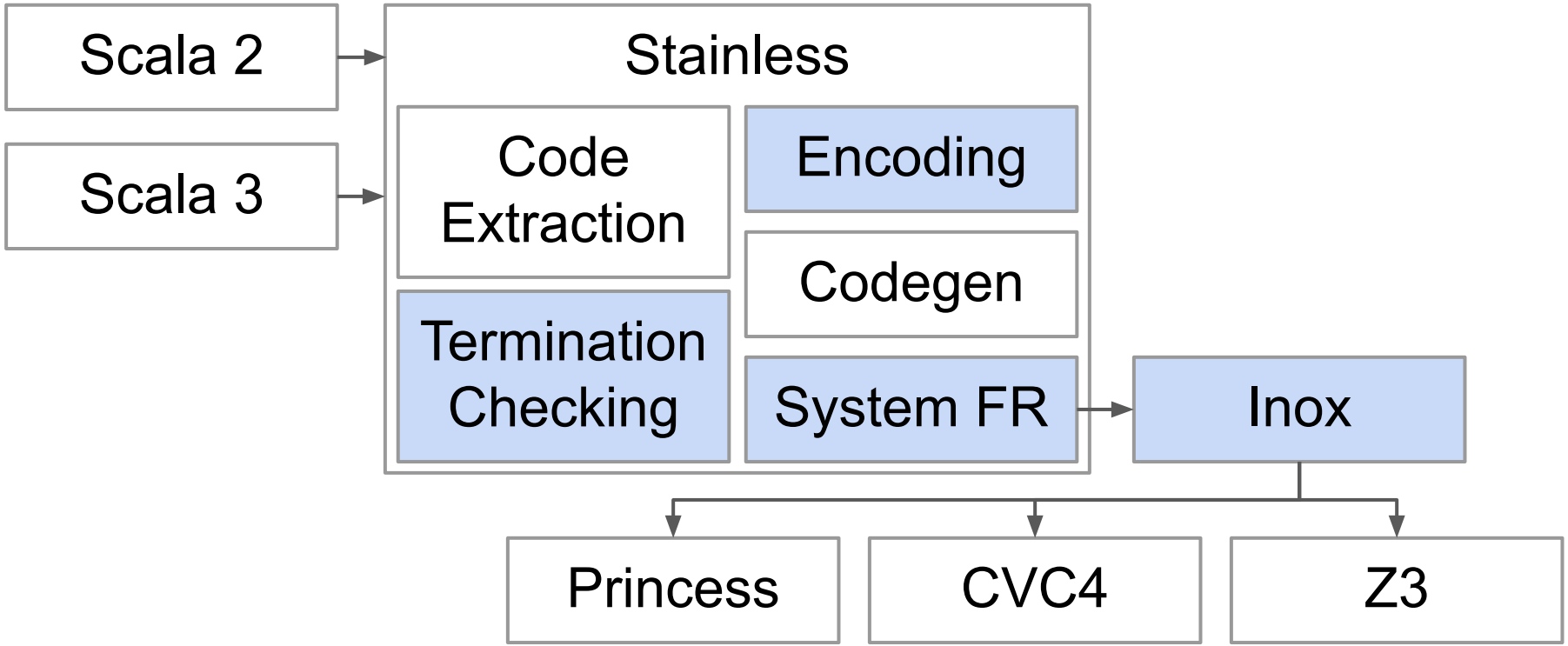
Stainless System

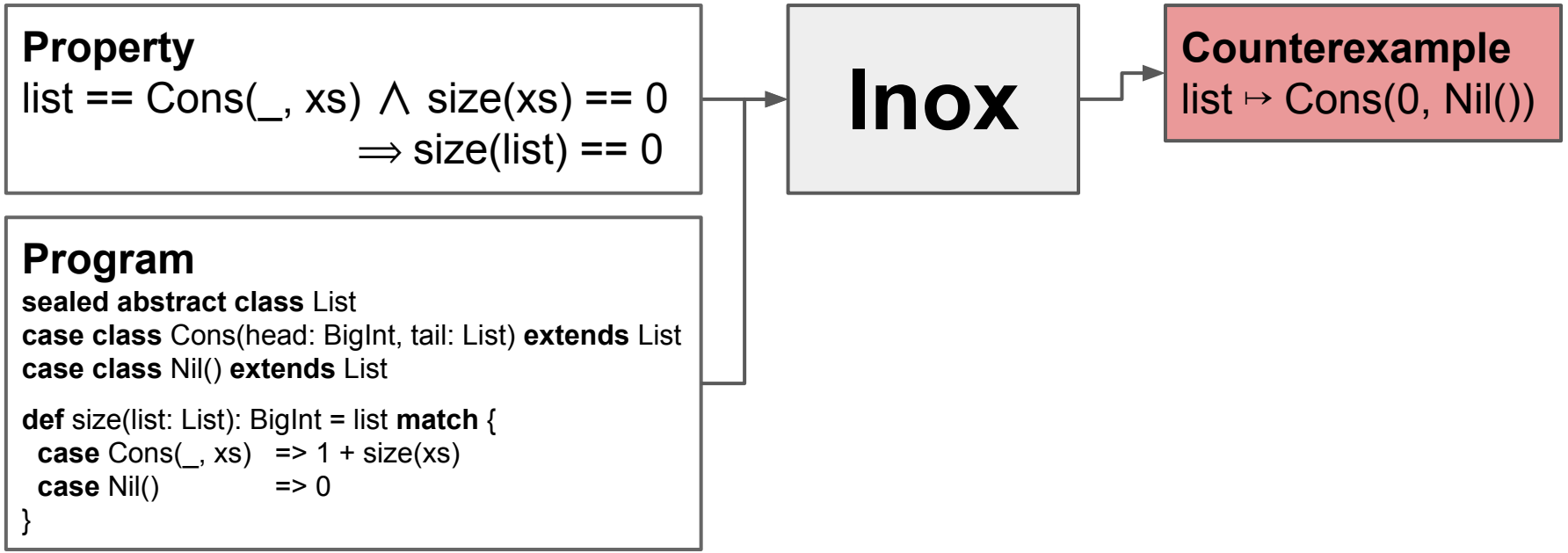
System Overview



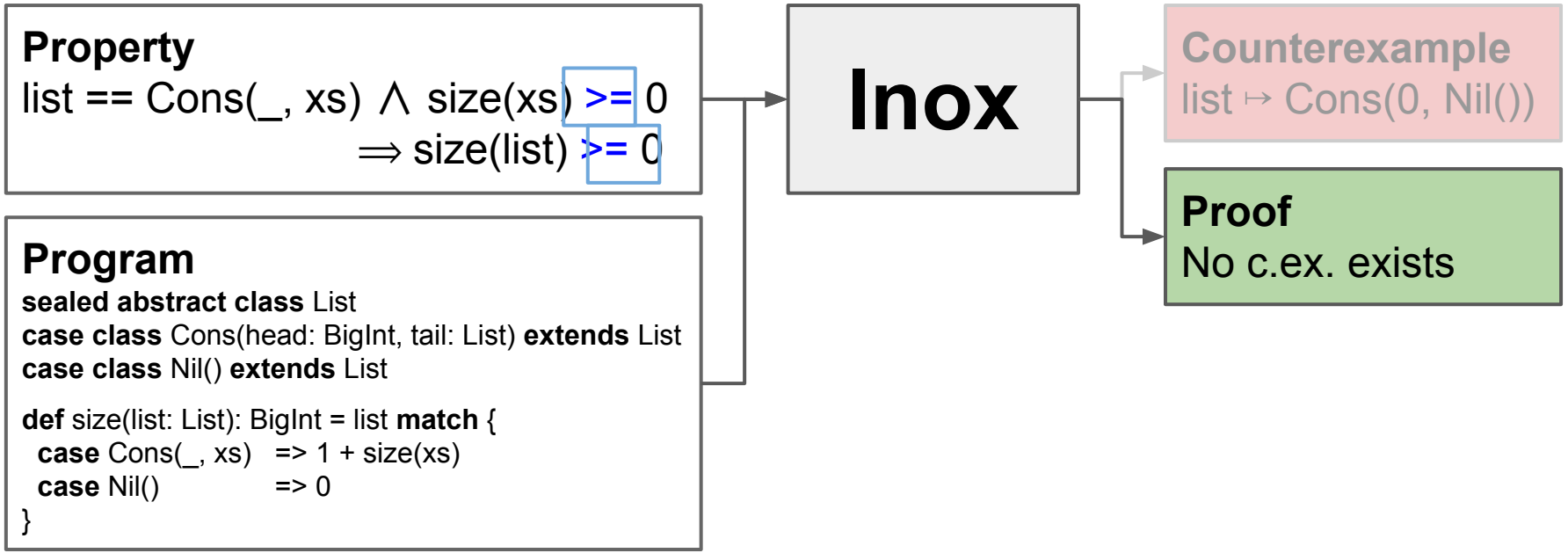
Inox

SMT-based counterexample-finding

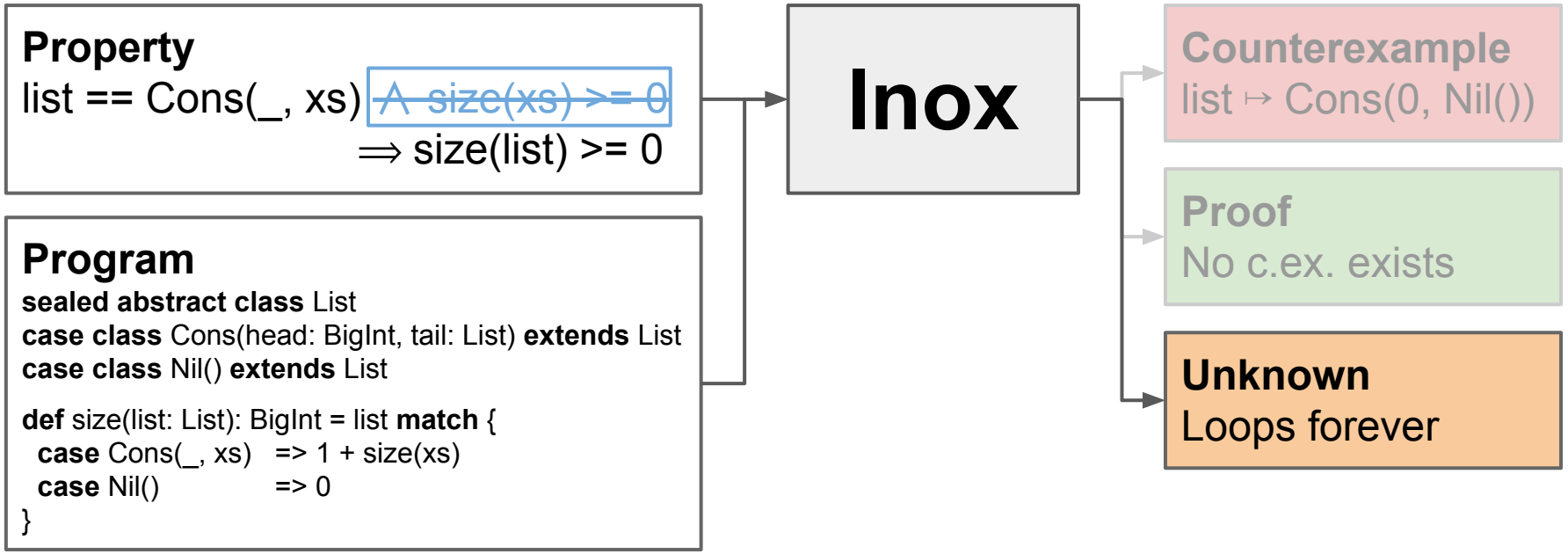
Inox System


Property
list == Cons(_, xs) ∧ size(xs) == 0
⟹ size(list) == 0
Program
sealed abstract class List
case class Cons(head: BigInt, tail: List) extends List
case class Nil() extends List
def size(list: List): BigInt = list match {
case Cons(_, xs) => 1 + size(xs)
case Nil() => 0
}
Inox
Counterexample
list ↦ Cons(0, Nil())

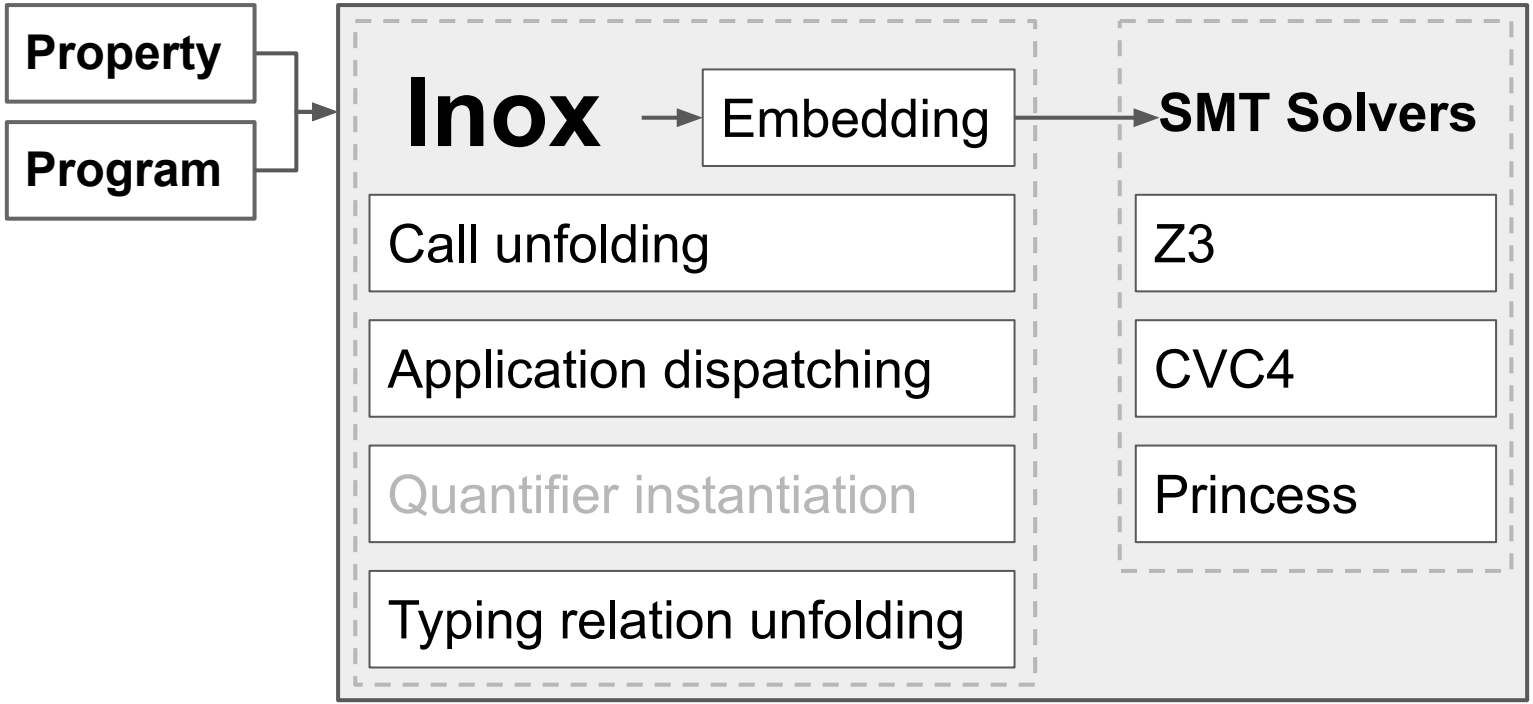
Inox System


Property
list == Cons(_, xs) ∧ size(xs) >= 0
⟹ size(list) >= 0
Program
sealed abstract class List
case class Cons(head: BigInt, tail: List) extends List
case class Nil() extends List
def size(list: List): BigInt = list match {
case Cons(_, xs) => 1 + size(xs)
case Nil() => 0
}
Inox
Counterexample
list ↦ Cons(0, Nil())
Proof
No c.ex. exists

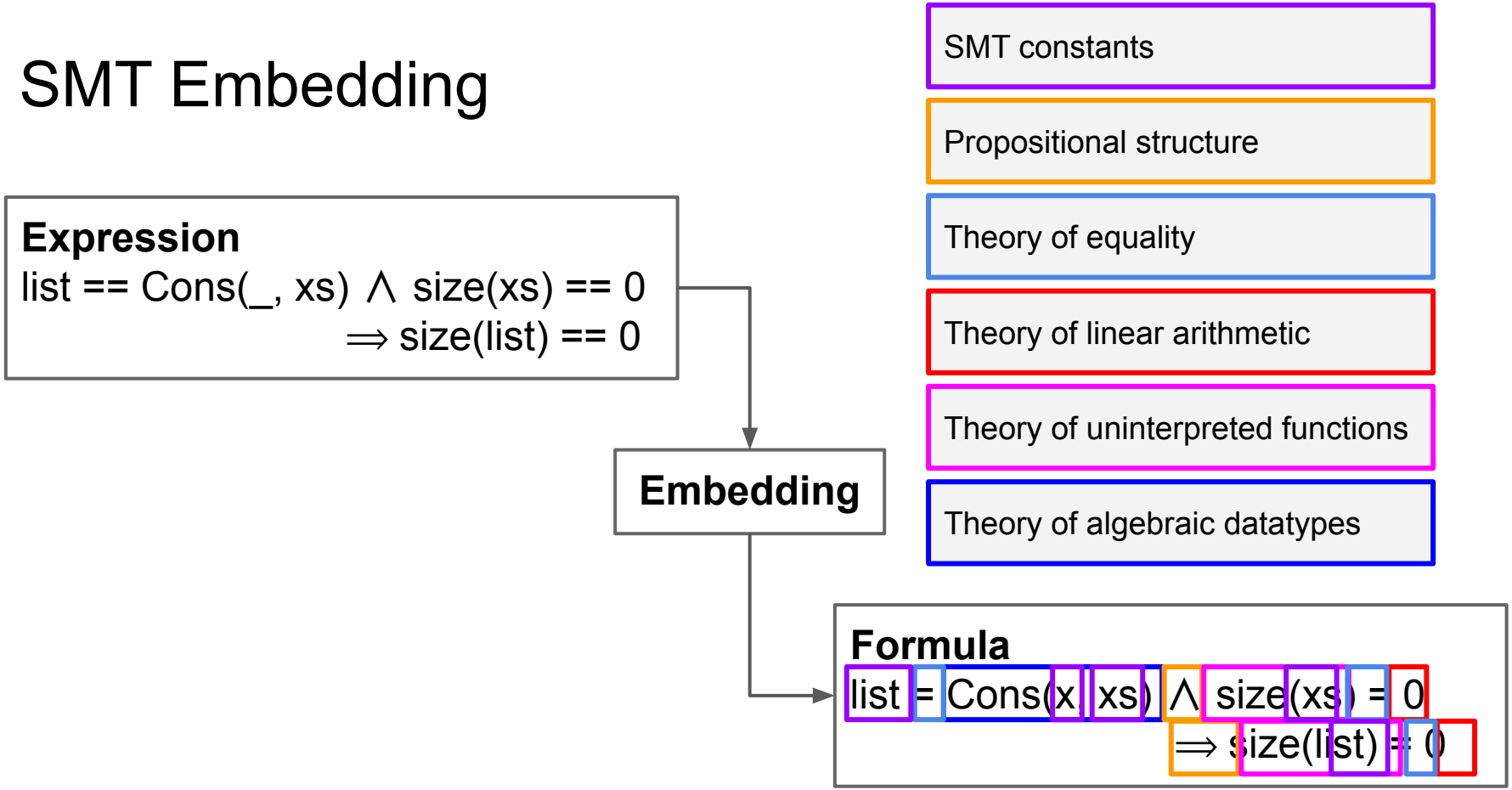
Inox System


Property
list == Cons(_, xs) ~~∧ size(xs) >= 0~~
⟹ size(list) >= 0
Program
sealed abstract class List
case class Cons(head: BigInt, tail: List) extends List
case class Nil() extends List
def size(list: List): BigInt = list match {
case Cons(_, xs) => 1 + size(xs)
case Nil() => 0
}
Inox
Counterexample
list ↦ Cons(0, Nil())
Proof
No c.ex. exists
Unknown
Loops forever

SMT-Backed Solving



SMT Embedding



SMT Embedding Procedure

b : "blocker" SMT constant
e : expression to be embedded
t : encoded SMT term
Φ : clause set

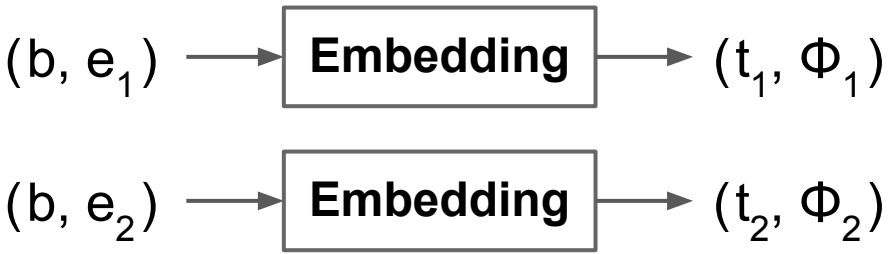
Embedding is precise[1] wrt. operational semantics:

given **M** ⊧ **Φ** ∪ **{b}**, we have **M(e)** →* **M(t)**

[1]: modulo function calls and applications

SMT Embedding: Recursive Procedure

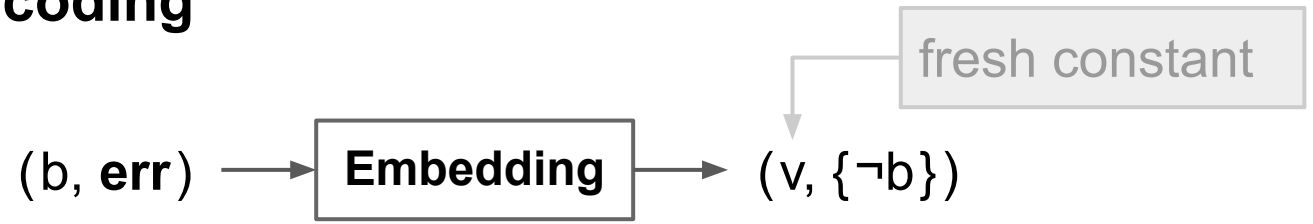
where



SMT Embedding: Blocker Constant

- Corresponds to the “path condition” of the expression
- b is **true** iff evaluation encounters the expression

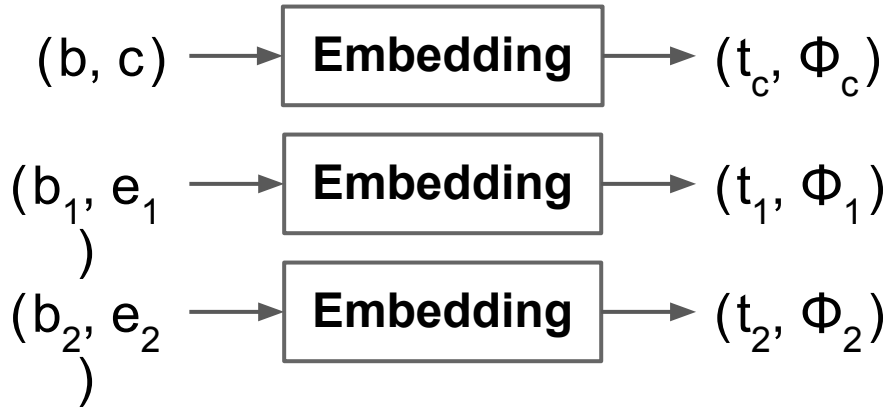
Stuck term encoding



As expected, there is no model **M** ⊨ **Φ** ∪ **{¬b}** (since **err** is stuck and will not evaluate to a value).

SMT Embedding: Branching

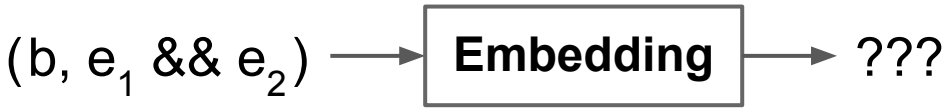
where



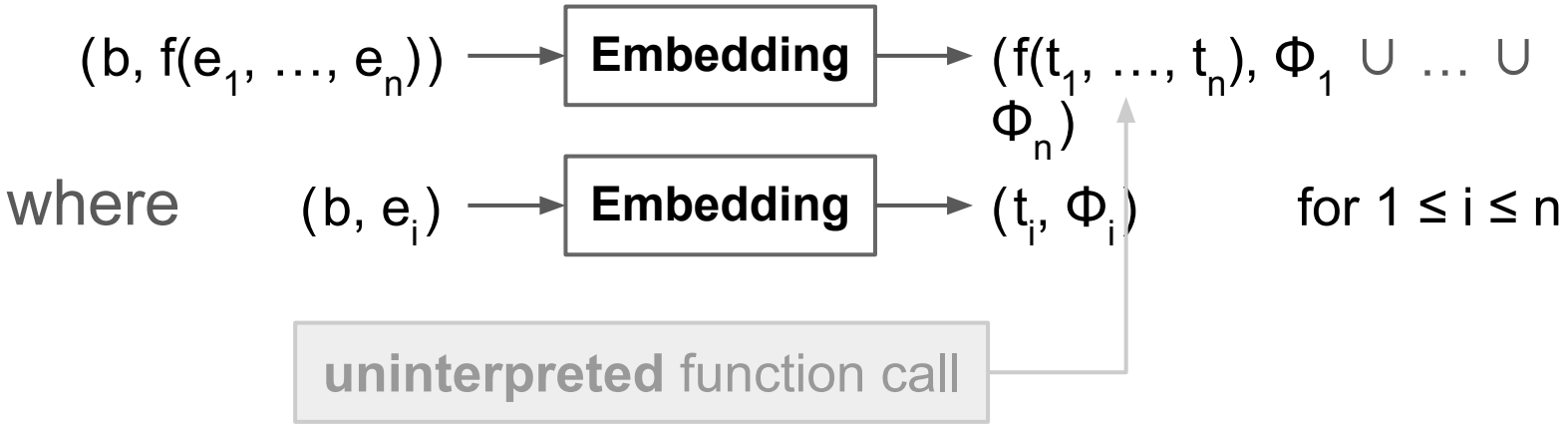
v, b_1, b_2 are fresh constants

$$\Phi_{\mathbf{if}} = \{\ b \wedge t_c \Leftrightarrow b_1,\ b_1 \Rightarrow v = t_1,\ b \wedge \neg t_c \Leftrightarrow b_2,\ b_2 \Rightarrow v = t_2\ \}$$

SMT Embedding: Exercise



SMT Embedding: Function Calls



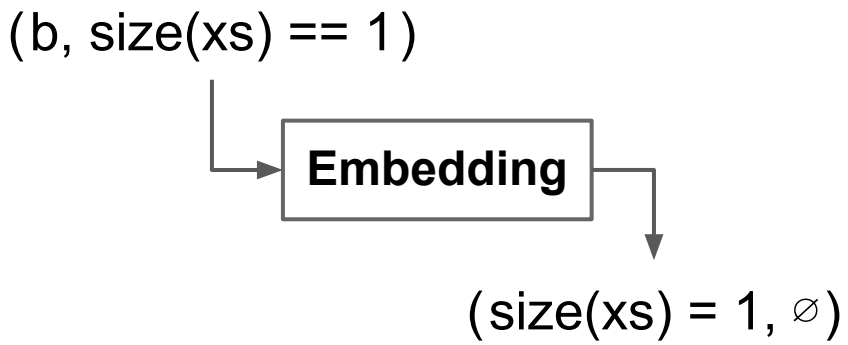
Embedding is only an under-approximation!

SMT Embedding: Uninterpreted Function Calls

Program

```
sealed abstract class List
case class Cons(head: BigInt, tail: List) extends List
case class Nil() extends List

def size(list: List): BigInt = list match {
  case Cons(_, xs)   => 1 + size(xs)
  case Nil()         => 0
}
```



Consider model **M** = {b ↦ **true**, xs ↦ Nil(), size(Nil()) ↦ 1}

We have **M** ⊨ ∅ ∪ **{b}** and **M** ⊨ **size(xs) = 1,**
but **M(size(xs) == 1) →* false**

Blocking Uninterpreted Function Calls

Observe blocker constants “encode” execution branching

Idea since function call embedding is not precise, block relevant branches by negating blocker constants

Consider expression **e** and fresh constant **b** with embedding (**t**, **Φ**).

Given model $\mathbf{M} \models \Phi \cup \{b\} \cup \{\neg b_c \mid b_c \text{ associated with function call in } e\}$,

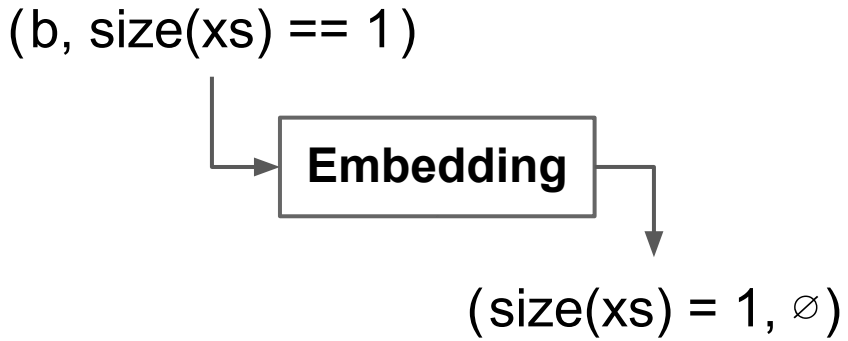
M is such that $\mathbf{M(e)} \rightarrow^* \mathbf{M(t)}$.

Unfolding Function Calls

Program

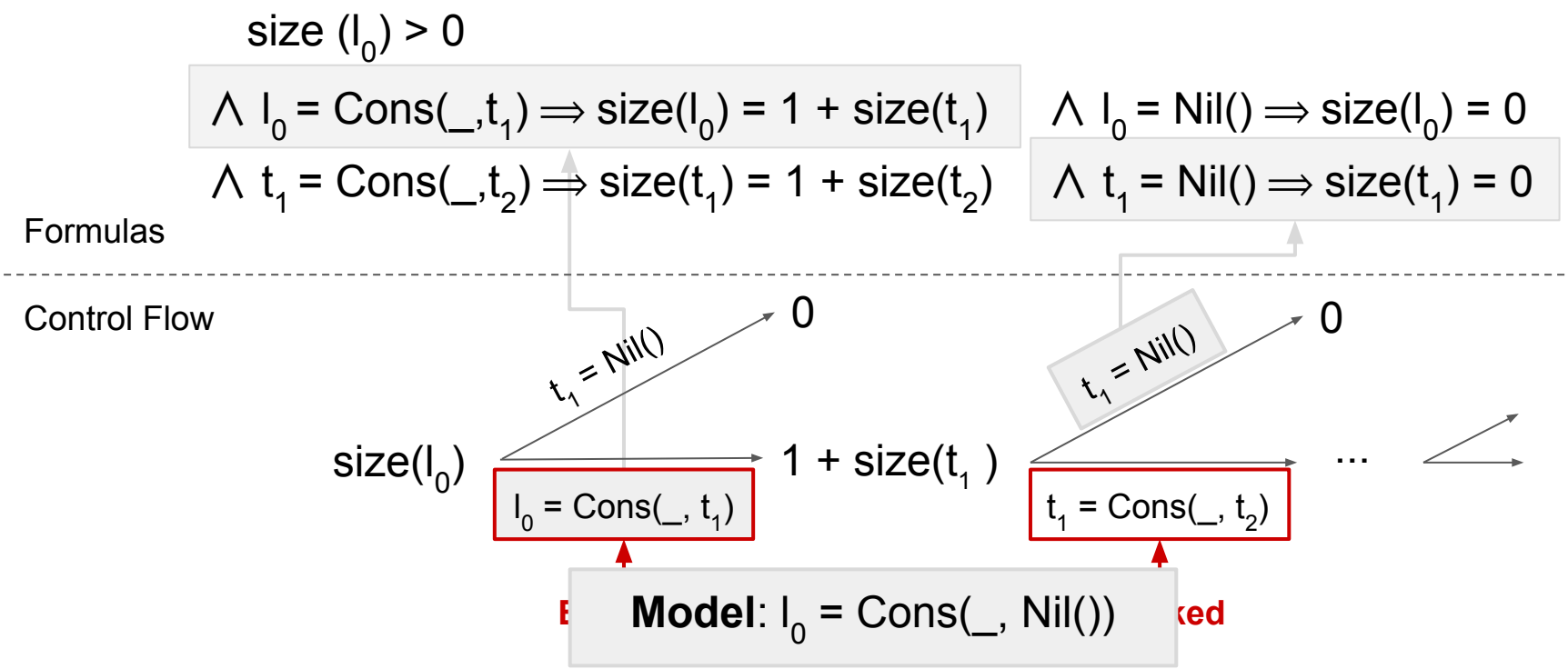
```
sealed abstract class List
case class Cons(head: BigInt, tail: List) extends List
case class Nil() extends List

def size(list: List): BigInt = list match {
  case Cons(_, xs)   => 1 + size(xs)
  case Nil()         => 0
}
```



- Embed body of size function with blocker constant **b**
- Substitute size function parameters with call arguments
- Allow models which satisfy **b**

Unfolding Function Calls: Example



Counterexample Finding Procedure

Given a property **p**

1. Compute embedding of (**b**, **p**) where **b** is a fresh constant: ($\mathbf{t_p}$, $\mathbf{\Phi_p}$)
2. Let $\mathbf{\Phi} := \mathbf{\Phi_p} \cup \{ \neg \mathbf{t_p} \} \cup \{ \mathbf{b} \}$ and $\mathbf{F} := \{$ all function calls in **p** $\}$
3. If there exists $\mathbf{M} \models \mathbf{\Phi} \cup \{ \neg \mathbf{b_c} \mid \mathbf{b_c}$ is a blocker for a call in **F** $\}$
   then output the counterexample **M**
4. Compute unfolding $\mathbf{\Phi_f}$ of some call $\mathbf{f} \in \mathbf{F}$
5. Let $\mathbf{\Phi} := \mathbf{\Phi} \cup \mathbf{\Phi_f}$ and $\mathbf{F} := \mathbf{F} \setminus \{ \mathbf{f} \} \cup \{$ all function calls in the unfolding of **f** $\}$
6. Return to step 3

Counterexample Finding: Guarantees

Counterexample finding is sound

Evaluation under reported counterexamples will terminate with **false**.

Counterexample finding is complete

If a counterexample exists, then the procedure will find one.

Counterexample Inexistence

Given a property **p**

1. Compute embedding of (**b**, **p**) where **b** is a fresh constant: ($\mathbf{t_p}$, $\mathbf{\Phi_p}$)
2. Let $\mathbf{\Phi} := \mathbf{\Phi_p} \cup \{ \neg \mathbf{t_p} \} \cup \{ \mathbf{b} \}$ and $\mathbf{F} := \{$ all function calls in **p** $\}$
3. If there exists no $\mathbf{M} \models \mathbf{\Phi}$
   then output that no counterexample exists
4. If there exists $\mathbf{M} \models \mathbf{\Phi} \cup \{ \neg \mathbf{b_c} \mid \mathbf{b_c}$ is a blocker for a call in **F** $\}$
   then output the counterexample **M**
5. Compute unfolding $\mathbf{\Phi_f}$ of some call $\mathbf{f} \in \mathbf{F}$
6. Let $\mathbf{\Phi} := \mathbf{\Phi} \cup \mathbf{\Phi_f}$ and $\mathbf{F} := \mathbf{F} \setminus \{ \mathbf{f} \} \cup \{$ all function calls in the unfolding of **f** $\}$
7. Return to step 4

Counterexample Finding: Guarantees

Counterexample finding is sound

Evaluation under reported counterexamples will terminate with **false**.

Counterexample finding is complete

If a counterexample exists, then the procedure will find one.

Counterexample finding is sound for proofs

The procedure is correct when reporting that no counterexample exists.

Higher-Order Functions

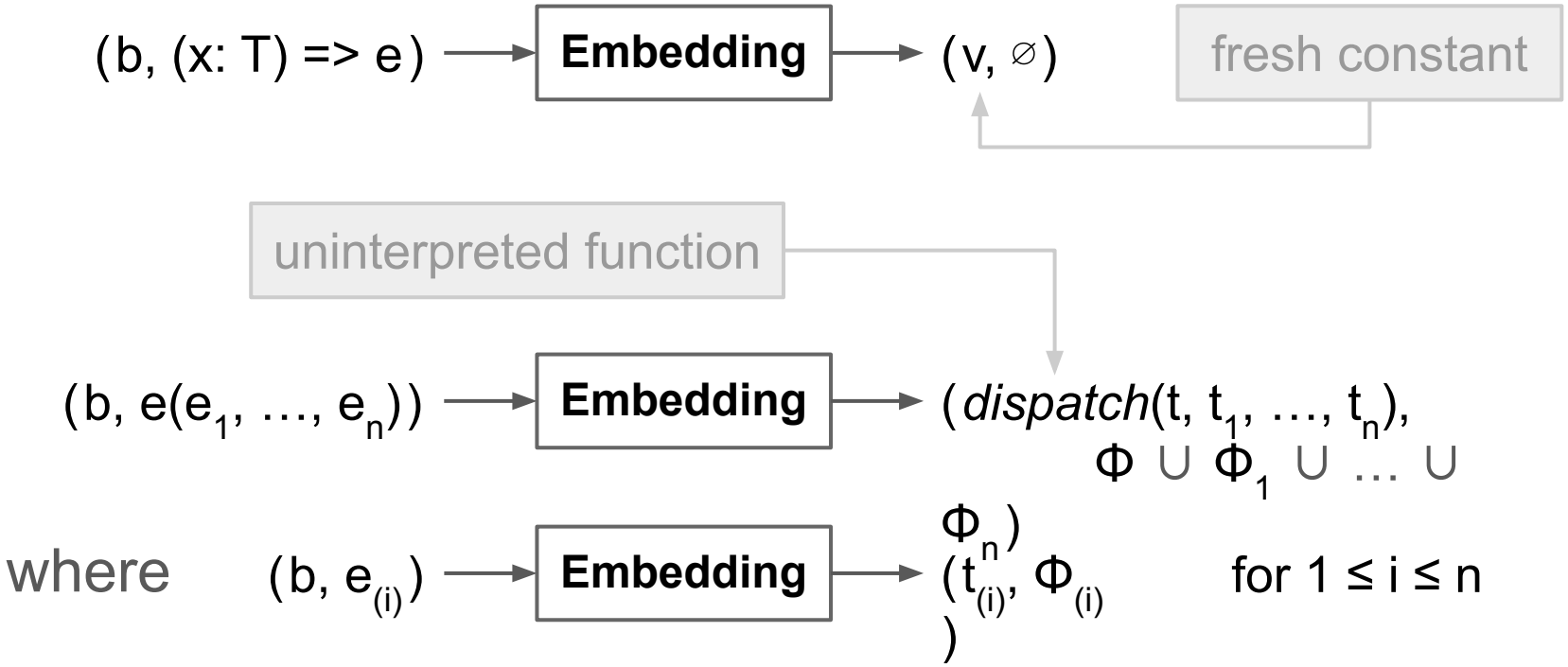
Consider the following (incorrect) definition of List exists:

```
def exists[T](list: List[T], p: T => Boolean): Boolean = list match {
  case Cons(x, xs) => p(x) && exists(xs, p)
  case Nil() => false
}
```

How can we find a counterexample to the forall/exists correspondence

```
exists(list, p) == !forall(list, x => !p(x))
```

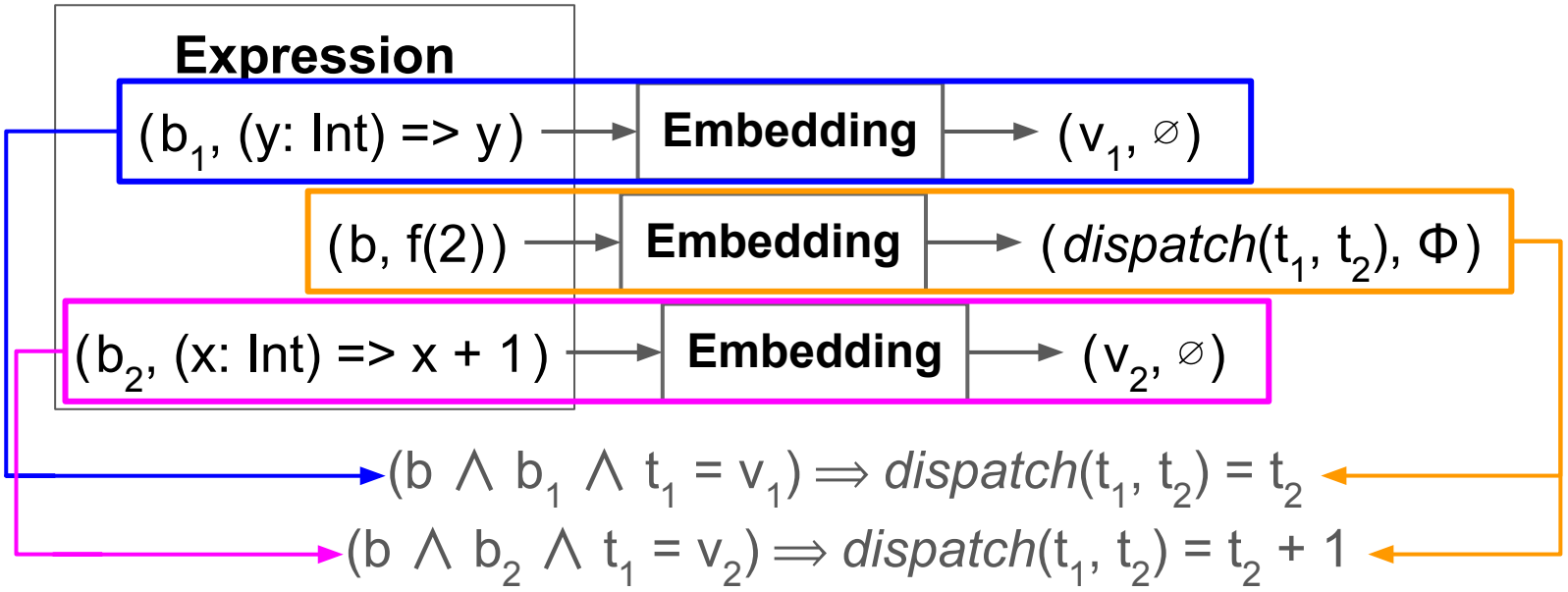
Higher-Order Functions: Embeddings



Higher-Order Functions: Unfolding

Challenge: we can't statically determine the lambda body to unfold.

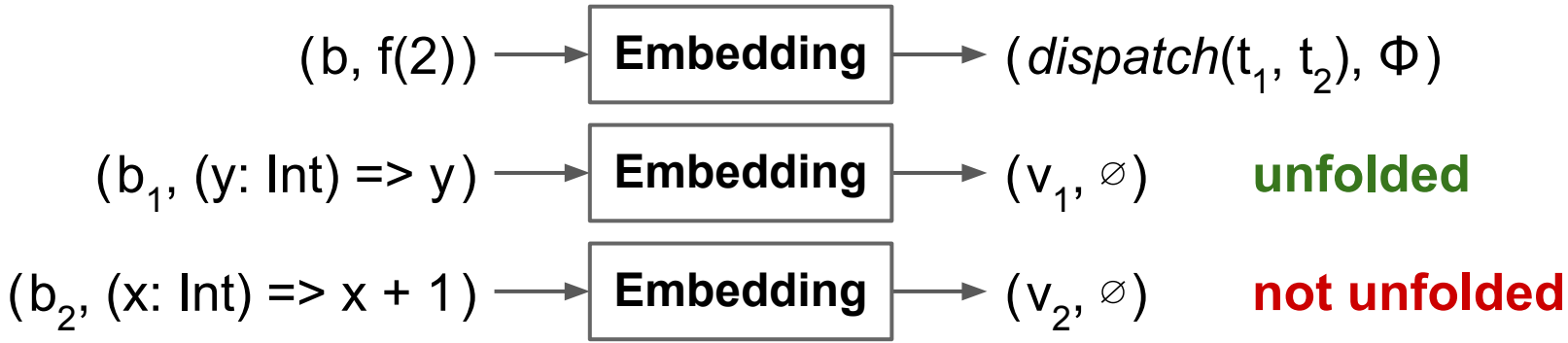
Solution: perform *dynamic dispatch* over the set of embedded lambdas.



Higher-Order Functions: Blocking

Challenge: we can't statically know when the *right* lambda was unfolded.

Solution: encode the *right unfolding* as a logical constraint.

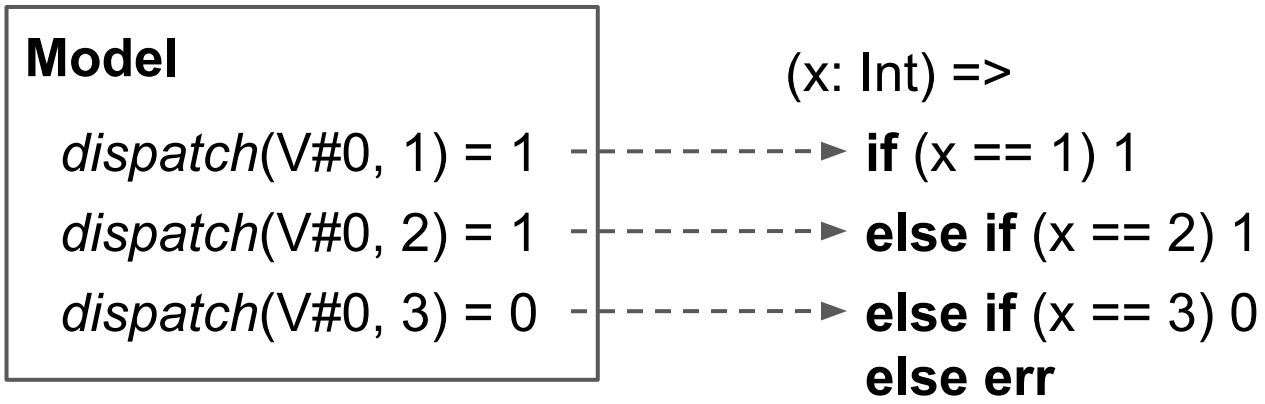


Constraint: $t_1 \in \{ \text{known lambdas} \} \setminus \{ \text{unfolded lambdas} \} \Longrightarrow \neg b$

Propositional encoding: $((b_2 \wedge t_1 = v_2) \wedge \neg(b_1 \wedge t_1 = v_1)) \Longrightarrow \neg b$

Extracting Counterexamples from Models

Idea: use interpretation of *dispatch* symbol



Dependent Types

Types that depend on values

- Refinement type: { x: τ | p[x] }
  - Even integers: { x: Int | x % 2 == 0 }
  - Sorted lists: { l: List[Int] | isSorted(l) }
- Dependent function type: (x: τ_1) => τ_2 [1]
  - Increasing functions: (x: Int) => { y: Int | y > x }
  - Proof of associativity:

    (x: Int, y: Int, z: Int) => { u: Unit | f(x, f(y, z)) == f(f(x, y), z) }

[1]: more commonly known as *Pi-type* (Πx: τ_1. τ_2)

Embedding the Typing Relation

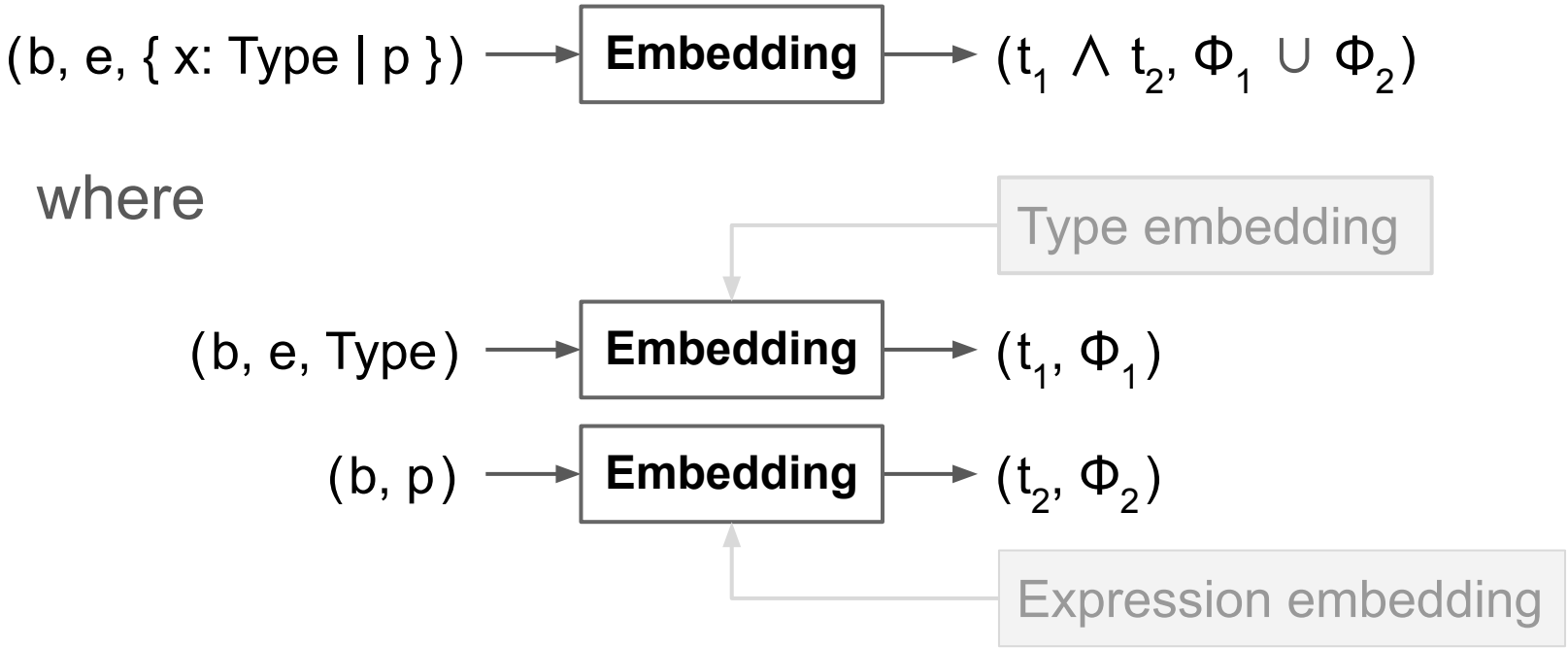
Idea: leverage types in the expression by embedding the typing relation

Embedding *under-approximates* the typing relation:

if there exists **C** such that **C(e) : Type**,
then there exists **M ⊨ Φ ∪ {b, t}**

The embedding can be used for proofs, but not counterexamples

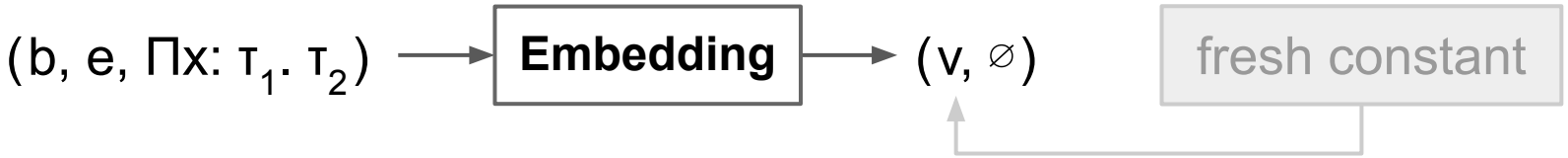
Embedding Refinement Types



Embedding Sigma-types

Blackboard

Embedding Pi-types



Fresh constant guarantees **under-approximation**, but can we do better?

Yes: use similar approach to application dispatching

Blackboard

Theoretical Results

	First-Order	Higher-Order	Dependent Types
Sound for Counterexamples	**Yes**	**Yes**	**No**
Complete for Counterexamples	**Yes**	**Yes**	**Yes**
Sound for Proofs	**Yes**	**Yes**	**Yes**

System FR

Dependent type checking

Beyond Counterexamples

- Counterexample inexistence is not enough
  - Termination is not guaranteed
  - Crashes can still occur
- Beyond counterexample inexistence is verification
- Dependent type checking enables verification
  - Allows showing total correctness
  - Supports expressive specifications
  - Predictable process

Bidirectional Type Checking

Mutually recursive judgements:

- **type checking** judgement: $\Gamma \vdash e \Downarrow \tau$

$$\frac{\Gamma \vdash e.\_1 \Downarrow \tau_1 \quad \Gamma \vdash e.\_2 \Downarrow \tau_2\{x \mapsto e.\_1\}}{\Gamma \vdash e \Downarrow \Sigma x{:}\ \tau_1.\ \tau_2}$$

- **type inference** judgement: $\Gamma \vdash e \Uparrow \tau$

$$\frac{\Gamma \vdash f \Uparrow \Pi x{:}\ \tau_1.\ \tau_2 \qquad \Gamma \vdash e \Downarrow \tau_1}{\Gamma \vdash f(e) \Uparrow \tau_2\{x \mapsto e\}}$$

Bidirectional Type Checking

Mutually recursive judgements:

- **type formation** judgement: $\Gamma \vdash \tau$ type

$$\frac{\Gamma \vdash \tau \text{ type} \qquad \Gamma, x{:}\,\tau \vdash p \Downarrow \text{Boolean}}{\Gamma \vdash \{\, x{:}\,\tau \mid p \,\} \text{ type}}$$

- **truth** judgement: $\Gamma \vdash p$ **true**

$$\frac{\Gamma \vdash p \Downarrow \text{Boolean} \quad p \text{ has no counterexample}}{\Gamma \vdash p \textbf{ true}}$$

System FR: Theoretical Result

Main theorem: if an expression type checks, then evaluation will terminate with a “good” value.

- Formalization of *typing rules* in Coq
- 20k lines of code
- Pen-and-paper proof of integration with Inox

Stainless

Verifying Scala programs

Verifying Scala Programs

Scala programs are **transformed** into the dependently typed language

- Inner class lifting
- Laws desugaring
- Method lifting
- Type encoding
- Imperative code elimination
- Inner function lifting
- Inlining / partial evaluation
- etc.

Laws - Ordering

There is an **implicit** contract on the Ordering type

```
/** … some unchecked description of the Ordering contract … */
trait Ordering[T] {
        def compare(x: T, y: T): Int
}
```

Laws - Ordering

We can make this contract **explicit** with the @law annotation.

```
trait Ordering[T] {
    def compare(x: T, y: T): Int
    @law def inverse(x: T, y: T): Boolean =
        sign(compare(x, y)) == −sign(compare(y, x))
    @law def transitive(x: T, y: T, z: T): Boolean =
        (compare(x, y) > 0 && compare(y, z) > 0) ⟹ (compare(x, z) > 0)
    @law def consistent(x: T, y: T, z: T): Boolean =
        (compare(x, y) == 0) ⟹ (sign(compare(x, z)) == sign(compare(y, z)))
}
```

Laws - Ordering

We can make this contract **explicit** with the @law annotation.

```
trait Ordering[T] {
    def compare(x: T, y: T): Int
    def inverse(x: T, y: T): { u: Unit |
        sign(compare(x, y)) == −sign(compare(y, x)) }
    def transitive(x: T, y: T, z: T): { u: Unit |
        (compare(x, y) > 0 && compare(y, z) > 0) ⟹ (compare(x, z) > 0) }
    def consistent(x: T, y: T, z: T): { u: Unit |
        (compare(x, y) == 0) ⟹ (sign(compare(x, z)) == sign(compare(y, z))) }
}
```

Laws - Proofs

- Implementations provide proofs through overrides
  - Consistent with the expected type
  - Can be omitted when trivial
- Laws can be invoked when the property is needed for a proof
  - For example: inductive proofs
  - Enables effective *proof control*

Demo

Termination Checking

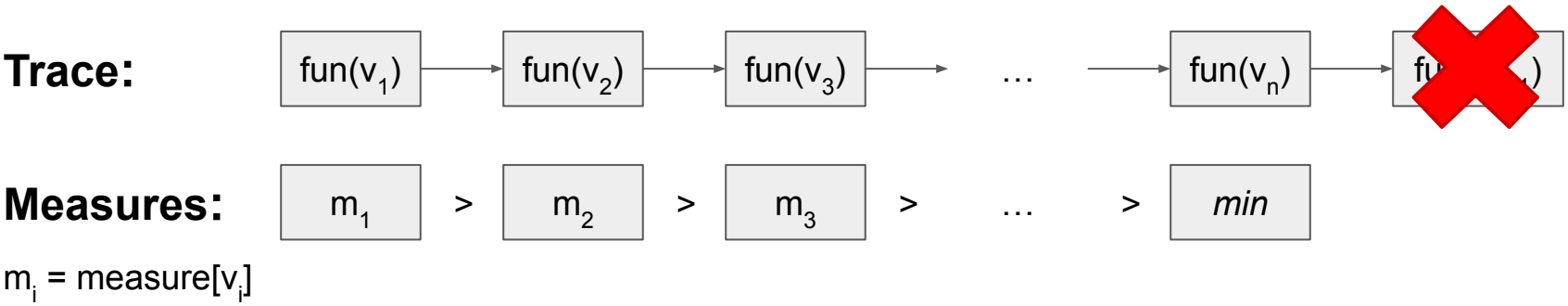
- Undecidable in **theory** (harder than the halting problem)
- But in **practice**, termination conditions are often simple
- Key idea: decreasing chains in well-founded domains
  - Well-founded domains have ordering relation
  - Well-founded domains have a minimal element

```
def fun(x: Type): Type = {
    decreases(measure[x])
    …
}
```

Decreasing Chains

```
def fun(x: Type): Type = {
    decreases(measure[x])
    … fun(e) ...
}
```

measure[x] > measure[e]



m_i = measure[v_i]

Lexicographic Ordering

QuickSort Demo

Measure Inference

- Inferring **decreases** clauses allows automating termination checking
- Termination checker considers likely measures
  - Absolute value of integer arguments
  - Structural size of datatype arguments
  - Sum of multiple argument measures
  - Lexicographic order over multiple argument measures
  - Lexicographic order encoding of decrease over multiple calls
  - etc.