2 Deriving the Implementation and Soundness Proof of Static Analyses

Developing a new analysis from scratch and proving it sound is difficult. This is especially the case if the soundness proof is conducted *after* the analysis was implemented. In such a case the analysis often differs significantly from the concrete semantics and hence is hard to relate in a soundness proof. To address this problem, several approaches systematically derive a static analysis and alongside a soundness proof.

The Calculational Design of a Generic Abstract Interpreter Cousot [1999] describes an approach based on abstract interpretation that uses the soundness proof as the guiding principle to derive an analysis. In particular, analysis developers apply a Galois connection to the collecting semantics of the concrete interpreter and then perform a series of small soundness proof steps until the result becomes computable. This approach guides the search for the abstract semantics, as often there is only one sensible reasoning step possible. This approach is especially useful for implementing analyses with complicated abstractions, such as Polyhedra [Chen et al. 2008; Singh et al. 2017] or regular tree grammars [Cousot and Cousot 1995]. Furthermore, it is useful for implementing analyses for complicated collecting semantics such as backward collecting semantics [Cousot and Cousot 1992a]. However, in our experience it is hard to scale the approach to a complete analysis for a real-world programming language. In particular, the proof steps for large language constructs become too complicated.

Our work complements this approach. In particular, we used the calculational approach to derive the abstract semantics and soundness proof of some operations of generic interpreters in our case studies. However, in comparison the reasoning steps are easier because the operations only describe a small piece of functionality. Furthermore, the semantics of each operation can be derived independently of each other.

Abstracting Abstract Machines (AAM) Horn and Might [2010] describes another systematic approach for deriving an abstract interpreter from an abstract machine. The derivation starts from the concrete semantics in form of an abstract machine. The approach then applies a series of systematic transformations to the abstract machine to derive a corresponding abstract interpreter. These transformations are easy and systematic because of the restrictiveness of the abstract machine semantics. In particular, an abstract machine consists of a tuple that captures the entire state of the machine and a transition relation between states. Each transformation translates a single transition of the old machine to a transition of a new machine. Furthermore, since each transformation is systematic, the soundness proof follows systematically as well.

The AAM approach has proven to scale beyond simple calculi to languages like Java.[1] However, using this approach to define analyses for other large languages becomes tedious, due to a

[1] https://github.com/Ucombinator/jaam/

lack of modularization. In particular, there is almost no code reuse between analyses for different languages and the same abstractions need to be reimplemented again and again. In contrast, this work enables a high degree of reuse of analysis functionality. More specifically, Chapter 3 explains how the same generic interpreter can be reused for different analyses for the same language and Chapter 4 and Chapter 5 describe how effect abstractions and fixpoint algorithms can be reused across different languages.

Scala-AM The Scala-AM [Stiévenart et al. 2016] framework closely follows the AAM approach, but addresses the problem of reusing functionality by modularizing the abstract machine semantics. To this end, Scala-AM separates the concerns of value abstraction, addresses and time stamps, language semantics, and machine abstractions. This allows analysis developers to reuse existing functionality or to swap out parts of the analysis without needing to change the rest. While the Scala-AM framework reduces the development effort of analyses and improves their maintainability, it does not explain how analyses can be proved sound. In particular, it is unclear if new analyses can be derived with soundness preserving steps like with AAM. Furthermore, it is unclear how parts of the analysis can be swapped out while preserving soundness.

7.2.3 Soundness Proofs via an Alternative Concrete Semantics

Many soundness proofs relate the abstract semantics directly to the concrete semantics in one step. However, for some analyses this may not be possible, because the concrete semantics may not be in a form in which a soundness proof is possible. In such a case the concrete semantics first needs to be related to an *alternative* concrete semantics and then in a second step to the abstract semantics.

Abstract Garbage Collection Such a detour is for example necessary for an analysis that performs abstract garbage collection [Might and Shivers 2006b; Es et al. 2019]. Like concrete garbage-collection, the abstract version frees unused addresses in the analysis store. However, the goal is not to save space, but to improve the precision of the analysis. It is difficult to relate such an analysis in a soundness proof directly to a concrete semantics that does not perform garbage collection. The reason is that the concrete store may contain addresses that the abstract store removed and hence the stores are not related via a galois connection. To solve this issue, analysis developers first relate the concrete semantics to an alternative concrete semantics with garbage collection, and in a second step to the abstract semantics.

Reaching Definitions Analysis Another example is a reaching definitions analysis [Nielson et al. 1999], a common dataflow analysis that calculates which definitions reach a certain statement. The analysis pairs with every address in the analysis store, the set of statements from which the address was last modified. Also, for this analysis it is difficult to prove soundness directly, because this extra information is not present in the concrete interpreter. Instead, analysis developers prove soundness with respect to a dynamic reaching definition analysis, that is, the concrete semantics extended with reaching definitions information.

Sturdy Sturdy supports such extra proof step via an alternative concrete semantics. More specifically, in Sturdy the concrete and alternative concrete semantics can be derived from the same generic interpreter, as we show in our case studies. This allows analysis developers to perform the first proof step via an alternative concrete semantics with the same techniques as an ordinary soundness proof.

7.3 Testing Soundness

Testing static analyses is a valuable tool for finding bugs in the analysis code. Testing analyses is complementary to traditional soundness proofs: Even though tests cannot prove that an analysis is sound, they often reveal unsoundness bugs faster than a manual soundness proof. This allows analysis developers to make changes to the analysis, while getting immediate feedback if

the change was unsound. There are several techniques for testing soundness of static analyses effectively.

Andreasen et al. [2017] propose an approach that compares the results of the analysis to a concrete execution of a program. If the results of the concrete execution are included in the analysis results, the test passes. Otherwise, the test fails as the analysis does not soundly overapproximate the concrete execution. If a test fails, their approach shrinks the analyzed program to a smaller program on which the test still fails. This reduces the size of the counterexample, which makes it easier to find the problem in the analysis. While this approach is relatively easy to implement as concrete language implementations are readily available, it does not work for all types of static analyses. For example, it does not work for static taint analyses, because the taint information is not available in the concrete execution of the program.

Klinger et al. [2019] follow a different approach for testing static analyses. They propose to compare the results of multiple static analyses of the same type to each other. If most analyses agree on a result, but few analyses underapproximate the result, the few analyses are likely unsound. Conversely, if few analyses overapproximate a result agreed upon by most analyses, the few analyses are likely imprecise. This technique has the benefit that it works for all types of static analyses. However, it requires that multiple analyses of the same type are available, which often is not the case.

To summarize, testing effectively discovers cases where the analysis is unsound. However, it is not sufficient to guarantees soundness of the analysis, as tests could miss soundness bugs.

8 FUTURE WORK

In this chapter, we discuss possible directions in which this work can be extended and ongoing efforts in these directions.

8.1 Proof Mechanization

In this work, we considered soundness proofs with pen and paper. While these types of proofs worked well for our case studies, they do not provide a high enough level of reliance for other analysis uses cases. However, as we explained in Section 7.2.1, we view our approach as orthogonal to proof mechanization. In particular, a mechanized soundness proof still becomes easier by using our principles of compositionality. Hence, an interesting direction for future work is the mechanization of our proof techniques with proof assistants.

Such a mechanization would require to formalize our methodology in a proof assistant. The fundamental definitions of abstract interpretations, such as partial orders, galois connections and fixpoint theory, already have been formalized in several prior works [Dubois 2000; Bertot 2008; Jourdan et al. 2015; Darais and Horn 2019]. The missing piece is the formalization of our core theorems: Theorem 3.4.1, Theorem 3.5.2 in Chapter 3, and Theorem 4.4.2, Theorem 4.4.1 in Chapter 4. However, these theorems are not straight-forward to formalize in a proof assistant because of the high-level of abstraction. For example, Theorem 3.4.1, Theorem 4.4.2, and Theorem 4.4.1 are parametric in the type of algebra that the generic interpreter or the analysis component uses. While mechanized formalizations of F-algebras exist [Timany and Jacobs 2016], it is still an open question how they can be applied to proof our theorems. As a partial solution, we could use a proof assistant to prove soundness lemmas of operations and assume that our core theorems hold without proving them. Such partially mechanized soundness proofs have a higher reliability compared to pure pen-and-paper proofs, however, they do not give the same level of assurance as end-to-end mechanized proofs.

As a preliminary step in this direction, we explored proof mechanization in the MSc thesis of Jens de Waard.[1] De Waard's work implements a part of our methodology in the Coq proof assistant [Coq Development Team 2019]. In particular, de Waard also derives a concrete and abstract interpreter from a generic interpreter. However, instead of using our core theorems, de Waard's work uses the proof automation of Coq to compose a soundness proof. While this proof automation was able to compose a soundness proof for a small case study, the proof automation may fail for larger case studies. To this end, it would be more desirable to mechanize our core theorems for soundness proof composition, because they hold universally even for the most complex languages and analyses.

8.2 Backward Analyses

An interesting direction for future work is the extension of our framework to other types of static analyses, such as analyses that approximate a *backward-collecting semantics* [Cousot 1999]. For example, consider the following program:

```
void safeIncrement(int i, int[] array) {
  if(0 ≤ i && i < array.length)
    array[i] += 1;
}
```

A forward interval analysis would first analyze the condition of the `if` statement and then the body. However, a forward analysis cannot determine that the array access `array[i]` is safe, because it does not refine the interval for the index variable `i` after the condition. This refinement can be done by an analysis that approximates a backward-collecting semantics. Such a backward

[1] https://svenkeidel.de/theses/jens-de-waard.pdf

analysis takes an abstract value as input and returns an approximation of the environment under which the analyzed program could have evaluated to this value.

To implement a backward analysis in Sturdy and make its soundness proof compositional, we would need to derive it from a generic interpreter. Most of the arrow operations are general enough, that they can be reversed for a backward analysis. Furthermore, our core theorems are general enough, that they allow us to compose a soundness for a backward analysis. However, it is unclear how pure functions embedded with `arr` can be reversed in general. Since functions are not reversible in general, they may require manually treatment, i.e., all pure functions in the generic interpreter need to be reversed by hand. This would significantly increase the effort of implementing backward analyses and proving them sound.

8.3 Performance Scalability

The focus of this work has been the soundness of static analyses. Nonetheless, real-world programs often pose large analysis problems and it is unclear how well Sturdy analyses scale to these larger analysis problems. While we demonstrate in Chapter 4 and Chapter 5, that our modularization does not induce a significant performance penalty, the question of scalability still remains open. For example, it could be that performant analyses require many optimizations and tweaks, that make it more difficult to implement the analysis in a modular style. Furthermore, it is unclear if our style of analysis semantics (big-step abstract interpreters) can be scaled to larger analysis problems. As of now, there have not been any works that investigated the performance characteristics of big-step abstract interpreters thoroughly.

To investigate the question of scalibility, we are actively working on implementing larger analyses in Sturdy. In particular, Tobias Hombücher is working on analyses for Jimpl,[2] a dialect of Java-Byte Code that is easier to analyze [Vallee-Rai and Hendren 1998]. Furthermore, Katharina Brandl is working on analyses for WebAssembly,[3] an assembly language which runs in browsers [Rossberg et al. 2018]. These case studies allow us to compare the performance of our analyses to existing state-of-the-art analyses.

[2] https://gitlab.rlp.net/plmz/sturdy/-/tree/jimple

[3] https://gitlab.rlp.net/kabrandl/sturdy/-/tree/wasm

CONCLUSION 9

In this dissertation, we identified a problem that makes soundness proofs of static analyses difficult: Their soundness proofs tend to be monolithic, which means that it is hard to decompose these proofs into smaller soundness lemmas, which can be proven independently. We identified three key problems that make the soundness proof monolithic: The impedance mismatch between concrete and abstract semantics, the mixing of concerns in the implementation of different effects, and monolithic fixpoint algorithms. To this end, we presented a methodology for structuring static analyses, that makes their soundness proof *compositional*.

The first problem we solved was the impedance mismatch between the concrete and abstract semantics. This impedance mismatch makes it difficult to decompose the soundness proof into smaller parts, which can be proven independently. We solved this problem by deriving both the concrete and abstract semantics from a generic semantics that captures their similarities. This generic semantics provides a common structure to compose the soundness proof. In particular, we showed that if the generic semantics is described with arrows, or in a language which enjoys parametricity, the soundness proof of the entire analysis follows from soundness lemmas for the concrete and abstract instance. To summarize, generic semantics reduce the complexity of the soundness proof, because we only need to prove smaller self-contained soundness lemmas. Furthermore, they reduce the effort of the soundness proof, because we do not need to prove anything about generic semantics.

The second problem we addressed is the mixing of concerns in the implementation of different effects of the language. We solved this problem by constructing analyses from small and reusable analysis components. Each analysis component is self-contained and captures a different aspect of the analysis. However, a challenge for making their soundness proofs compositional is the interaction between different effects. To this end, we defined a specific composition operation for analysis components, that explicitly describes their interaction. Proving the interactions sound adds extra proof work. However, we show in our evaluation that most interactions are trivial and hence their implementation and soundness proof can be derived automatically. To summarize, analysis components reduce the complexity of the soundness proof, because we can prove each component sound independently. Furthermore, they also reduce the effort of the soundness proof, because language-independent components can be proven sound a priori as part of a library.

The third problem we solved was to modularize monolithic fixpoint algorithms. A fixpoint algorithm is one of the most complicated parts of a static analysis and hence difficult to prove sound. We solve this problem by composing fixpoint algorithms from small and reusable fixpoint combinators. Each of these combinators addresses a specific concern of the fixpoint algorithm. These combinators allow us to define new fixpoint algorithms more easily and to fine-tune existing fixpoint algorithms more easily by adding, replacing, or reordering combinators. To summarize, our fixpoint combinators reduce the complexity of the soundness proof, because each combinator can be proven sound independently. Furthermore, they reduce the effort of the soundness proof, because the language-independent combinators can be proven sound a priori as part of a library.

We evaluate our approach by developing the open-source Haskell library *Sturdy*,[1] a library of 22 reusable analysis components and 14 fixpoint combinators. This library demonstrates how our approach allows us to split common analysis functionality into smaller reusable pieces. We use this library to develop several analyses for different languages, which can be found in the Sturdy code repository. Table 9.1 shows which analyses use analysis components and fixpoint combinators from the Sturdy library. This table demonstrates that analyses for different languages require a high amount of customization, since each column only shares some check marks with other columns. However, there is also a potential for reusing common analysis functionality, which can be seen when there multiple check marks in a row. From this table, we conclude that (1) analysis functionality can be split into smaller components and (2) many of these components are language-independent and reusable. Furthermore, this reduces the complexity of the soundness proof, as components are smaller and easier to prove sound. Lastly, the soundness proof takes

[1] https://github.com/svenkeidel/sturdy

	While	PCF	Scheme	Stratego
Values				
$\widehat{\mathtt{Interval}}$	✓	✓		
$\widehat{\mathtt{Boolean}}$	✓		✓	
$\widehat{\mathtt{Closure}}$		✓	✓	
$\widehat{\mathtt{Term}}$				✓
$\widehat{\mathtt{Powerset}}$			✓	
Types			✓	
Effects				
$\widehat{\mathtt{Env}}$	✓		✓	✓
$\widehat{\mathtt{FiniteEnv}}$		✓		
$\widehat{\mathtt{Store}}$	✓			
$\widehat{\mathtt{MonotoneStore}}$			✓	
$\widehat{\mathtt{Exception}}$	✓			✓
$\widehat{\mathtt{PropagateError}}$	✓	✓		✓
$\widehat{\mathtt{LogError}}$			✓	
$\widehat{\mathtt{Terminating}}$	✓	✓	✓	✓
Fixpoint Data				
$\widehat{\mathtt{SCC}}$	✓	✓	✓	✓
$\widehat{\mathtt{Stack}}$	✓	✓	✓	✓
$\widehat{\mathtt{Cache}}$	✓			✓
$\widehat{\mathtt{MonotoneCache}}$		✓	✓	
$\widehat{\mathtt{CallCount}}$	✓			
$\widehat{\mathtt{CallContext}}$		✓	✓	
Fixpoint Combinators				
$\varphi_{\mathtt{callsiteSensitivity}}$		✓		
$\varphi_{\mathtt{loopUnroll}}$	✓			
$\varphi_{\mathtt{stackReuse}}$				✓
$\varphi_{\mathtt{filter}}$	✓	✓	✓	✓
$\varphi_{\mathtt{innermost}}$	✓	✓	✓	✓
$\varphi_{\mathtt{outermost}}$			✓	

Table 9.1: Reuse of analysis components across different languages and analyses that appeared as case studies throughout this thesis.

less effort, because the language-independent components can be proven sound a priori as part of the library.

With our evaluation, we confirm the central thesis of this dissertation:

> Compositional soundness proofs are feasible and reduce the effort and complexity of developing sound static analyses.

BIBLIOGRAPHY

Harold Abelson, R. Kent Dybvig, Christopher T. Haynes, Guillermo Juan Rozas, N. I. Adams IV, Daniel P. Friedman, Eugene E. Kohlbecker, Guy L. Steele Jr., David H. Bartley, Robert H. Halstead Jr., Don Oxley, Gerald J. Sussman, G. Brooks, Chris Hanson, Kent M. Pitman, and Mitchell Wand. 1998. Revised Report on the Algorithmic Language Scheme. *High. Order Symb. Comput.* 11, 1 (1998), 7–105. https://doi.org/10.1023/A:1010051815785

Samson Abramsky. 1994. *Handbook of logic in computer science.* Clarendon Press. http://www.worldcat.org/oclc/312138578

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley. https://www.worldcat.org/oclc/12285707

Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification - The KeY Book - From Theory to Practice.* Lecture Notes in Computer Science, Vol. 10001. Springer. https://doi.org/10.1007/978-3-319-49812-6

Ahmad Salim Al-Sibahi, Thomas P. Jensen, Aleksandar S. Dimovski, and Andrzej Wasowski. 2018. Verification of high-level transformations with inductive refinement types. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2018, Boston, MA, USA, November 5-6, 2018*, Eric Van Wyk and Tiark Rompf (Eds.). ACM, 147–160. https://doi.org/10.1145/3278122.3278125

Md. Imran Alam, Raju Halder, Harshita Goswami, and Jorge Sousa Pinto. 2018. K-Taint: An Executable Rewriting Logic Semantics for Taint Analysis in the K Framework. In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2018, Funchal, Madeira, Portugal, March 23-24, 2018*, Ernesto Damiani, George Spanoudakis, and Leszek A. Maciaszek (Eds.). SciTePress, 359–366. https://doi.org/10.5220/0006786603590366

Karim Ali and Cristina Cifuentes (Eds.). 2017. *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017, Barcelona, Spain, June 18, 2017.* ACM. https://doi.org/10.1145/3088515

Gianluca Amato, Francesca Scozzari, Helmut Seidl, Kalmer Apinis, and Vesal Vojdani. 2016. Efficiently intertwining widening and narrowing. *Sci. Comput. Program.* 120 (2016), 1–24. https://doi.org/10.1016/j.scico.2015.12.005

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.), Vol. 9600. Springer, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

Nada Amin, Adriaan Moors, and Martin Odersky. 2012. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages.*

Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of path-dependent types. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 233–249. https://doi.org/10.1145/2660193.2660216

Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. 2017. Systematic approaches for increasing soundness and precision of static analyzers, See [Ali and Cifuentes 2017], 31–36. https://doi.org/10.1145/3088515.3088521

Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting doop to Soufflé: a tale of inter-engine portability for Datalog-based analyses, See [Ali and Cifuentes 2017], 25–30. https://doi.org/10.1145/3088515.3088522

Robert Atkey. 2012. Relational Parametricity for Higher Kinds. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France (LIPIcs)*, Patrick Cégielski and Arnaud Durand (Eds.), Vol. 16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 46–61. https://doi.org/10.4230/LIPIcs.CSL.2012.46

Pavel Avgustinov, Elnar Hajiyev, Neil Ongkingco, Oege de Moor, Damien Sereni, Julian Tibble, and Mathieu Verbaere. 2007. Semantics of static pointcuts in aspectJ. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 11–23. https://doi.org/10.1145/1190216.1190221

Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (2008), 22–29. https://doi.org/10.1109/MS.2008.130

Anya Helene Bagge and Karl Trygve Kalleberg. 2006. DSAL = library+ notation: Program transformation for domain-specific aspect languages. In *Proc. Domain-Specific Asp. Lang. Work.*

Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. 2007. Tom: Piggybacking Rewriting on Java. In *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings (Lecture Notes in Computer Science)*, Franz Baader (Ed.), Vol. 4533. Springer, 36–47. https://doi.org/10.1007/978-3-540-73449-9_5

Anindya Banerjee, Olivier Danvy, Kyung-Goo Doh, and John Hatcliff (Eds.). 2013. *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013.* EPTCS, Vol. 129. https://doi.org/10.4204/EPTCS.129

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 14–25. https://doi.org/10.1145/964001.964003

Yves Bertot. 2008. Structural Abstract Interpretation: A Formal Study Using Coq. In *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures (Lecture Notes in Computer Science)*, Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto (Eds.), Vol. 5520. Springer, 153–194. https://doi.org/10.1007/978-3-642-03153-3_4

Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75. https://doi.org/10.1145/1646353.1646374

Martin Bodin, Philippa Gardner, Thomas P. Jensen, and Alan Schmitt. 2019. Skeletal semantics and their interpretations. *Proc. ACM Program. Lang.* 3, POPL (2019), 44:1–44:31. https://doi.org/10.1145/3290357

Martin Bodin, Thomas P. Jensen, and Alan Schmitt. 2015. Certified Abstract Interpretation with Pretty-Big-Step Semantics. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, Xavier Leroy and Alwen Tiu (Eds.). ACM, 29–40. https://doi.org/10.1145/2676724.2693174

François Bourdoncle. 1993. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications, International Conference, Akademgorodok, Novosibirsk, Russia, June 28 - July 2, 1993, Proceedings (Lecture Notes in Computer Science)*, Dines Bjørner, Manfred Broy, and Igor V. Pottosin (Eds.), Vol. 735. Springer, 128–141. https://doi.org/10.1007/BFb0039704

Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. 2006. Program Transformation with Scoped Dynamic Rewrite Rules. *Fundam. Inform.* 69, 1-2 (2006), 123–178. http://content.iospress.com/articles/fundamenta-informaticae/fi69-1-2-06

Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings (Lecture Notes in Computer Science)*, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.), Vol. 9058. Springer, 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. 1986. Interprocedural constant propagation. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, Palo Alto, California, USA, June 25-27, 1986*, Richard L. Wexelblat (Ed.). ACM, 152–161. https://doi.org/10.1145/12276.13327

Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. 2017. HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving. *CoRR* abs/1707.07866 (2017). arXiv:1707.07866 http://arxiv.org/abs/1707.07866

Joana Campos and Vasco T. Vasconcelos. 2018. Dependent Types for Class-based Mutable Objects, See [Millstein 2018], 13:1–13:28. https://doi.org/10.4230/LIPIcs.ECOOP.2018.13

Luca Cardelli. 1996. Type Systems. *ACM Comput. Surv.* 28, 1 (1996), 263–264. https://doi.org/10.1145/234313.234418

Arthur Charguéraud. 2013. Pretty-Big-Step Semantics. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, 41–60. https://doi.org/10.1007/978-3-642-37036-6_3

Liqian Chen, Antoine Miné, and Patrick Cousot. 2008. A Sound Floating-Point Polyhedra Abstract Domain. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings (Lecture Notes in Computer Science)*, G. Ramalingam (Ed.), Vol. 5356. Springer, 3–18. https://doi.org/10.1007/978-3-540-89330-1_2

Yih-Farn Chen, Emden R. Gansner, and Eleftherios Koutsofios. 1997. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In *Software Engineering - ESEC/FSE '97, 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering, Zurich, Switzerland, September 22-25, 1997, Proceedings (Lecture Notes in Computer Science)*, Mehdi Jazayeri and Helmut Schauer (Eds.), Vol. 1301. Springer, 414–431. https://doi.org/10.1007/3-540-63531-9_28

Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. 2002. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.* 285, 2 (2002), 187–243. https://doi.org/10.1016/S0304-3975(01)00359-0

Coq Development Team. 2019. *The Coq Proof Assistant Reference Manual*. Available at https://coq.inria.fr/doc/.

Thierry Coquand and Christine Paulin. 1988. Inductively defined types. In *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings (Lecture Notes in Computer Science)*, Per Martin-Löf and Grigori Mints (Eds.), Vol. 417. Springer, 50–66. https://doi.org/10.1007/3-540-52335-9_47

Agostino Cortesi, Giulia Costantini, and Pietro Ferrara. 2013. A Survey on Product Operators in Abstract Interpretation, See [Banerjee et al. 2013], 325–336. https://doi.org/10.4204/EPTCS.129.19

P Cousot. 1999. The Calculational Design of a Generic Abstract Interpreter. In *Calc. Syst. Des.*, M Broy and R Steinbrüggen (Eds.). NATO ASI Series F. IOS Press, Amsterdam.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen (Eds.). ACM Press, 269–282. https://doi.org/10.1145/567752.567778

Patrick Cousot and Radhia Cousot. 1992a. Abstract Interpretation Frameworks. *J. Log. Comput.* 2, 4 (1992), 511–547. https://doi.org/10.1093/logcom/2.4.511

Patrick Cousot and Radhia Cousot. 1992b. Comparing the Galois Connection and Widening/-Narrowing Approaches to Abstract Interpretation. In *Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Leuven, Belgium, August 26-28, 1992, Proceedings (Lecture Notes in Computer Science)*, Maurice Bruynooghe and Martin Wirsing (Eds.), Vol. 631. Springer, 269–295. https://doi.org/10.1007/3-540-55844-6_142

Patrick Cousot and Radhia Cousot. 1995. Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, John Williams (Ed.). ACM, 170–181. https://doi.org/10.1145/224164.224199

Patrick Cousot and Radhia Cousot. 2002. Systematic design of program transformation frameworks by abstract interpretation, See [Launchbury and Mitchell 2002], 178–190. https://doi.org/10.1145/503272.503290

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2006. Combination of Abstractions in the ASTRÉE Static Analyzer. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers (Lecture Notes in Computer Science)*, Mitsu Okada and Ichiro Satoh (Eds.), Vol. 4435. Springer, 272–300. https://doi.org/10.1007/978-3-540-77505-8_23

Ron K. Cytron and Peter Lee (Eds.). 1995. *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*. ACM Press. http://dl.acm.org/citation.cfm?id=199448

David Darais and David Van Horn. 2019. Constructive Galois Connections. *J. Funct. Program.* 29 (2019), e11. https://doi.org/10.1017/S0956796819000066

David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *Proc. ACM Program. Lang.* 1, ICFP (2017), 12:1–12:25. https://doi.org/10.1145/3110256

David Darais, Matthew Might, and David Van Horn. 2015. Galois transformers and modular abstract interpreters: reusable metatheory for program analysis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 552–571. https://doi.org/10.1145/2814270.2814308

David Charles Darais. 2017. *Mechanizing Abstract Interpretation*. Ph.D. Dissertation. University of Maryland, College Park, MD, {USA}. https://doi.org/10.13016/M2J96097D

Maartje de Jonge and Eelco Visser. 2012. A language generic solution for name binding preservation in refactorings. In *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*, Anthony Sloane and Suzana Andova (Eds.). ACM, 2. https://doi.org/10.1145/2427048.2427050

Jens de Waard. 2020. Soundness Proofs of Static Analyses in Coq.

Edsgar Dijkstra. 1969. *Notes on structured programming*. Technische Hogeschool Eindhoven.

Eelco Dolstra and Eelco Visser. 2002. Building Interpreters with Rewriting Strategies. *Electron. Notes Theor. Comput. Sci.* 65, 3 (2002), 57–76. https://doi.org/10.1016/S1571-0661(04)80427-4

Catherine Dubois. 2000. Proving ML Type Soundness Within Coq. In *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings (Lecture Notes in Computer Science)*, Mark Aagaard and John Harrison (Eds.), Vol. 1869. Springer, 126–144. https://doi.org/10.1007/3-540-44659-1_9

Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. 2014. The Matter of Heartbleed. In *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*, Carey Williamson, Aditya Akella, and Nina Taft (Eds.). ACM, 475–488. https://doi.org/10.1145/2663716.2663755

Giorgios Rob Economopoulos and Bernd Fischer. 2011. Higher-order transformations with nested concrete syntax. In *Language Descriptions, Tools and Applications, LDTA 2011, Saarbrücken, Germany, March 26-27, 2011. Proceeding*, Claus Brabrand and Eric Van Wyk (Eds.). ACM, 4. https://doi.org/10.1145/1988783.1988787

Pär Emanuelsson and Ulf Nilsson. 2008. A Comparative Study of Industrial Static Analysis Tools. *Electron. Notes Theor. Comput. Sci.* 217 (2008), 5–21. https://doi.org/10.1016/j.entcs.2008.06.039

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: library-based syntactic language extensibility. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 391–406. https://doi.org/10.1145/2048066.2048099

Sebastian Erdweg, Vlad A. Vergu, Mira Mezini, and Eelco Visser. 2014. Modular specification and dynamic enforcement of syntactic language constraints when generating code. In *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld (Eds.). ACM, 241–252. https://doi.org/10.1145/2577080.2577089

Noah Van Es, Quentin Stiévenart, and Coen De Roover. 2019. Garbage-Free Abstract Interpretation Through Abstract Reference Counting. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom (LIPIcs)*, Alastair F. Donaldson (Ed.), Vol. 134. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:33. https://doi.org/10.4230/LIPIcs.ECOOP.2019.10

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 234–245. https://doi.org/10.1145/512529.512558

Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. 2018. Static Value Analysis of Python Programs by Abstract Interpretation. In *NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings (Lecture Notes in Computer Science)*, Aaron Dutle, César A. Muñoz, and Anthony Narkawicz (Eds.), Vol. 10811. Springer, 185–202. https://doi.org/10.1007/978-3-319-77935-5_14

Richard P Gabriel. 1985. *Performance and Evaluation of LISP Systems.* Massachusetts Institute of Technology, USA.

Alfons Geser, Jens Knoop, Gerald Lüttgen, Bernhard Steffen, and Oliver Ruthing. 1994. Chaotic fixed point iterations.

Neil Ghani, Patricia Johann, Fredrik Nordvall Forsberg, Federico Orsanigo, and Tim Revell. 2015. Bifibrational Functorial Semantics of Parametric Polymorphism. In *The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015 (Electronic Notes in Theoretical Computer Science)*, Dan R. Ghica (Ed.), Vol. 319. Elsevier, 165–181. https://doi.org/10.1016/j.entcs.2015.12.011

Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. 2020. Scala step-by-step: soundness for DOT with step-indexed logical relations in Iris. *Proc. ACM Program. Lang.* 4, ICFP (2020), 114:1–114:29. https://doi.org/10.1145/3408996

Neville Grech and Yannis Smaragdakis. 2017. P/Taint: unified points-to and taint analysis. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 102:1–102:28. https://doi.org/10.1145/3133926

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. 1996. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.* 18, 2 (1996), 109–138. https://doi.org/10.1145/227699.227700

Makoto Hamana and Marcelo P. Fiore. 2011. A foundation for GADTs and inductive families: dependent polynomial functor approach. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Jaakko Järvi and Shin-Cheng Mu (Eds.). ACM, 59–70. https://doi.org/10.1145/2036918.2036927

Michael Hind and Anthony Pioli. 1998. Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In *Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings (Lecture Notes in Computer Science)*, Giorgio Levi (Ed.), Vol. 1503. Springer, 57–81. https://doi.org/10.1007/3-540-49727-7_4

David Van Horn and Matthew Might. 2010. Abstracting abstract machines. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 51–62. https://doi.org/10.1145/1863543.1863553

David Hovemeyer, Jaime Spacco, and William Pugh. 2005. Evaluating and tuning a static analysis to find null pointer bugs. In *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05, Lisbon, Portugal, September 5-6, 2005*, Michael D. Ernst and Thomas P. Jensen (Eds.). ACM, 13–19. https://doi.org/10.1145/1108792.1108798

John Hughes. 2000. Generalising monads to arrows. *Sci. Comput. Program.* 37, 1-3 (2000), 67–111. https://doi.org/10.1016/S0167-6423(99)00023-4

Alex Jeffery. 2019. Dependent object types with implicit functions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala, Scala@ECOOP 2019, London, UK, July 17, 2019*, Jonathan Immanuel Brachthäuser, Sukyoung Ryu, and Nathaniel Nystrom (Eds.). ACM, 1–11. https://doi.org/10.1145/3337932.3338811

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings (Lecture Notes in Computer Science)*, Jens Palsberg and Zhendong Su (Eds.), Vol. 5673. Springer, 238–255. https://doi.org/10.1007/978-3-642-03237-0_17

N Jones and Flemming Nielson. 1994. Abstract interpretation: a semantics-based tool for program analysis. *Handb. Log. Comput. Sci.* 4 (1994), 527–636.

Simon L. Peyton Jones and Ralf Lämmel. 2003. Scrap Your Boilerplate. In *Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003, Proceedings (Lecture Notes in Computer Science)*, Atsushi Ohori (Ed.), Vol. 2895. Springer, 357. https://doi.org/10.1007/978-3-540-40018-9_23

Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 247–259. https://doi.org/10.1145/2676726.2676966

Aditya Kanade, Amitabha Sanyal, and Uday P. Khedker. 2007. Structuring Optimizing Transformations and Proving Them Sound. *Electron. Notes Theor. Comput. Sci.* 176, 3 (2007), 79–95. https://doi.org/10.1016/j.entcs.2006.06.018

Sven Keidel and Sebastian Erdweg. 2017. Toward abstract interpretation of program transformations. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection, META@SPLASH 2017, Vancouver, BC, Canada, October 23 - 27, 2017*, Shigeru Chiba, Elisa Gonzalez Boix, and Stefan Marr (Eds.). ACM, 1–5. https://doi.org/10.1145/3141517.3141855

Sven Keidel and Sebastian Erdweg. 2019. Sound and reusable components for abstract interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 176:1–176:28. https://doi.org/10.1145/3360602

Sven Keidel and Sebastian Erdweg. 2020. A Systematic Approach to Abstract Interpretation of Program Transformations. In *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings (Lecture Notes in Computer Science)*, Dirk Beyer and Damien Zufferey (Eds.), Vol. 11990. Springer, 136–157. https://doi.org/10.1007/978-3-030-39322-9_7

Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. 2018. Compositional soundness proofs of abstract interpreters. *Proc. ACM Program. Lang.* 2, ICFP (2018), 72:1–72:26. https://doi.org/10.1145/3236767

Sung Kook Kim, Arnaud J. Venet, and Aditya V. Thakur. 2020. Deterministic parallel fixpoint computation. *Proc. ACM Program. Lang.* 4, POPL (2020), 14:1–14:33. https://doi.org/10.1145/3371082

Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 239–250. https://doi.org/10.1145/3293882.3330553

Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*. IEEE Computer Society, 168–177. https://doi.org/10.1109/SCAM.2009.28

Jens Knoop and Oliver Rüthing. 1999. Optimization Under the Perspective of Soundness, Completeness, and Reusability. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel) (Lecture Notes in Computer Science)*, Ernst-Rüdiger Olderog and Bernhard Steffen (Eds.), Vol. 1710. Springer, 288–315. https://doi.org/10.1007/3-540-48092-7_13

Ralf Lämmel. 2003. Typed generic traversal with term rewriting strategies. *J. Log. Algebraic Methods Program.* 54, 1-2 (2003), 1–64. https://doi.org/10.1016/S1567-8326(02)00028-0

Saunders Mac Lane. 1971. *Categories for the Working Mathematician*. Graduate Texts in Mathematics, Vol. 5. Springer New York, New York, NY. https://doi.org/10.1007/978-1-4612-9839-7

John Launchbury and John C. Mitchell (Eds.). 2002. *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*. ACM. http://dl.acm.org/citation.cfm?id=503272

Sorin Lerner, David Grove, and Craig Chambers. 2002. Composing dataflow analyses and transformations, See [Launchbury and Mitchell 2002], 270–282. https://doi.org/10.1145/503272.503298

Sorin Lerner, Todd D. Millstein, and Craig Chambers. 2003. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, Ron Cytron and Rajiv Gupta (Eds.). ACM, 220–231. https://doi.org/10.1145/781131.781156

Sorin Lerner, Todd D. Millstein, Erika Rice, and Craig Chambers. 2005. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 364–377. https://doi.org/10.1145/1040305.1040335

Xavier Leroy and Others. 2012. The CompCert verified compiler. *Doc. user's manual. INRIA Paris-Rocquencourt* 53 (2012).

Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters, See [Cytron and Lee 1995], 333–343. https://doi.org/10.1145/199448.199528

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundness: a manifesto. *Commun. ACM* 58, 2 (2015), 44–46. https://doi.org/10.1145/2644805

Magnus Madsen and Ondrej Lhoták. 2018. Safe and sound program analysis with Flix. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 38–48. https://doi.org/10.1145/3213846.3213847

Simon Marlow. 2010. *Haskell 2010 language report.* Technical Report. https://www.haskell.org/onlinereport/haskell2010/

Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. 2004. A Visual Environment for Developing Context-Sensitive Term Rewriting Systems. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings (Lecture Notes in Computer Science)*, Vincent van Oostrom (Ed.), Vol. 3091. Springer, 301–311. https://doi.org/10.1007/978-3-540-25979-4_21

Laurent Mauborgne and Xavier Rival. 2005. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science)*, Shmuel Sagiv (Ed.), Vol. 3444. Springer, 5–20. https://doi.org/10.1007/978-3-540-31987-0_2

Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. 2012. Eval begone!: semi-automated removal of eval from javascript programs. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 607–620. https://doi.org/10.1145/2384616.2384660

Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. *IEEE Trans. Software Eng.* 30, 2 (2004), 126–139. https://doi.org/10.1109/TSE.2004.1265817

Jan Midtgaard. 2012. Control-flow analysis of functional programs. *ACM Comput. Surv.* 44, 3 (2012), 10:1–10:33. https://doi.org/10.1145/2187671.2187672

Matthew Might and Olin Shivers. 2006a. Environment analysis via Delta CFA. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 127–140. https://doi.org/10.1145/1111037.1111049

Matthew Might and Olin Shivers. 2006b. Improving flow analyses via GammaCFA: abstract garbage collection and counting. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, John H. Reppy and Julia L. Lawall (Eds.). ACM, 13–25. https://doi.org/10.1145/1159803.1159807

Todd D. Millstein (Ed.). 2018. *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands.* LIPIcs, Vol. 109. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. http://www.dagstuhl.de/dagpub/978-3-95977-079-8

John C. Mitchell. 1990. Type Systems for Programming Languages. See [van Leeuwen 1990], 365–458. https://doi.org/10.1016/b978-0-444-88074-1.50013-5

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis.* Springer. https://doi.org/10.1007/978-3-662-03811-6

Oystein Ore. 1944. Galois connexions. *Trans. Am. Math. Soc.* 55, 3 (1944), 493–513.

David Lorge Parnas. 1972. On the Criteria To Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (1972), 1053–1058. https://doi.org/10.1145/361598.361623

Ross Paterson. 2001. A New Notation for Arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, Benjamin C. Pierce (Ed.). ACM, 229–240. https://doi.org/10.1145/507635.507664

Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press.

Gordon D. Plotkin. 1977. LCF Considered as a Programming Language. *Theor. Comput. Sci.* 5, 3 (1977), 223–255. https://doi.org/10.1016/0304-3975(77)90044-5

Gordon D Plotkin. 1980. Lambda-definability in the full type hierarchy. *To HB Curry Essays Comb. Logic, Lambda Calc. Formalism* (1980), 363–373.

Marianna Rapoport and Ondrej Lhoták. 2019. A path to DOT: formalizing fully path-dependent types. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 145:1–145:29. https://doi.org/10.1145/3360571

Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability, See [Cytron and Lee 1995], 49–61. https://doi.org/10.1145/199448.199462

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, R. E. A. Mason (Ed.). North-Holland/IFIP, 513–523.

John C. Reynolds. 1998. Definitional Interpreters for Higher-Order Programming Languages. *High. Order Symb. Comput.* 11, 4 (1998), 363–397. https://doi.org/10.1023/A:1010027404223

Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 624–641. https://doi.org/10.1145/2983990.2984008

Mads Rosendahl. 1995. Introduction to abstract interpretation. *Computer Science University of Copenhagen* (1995).

Mads Rosendahl. 2013. Abstract Interpretation as a Programming Language, See [Banerjee et al. 2013], 84–104. https://doi.org/10.4204/EPTCS.129.7

Andreas Rossberg, Ben L. Titzer, Andreas Haas, Derek L. Schuff, Dan Gohman, Luke Wagner, Alon Zakai, J. F. Bastien, and Michael Holman. 2018. Bringing the web up to speed with WebAssembly. *Commun. ACM* 61, 12 (2018), 107–115. https://doi.org/10.1145/3282510

Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010.* IEEE Computer Society, 186–199. https://doi.org/10.1109/CSF.2010.20

Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. 2004. A Comparison of Bug Finding Tools for Java. In *15th International Symposium on Software Reliability Engineering (ISSRE 2004), 2-5 November 2004, Saint-Malo, Bretagne, France.* IEEE Computer Society, 245–256. https://doi.org/10.1109/ISSRE.2004.1

Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 598–608. https://doi.org/10.1109/ICSE.2015.76

Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.* 167, 1&2 (1996), 131–170. https://doi.org/10.1016/0304-3975(96)00072-2

Alexandru Salcianu and Martin C. Rinard. 2005. Purity and Side Effect Analysis for Java Programs. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings (Lecture Notes in Computer Science)*, Radhia Cousot (Ed.), Vol. 3385. Springer, 199–215. https://doi.org/10.1007/978-3-540-30579-8_14

David A. Schmidt. 1995. Natural-Semantics-Based Abstract Interpretation (Preliminary Version). In *Static Analysis, Second International Symposium, SAS'95, Glasgow, UK, September 25-27, 1995, Proceedings (Lecture Notes in Computer Science)*, Alan Mycroft (Ed.), Vol. 983. Springer, 1–18. https://doi.org/10.1007/3-540-60360-3_28

David A. Schmidt. 1996. Abstract Interpretation of Small-Step Semantics. In *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop, Stockholm, Sweden, June 24-26, 1996, Selected Papers (Lecture Notes in Computer Science)*, Mads Dam (Ed.), Vol. 1192. Springer, 76–99. https://doi.org/10.1007/3-540-62503-8_4

David A. Schmidt. 1998. Trace-Based Abstract Interpretation of Operational Semantics. *LISP Symb. Comput.* 10, 3 (1998), 237–271.

Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. *CoRR* abs/2005.06227 (2020). arXiv:2005.06227 https://arxiv.org/abs/2005.06227

Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. 2013. Monadic abstract interpreters. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 399–410. https://doi.org/10.1145/2491956.2491979

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Tom Ridge, Susmit Sarkar, and Rok Strnisa. 2007. Ott: effective tool support for the working semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, Ralf Hinze and Norman Ramsey (Eds.). ACM, 1–12. https://doi.org/10.1145/1291151.1291155

Olin Shivers. 1991. *Control-flow analysis of higher-order languages.* Ph.D. Dissertation. Carnegie Mellon University.

Warclaw Sierpinski. 1915. Sur une courbe dont tout point est un point de ramification. *CR Acad. Sci.* 160 (1915), 302–305.

Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2017. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 46–59. https://doi.org/10.1145/3009837

Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (2015), 1–69. https://doi.org/10.1561/2500000014

Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More Sound Static Handling of Java Reflection. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings (Lecture Notes in Computer Science)*, Xinyu Feng and Sungwoo Park (Eds.), Vol. 9458. Springer, 485–503. https://doi.org/10.1007/978-3-319-26529-2_26

Yannis Smaragdakis and George Kastrinis. 2018. Defensive Points-To Analysis: Effective Soundness via Laziness, See [Millstein 2018], 23:1–23:28. https://doi.org/10.4230/LIPIcs.ECOOP.2018.23

Jeff Smits and Eelco Visser. 2017. FlowSpec: declarative dataflow analysis specification. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, Benoît Combemale, Marjan Mernik, and Bernhard Rumpe (Eds.). ACM, 221–231. https://doi.org/10.1145/3136014.3136029

Quentin Stiévenart, Maarten Vandercammen, Wolfgang De Meuter, and Coen De Roover. 2016. Scala-AM: A Modular Static Analysis Framework. In *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016*. IEEE Computer Society, 85–90. https://doi.org/10.1109/SCAM.2016.14

Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 139:1–139:29. https://doi.org/10.1145/3276509

Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: a DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 320–331. https://doi.org/10.1145/2970276.2970298

Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5, 2 (1955), 285–309.

Amin Timany and Bart Jacobs. 2016. Category Theory in Coq 8.5. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal (LIPIcs)*, Delia Kesner and Brigitte Pientka (Eds.), Vol. 52. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:18. https://doi.org/10.4230/LIPIcs.FSCD.2016.30

Raja Vallee-Rai and Laurie J Hendren. 1998. Jimple: Simplifying Java bytecode for analyses and transformations. (1998).

Elmer van Chastelet, Eelco Visser, and Craig Anslow. 2015. Conf.Researchr.Org: towards a domain-specific content management system for managing large conference websites. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 50–51. https://doi.org/10.1145/2814189.2817270

Jan van Leeuwen (Ed.). 1990. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. In van Leeuwen [van Leeuwen 1990]. https://www.sciencedirect.com/book/9780444880741/formal-models-and-semantics

Arnaud Venet. 1996. Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs. In *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings (Lecture Notes in Computer Science)*, Radhia Cousot and David A. Schmidt (Eds.), Vol. 1145. Springer, 366–382. https://doi.org/10.1007/3-540-61739-6_53

Eelco Visser. 2007. WebDSL: A Case Study in Domain-Specific Language Engineering. In *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers (Lecture Notes in Computer Science)*, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.), Vol. 5235. Springer, 291–373. https://doi.org/10.1007/978-3-540-88643-3_7

Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, 13–26. https://doi.org/10.1145/289423.289425

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, Joseph E. Stoy (Ed.). ACM, 347–359. https://doi.org/10.1145/99370.99404

Philip Wadler. 1992. Monads for functional programming. In *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi, Marktoberdorf, Germany, July 28 - August 9, 1992 (NATO ASI Series)*, Manfred Broy (Ed.), Vol. 118. Springer, 233–264. https://doi.org/10.1007/978-3-662-02880-3_8

Yan Wang, Hailong Zhang, and Atanas Rountev. 2016. On the unsoundness of static analysis for Android GUIs. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2016, Santa Barbara, CA, USA, June 14, 2016*, Charles Zhang and Xavier Rival (Eds.). ACM, 18–23. https://doi.org/10.1145/2931021.2931026

Gary Wassermann and Zhendong Su. 2007. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 32–41. https://doi.org/10.1145/1250734.1250739

Fadi Wedyan, Dalal Alrmuny, and James M. Bieman. 2009. The Effectiveness of Automated Static Analysis Tools for Fault Detection and Refactoring Prediction. In *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009*. IEEE Computer Society, 141–150. https://doi.org/10.1109/ICST.2009.21

Guannan Wei, Yuxuan Chen, and Tiark Rompf. 2019. Staged abstract interpreters: fast and modular whole-program analysis via meta-programming. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 126:1–126:32. https://doi.org/10.1145/3360552

David A. Wheeler. 2014. Preventing Heartbleed. *IEEE Computer* 47, 8 (2014), 80–83. https://doi.org/10.1109/MC.2014.217

Yichen Xie, Andy Chou, and Dawson R. Engler. 2003. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003*, Jukka Paakki and Paola Inverardi (Eds.). ACM, 327–336. https://doi.org/10.1145/940071.940115