Compositional and Mechanically Verified Program Analyzers

David Darais[1]

1 **University of Maryland**
College Park, Maryland, USA
darais@cs.umd.edu

1 Introduction

Program analyzers have proven effective in detecting undesired behavior in programs such as crashes, bugs, and security vulnerabilities. Some settings require high assurance in the results of program analysis, such as software embedded in automobiles or airplanes. To achieve high assurance in the correctness or security of a piece of software, formal methods are used to automatically construct or check proofs of these properties using computers.

Achieving high assurance for a piece of software is a monumental task, and is widely considered by experts to be out of reach for mainstream use using current methods. As a result, verification is only attempted for the most critical software components. In this thesis, I describe how to bring high assurance software closer to a reality by improving the methods used to develop implementations and proofs for program analyzers.

There are two fundamental challenges in the state of the art for mechanized verification of program analyzers:

1. small changes to program analyzers require large changes in their proofs; and
2. a proof for a program analyzer takes much longer to complete than its implementation.

I improve on the first problem by demonstrating how to design *Compositional Program Analyzers* and *Definitional Program Analyzers* which allow one to rapidly prototype different analysis designs without having to rewrite proofs of correctness. I improve on the second problem by developing a new framework for *Mechanizing Program Analyzers* which supports the development of verified program analyzers directly from their proofs of correctness.

2 Compositional Program Analyzers

The design and implementation of static analyzers has become increasingly systematic. Yet although the design is systematic, it often requires tedious and error prone work to implement an analyzer and prove it sound. The issue is that static analysis features and their proofs of soundness do not compose well, preventing reuse in both implementation and metatheory.

I solve the problem of constructing static analyzers and their proofs from reusable components by introducing Galois transformers[2]: monad transformers that transport Galois connection properties. In concert with a monadic interpreter, I define a library of monad transformers that implement building blocks for classic analysis parameters like context, heap, path and flow (in)sensitivity. Moreover, these can be composed together independent of the language being analyzed.

Significantly, a Galois transformer can be proved sound once and for all, making it a reusable analysis component. As new analysis features and abstractions are developed and mixed in, soundness proofs need not be reconstructed, as the composition of a monad transformer stack is sound by virtue of its constituents. Galois transformers provide a viable foundation for reusable and composable metatheory for program analysis.

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Finally, these Galois transformers shift the level of abstraction in analysis design and implementation to a level where non-specialists have the ability to synthesize sound analyzers over a number of parameters.

The Technical Problem

Consider the following program:

```
1  function myfun(I : int) → int
2    var x,y : int
3    if I ≠ 0
4      then x := 0
5      else x := 1
6    if I ≠ 0
7      then y := 100 / I
8      else y := 100 / x
9    return y
```

This program branches on the function argument I to define x such that $x = 0 \Leftrightarrow I \neq 0$. The variable y is then assigned a value by dividing by I if $I \neq 0$, or x if $I = 0$. The goal of the analysis is to discover division by zero errors, and this program is always safe because of the correlation between x and I. However, only sophisticated and computationally expensive techniques like *path sensitivity* are capable of discovering this information. In a security-critical setting, path sensitivity is the right choice for the analysis despite the added computational complexity. In performance-critical settings path sensitivity is undesirable because of the cost of the analysis. What is needed is a technique for designing a program analysis parameterized by its path sensitivity properties.

Galois Transformers are reusable building blocks for building analysers that provide each choice in the path sensitivity spectrum: flow insensitive, flow sensitive and path sensitive. A flow insensitive Galois Transformers can simply be replaced by a path sensitive Galois transformer, requiring no further change to the analyzer or its proof. In this way, one can rapidly prototype this design space for a given program analysis setting. Proofs of correctness for the analyzer also carry over between different instantiations of Galois Transformers.

3 Definitional Program Analyzers

Two dominant schools of through for designing program analyzers are the constraint-based approach and small-step state-machines-based approach. In both paradigms, the analysis is computed by the least-fixed-point of a set of constraint equations, or small-step collecting semantics respectively.

On the other hand, there is a large body of work on denotational semantics and definitional interpreters, or so-called "big-step" interpreters. This style is popular for describing concrete semantics, but has not yet seen adoption for describing abstract semantics, or as the basis for defining program analyzers.

To bridge this gap I develop Abstract Definitional Interpreters and show that definitional interpreters written in monadic style can express not only the usual notion of (concrete) interpretation, but also a wide variety of collecting semantics, abstract interpretations, symbolic execution, and their intermixings.

In this work I reconstruct a definitional abstract interpreter for a higher-order language to use monadic operations and a novel fixpoint iteration strategy. Through a monadic defin-

itional design, I achieve a computable abstract interpreter that arises from the composition of simple, independent components.

Remarkably, the resulting program analyzer implements a form of pushdown control flow analysis (PDCFA) in which calls and returns are always properly matched in the abstract semantics. True to the definitional style of Reynolds, the evaluator involves no explicit mechanics to achieve this property; it is simply inherited from the defining language.

The Technical Problem

Consider the following program:

```
1  function id(x : any) → any
2    return x
3  function main() → void
4    var y := id(1)
5    print("Y")
6    var z := id(2)
7    print("Z")
```

Clearly, the printed output of this program is `"YZ"`. However, most control-flow analyzers will report that the output could be any string that matches the regular expression `"Y*Z"`. The problem is that control flow analyzers typically construct a graph of call edges, in this case from lines 5 and 7 to the body of *id*, and return edges, in this case from *id* back to lines 5 and 7. Without precise call-return matching, control-flow analyzers get confused and think the program could call *id* at line 5 and then return to line 7, or call *id* at line 7 and return to line 5. A "k-call-site-sensitive" analysis can distinguish these cases, but only up to a finite call-depth.

A pushdown analysis solves the call/return matching problem up to infinite depth. Prior descriptions of pushdown analysis are set in the context of actual pushdown automata[10], Dyck state graphs[4] or small-step state machines[11, 7, 6], and each approach require ad-hoc extensions and instrumentation to the design of the program analyzer. By writing program analyzers in a definitional style, precise call/return matching is inherited from the defining metalanguage. To achieve this, I solve the key technical challenging in defining definitional program analyzers through a novel big-step fixpoint finding algorithm.

4 Mechanizing Program Analyzers

Galois connections are a foundational tool for structuring abstraction in semantics and their use lies at the heart of the theory of abstract interpretation. Yet, mechanization of Galois connections remains limited to restricted modes of use, preventing their general application in mechanized metatheory and certified programming.

I present Constructive Galois Connections[3], a variant of Galois connections that is effective both on paper and in proof assistants; is complete with respect to a large subset of classical Galois connections; and enables more general reasoning principles, including the "calculational" style advocated by Cousot.

To design constructive Galois connection I identify a restricted mode of use of classical ones which is both general and amenable to mechanization in dependently-typed functional programming languages. Crucial to the metatheory is the addition of monadic structure to Galois connections to control a "specification effect". Effectful calculations may reason classically, while pure calculations have extractable computational content. Explicitly moving between the worlds of specification and implementation is enabled by the metatheory.

To validate the approach, I provide two case studies in mechanizing existing proofs from the literature: one uses calculational abstract interpretation to design a static analyzer[1], the other forms a semantic basis for gradual typing[5]. Both mechanized proofs closely follow their original paper-and-pencil counterparts, employ reasoning principles not captured by previous mechanization approaches[8, 9], support the extraction of verified algorithms, and are novel.

The Technical Problem

Consider a simple parity program analyzer designed using the following Galois connection between natural numbers $\mathbb{N}$ and parities $\mathbb{P}$ (plus Galois Connection laws not shown):

$$\mathbb{P} \coloneqq \{\texttt{EVEN}, \texttt{ODD}\}$$
$$\alpha : \wp(\mathbb{N}) \to \wp(\mathbb{P})$$
$$\alpha(N) \coloneqq \bigcup_{n \in N} \begin{cases} \{\texttt{EVEN}\} & \text{if} \quad even(n) \\ \{\texttt{ODD}\} & \text{if} \quad odd(n) \end{cases}$$
$$\gamma : \wp(\mathbb{P}) \to \wp(\mathbb{N})$$
$$\gamma(P) \coloneqq \bigcup_{p \in P} \begin{cases} \{n \mid even(n)\} & \text{if} \quad p = \texttt{EVEN} \\ \{n \mid odd(n)\} & \text{if} \quad p = \texttt{ODD} \end{cases}$$

A static analyzer $\mathcal{A} : \wp(\mathbb{P}) \to \wp(\mathbb{P})$ for a concrete semantics $\mathcal{C} : \wp(\mathbb{N}) \to \wp(\mathbb{N})$ is then justified by relating to (or even calculating from) the composition of $\mathcal{C}$ with α and γ:

$$sound : \alpha \circ \mathcal{C} \circ \gamma \subseteq \mathcal{A}$$

The trouble in mechanizing *sound* is that $\mathcal{A}$ is expected to be computable, meaning its type $\wp(\mathbb{P}) \to \wp(\mathbb{P})$ represents an *algorithm* mapping between finite sets of parities. However, the specification $\alpha \circ \mathcal{C} \circ \gamma$, also at type $\wp(\mathbb{P}) \to \wp(\mathbb{P})$, represents an induced *specification* which cannot be computed.

In a constructive setting which supports program extraction, these two powerset types have different representations. Constructed powersets $\wp(\mathbb{P})$ are modelled with a datatype like a list or binary tree, or in the case of $\wp(\mathbb{P})$ as an enumeration of its inhabitants:

$$\wp(\mathbb{P}) \approx \mathbb{P}^+ \coloneqq \{\bot, \texttt{EVEN}, \texttt{ODD}, \top\}$$

However, specification powersets $\wp(\mathbb{P})$ are modelled as predicates on $\mathbb{P}$:

$$\wp(\mathbb{P}) \approx \mathbb{P} \to prop$$

On paper the encoding of $\wp(\mathbb{P})$ doesn't matter, but to perform verified program extraction on $\mathcal{A}$, a solution must be found for encoding proofs like *sound* which transition between specification and algorithm.

The solution is to design a constructive theory of Galois connections, which can be seen as a restricted mode of use of classical Galois connections. The essence of the theory is a different adjunction η/μ instead of α/γ:

$$\eta : \mathbb{N} \to \mathbb{P}$$
$$\eta(n) \coloneqq \begin{cases} \texttt{EVEN} & \text{if} \quad even(n) \\ \texttt{ODD} & \text{if} \quad odd(n) \end{cases}$$
$$\mu : \mathbb{P} \to \wp(\mathbb{N})$$
$$\mu(p) \coloneqq \begin{cases} \{n \mid even(n)\} & \text{if} \quad p = \texttt{EVEN} \\ \{n \mid odd(n)\} & \text{if} \quad p = \texttt{ODD} \end{cases}$$

In this restricted theory, executable algorithms can be extracted directly from the results of proofs in the abstract interpretation paradigm.

References

1. Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
2. David Darais, Matthew Might, and David Van Horn. 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, pages 552–571, New York, NY, USA, 2015. ACM.
3. David Darais and David Van Horn. Constructive galois connections: Taming the galois connection framework for mechanized metatheory. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP '16. ACM, 2016.
4. Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. Introspective pushdown analysis of higher-order programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12. ACM, 2012.
5. Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, St Petersburg, FL, USA, January 2016. ACM Press. To appear.
6. Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 691–704, New York, NY, USA, 2016. ACM.
7. James Ian Johnson and David Van Horn. Abstracting abstract control. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 11–22, New York, NY, USA, 2014. ACM.
8. David Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Rapport de DEA, Université Paris VII, 1998. French.
9. David Pichardie. *Interprétation abstraite en logique intuitionniste: extraction d'analyseurs Java certifiés*. PhD thesis, Université Rennes, 2005.
10. Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95. ACM, 1995.
11. Dimitrios Vardoulakis and Olin Shivers. CFA2: a Context-Free Approach to Control-Flow Analysis. In *European Symposium on Programming*, pages 570–589, 2010.