Developing a Framework for Extensible Static Analysis

Abstract

This paper seeks to explore and develop a fully extensible, language- and domain-agnostic abstract interpreter framework without the need for any code revision. We present snippets of this framework and discuss its potential applications and limitations. Most pre-existing tools are limited in their use cases and only work for one or two languages. Additionally research in the area is stymied by the need to reinvent the wheel with each project - creating a new abstract interpreter from scratch. As a result, there exists a multitude of smaller, very functional but practically useless interpreters. If there was a way to use this pre-existing research in combination with other research easily, these tools could become far more useful. We explore a framework that allows a programmer to add new features as well as new domains, all while requiring no revision to previous code. By leveraging a monadic type class system we seek to create a framework where pre-existing domains can be integrated with new features easily. A monadic structure allows us to extend the features of the language indefinitely, while ensuring proper chaining and evaluation of the features. Type classes allow us to define different behavior depending on domain for each feature. More importantly, with the combination of these two we can still maintain type safety. While this framework is still in its early stages, we believe it has great potential to aid research in static analysis. Ultimately, however, more research is need to determine if this framework will be viable in practice, and more importantly useful to the community.

Keywords

Static Analysis, Abstract Interpretation, Extensible, Modular

1 Introduction

Static Analysis is an integral part of the development process, ensuring that programs are bug-free, reliable, and secure. These tools are used in various fields including software development and cybersecurity; they analyze code before it is run, checking for potential errors and vulnerabilities. One of the problems plaguing new research into static analysis is the need to develop new abstract interpreters from scratch repeatedly. Each of these new interpreters is built with a narrow focus, tailored to a specific language and task. This results in a large redundancy across projects that is difficult to reduce. Not only does this increase the time and cost of development, but it also limits the reusability of projects in the future. Consequently, there exists a fragmented ecosystem with a plethora of small, hyperfunctional interpreters that will never be used again. Every project requires reinvention. The lack of a unified, flexible framework restricts the ability to apply powerful techniques across different languages or with new domains without significant effort like is required with the current state-of-the-art: MOPSA [6].

The introduction of a general-purpose abstract interpreter that can be adapted for any new task would be revolutionary. This interpreter would invigorate and expedite research into static analyzers. A unified, extensible framework would reduce both the cost and the time of development of research. Additionally, most modern codebases have a mix of languages in them, and there exists few methods of static analysis for this. Such a framework would also streamline the use and application of new abstract domains in existing projects by allowing use of already existing domain libraries and functionalities.

In this paper we refer to extensibility along two axes: features, and domains. Feature extensibility allows one to add new language features, e.g. operations, variables, lambda functions. Domain extensibility allows for analysis along multiple abstract domains, e.g. Interval, Parity, Sign. Extensibility along one of these axes is trivial, and depending on the language can be accomplished in any number of ways. However, when you wish to be extensible along both of these axes it becomes more difficult. This is an instance of the famous expression problem [7]. Creating an interpreter across languages and domains is challenging due to the difference in paradigms, syntax, and semantics of languages as well as the differences in how domains are analyzed. Furthermore, achieving generality without becoming burdeningly complicated and contrived becomes difficult as the complexity of the features and domains increase. Our framework addresses these challenges by separating the domain from the language, allowing there to be two, non-conflicting axes that can easily be extended independently of each other.

This paper aims to address this need by developing a fully extensible, language- and domain-agnostic abstract interpreter framework. To do this, we leverage a monadic structure along with typeclass polymorphism to modularize and isolate evaluation logic. The goal is to develop a uniform method of creating future interpreters such that they can be reused, modified, and combined to propel research further. To accomplish this we will not delve into any specific static analysis techniques, but will keep it to shorter, simpler examples.

The remainder of this paper is organized as follows: In the remainder of this section we present our overarching problem statements. Section 2 and describes our motivation and approach for the monadic and typeclass architecture for simple through complex features. Section 3 discusses the limitations and assumptions of our framework. Section 4 situates our work in the context of previous work.

1.1 Problem Statement

Abstract Interpreters are typically built with a narrow focus, tailored to specific tasks and languages. This fragmentation limits their reusability, slows research, and inhibits multi-language analysis. Our goal is to create a framework that overcomes these limitations. Specifically, we aim to:

- Develop an interpreter framework that is agnostic to language and domain.
- Ensure new features and new domains can be analyzed *without* ever revising any code.

2 Overview

To begin with, let us simplify the combination of two languages as the combination of two or three features. This is a simplification,

but it is a reasonable starting point. We will work with three distinct features with escalating complexity: increment, conditionals, and variables. If one made a more standard interpreter for each of these, it would be trivial to structure it such that you can add new features, but changing the domain would be harder. Similarly, if we wanted to build an interpreter for a fixed language across extensible domains, this too would be easy; however, it would be difficult to add new features. Thus arises the need for the two axes of extensibility.

A note on the code snippets included here: some will be truncated or have otherwise modified syntax for conciseness and neatness.

This project builds on Oleg Kiselyov's "Having an Effect" talk [4] in which he defines a feature-extensible expression language. In his design, every computation is treated as either completed or suspended. Any suspended computations are treated as requests to another automata such as a file system or another program. A key part of this framework is that there is no single evaluation function, but rather each computation is simplified as much as can be until resolved by a handler (other automata). By treating everything as an effect, unstable semantics can be avoided and no revision to the core framework is necessary. We seek to leverage this isolation of evaluations to make the language domain-extensible while retaining the same fluid method for adding features. For this paper We will explore features in the Concrete and Interval domains only, but the approach can be extended to most other abstract domains (which is discussed more in section 3). We will also only be working on simple language features because if the framework is modular at the smallest level, then it will be modular at all levels. All of this helps us to consider the framework from a simpler perspective, without getting overwhelmed by complicated implementation details.

Consider the following Scala implementation of Kiselyov's framework:

```
class Comp
case class Done(v:DomC) extends Comp
case class Req(r:ReqType, f:DomC => Comp) ...
```

Note that both Done and Req extends Comp. Our goal here is to make computations parametric on domain:

```
class Comp[D]
case class Done[D](e:D) extends Comp[D]
case class Req[D](r:ReqType, f: D => Comp[D]) ...
```

To accomplish this we can use Scala's type parameter feature which can be used to make most functions generic without too much effort. Where we run into trouble is when we need to define different behavior depending on domain. For example, consider the following increment function over the Concrete and Interval domains:

```
def inc: Comp[DomC]
def inc: Comp[DomI]
```

Due to Scala's type erasure, this method won't work. We also cannot put the two implementations in the same function via pattern matching, as this erodes the extensibility of the language. Somehow, we need a way to define different behavior for different domains without losing the ability to add new domains in the future. A first naïve attempt would be to have the same function for every attempt, requiring class methods inside domains to define the + operator, as shown below:

```
def inc: Comp = Done(Func{v => Done(v + DInt(1))})
```

Theoretically, this method could be scaled for most elementary operations; however, there are two major reasons why this won't work. The first is that the line between "elementary" features and "non-elementary" features is not always clear. Are simple logic statements elementary? What about loops? Are all mathematical operators universally applied across languages? It is difficult to determine where to draw the line. The second reason is that this goes against the ethos of this work; We want to have no implementation in the features or in the domain, but rather in a space between. Thus arises the need for a typeclass. If we declare a generic type class for each language feature, like increment, we can then define different implementations for each domain, relying on Scala's implicit resolution to select the implementation for the chosen domain. Furthermore, we can rely on Scala's type checking to ensure type safety of all the code, avoiding any runtime errors. With the combination of typeclass polymorphism, generic type parameters, and Kiselyov's monadic structure, we can maintain extensibility of features while being able to add new domains with relative ease:

```
trait DomainOps[D]
  def increment(value: D): Comp[D]
implicit object DomCIncrement extends DomainOps[DomC]
  def increment(value: DomC): Comp[DomC]
implicit object DomIIncrement extends DomainOps[DomI]
  def increment(value: DomI): Comp[DomI]
def inc[D](value: D)(implicit ops: DomainOps[D]):
                Comp[D] = ops.increment(value)
```

Adding logical expressions is the next part of the language that we will expand. This is the first point in which the implementation of the two domains is truly different. For example consider the following if statement:

```
if (x<0) then {x=0} else {x=1}
```

In the concrete domain, depending on the value of x only one branch is reachable, and so we can simply evaluate x, choose the correct branch and continue on. In a simple interval domain however, the interval may be split by the conditional, and both branches of the statement are reachable, requiring a union of the two branches in the end. The type class pattern can work for this as well. We can define the concrete domain to work as a simple if statement and comparison operator would, while changing how the two work for the interval domain.

We now look to more complicated features such as higher order requests which are able to express variable bindings and closures. This allows the language to handle lambda functions and variable resolution; While these seem more complicated, they are in fact easier than other language features. This is because for most abstract domains, state is essentially the same. We can extend Kiselyov's implementation of these requests for variables very easily. Consider the following code in which we have a domain ambiguous state, lambda, and variable functions:

```
type Env[D] = List[(String, D)]
def var_[D](varName: String): Comp[D] =
        Req(ReqHO(ReqVar(varName)), Done)
def lam[D](varName: String, body: Comp[D]): Comp[D] =
        Req(ReqHO(ReqClosure(varName, body)), Done)
```

The handler can similarly be extended with minimal effort, which is not shown here. This framework shows great promise for future

extensibility. We have shown that we can add new domains with relative ease, and that we can add new features with minimal effort. However, there are some limitations to this framework that will be discussed in the next section.

3 Discussion

The proposed framework offers significant benefits to abstract interpretation, but it is not without its limitations. First, the range of supported domains is potentially constrained, which restricts the applicability to real-world problems. Second, the system's complexity makes the framework require a large amount of code that could quickly become too large or confusing. Third, the system is not yet fully implemented and so it is difficult to say how well it will work in practice. These limitations underscore areas for future refinement and highlight the trade-offs inherent in designing a highly extensible and powerful analysis tool.

This framework was developed on the idea of numerical abstract domains: Concrete, Interval, Sign, Parity, etc. All of these domains would work flawlessly in the framework, however other classes of domains exists. Relational domains, such as octagonal, polyhedra and affine, store state differently than numerical domains. These domains are not just a direct mapping of variable name to a value of some type and so more research is required to see if they could be used in this framework. Other domains such as the pointer domain, work differently from other types of domains as well, and so pose the same problem as relational domains do.

Type classes, while a powerful and effective solution to the expression problem here are not without their own limitations. The main limitation is that they require a large overhead of boilerplate code to implement. For every new feature that is added to the language, a new typeclass must be defined. This can quickly become cumbersome and difficult to manage. This could be solved by using a language other than Scala (such as Haskell) where they are more powerful and easier to use. However, it may also be reduced as the language features and domains increase, the relative size of the typeclass definitions will decrease. Furthermore, it could be argued that there doesn't need to be the separation of completed and suspended computations. One could simply build an interpreter with the same framework based off of one expression trait that is the monad and type classes for each domain. This could reduce the complexity of the code and more research is needed to see if that method is more viable.

The framework also is not fully implemented, so our knowledge about it is limited. While the framework has been tested on a few simple language features, it is not yet known how well it will work on more complex features. Furthermore, the framework has not been tested on any real-world code and so it is not known how well it will work in practice. This limitation can only be addressed by further research and testing.

4 Related Works

Extensible Domain Interpreters

In 2023, Brandl et al. developed a static analyzer based on the principal of modular domains, so that new domains could be added very easily without requiring any edits to previous code [2]. This design, while very efficient and creative, lacks the flexibility to add any new features from the language, and so it is limited in that aspect. Michelland et al. created a modular proof big step interpreter based on the use of monadic interpreters and interaction trees. They used handlers and transformers to maintain the same syntax for concrete and abstract interpretation [5]. This approach allows for easy domain extensibility, and while it lends itself to easily adding new language features these additions do require code revision.

Extensible Feature Interpreters

In 2023 Jin et al. created FPOP, an extensible proof based interpreter built with polymorphic family structures [3]. This ingenious design is built rather well, however it is specifically for a proof base language and thus not applicable to commonly used languages like C and Python. Because of its limited use case, it is not a good solution to the problem, but it does demonstrate how Object-Oriented code can help to solve the expression problem.

Previous Language Agnostic Interpreters

The ORBS interpreter created by Binkley et al. is capable of language agnostic slicing, meaning it can efficiently analyze snippets of code from multiple languages at once [1]. However, this design is inherently only capable of analyzing one specific feature from an ensemble of programs, and not the entirety [8]. The Modular Open Platform for Static Analysis (MOPSA) is one of the most extensible static analyzers to date, built on the principal of extensibility of new program features, so new languages can be added easily and efficiently without major code revisions. [6]. Their design is largely used and a good solution to the problem, however it involves repeated calls to evaluators until one returns a non-null value, which in a large codebase for many fully integrated languages could become costly and too expensive. MOPSA is by far the current state of the art and has many use cases, but I believe has room for improvement.

References

[1] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 109–120. https://doi.org/10.1145/2635868.2635893

[2] Katharina Brandl, Sebastian Erdweg, Sven Keidel, and Nils Hansen. 2023. Modular Abstract Definitional Interpreters for WebAssembly. In *DROPS-IDN/v2/document/10.4230/LIPIcs.ECOOP.2023.5*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2023.5

[3] Ende Jin, Nada Amin, and Yizhou Zhang. 2023. Extensible Metatheory Mechanization via Family Polymorphism. *Artifact for Extensible Metatheory Mechanization via Family Polymorphism* 7, PLDI (June 2023), 172:1608–172:1632. https://doi.org/10.1145/3591286

[4] Oleg Kiselyov. 2017. Having an Effect. https://docs.huihoo.com/okmij.org/ftp/Computation/having-effect.html.

[5] Sébastien Michelland, Yannick Zakowski, and Laure Gonnord. 2024. Abstract Interpreters: A Monadic Approach to Modular Verification. *Replication package for article: Abstract Interpreters: a Monadic Approach to Modular Verification* 8, ICFP (Aug. 2024), 257:602–257:629. https://doi.org/10.1145/3674646

[6] Antoine Miné, Abdelraouf Ouadjaout, and Matthieu Journault. 2018. Design of a Modular Platform for Static Analysis. (2018).

[7] Philip Wadler. 1998. The Expression Problem.

[8] Haoran Yang, Wen Li, and Haipeng Cai. 2022. Language-agnostic dynamic analysis of multilingual code: promises, pitfalls, and prospects. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1621–1626. https://doi.org/10.1145/3540250.3560880