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

x/0

Program

```
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;}}
```

Let's Design an Analysis

Property

x/0

Program

```
int x y;
void safe_fun(int N) {
  if (N≠0) {x := 0;}
  else     {x := 1;}
```

Value Abstraction

$$\mathbb{Z} \sqsubseteq \{-,0,+\}$$

Let's Design an Analysis

Property

x/0

Program

```
int x y;
void safe_fun(int N) {
  if   (N≠0)  {x := 0;}
  else        {x := 1;}
```

Implement

Value Abstraction

$\mathbb{Z} \sqsubseteq \{-,0,+\}$

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

Let's Design an Analysis

Property

x/0

Program

```
0: int x y;
1: void s          N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
```

Value Abstraction

$\mathbb{Z} \sqsubseteq \{-,0,+\}$

Implem

```
analyze : exp →
analyze(x := æ)
   .. x .. æ .
analyze(IF(æ){e
   .. æ .. e₁
```

Results

N ∈ {-,0,+}
x ∈ {0,+}
y ∈ {-,0,+}

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

Let's Design an Analysis

Property

x/0

Program

```
int x y;
void safe_fun(int N){
  if (N≠0) {x := 0;}
  else     {x := 1;}
```

Prove Correct

Value Abstraction

$\mathbb{Z} \sqsubseteq \{-,0,+\}$

Implem

```
analyze : exp →
analyze(x := æ)
   .. x .. æ .
analyze(IF(æ){e
   .. æ .. e1
```

$$[\![e]\!] \in [\![\mathrm{analyze}(e)]\!]$$

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

$\mathbb{Z} \sqsubseteq \{-, 0, +\}$

Implement

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

Results

N ∈ {-,0,+}
x ∈ {0,+}
y ∈ {-,0,+}

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

Prove Correct

$[\![e]\!] \in [\![\mathrm{analyze}(e)]\!]$

Let's Design an Analysis

```
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;}}
```

```
N ∈ {-,0,+}
x ∈ {0,+}
y ∈ {-,0,+}

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

Flow-insensitive

$$\texttt{results} : \texttt{var} \mapsto \wp(\{-,0,+\})$$

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;}}
```

```
4:     x ∈ {0,+}
4.T: N ∈ {-,+}
5.F: x ∈ {0,+}

N,y ∈ {-,0,+}

UNSAFE: {100/x}
```

Flow-sensitive

$$\texttt{results} : \texttt{loc} \mapsto (\texttt{var} \mapsto \wp(\{\texttt{-},\texttt{0},\texttt{+}\}))$$

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;}}
```

```
4: N∈{-,+},x∈{0}
4: N∈{0},x∈{+}

N∈{-,+},y∈{-,0,+}
N∈{0},y∈{0,+}

SAFE
```

Path-sensitive

results : loc ↦ ℘(var ↦ ℘({-,0,+}))

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

ℤ ⊑ {-,0,+}

Implement

```
analyze : exp → results
analyze(x : ...) :=
   .. x .. ..
analyze(IF(æ){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⟧ ∈ ⟦analyze(e)⟧

Let's Design an Analysis

Property

x/0

Program

safe_fun.js ?

Value Abstraction

$\mathbb{Z} \sqsubseteq \{-,0,+\}$ ✓

Implement

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

✗

Results

```
4: N∈{-,+},x∈{0}
4: N∈{0},x∈{+}

N∈{-,+},y∈{-,0,+}
N∈{0},y∈{0,+}
```

SAFE

Prove Correct

$⟦e⟧ \in ⟦analyze(e)⟧$ ✗

Contributions

Orthogonal Components	**Systematic Design**	**Mechanized Proofs**
Galois Transformers [OOPSLA’15]	Abstracting Definitional Interpreters [draft]	Constructive Galois Connections [ICFP’16]

Contributions

Orthogonal Components	Systematic Design	Mechanized Proofs
Galois Transformers [OOPSLA'15]	Abstracting Definitional Interpreters [draft]	Constructive Galois Connections [ICFP'16]

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

$\mathbb{Z} \sqsubseteq \{-,0,+\}$

Implement

```
analyze : exp → results
analyze(x : ...) :=
    .. x .. ..
analyze(IF(æ){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⟧ ∈ ⟦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

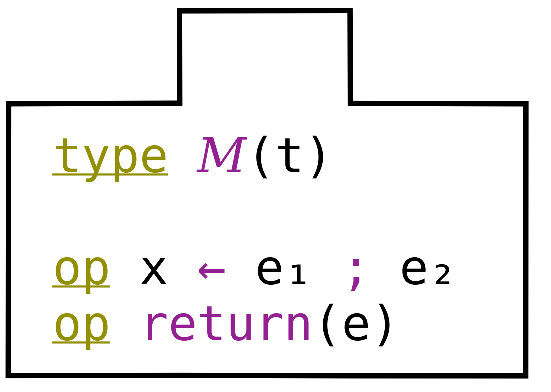
```
type M(t)

op x ← e₁ ; e₂
op return(e)
```

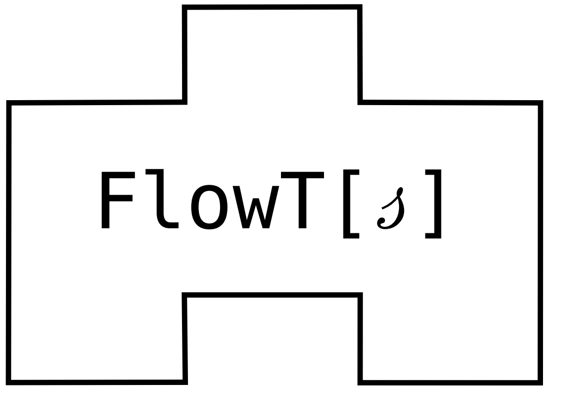
Galois Transformers

Monadic small-step interpreter

+

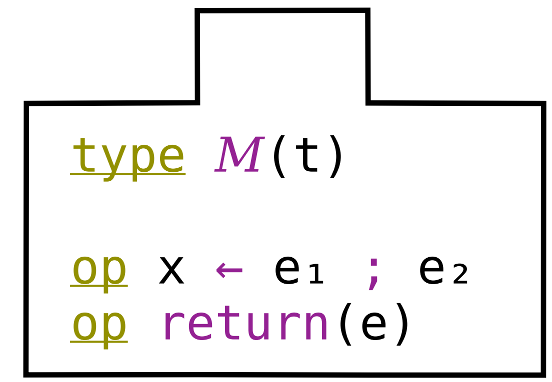
Monad **Transformers**





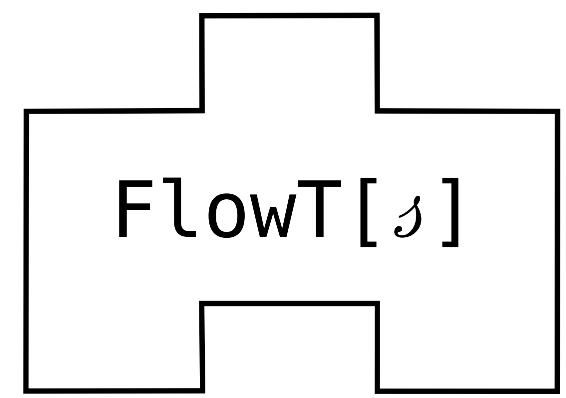
Galois Transformers

Monadic small-step interpreter



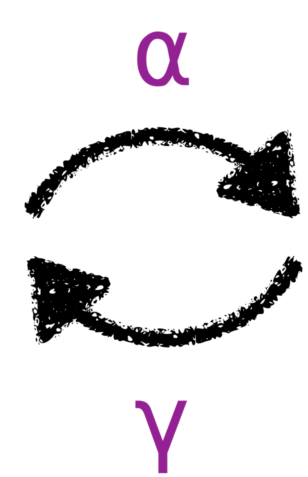
+

Monad **Transformers**



+

Galois Connections



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

```
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

$\mathbb{Z} \sqsubseteq \{-,0,+\}$

Implement

```
analyze : exp → sults
analyze(x := æ :=
    .. x   æ ..
analyze(I   {e₁}{e₂}) :=
    .. æ .  e₁ .. e₂ ..
```

Results

```
4: N∈{-,+},x∈{0}
4: N∈{0},x∈{+}

N∈{-,+},y∈{ 0,+}
N∈{0},y∈{0,+}

SAFE
```

Prove Correct

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

Contributions

Orthogonal Components	**Systematic Design**	**Mechanized Proofs**
Galois Transformers [OOPSLA'15]	Abstracting Definitional Interpreters [draft]	Constructive Galois Connections [ICFP'16]

Contributions

Orthogonal Components	Systematic Design	Mechanized Proofs
Galois Transformers [OOPSLA'15]	Abstracting Definitional Interpreters [draft]	Constructive Galois Connections [ICFP'16]

Systematic Design

Property

x/0

Program

Value Abstraction

$\mathbb{Z} \sqsubseteq \{-,0,+\}$

Implement

```
analyze : exp → results
analyze(x : ℤ) :=
   .. x .. ..
analyze(IF(æ){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⟧ ∈ ⟦analyze(e)⟧

Systematic Design

Property

```
x/0
```

Program

```
safe_fun.js
```

?

Value Abstraction

$\mathbb{Z} \sqsubseteq \{-,0,+\}$

Implement

```
analyze : e p → results
analyze(x :   ) :=
   .. x ..   ..
analyze(I   e)   ₁}{e₂}) :=
   .. æ  . e₁ .. e₂ ..
```

Results

```
4: N∈{-,+},x∈{0}
4: N∈{0},x∈{+}

N∈{-,+},y∈{-,0,+}
N∈{0},y∈{0,+}
```

SAFE

Prove Correct

⟦e⟧ ∈ ⟦a lyze(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 ⟦e⟧ : exp → val

Definitional Abstract Interpreters

Definitional Interpreters $[\![e]\!] : \mathtt{exp} \to \mathtt{val}$

\+

Open Recursion $[\![e]\!]^{o} : (\mathtt{exp} \to \mathtt{val}) \to (\mathtt{exp} \to \mathtt{val})$

Definitional Abstract Interpreters

Definitional Interpreters $[\![e]\!]$: exp → val

+

Open Recursion $[\![e]\!]^{o}$: (exp → val) → (exp → val)

+

Monads (again) $[\![e]\!]^{M}$: exp → M(val)

Definitional Abstract Interpreters

Definitional Interpreters	$[\![e]\!]$: exp → val
+	
Open Recursion	$[\![e]\!]^{O}$: (exp → val) → (exp → val)
+	
Monads (again)	$[\![e]\!]^{M}$: exp → M(val)
+	
Custom Fixpoints	Y($[\![e]\!]^{OM}$) vs F($[\![e]\!]^{OM}$)

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

`safe_fun.js` ?

Value Abstraction

$\mathbb{Z} \sqsubseteq \{-,0,+\}$

Implement

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

Results

```
4: N∈{-,+},x∈{0}
4: N∈{0},x∈{+}

N∈{-,+},y∈{-,0,+}
N∈{0},y∈{0,+}
```

SAFE

Prove Correct

$[\![e]\!] \in [\![analyze(e)]\!]$

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]

Mechanized Proofs

Property

```
x/0
```

Program

```
safe_fun.js
```

Value Abstraction

$\mathbb{Z} \sqsubseteq \{-,0,+\}$

Implement

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

Results

```
4: N∈{-,+},x∈{0}
4: N∈{0},x∈{+}

N∈{-,+},y∈{-,0,+}
N∈{0},y∈{0,+}

SAFE
```

Prove Correct

$[\![e]\!] \in [\![analyze(e)]\!]$?

Mechanized Proofs

Implement

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

↤

Prove Correct

⟦e⟧ ∈ ⟦analyze(e)⟧

Mechanized Proofs

Implement

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

↤

Prove Correct

$$[\![e]\!] \in [\![\mathtt{analyze}(e)]\!]$$

“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

$$\alpha : \wp(C) \rightarrow A$$
$$\gamma : A \rightarrow \wp(C)$$

+

Restricted Form

$$\eta : C \rightarrow A$$
$$\mu : A \rightarrow \wp(C)$$

Calculational Galois Connections

Classical Galois Connections

$\alpha : \wp(C) \rightarrow A$
$\gamma : A \rightarrow \wp(C)$

+

Restricted Form

$\eta : C \rightarrow A$
$\mu : A \rightarrow \wp(C)$

+

Monads (again)

`calculate` $: \wp(A) \rightarrow \wp(A)$

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)

$$\texttt{calculate} : \wp(A) \to \wp(A)$$

"has effects" "no effects"

Calculational Galois Connections

✓ First theory to support calculation and extraction

Calculational 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

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

↤

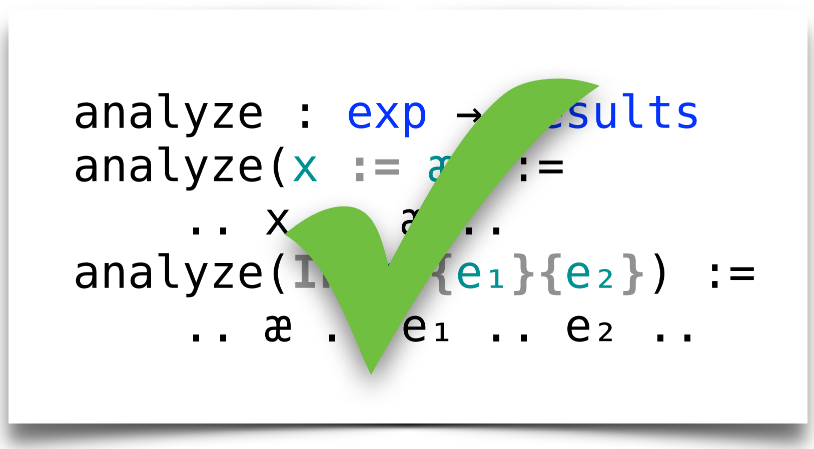
Prove Correct

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

"Calculational Abstract Interpretation" [Cousot99]

Mechanized Proofs

Implement



↤

Prove Correct

AGDA

AGDA

"Calculational Abstract Interpretation" [Cousot99]

Contributions

Orthogonal Components	**Systematic Design**	**Mechanized Proofs**
Galois Transformers [OOPSLA'15]	Abstracting Definitional Interpreters [draft]	Constructive Galois Connections [ICFP'16]

Program Analysis Design