Mechanizing Abstract Interpretation

Thesis Defense

David Darais
University of Maryland
Software Reliability
The Usual Story

<table>
<thead>
<tr>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing</td>
</tr>
<tr>
<td>Analysis</td>
</tr>
<tr>
<td>Compiler</td>
</tr>
<tr>
<td>Operating System</td>
</tr>
<tr>
<td>Hardware</td>
</tr>
</tbody>
</table>
The Usual Story

<table>
<thead>
<tr>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing</td>
</tr>
<tr>
<td>Analysis</td>
</tr>
<tr>
<td>Compiler</td>
</tr>
<tr>
<td>Operating System</td>
</tr>
<tr>
<td>Hardware</td>
</tr>
</tbody>
</table>
The Reality

<table>
<thead>
<tr>
<th>Category</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>✓</td>
</tr>
<tr>
<td>Testing</td>
<td></td>
</tr>
<tr>
<td>Analysis</td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td></td>
</tr>
<tr>
<td>Operating System</td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td></td>
</tr>
</tbody>
</table>
The Reality

<table>
<thead>
<tr>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
</tr>
<tr>
<td>Testing</td>
</tr>
<tr>
<td>Analysis</td>
</tr>
<tr>
<td>Compiler</td>
</tr>
<tr>
<td>Operating System</td>
</tr>
<tr>
<td>Hardware</td>
</tr>
</tbody>
</table>

![Bug on a phone]
Security Exploit
In Linux Kernel

Time →
(2009)
Security Exploit in Linux Kernel

Kernel Patch to Fix Exploit (2009)
Time (2009)

Security Exploit In Linux Kernel

Kernel Patch to Fix Exploit

Security Exploit In Linux Kernel
Time \(\rightarrow\)

(2009)

Security Exploit in Linux Kernel

Kernel Patch to Fix Exploit
Story 1: Linux Kernel Exploit

static unsigned int tun_chr_poll(struct file *file, 
{ 
    struct tun_file *tfile = file->private_data; 
    struct tun_struct *tun = __tun_get(tfile); 
    struct sock *sk = tun->sk; 
    unsigned int mask = 0; 

    if (!tun) 
        return POLLERR;

–Linux 2.6.30 kernel exploit [2009]
Story 1: Linux Kernel Exploit

```c
static unsigned int tun_chr_poll(struct file *file,
{
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk = tun->sk;
    unsigned int mask = 0;

    if (!tun)
        return POLLERR;
    return POLLIN;
}
```

–Linux 2.6.30 kernel exploit [2009]
<table>
<thead>
<tr>
<th>Program</th>
<th>Testing</th>
<th>Compiler</th>
<th>Operating System</th>
<th>Hardware</th>
</tr>
</thead>
</table>

GCC Compiler
GCC Compiler

Linux OS Kernel

Program

Testing

Compiler

Hardware
Linux OS
Kernel

Self-driving Cars
Airplanes
SpaceX
Secure Web Infr.
Pacemakers
Medical Records DB

✓ Program
✓ Compiler
Operating System
Hardware
Trust in Software Runs Deep
<table>
<thead>
<tr>
<th>Component</th>
<th>✔</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>✔</td>
</tr>
<tr>
<td>Testing</td>
<td>✔</td>
</tr>
<tr>
<td>Analysis</td>
<td>✔</td>
</tr>
<tr>
<td>Compiler</td>
<td>✔</td>
</tr>
<tr>
<td>Operating System</td>
<td>✔</td>
</tr>
<tr>
<td>Hardware</td>
<td>✔</td>
</tr>
</tbody>
</table>

Critical Software Requires Trustworthy Tools
Trustworthy Tools are Critical Software
My Research: Tools with 0 Bugs
The Tools I Build: 
Program Analyzers (lightweight)

Difficult to Implement Correctly
The Tool I Use:  
**Mechanized Verification**  
(heavyweight)

Verify 0 Bugs in Program Analyzers
<table>
<thead>
<tr>
<th>Program Analyzers</th>
<th>Usable</th>
<th>Trustworthy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>Usable</td>
<td>Trustworthy</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>Program Analyzers</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Mechanized Verification</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Usable</td>
<td>Trustworthy</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>Program Analyzers</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Mechanized Verification</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Mechanically Verified</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

My Research
Problem

Building one verified analyzer is extremely difficult.

(decades for first compiler)
Assumption

*Calculational and compositional methods can make analyzers easier to construct.*
Research Question

How can we construct *mechanically verified program analyzers* using calculational and compositional methods?
Thesis

Constructing mechanically verified program analyzers via calculation and composition is feasible using constructive Galois connections and modular abstract interpreters.
Contribution 1

State of the art in program analysis and mechanized verification:

*Abstract interpretation*: 0 bugs in analyzer design+specification

*Mechanized verification*: 0 bugs in analyzer implementation

~20 year old problem: how to combine these two techniques
Contribution 1

State of the art in program analysis and mechanized verification:

Abstract interpretation: 0 bugs in analyzer design+specification

Mechanized verification: 0 bugs in analyzer implementation

~20 year old problem: how to combine these two techniques

Result: achieved mechanically verified calculational AI

Idea: new AI framework which supports mechanization

[Darais and Van Horn, ICFP ’16]
Contribution 2

State of the art in reusable program analyzers:

Some features easy to reuse: context and object sens.

Some features had to reuse: path and flow sens.

Challenge: achieve reuse in both implementation and proof
Contribution 2

State of the art in *reusable* program analyzers:

*Some features easy to reuse*: context and object sens.

*Some features had to reuse*: path and flow sens.

Challenge: achieve reuse in both implementation and proof

**Result**: compositional PA components, implementation + proofs

**Idea**: combine monad transformers and Galois connections

[Darais, Might and Van Horn, OOPSLA '15]
Contribution 3

State of the art in *reusable* program analysis:

*Control flow abstraction:* often too imprecise

*Pushdown precision:* precise abstraction for control

No technique which supports compositional interpreters
Contribution 3

State of the art in *reusable* program analysis:

*Control flow abstraction:* often too imprecise

*Pushdown precision:* precise abstraction for control

No technique which supports compositional interpreters

Result: *pushdown precision for definitional interpreters*

Idea: *inherit precision from defining metalanguage*

[Darais, Labich, Nguyễn and Van Horn, ICFP ‘17]
Constructive Galois Connections

Galois Transformers

Abstracting Definitional Interpreters
Constructive Galois Connections
Classical Galois Connections

```c
int a[3];
if (b) {x = 2} else {x = 4};
a[4 - x] = 1;
```
Classical Galois Connections

```c
int a[3];
if (b) {x = 2} else {x = 4};
a[4 - x] = 1;
```

\[ x \in \{2, 4\} \]
Classical Galois Connections

```c
int a[3];
if (b) {x = 2} else {x = 4};
a[4 - x] = 1;
```

\[ x \in \{2, 4\} \quad \Rightarrow \quad x \in [2, 4] \]
Classical Galois Connections

```c
int a[3];
if (b) {x = 2} else {x = 4};
a[4 - x] = 1;
```

\[ x \in \{2, 4\} \quad \Rightarrow \quad x \in [2, 4] \]

\[ x \in \{2, 3, 4\} \quad \Rightarrow \quad x \in [2, 4] \]
Classical Galois Connections

\[ \mathcal{O}(\mathbb{Z}) \rightarrow \mathbb{Z} \times \mathbb{Z} \]

- \( x \in \{2, 4\} \)
  - in
- \( x \in \{2, 3, 4\} \)
- \( x \in [2, 4] \)

```c
int a[3];
if (b) {x = 2} else {x = 4};
a[4 - x] = 1;
```
\( \varphi(\mathbb{Z}) \)
\[ \mathfrak{g}(\mathbb{Z}) \rightarrow \mathbb{Z} \times \mathbb{Z} \]

\[ \{2, 3, 4\} \oplus \{5\} \rightarrow [2, 4] \oplus [5, 5] \]
\[ \varphi(\mathbb{Z}) \cong \mathbb{Z} \times \mathbb{Z} \]

\[ \begin{align*}
\{2, 3, 4\} & \cong \{5\} \\
\{7, 8, 9\} & \cong [2, 4] \\
\{5\} & \cong [5, 5]
\end{align*} \]
\[
[2, 4] \hat{\oplus} [5, 5] = \alpha(\gamma([2, 4]) \hat{\oplus} \gamma([5, 5]))
\]
\[ \alpha(\gamma([2,4]) \diamond \gamma([5,5])) \]
\[ \alpha(\gamma([2,4]) \ast \gamma([5,5])) = \alpha(\{ i + j \mid i \in \gamma([2,4]) \land j \in \gamma([5,5]) \}) \]

\[ [2,4] \ast [5,5] \]
\[ \alpha(\gamma([2,4]) \hat{\land} \gamma([5,5])) \]
\[ = \]
\[ \alpha(\{ i + j \mid i \in \gamma([2,4]) \land j \in \gamma([5,5]) \}) \]
\[ = \]
\[ \alpha(\{ 7,8,9 \}) \]
\[ [2,4] \hat{\land} [5,5] \]
\[ \alpha(\gamma([2,4]) \hat{\circ} \gamma([5,5])) = \alpha(\{ \ i + j \ | \ i \in \gamma([2,4]) \land j \in \gamma([5,5]) \ \} ) = \alpha(\{7,8,9\}) = \alpha(\{7\}) \sqcup \alpha(\{8\}) \sqcup \alpha(\{9\}) \]

\[ [2,4] \hat{\circ} [5,5] \]
\[\alpha(\gamma([2,4]) \hat{\cup} \gamma([5,5])) =\]
\[\alpha(\{ i + j \mid i \in \gamma([2,4]) \land j \in \gamma([5,5]) \}) =\]
\[\alpha([7,8,9]) =\]
\[\alpha([7]) \sqcup \alpha([8]) \sqcup \alpha([9]) =\]
\[[7,9] =\]
\[[2,4] \hat{\cup} [5,5]\]
\[ \alpha(\gamma([2,4]) \hat{+} \gamma([5,5])) = \]
\[ \alpha(\{ i + j \mid i \in \gamma([2,4]) \land j \in \gamma([5,5]) \}) = \]
\[ \alpha(\{7,8,9\}) = \]
\[ \alpha(\{7\}) \sqcup \alpha(\{8\}) \sqcup \alpha(\{9\}) = \]
\[ [7,9] \]
\[ \triangleq [2,4] \hat{+} [5,5] \]
\[\alpha(\gamma([w,x]) \uplus \gamma([y,z])) = \alpha(\{ \ i + j \mid i \in \gamma([w,x]) \land j \in \gamma([y,z]) \ \}\} = \alpha(\{w+y,...,x+z\}) = \alpha(\{w+y\}) \uplus \ldots \uplus \alpha(\{x+z\}) = [w+y,x+z] \triangleq [w,x] \uplus [y,z]\]
\[ \alpha(\gamma([w,x]) \uplus \gamma([y,z])) = \]
\[ \alpha(\left\{ i + j \mid i \in \gamma([w,x]) \land j \in \gamma([y,z]) \right\}) = \]
\[ \alpha(\{w+y,\ldots,x+z\}) = \]
\[ \alpha(\{w+y\}) \uplus \ldots \uplus \alpha(\{x+z\}) = [w+y,x+z] \]
\[ \triangleq [w,x] \uplus [y,z] \]
Mechanized Verification (MV)
Traditional Approach

Verified Model
Traditional Approach

The Calculational Design of a Generic Abstract Interpreter
[Cousot, 1998]
Traditional Approach

(* let b_unary b_uop r x = *)
let b_unary b_uop x r =

(* let b_binary b_bop r x y = *)
let b_binary b_bop x y r =

*** bug corrected on 02/09/2000 ***

CDGAI Errata [Cousot, 2000]
Traditional Approach

“Beware of bugs in the above code; I have only proved it correct, not tried it.”

–Donald Knuth
Traditional Approach

Verified Model
Mechanized Verification

Verified Model

Certified Implementation
The Plan

\[
\alpha(\gamma([w,x]) \downarrow \gamma([y,z])) = \alpha(\{ i + j \mid i \in \gamma([w,x]) \land j \in \gamma([y,z]) \}) = \alpha(\{w+y,...,x+z\}) = \alpha(\{w+y\}) \sqcup ... \sqcup \alpha(\{x+z\}) = [w+y,x+z] \triangleq [w,x] \downarrow [y,z]
\]
The Plan

Step 1: **Check** These Calculations Using a Proof Assistant

\[
\alpha(\gamma([w,x]) \uplus \gamma([y,z])) = \\
\alpha(\{ i + j \mid i \in \gamma([w,x]) \\
\quad \land j \in \gamma([y,z]) \}) = \\
\alpha(\{w+y,\ldots,x+z\}) = \\
\alpha(\{w+y\}) \uplus \ldots \uplus \alpha(\{x+z\}) = \\
[w+y,x+z] \uplus [w,x] \vdash [y,z]
\]
The Plan

Step 1: **Check** These Calculations Using a Proof Assistant

Step 2: **Extract** a Certified Implementation

\[
\alpha(\gamma([w,x]) \uparrow \gamma([y,z]))
\]
\[
= \alpha(\{ i + j \mid i \in \gamma([w,x]) \land j \in \gamma([y,z]) \})
\]
\[
= \alpha(\{w+y,...,x+z\})
\]
\[
= \alpha(\{w+y\}) \sqcup \cdots \sqcup \alpha(\{x+z\})
\]
\[
= [w+y,x+z] \uparrow [w,x] \uparrow [y,z]
\]
The Plan

**Step 1:**
**Check** These Calculations Using a Proof Assistant

**Step 2:**
**Extract** a Certified Implementation
“This looks like an algorithm”
(to a human)

“I know how to execute this”
(to a machine)
“This looks like an algorithm” (to a human)  “I know how to execute this” (to a machine)

Mathematical Formulas

Backed by an Algorithm
“This looks like an algorithm”
(to a human)

“I know how to execute this”
(to a machine)
“This looks like an algorithm”
(to a human)

“I know how to execute this”
(to a machine)

**PROBLEM:** how to know when boundary is crossed

**SOLUTION:** explicitly account for algorithmic content

Classical Mathematics

Constructive Mathematics
Classical Galois Connections

\[ \mathcal{O}(\mathbb{Z}) \xrightarrow{\alpha} \mathbb{Z} \times \mathbb{Z} \]
Constructive Galois Connections

\[ \mathbb{Z} \xrightarrow{\eta} \mathbb{Z} \times \mathbb{Z} \]
Constructive Galois Connections

\[ \begin{array}{c}
\mathbb{Z} \\
\eta \\
\mathbb{Z} \times \mathbb{Z}
\end{array} \]

\[ \eta(i) = [i, i] \]

algorithmic content of abstraction
Constructive Galois Connections

\[ \mathbb{Z} \xrightarrow{\eta} \mathbb{Z} \times \mathbb{Z} \]

\[ \eta(i) = [i, i] \]

\[ \alpha = \langle \eta \rangle \]
Constructive Galois Connections

\[ \mathbb{Z} \xrightarrow{\eta} \mathbb{Z} \times \mathbb{Z} \]

**definition**

\[ \eta(i) = [i, i] \]

**law 1**

\[ \alpha = \langle \eta \rangle \]

**law 2**

\[ \langle \eta \rangle (\{x\}) = \langle \eta(x) \rangle \]

singleton powersets

compute
\[
\alpha(\gamma([w,x]) \uplus \gamma([y,z])) \\
= \\
\alpha(\{ i + j \mid i \in \gamma([w,x]) \land j \in \gamma([y,z]) \}) \\
= \\
\alpha(\{w+y,\ldots,x+z\}) \\
= \\
\alpha(\{w+y\}) \uplus \ldots \uplus \alpha(\{x+z\}) \\
= \\
[w+y,x+z] \\
\triangleq \\
[w,x] \uplus [y,z]
\]
\[ \alpha(\gamma([w, x]) \oplus \gamma([y, z])) \]
law 1

\[ \alpha = \langle \eta \rangle \]
\( \alpha(\gamma([w,x]) \oplus \gamma([y,z])) \)
\langle \eta \rangle (\gamma([w,x]) \dagger \gamma([y,z]))
\[ \langle \eta \rangle (\gamma([w,x]) \uplus \gamma([y,z])) = \langle \eta \rangle (\{ i + j \mid i \in \gamma([w,x]) \land j \in \gamma([y,z]) \}) = \langle \eta \rangle (\{w+y, \ldots, x+z\}) = \langle \eta \rangle (\{w+y\}) \uplus \ldots \uplus \langle \eta \rangle (\{x+z\}) \]
\[ \langle \eta \rangle (\gamma ([w,x]) \uplus \gamma ([y,z])) \]
\[ = \langle \eta \rangle (\{ i + j \mid i \in \gamma ([w,x]) \land j \in \gamma ([y,z]) \}) \]
\[ = \langle \eta \rangle (\{ w+y, \ldots, x+z \}) \]

law 2

\[ \langle \eta \rangle (\{ x \}) = \langle \eta(x) \rangle \]
\[ \langle \eta \rangle (\gamma([w,x]) \dagger \gamma([y,z])) = \langle \eta \rangle (\{ i + j \mid i \in \gamma([w,x]) \land j \in \gamma([y,z]) \}) = \langle \eta \rangle (\{w+y,...,x+z\}) = \langle \eta \rangle (\{w+y\}) \cup ... \cup \langle \eta \rangle (\{x+z\}) \]
\[ \langle \eta \rangle (\gamma([w,x]) \uplus \gamma([y,z])) = \langle \eta \rangle (\{ i + j \mid i \in \gamma([w,x]) \land j \in \gamma([y,z]) \}) = \langle \eta \rangle (\{w+y,...,x+z\}) = \langle \eta(w+y) \rangle \uplus ... \uplus \langle \eta(x+z) \rangle \]
\[ \langle \eta \rangle (\gamma([w,x]) \hat{+} \gamma([y,z])) \]

\[ = \langle \eta \rangle (\{ i + j \mid i \in \gamma([w,x]) \land j \in \gamma([y,z]) \}) \]

\[ = \langle \eta \rangle (\{w+y,\ldots,x+z\}) \]

\[ = \langle \eta(w+y) \rangle \cup \ldots \cup \langle \eta(x+z) \rangle \]

\[ = \langle [w+y,x+z] \rangle \]

\[ \triangleq [w,x] \hat{+} [y,z] \]
The Plan

Step 1: Check These Calculations Using a Proof Assistant

Step 2: Extract a Certified Implementation

Spec

\[
\{\eta\}(\gamma([w,x]) \uplus \gamma([y,z])) = \\
\{\eta\}(\{i + j \mid i \in \gamma([w,x]) \land j \in \gamma([y,z])\}) = \\
\{\eta\}([w+y,\ldots,x+z]) = \\
\{\eta(w+y)\} \uplus \ldots \uplus \{\eta(x+z)\} = \\
\{[w+y,x+z]\} \triangleq [w,x] \uplus [y,z]
\]
calc.cousot
calc.agda
\[
\alpha(\text{eval}[n])(\rho^#) \\
\triangleq \text{defn of } \alpha \Downarrow \\
= \alpha^I(\text{eval}[n](\gamma^R(\rho^#))) \\
\triangleq \text{defn of } \text{eval}[n] \Downarrow \\
= \alpha^I(\{i \mid \rho \vdash n \mapsto i\}) \\
\triangleq \text{defn of } \_ \vdash \_ \mapsto \\
= \alpha^I(\{n\}) \\
\triangleq \text{defn of } \text{eval}^#[n] \Downarrow \\
= \text{eval}^#[n](\rho^#) \\
\]
\[
\begin{align*}
[ \alpha[ \cong^R \cong^I ] \\
&\cdot \text{eval}[\text{Num } n ] \cdot \rho^# ] \\
[ \eta^I \ast \cdot (\text{eval}[\text{Num } n ] \ast \\
&\cdot (\mu^R \cdot \rho^#)) ] \\
[ \text{focus-right }[\cdot] \text{ of } \eta^I \ast ] \\
\triangleq \text{defn}[\text{eval}[\text{Num } n ]] \Downarrow \\
[ \eta^I \ast \cdot (\text{return } \cdot n) ] \\
\triangleq \text{right-unit}[*] \Downarrow \\
[ \text{pure } \cdot (\eta^I \cdot n) ] \\
[ \text{pure } \cdot \text{eval}^#[\text{Num } n ] \cdot \rho^# ]
\end{align*}
\]
Classical GCs

\[ A : \text{poset} \quad \quad \alpha : A \to B \]

\[ B : \text{poset} \quad \quad \gamma : B \to A \]

\[ x \subseteq \gamma(\alpha(x)) \land \alpha(\gamma(y)) \subseteq y \]

\[ x \subseteq \gamma(y) \iff \alpha(x) \subseteq y \]
Classical GCs

A : poset  \quad \alpha : A \to B

B : poset  \quad \gamma : B \to A

\text{id} \subseteq \gamma \circ \alpha \quad \land \quad \alpha \circ \gamma \subseteq \text{id}

\text{id}(x) \subseteq \gamma(y) \iff \alpha(x) \subseteq \text{id}(y)
Constructive GCs

A : poset  \quad \eta : A \rightarrow \wp(B)

B : poset  \quad \mu : B \rightarrow \wp(A)

\text{ret} \subseteq \mu \circ \eta \quad \land \quad \eta \circ \mu \subseteq \text{ret}

\text{ret}(n) \subseteq \mu^*(r) \iff \eta^*(n) \subseteq \text{ret}(r)
Constructive GCs

\[ A : \text{poset} \quad \eta : A \to B \]
\[ B : \text{poset} \quad \mu : B \to \wp(A) \]

\[ \text{ret} \subseteq \mu \otimes \langle \eta \rangle \land \langle \eta \rangle \otimes \mu \subseteq \text{ret} \]

\[ \text{ret}(n) \subseteq \mu^*(r) \iff \langle \eta \rangle^*(n) \subseteq \text{ret}(r) \]
Constructive GCs

\begin{align*}
A & : \text{poset} & \eta & : A \to B \\
B & : \text{poset} & \mu & : B \to \wp(A)
\end{align*}

\[
\begin{align*}
\text{ret} \subseteq & \mu \otimes \langle \eta \rangle \land \langle \eta \rangle \otimes \mu \subseteq \text{ret} \\
\text{ret}(n) \subseteq & \mu^*(r) \iff \langle \eta \rangle^*(n) \subseteq \text{ret}(r)
\end{align*}
\]
Constructive GCs

\[ \begin{align*}
    \text{A} & : \text{poset} & \quad \eta & : \text{A} \rightarrow \text{B} \\
    \text{B} & : \text{poset} & \quad \mu & : \text{B} \rightarrow \mathcal{P}(\text{A}) \\
\end{align*} \]

\[ \begin{align*}
    \text{ret} \subseteq & \quad \mu \circ \langle \eta \rangle \quad \land \quad \langle \eta \rangle \circ \mu \subseteq \text{ret} \\
    \text{ret}(n) \subseteq & \quad \mu^{*}(r) \quad \Leftrightarrow \quad \langle \eta \rangle^{*}(n) \subseteq \text{ret}(r) \\
\end{align*} \]
Classical GCs

= adjunction in category of posets
  (adjoints are mono. functions)

Constructive GCs

= biadjunction in category of posets enriched over \( \wp \)-Kleisli
  (adjoints are mono. \( \wp \)-monadic functions)
Constructive Galois Connections

✓ First theory to support both calculation and extraction
✓ Soundness and completeness w.r.t. classical GCs
✓ Two case studies: calculational AI and gradual typing
✗ Only (constr.) equivalent to subset of classical GCs
✗ Same limitations as classical GCs ($\exists \alpha$ for some $\gamma$)
Galois Transformers

```c
0: int x y;
1: void safe_fun(int N) {
2:   if (N\neq 0) {x := 0;}
3:   else {x := 1;}
4:   if (N\neq 0) {y := 100/N;}
5:   else {y := 100/x;}
```
Galois Transformers

```c
0: int x y;
1: void safe_fun(int N) {
2:   if (N != 0) {x := 0;}
3:   else {x := 1;}
4:   if (N != 0) {y := 100/N;}
5:   else {y := 100/x;} }
```

Flow-insensitive

\[
\text{results: } \text{var} \mapsto \emptyset (\{-, 0, +\})
\]
Galois Transformers

```
0: int x y;
1: void safe_fun(int N) {
  2: if (N≠0) {x := 0;}
  3: else {x := 1;}
  4: if (N≠0) {y := 100/N;}
  5: else {y := 100/x;}}
```

\[ N \in \{-, \emptyset, +\} \]
\[ x \in \{\emptyset, +\} \]
\[ y \in \{-, \emptyset, +\} \]

UNSAFE: \{100/N\}
UNSAFE: \{100/x\}

Flow-insensitive results: var \mapsto \emptyset(\{-, \emptyset, +\})
Galois Transformers

Flow-sensitive results:

\[
\text{loc} \mapsto (\text{var} \mapsto \emptyset (\{-,0,+\}))
\]
Galois Transformers

Path-sensitive results:

\[ \text{loc} \mapsto \mathcal{G}(\text{var} \mapsto \mathcal{G}(\{-,-,\emptyset,+,\})) \]

0: int x y;
1: void safe_fun(int N) {
2:   if (N\neq0) {x := 0;}
3:   else {x := 1;}
4:   if (N\neq0) {y := 100/N;}
5:   else {y := 100/x;}}

4: \( N \in \{-,+,\}, x \in \{\emptyset\} \)
4: \( N \in \{\emptyset\}, x \in \{+\} \)

\( N \in \{-,+,\}, y \in \{-,\emptyset,+,\} \)
\( N \in \{\emptyset\}, y \in \{\emptyset,+,\} \)

SAFE
Precision \simeq Performance
Insight

\[
\text{results: } \text{var} \mapsto \wp(\{-,0,+\}) \\
\text{results: } \text{loc} \mapsto (\text{var} \mapsto \wp(\{-,0,+\})) \\
\text{results: } \text{loc} \mapsto \wp(\text{var} \mapsto \wp(\{-,0,+\}))
\]
$\mathcal{C} : \text{loc} \times \text{store} \rightarrow \text{loc} \times \text{store}$
\[ C : \text{loc} \times \text{store} \rightarrow \text{loc} \times \text{store} \]

\[ A : \text{loc} \times \text{store}^\# \rightarrow \wp(\text{loc} \times \text{store}^\#) \]
\[C : \text{loc} \times \text{store} \rightarrow \text{loc} \times \text{store}\]
\[A : \text{loc} \times \text{store}^\# \rightarrow \wp(\text{loc} \times \text{store}^\#)\]
\[ C : \text{loc} \times \text{store} \rightarrow \text{loc} \times \text{store} \]

\[ A : \text{loc} \times \text{store}# \rightarrow \wp(\text{loc} \times \text{store}#) \]

\[
C \quad \bullet \quad \cdots \quad \rightarrow \quad \bullet \quad \cdots \quad \rightarrow \quad \bullet \quad \cdots \quad \rightarrow \quad \bullet \quad \cdots \quad \rightarrow \quad \bullet \quad \cdots \quad \rightarrow \quad \bullet \quad \cdots \quad \rightarrow \quad \bullet \quad \cdots
\]

\[
A \quad \bullet \quad \cdots \quad \rightarrow \quad \bullet \quad \cdots \quad \rightarrow \quad \bullet \quad \cdots \quad \rightarrow \quad \bullet \quad \cdots \quad \rightarrow \quad \bullet \quad \cdots
\]

\[
\text{analyzer} = \text{lfp } X. \ X \cup A^*(X) \cup \{\langle l_0, \bot \rangle\}
\]
\[ \mathcal{A} : \wp(\text{loc} \times \text{store#}) \rightarrow \wp(\text{loc} \times \text{store#}) \]
\[ \mathcal{A} : \wp(\text{loc } \times \text{store#}) \to \wp(\text{loc } \times \text{store#}) \]

Path Sensitive
\[ \Sigma = \text{loc } \mapsto \wp(\text{store#}) \]
\[ \approx \wp(\text{loc } \times \text{store#}) \]

Flow Sensitive
\[ \Sigma = \text{loc } \mapsto \text{store#} \]

Flow Insensitive
\[ \Sigma = \wp(\text{loc}) \times \text{store#} \]
\( \mathcal{M} : \text{loc} \rightarrow m(\text{loc}) \)
\[ \mathcal{M} : \text{loc} \rightarrow m(\text{loc}) \]

**Path Sensitive**
\[ m(A) = \text{store}# \rightarrow A \mapsto \wp(\text{store}#) \]

**Flow Sensitive**
\[ m(A) = \text{store}# \rightarrow A \mapsto \text{store}# \]

**Flow Insensitive**
\[ m(A) = \text{store}# \rightarrow \wp(A) \times \text{store}# \]
\[ M : \text{loc} \rightarrow m(\text{loc}) \]

**Path Sensitive**

\[ m(A) = (S[\text{store#}] \circ ND)(\text{ID})(A) \]

**Flow Sensitive**

\[ m(A) = FS[\text{store#}](\text{ID})(A) \]

**Flow Insensitive**

\[ m(A) = (ND \circ S[\text{store#}]) (\text{ID})(A) \]
Collecting Semantics

Path Sensitive

Flow Sensitive

Flow Insensitive

Collecting Semantics: $S[\sigma]$, $ND$, $ID$

Path Sensitive: $S[\sigma#]$, $ND$, $ID$

Flow Sensitive: $FS[\sigma#]$, $ID$

Flow Insensitive: $ND$, $S[\sigma#]$, $ID$
Collecting Analyzer

\[ S[σ] \]

\[ S[σ#] \]
Analyzer = Monad Stack + Monadic Interpreter

One Monadic Interpreter
Must Be Monotonic
Must Recover Collecting Semantics
Galois Transformers

✓ Flow sensitive and path sensitive precision
✓ Compositional end-to-end correctness proofs
✓ Implemented in Haskell and available on Github
✗ Not whole story for path-sensitive refinement
✗ Naive fixpoint strategies
Constructive Galois Connections

Galois Transformers

Abstracting Definitional Interpreters
Abstracting Definitional Interpreters

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")
Abstracting Definitional Interpreters

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")
Abstracting Definitional Interpreters

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")
Abstracting Definitional Interpreters

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")
Abstracting Definitional Interpreters

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")`  

> Z  
> YYYYYYYZ
Pushdown Precision

Reps et al 1995

Earl Diss 2012

Vardoulakis Diss 2012

Johnson and Van Horn 2014

Dyck State Graphs

“Big”CFA

Instrumented AAM

Instrumented AAM
Definitional Interpreters

- Modeled features vs inherited features
- (e.g., Reynolds’ inherited CBV and CBN)
- Things often modeled in Abstract Interpreters
  - Control (continuations)
  - Fixpoints
Definitional Interpreters

• Modeled features vs inherited features
• (e.g., Reynolds’ inherited CBV and CBN)
• Things often modeled in Abstract Interpreters
  • Control (continuations)
  • Fixpoints

Idea:
Inherit from metalanguage
\[ \mathcal{E}[\cdot] : \text{exp} \rightarrow \text{env} \times \text{store} \rightarrow (\text{val} \times \text{env} \times \text{store}) \]
\[ \mathcal{E}[\cdot] : \text{exp} \to \text{env} \times \text{store} \to (\text{val} \times \text{env} \times \text{store}) \]

\[
\begin{align*}
\mathcal{E}[\text{if}(e_1)\{e_2\}\{e_3\}](\rho,\sigma) &= \text{match } \mathcal{E}[e_1](\rho,\sigma) \\
| \langle \text{true},\sigma' \rangle &\Rightarrow \mathcal{E}[e_2](\rho,\sigma') \\
| \langle \text{false},\sigma' \rangle &\Rightarrow \mathcal{E}[e_3](\rho,\sigma')
\end{align*}
\]
\[ \mathcal{E}[\cdot] : \exp \rightarrow \text{env} \times \text{store} \rightarrow (\text{val} \times \text{env} \times \text{store}) \]

No explicit model for control (continuations). It’s inherited from the metalanguage.
\[ E[\cdot] : \text{exp} \rightarrow m(\text{val}) \]

**Step 1**

*Monadic Interpreter*
\[ \mathcal{E}[\cdot] : \text{exp} \to m(\text{val}) \]

\[ \mathcal{E}[\text{if}(e_1)\{e_2\}\{e_3\}] = \text{do} \]
\[ v \leftarrow \mathcal{E}[e_1] \]
\[ \text{match } v | \text{ true } \Rightarrow \mathcal{E}[e_2] \]
\[ | \text{ false } \Rightarrow \mathcal{E}[e_3] \]

\[ \ldots \]

\textit{Step 1}

\textit{Monadic Interpreter}
\[ \varepsilon[\cdot] : \exp \rightarrow (\exp \rightarrow m(val)) \rightarrow m(val) \]

**Step 2**

*Unfixed Recursion*
\[ \mathcal{E} \[ \cdot \] : \text{exp} \to (\text{exp} \to m(\text{val})) \to m(\text{val}) \]

\[
\ldots \\
\mathcal{E}[\text{if}(e_1)\{e_2\}\{e_3\}](\mathcal{E}^{'}) = \text{do} \\
v \leftarrow \mathcal{E}^{' \ [e_1]} \\
\text{match } v \mid \text{true} \Rightarrow \mathcal{E}^{' \ [e_2]} \\
\mid \text{false} \Rightarrow \mathcal{E}^{' \ [e_3]} \\
\ldots \\
\]

\textit{Step 2} \\
\textit{Unfixed Recursion}
\[ \varepsilon[\cdot] : \text{exp} \to (\text{exp} \to m\#(\text{val})) \to m\#(\text{val}) \]

\[
\ldots
\]

\[
\varepsilon[\text{if}(e_1)\{e_2\}\{e_3\}](\varepsilon\,') = \text{do}
\]

\[
\begin{align*}
v & \leftarrow \varepsilon\,'[e_1] \\
\text{match } v \mid \text{true} & \Rightarrow \varepsilon\,'[e_2] \\
\mid \text{false} & \Rightarrow \varepsilon\,'[e_3]
\end{align*}
\]

\[
\ldots
\]

\textbf{Step 3}

\textit{Abstract Monad}
\[ E[\cdot] : \exp \rightarrow (\exp \rightarrow m\#(\text{val})) \rightarrow m\#(\text{val}) \]

\[ Y(\lambda E'. \lambda e. E[e](E')) \quad \text{Abstract Evaluator (Doesn't Terminate)} \]
$$\varepsilon[\cdot] : \text{exp} \rightarrow (\text{exp} \rightarrow \text{m#}(\text{val})) \rightarrow \text{m#}(\text{val})$$

$$Y(\lambda \varepsilon’. \lambda e. \varepsilon[e](\varepsilon’))$$

Abstract Evaluator (Doesn’t Terminate)

$$CY(\lambda \varepsilon’. \lambda e. \varepsilon[e](\varepsilon’))$$

Caching Evaluator (Terminates)

Pushdown Precision
Formalism

\[ \rho, \tau \vdash e, \sigma \downarrow v, \sigma \]  
Evaluation

\[ \rho, \tau \vdash e, \sigma \uparrow \langle e, \rho, \tau, \sigma \rangle \]  
Reachability
Formalism

\[ \rho, \tau \vdash e, \sigma \downarrow v, \sigma \]

Evaluation

\[ \rho, \tau \vdash e, \sigma \uparrow \langle e, \rho, \tau, \sigma \rangle \]

Reachability

\[
\llbracket e \rrbracket(\rho, \tau, \sigma) = \\
\{\langle v, \sigma'' \rangle \mid \rho, \tau \vdash e, \sigma \uparrow \langle e', \rho', \tau', \sigma' \rangle \\
\land \rho', \tau' \vdash e', \sigma' \downarrow \langle v, \sigma'' \rangle \}
\]
Definitional Abstract Interpreters

✓ Compositional program analyzers
✓ Formalized w.r.t. big-step reachability semantics
✓ Pushdown precision inherited from metalanguage
✓ Implemented in Racket and available on Github
✗ Naive caching algorithm (could be improved)
✗ Monadic, open-recursive interpreters
<table>
<thead>
<tr>
<th></th>
<th>Usable</th>
<th>Trustworthy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Program Analysis</strong></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Mechanized Verification</strong></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>MVPA</strong></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

*My Research*
Thesis

Constructing mechanically verified program analyzers via calculation and composition is feasible using constructive Galois connections and modular abstract interpreters.
Constructive Galois Connections

Galois Transformers

Abstracting Definitional Interpreters

Mechanization + Calculation


Compositional Interpreters + Pushdown Precision