Compositional and Mechanically Verified Program Analyzers

David Darais
University of Maryland
Let's Design an Analysis
Let's Design an Analysis

Property

\[ x/0 \]
Let's Design an Analysis

Property

0: \texttt{int x y;}
1: \texttt{void safe\_fun(int N) \{} \texttt{\}
2: \texttt{if (N\neq 0) \{} \texttt{x := 0;}} \texttt{\}
3: \texttt{else \{} \texttt{x := 1;}} \texttt{\}
4: \texttt{if (N\neq 0) \{} \texttt{y := 100/N;} \texttt{\}
5: \texttt{else \{} \texttt{y := 100/x;} \texttt{\}}
Let's Design an Analysis

Property:

\[
\frac{x}{0}
\]

Program:

```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; }
}
```

Value Abstraction:

\[
\mathbb{Z} ⊆ \{-, 0, +\}
\]
Let's Design an Analysis

Property

Program

Value Abstraction

\[
\begin{align*}
\text{x/0} & \quad \text{Implement} \\
\text{0: int } x \ y; & \quad \text{0: int } x \ y; \\
1: \text{void safe_fun(int N) } & \quad 1: \text{void safe_fun(int N) } \\
2: \quad \text{if } (N \neq 0) \{ x := 0; \} & \quad 2: \quad \text{if } (N \neq 0) \{ x := 0; \} \\
3: \quad \text{else } \{ x := 1; \} & \quad 3: \quad \text{else } \{ x := 1; \} \\
4: \quad \text{if } (N \neq 0) \{ y := 100/N; \} & \quad 4: \quad \text{if } (N \neq 0) \{ y := 100/N; \} \\
5: \quad \text{else } \{ y := 100/x; \} & \quad 5: \quad \text{else } \{ y := 100/x; \} \\
\end{align*}
\]

\[\mathbb{Z} \subseteq \{-,0,+\}\]

\[\text{analyze : exp } \rightarrow \text{ results}\]
\[\text{analyze(x := \&\&): =}\]
\[.. x .. \&\& ..\]
\[\text{analyze(IF(\&\&){e_1}{e_2}) : =}\]
\[.. \&\& .. e_1 .. e_2 ..\]
Let's Design an Analysis

Property

Program

Value Abstraction

\[ x/\theta \]

\[
\begin{align*}
0: & \text{int } x \ y; \\
1: & \text{void safe_fun} \ (\text{int } \ N) \ \{ \\
2: & \quad \text{if} \ (N \neq 0) \ \{ x := 0; \} \\
3: & \quad \text{else} \quad \{ x := 1; \} \\
4: & \quad \text{if} \ (N \neq 0) \ \{ y := 100/N; \} \\
5: & \quad \text{else} \quad \{ y := 100/x; \} \\
\end{align*}
\]

\[ \mathbb{Z} \subseteq \{-,0,+\} \]

Implement

\[
\begin{align*}
\text{analyze : exp} & \rightarrow \text{results} \\
\text{analyze}(x := \alpha) & \rightarrow \ldots x \ldots \alpha \ldots \\
\text{analyze}(\text{IF}(\alpha)\{e_1\}) & \rightarrow \ldots \alpha \ldots e_1 \ldots
\end{align*}
\]

Results

\[
\begin{align*}
N \in \{-,0,+\} \\
x \in \{0,+\} \\
y \in \{-,0,+\}
\end{align*}
\]

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

\textbf{UNSAFE:} \{100/x\}
Let's Design an Analysis

Property

Program

Value Abstraction

Prove Correct

\[ [e] \in [\text{analyze}(e)] \]
Let's Design an Analysis

Property  
\( x/0 \)

Program  
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;}}`

Value Abstraction  
\( \mathbb{Z} \subseteq \{-, 0, +\} \)

Implement  
\[
\begin{align*}
\text{analyze} : \ & \text{exp} \to \text{results} \\
\text{analyze}(x := \alpha) :&= \\
& \ldots x \ldots \alpha \ldots \\
\text{analyze}(\text{IF}(\alpha)\{e_1\}\{e_2\}) :&= \\
& \ldots \alpha \ldots e_1 \ldots e_2 \ldots 
\end{align*}
\]

Results  
\[
\begin{align*}
N & \in \{-, 0, +\} \\
x & \in \{0, +\} \\
y & \in \{-, 0, +\} \\
\text{UNSAFE} : & \{100/N\} \\
\text{UNSAFE} : & \{100/x\}
\end{align*}
\]

Prove Correct  
\( [e] \in [\text{analyze}(e)] \)
Let's Design an Analysis

```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;}}

N ∈ {-,0,+}
x ∈ {0,+}
y ∈ {-,0,+}

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

Flow-insensitive results: \( \text{var} \mapsto \emptyset (\{-,0,\+\}) \)
Let's Design an Analysis

Flow-sensitive results: \( \text{loc} \mapsto (\text{var} \mapsto \emptyset (\{-, 0, +\})) \)
Let's Design an Analysis

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;}}

Path-sensitive results: loc ↦ ℘(var ↦ ℘({-,-,0,+,+}))

SAFE
Let's Design an Analysis

**Property**

\[
\frac{x}{0}
\]

**Program**

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

**Value Abstraction**

\[\mathbb{Z} \subseteq \{-, 0, +\}\]

**Implement**

```c
analyze : exp \rightarrow results
analyze(x := a) :=
.. x ...
analyze(IF(e_1)_{e_2}) :=
.. a \ldots e_1 \ldots e_2 ..
```

**Results**

4: \(N \in \{-, +\}, x \in \{0\}\)
4: \(N \in \{0\}, x \in \{+\}\)
\(N \in \{-, +\}, y \in \{-0, +\}\)
\(N \in \{0\}, y \in \{0, +\}\)
SAFE

**Prove Correct**

\([e] \in [\text{analyze}(e)]\)
Let's Design an Analysis

Property

\( x/0 \)

Program

safe_fun.js

Value Abstraction

\( \mathbb{Z} \subseteq \{-, 0, +\} \)

Implement

\[
\text{analyze} : \text{expr} \rightarrow \text{results} \\
\text{analyze}(x \leftarrow \text{expr}) := \\
\quad \ldots \times \ldots \ldots \\
\text{analyze}((\text{IF}\{\text{expr}\} \{e_1\} \{e_2\}) := \\
\quad \ldots x \ldots e_1 \ldots e_2 \ldots
\]

Results

4: \( N \in \{-, +\}, x \in \{0\} \)
4: \( N \in \{0\}, x \in \{+\} \)

\( N \in \{-, +\}, y \in \{-, 0, +\} \)
\( N \in \{0\}, y \in \{0, +\} \)

SAFE

Prove Correct

\( [e] \in [\text{analyze}(e)] \)
Contributions

Orthogonal Components
Galois Transformers [OOPSLA’15]

Systematic Design
Abstracting Definitional Interpreters [draft]

Mechanized Proofs
Constructive Galois Connections [ICFP’16]
Contributions

Orthogonal Components
Galois Transformers [OOPSLA’15]

Systematic Design
Abstracting Definitional Interpreters [draft]

Mechanized Proofs
Constructive Galois Connections [ICFP’16]
Orthogonal Components

Property

\[ x/0 \]

Program

\[
0: \text{int } x, y; \\
1: \text{void safe_fun(int } N) \{ \\
2: \quad \text{if } (N\neq 0) \{ x := 0; \} \\
3: \quad \text{else } \{ x := 1; \} \\
4: \quad \text{if } (N\neq 0) \{ y := 100/N; \} \\
5: \quad \text{else } \{ y := 100/x; \}\}
\]

Value Abstraction

\[ \mathbb{Z} \subseteq \{-,0,+\} \]

Implement

\[
\begin{align*}
\text{analyze : exp } \rightarrow & \text{ results} \\
\text{analyze}(x := e) := & \ldots x \ldots \\
\text{analyze(IF } e_1\{e_2\}) := & \ldots \text{SAFE } \ldots e_1 \ldots e_2 \ldots
\end{align*}
\]

Results

\[
4: N\in\{-,+\}, x\in\{0\} \\
4: N\in\{0\}, x\in\{+\} \\
N\in\{-,+\}, y\in\{-,0,+\} \\
N\in\{0\}, y\in\{0,+\}
\]

Prove Correct

\[
[e] \in [\text{analyze}(e)]
\]
Orthogonal Components

**Problem:** Isolate path and flow sensitivity in analysis
Orthogonal Components

**Problem:** Isolate path and flow sensitivity in analysis

**Challenge:** Path and flow sensitivity are deeply integrated
Orthogonal Components

**Problem:** Isolate path and flow sensitivity in analysis

**Challenge:** Path and flow sensitivity are deeply integrated

**State-of-the-art:** Redesign from scratch
Orthogonal Components

**Problem:** Isolate path and flow sensitivity in analysis

**Challenge:** Path and flow sensitivity are deeply integrated

**State-of-the-art:** Redesign from scratch

**Our Insight:** Monads capture path and flow sensitivity
Galois Transformers

**Monadic** small-step interpreter

```
<table>
<thead>
<tr>
<th>type M(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>op x ← e₁ ; e₂</td>
</tr>
<tr>
<td>op return(e)</td>
</tr>
</tbody>
</table>
```
Galois Transformers

**Monadic** small-step interpreter

+ Monad **Transformers**

```
{\texttt{type}} \ M(t)
{\texttt{op}} \ x \leftarrow \ e_1 ; \ e_2
{\texttt{op}} \ \text{return}(e)
```

```
\text{FlowT}[\delta]
```
Galois Transformers

**Monadic** small-step interpreter

+ Monad **Transformers**

+ **Galois Connections**

```
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>type M(t)</td>
</tr>
<tr>
<td>op x ← e₁ ; e₂</td>
</tr>
<tr>
<td>op return(e)</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
</tbody>
</table>
```

```
FlowT[α] γ
```
Galois Transformers

✓ Prototype flow insensitive, flow sensitive and path sensitive CFA—no change to code or proof
Galois Transformers

✓ Prototype flow insensitive, flow sensitive and path sensitive CFA—no change to code or proof

✓ End-to-end correctness proofs given parameters
Galois Transformers

✓ Prototype flow insensitive, flow sensitive and path sensitive CFA—no change to code or proof

✓ End-to-end correctness proofs given parameters

✓ Implemented in Haskell and available on Github
Galois Transformers

✓ Prototype flow insensitive, flow sensitive and path sensitive CFA—no change to code or proof

✓ End-to-end correctness proofs given parameters

✓ Implemented in Haskell and available on Github

✗ Not whole story for path-sensitivity refinement
Galois Transformers

✓ Prototype flow insensitive, flow sensitive and path sensitive CFA—no change to code or proof

✓ End-to-end correctness proofs given parameters

✓ Implemented in Haskell and available on Github

✗ Not whole story for path-sensitivity refinement

✗ Somewhat naive fixpoint iteration strategies
Orthogonal Components

**Property**

\[x/0\]

**Program**

```
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;}}
```

**Value Abstraction**

\(\mathbb{Z} \subseteq \{-, 0, +\}\)

**Implement**

```
analyze : exp → results
analyze(x := æ) := .. x := æ ..
analyze(1 + {e1}{e2}) := .. æ .. e1 .. e2 ..
```

**Results**

```
4: N∈{-,+} x∈{0}
4: N∈{0},x∈{+}
N∈{-,+}, y∈{-,0,+}
N∈{0}, y∈{0,+}
SAFE
```

**Prove Correct**

\([e] \in \llbracket analyze(e)\rrbracket\)
## Contributions

<table>
<thead>
<tr>
<th>Orthogonal Components</th>
<th>Systematic Design</th>
<th>Mechanized Proofs</th>
</tr>
</thead>
</table>
Contributions

**Orthogonal Components**
- Galois Transformers
  - [OOPSLA’15]

**Systematic Design**
- Abstracting Definitional Interpreters
  - [draft]

**Mechanized Proofs**
- Constructive Galois Connections
  - [ICFP’16]
### Systematic Design

#### Property

\[ x / 0 \]

#### Program

\[
\text{analyze : expr } \rightarrow \text{ results}
\]
\[
\text{analyze}(x : \text{æ}) := \\
\text{.. } x \text{ .. æ ..}
\]
\[
\text{analyze}(\text{IF}(\text{æ} \{ e_1 \} \{ e_2 \}) := \\
\text{.. æ .. } e_1 \text{ .. } e_2 ..
\]

#### Value Abstraction

\[ \mathbb{Z} \subseteq \{- , 0 , +\} \]

#### Implement

4: \( N \in \{- , +\} , x \in \{0\} \)
4: \( N \in \{0\} , x \in \{+\} \)

#### Results

4: \( N \in \{- , +\} , y \in \{- , 0 , +\} \)
4: \( N \in \{0\} , y \in \{0 , +\} \)

SAFE

#### Prove Correct

\[ [e] \in [\text{analyze}(e)] \]
Systematic Design

Property

\[ x/0 \]

Program

safe_fun.js

Value Abstraction

\[ \mathbb{Z} \subseteq \{-, 0, +\} \]

Implement

\[
\begin{align*}
\text{analyze} : \text{exp} & \rightarrow \text{results} \\
\text{analyze}(x : \text{exp}) & := \\
& \ldots x \ldots \\
\text{analyze}(\text{IF}(\alpha\{\text{e}_{1}\}\{\text{e}_{2}\}) & := \\
& \ldots \text{\_\_\_\_} \ldots \text{e}_{1} \ldots \text{e}_{2} \ldots
\end{align*}
\]


Results

4: \( N \in \{-, +\}, x \in \{0\} \)
4: \( N \in \{0\}, x \in \{+\} \)
\( N \in \{-, +\}, y \in \{-, 0, +\} \)
\( N \in \{0\}, y \in \{0, +\} \)
SAFE

Prove Correct

\[ [e] \in [\text{analyze}(e)] \]
Systematic Design

Problem: Turn interpreters into program analyzers
Systematic Design

Problem: Turn interpreters into program analyzers

Challenge: Interpreters don’t expose reachable configurations
Systematic Design

**Problem:** Turn interpreters into program analyzers

**Challenge:** Interpreters don’t expose reachable configurations

**State-of-the-art:** Small-step machines or constraint systems
Systematic Design

Problem: Turn interpreters into program analyzers

Challenge: Interpreters don’t expose reachable configurations

State-of-the-art: Small-step machines or constraint systems

Our Insight: Intercept recursion and monad of interpretation
Definitional Abstract Interpreters

Definitional Interpreters $\llbracket e \rrbracket : \text{exp} \rightarrow \text{val}$
Definitional Abstract Interpreters

Definitional Interpreters \( [e] : \text{exp} \rightarrow \text{val} \)

+ 

Open Recursion \( [e]^0 : (\text{exp} \rightarrow \text{val}) \rightarrow (\text{exp} \rightarrow \text{val}) \)
Definitional Abstract Interpreters

Definitional Interpreters \[\llbracket e \rrbracket : \text{exp} \to \text{val}\]

+ 

Open Recursion \[\llbracket e \rrbracket^0 : (\text{exp} \to \text{val}) \to (\text{exp} \to \text{val})\]

+ 

Monads (again) \[\llbracket e \rrbracket^M : \text{exp} \to M(\text{val})\]
Definitional Abstract Interpreters

Definitional Interpreters \[[e] : \text{exp} \to \text{val}\]

+ 

Open Recursion \[[e]^0 : (\text{exp} \to \text{val}) \to (\text{exp} \to \text{val})\]

+ 

Monads (again) \[[e]^M : \text{exp} \to M(\text{val})\]

+ 

Custom Fixpoints \(Y([e]^0^M) \text{ vs } F([e]^0^M)\)
Definitional Abstract Interpreters

✓ Analyzers instantly from definitional interpreters
Definitional Abstract Interpreters

- Analyzers instantly from definitional interpreters
- Soundness w.r.t. big-step reachability semantics
Definitional Abstract Interpreters

✓ Analyzers instantly from definitional interpreters
✓ Soundness w.r.t. big-step reachability semantics
✓ Pushdown analysis inherited from meta-language
Definitional Abstract Interpreters

✓ Analyzers instantly from definitional interpreters
✓ Soundness w.r.t. big-step reachability semantics
✓ Pushdown analysis inherited from meta-language
✓ Implemented in Racket and available on Github
Definitional Abstract Interpreters

✓ Analyzers instantly from definitional interpreters
✓ Soundness w.r.t. big-step reachability semantics
✓ Pushdown analysis inherited from meta-language
✓ Implemented in Racket and available on Github
✗ More complicated meta-theory
Definitional Abstract Interpreters

✓ Analyzers instantly from definitional interpreters
✓ Soundness w.r.t. big-step reachability semantics
✓ Pushdown analysis inherited from meta-language
✓ Implemented in Racket and available on Github
✗ More complicated meta-theory
✗ Monadic, open-recursive interpreters aren’t “simple”
Systematic Design

Property

\[ x / 0 \]

Program

\texttt{safe_fun.js}

Value Abstraction

\[ \mathbb{Z} \subseteq \{-,0,+\} \]

Implement

\begin{align*}
\text{analyze} : \ & \text{exp} \rightarrow \text{results} \\
\text{analyze}(x := \alpha) := \\
& \ldots x := \alpha \\
\text{analyze}(1 \oplus e_1 \{e_1\} e_2) := \\
& \ldots \alpha \ldots e_1 \ldots e_2 \\
\end{align*}

Results

\begin{align*}
4: & \ N \in \{-,+\}, x \in \{0\} \\
4: & \ N \in \{0\}, x \in \{+\} \\
& N \in \{-,+\}, y \in \{-,0,\} \\
& N \in \{0\}, y \in \{0,\} \\
& \text{SAFE} \\
\end{align*}

Prove Correct

\[ [e] \in \{\text{analyze}(e)\} \]

\[ \checkmark \]
Contributions

Orthogonal Components

Galois Transformers [OOPSLA’15]

Systematic Design

Abstracting Definitional Interpreters [draft]

Mechanized Proofs

Constructive Galois Connections [ICFP’16]
Contributions

Orthogonal Components
- Galois Transformers [OOPSLA’15]

Systematic Design
- Abstracting Definitional Interpreters [draft]

Mechanized Proofs
- Constructive Galois Connections [ICFP’16]
Mechanized Proofs

Property

\( x/0 \)

Program

\texttt{safe\_fun.js}

Value Abstraction

\( \mathbb{Z} \subseteq \{-, 0, +\} \)

Implement

\begin{align*}
\text{analyze} : & \quad \text{exp} \to \text{results} \\
\text{analyze}(x := \emptyset) : = & \quad \ldots x \ldots \emptyset \ldots \\
\text{analyze}(\text{IF}(\emptyset)\{e_1\}\{e_2\}) : = & \quad \ldots \emptyset \ldots e_1 \ldots e_2 \ldots
\end{align*}

Results

4: \( N \in \{-, +\}, x \in \{0\} \)
4: \( N \in \{0\}, x \in \{+\} \)

\( N \in \{-, +\}, y \in \{-, 0, +\} \)
\( N \in \{0\}, y \in \{0, +\} \)

SAFE

Prove Correct

\([e] \in [\text{analyze}(e)]\)
Mechanized Proofs

**Implement**

\[
\text{analyze} : \ exp \rightarrow \text{results}
\]
\[
\text{analyze}(x := a) :=
.. x .. a ..
\]
\[
\text{analyze}(\text{IF}(a\{e_1\}e_2)) :=
.. a .. e_1 .. e_2 ..
\]

**Prove Correct**

\[
[e] \in \llbracket \text{analyze}(e) \rrbracket
\]
Mechanized Proofs

Implement

\[
\text{analyze} : \ exp \rightarrow \ results \\
\text{analyze}(x := \varnothing) := \\
\quad \ldots \ x \ldots \ \varnothing \ldots \\
\text{analyze}(\text{IF}(\varnothing)\{e_1\}{e_2}) := \\
\quad \ldots \ \varnothing \ldots \ e_1 \ldots \ e_2 \ldots
\]

Prove Correct

\[
[e] \in [\text{analyze}(e)]
\]

“Calculational Abstract Interpretation” [Cousot99]
Mechanized Proofs

Problem: Calculation, abstraction and mechanization don’t mix
Mechanized Proofs

**Problem:** Calculation, abstraction and mechanization don’t mix

**Challenge:** Transition from specifications to algorithms
Mechanized Proofs

**Problem:** Calculation, abstraction and mechanization don’t mix

**Challenge:** Transition from specifications to algorithms

**State-of-the-art:** Avoid Galois connections in mechanizations
Mechanized Proofs

**Problem:** Calculation, abstraction and mechanization don’t mix

**Challenge:** Transition from specifications to algorithms

**State-of-the-art:** Avoid Galois connections in mechanizations

**Our Insight:** A constructive sub-theory of Galois connections
Calculational Galois Connections

Classical Galois Connections

\[ \alpha : \wp(C) \to A \]
\[ \gamma : A \to \wp(C) \]
Calculational Galois Connections

Classical Galois Connections

\[ \alpha : \mathcal{P}(C) \rightarrow A \]
\[ \gamma : A \rightarrow \mathcal{P}(C) \]

Restricted Form

\[ \eta : C \rightarrow A \]
\[ \mu : A \rightarrow \mathcal{P}(C) \]
Calculational Galois Connections

Classical Galois Connections

\[ \alpha : \wp(C) \to A \]
\[ \gamma : A \to \wp(C) \]

Restricted Form

\[ \eta : C \to A \]
\[ \mu : A \to \wp(C) \]

Monads (again)

\[ \text{calculate : } \wp(A) \to \wp(A) \]
Calculational Galois Connections

Classical Galois Connections

\[ \alpha : \wp(C) \to A \]
\[ \gamma : A \to \wp(C) \]

Restricted Form

\[ \eta : C \to A \]
\[ \mu : A \to \wp(C) \]

Monads (again)

\[ \text{calculate} : \wp(A) \to \wp(A) \]

“has effects”

“no effects”
Calculational Galois Connections

✓ First theory to support calculation and extraction
Calculusional Galois Connections

✓ First theory to support calculation and extraction

✓ Soundness and completeness, also mechanized
Calculational Galois Connections

- First theory to support calculation and extraction
- Soundness and completeness, also mechanized
- Provably less boilerplate than classical theory
Calculational Galois Connections

✓ First theory to support calculation and extraction
✓ Soundness and completeness, also mechanized
✓ Provably less boilerplate than classical theory
✓ Two case studies: calculational AI and gradual typing
Calculational Galois Connections

✓ First theory to support calculation and extraction
✓ Soundness and completeness, also mechanized
✓ Provably less boilerplate than classical theory
✓ Two case studies: calculational AI and gradual typing
✗ Still some reasons not to use Galois connections
Calculational Galois Connections

✓ First theory to support calculation and extraction

✓ Soundness and completeness, also mechanized

✓ Provably less boilerplate than classical theory

✓ Two case studies: calculational AI and gradual typing

✗ Still some reasons not to use Galois connections

✗ Calculating abstract interpreters is still very difficult
Mechanized Proofs

Implement

\[
\begin{align*}
\text{analyze} & : \text{exp} \rightarrow \text{results} \\
\text{analyze}(x := \var) & := \\
& \quad \ldots x \ldots \var \ldots \\
\text{analyze}(\text{IF}(\var)\{e_1\}{e_2}) & := \\
& \quad \ldots \var \ldots e_1 \ldots e_2 \ldots
\end{align*}
\]

Prove Correct

\[ [e] \in [\text{analyze}(e)] \]

"Calculational Abstract Interpretation" [Cousot99]
Mechanized Proofs

Implement

```
analyze : exp \rightarrow results
analyze(x := æ, := .. x := .. æ .. 
analyze(IF(æ) {e₁}{e₂}) := .. æ .. e₁ .. e₂ ..
```

Prove Correct

```
[e] ∈ [analyze(e)]
```

“Calculational Abstract Interpretation” [Cousot99]

AGDA

AGDA
Contributions

Orthogonal Components
Galois Transformers [OOPSLA’15]

Systematic Design
Abstracting Definitional Interpreters [draft]

Mechanized Proofs
Constructive Galois Connections [ICFP’16]
Program Analysis Design