A Conceptual Framework for Safe Object Initialization
Clément Blaudeau, Inria & Université de Paris Cité, France
Fengyun Liu, Oracle Labs, Switzerland
Cambium Seminar 2023, 16/10/23
Examples

```java
class A {
    var y = 42 :: this.x
    var x = List()
}
class A {
    var x : List[Int] = this.m()
    def m() = 42 :: this.x
}
class A {
    var b = new B(this)
    var x = List()
}
class B (a:A) {
    var y = 42 :: a.x
}
```

Initialization errors
• Early field access
• Early method call
• Incorrect escaping

Key issue
Objects under initialization do not fulfill their class specification yet
Examples

```java
class A {
  var y = 42 :: this.x
  var x = List()
}
```

Initialization errors
- Early field access
### Examples

```scala
class A {
  var y = 42 :: this.x
  var x = List()
}
```

```scala
class A {
  var x : List[Int] = this.m()
  def m() = 42 :: this.x
}
```

**Initialization errors**
- Early field access
- Early method call

---

Key issue: Objects under initialization do not fulfill their class specification yet.
Examples

```java
1 class A {
2   var y = 42 :: this.x
3   var x = List()
4 }
```

```java
1 class A {
2   var x : List[Int] = this.m()
3   def m() = 42::this.x
4 }
```

```java
1 class A {
2   var b = new B(this)
3   var x = List()
4 }
5 class B (a:A) {
6   var y = 42 :: a.x
7 }
```

**Initialization errors**
- Early field access
- Early method call
- Incorrect escaping
Examples

```scala
1 class A {
2   var y = 42 :: this.x
3   var x = List()
4 }
```

```scala
1 class A {
2   var x : List[Int] = this.m()
3   def m() = 42::this.x
4 }
```

```scala
1 class A {
2   var b = new B(this)
3   var x = List()
4 }
5 class B (a:A) {
6   var y = 42 :: a.x
7 }
```

Initialization errors
- Early field access
- Early method call
- Incorrect escaping
### Examples

```scala
1 class A {
2   var y = 42 :: this.x
3   var x = List()
4 }
```

```scala
1 class A {
2   var x : List[Int] = this.m()
3   def m() = 42 :: this.x
4 }
```

```scala
1 class A {
2   var b = new B(this)
3   var x = List()
4 }
5 class B (a:A) {
6   var y = 42 :: a.x
7 }
```

### Initialization errors
- Early field access
- Early method call
- Incorrect escaping

### Key issue
Objects under initialization do not fulfill their class specification yet
Complex initializations

Cyclic data structures

```java
class A () {
    var b = new B(this)
    var c = this.b.c
}
class B (arg:A) {
    var a = arg
    var c = new C(this)
}
class C (arg:B) {
    var a = arg.a
    var b = arg
}
```
Complex initializations

Cyclic data structures

```java
class A () {
  var b = new B(this)
  var c = this.b.c
}
class B (arg:A) {
  var a = arg
  var c = new C(this)
}
class C (arg:B) {
  var a = arg.a
  var b = arg
}
```

Early method call

```java
class Server (a: Address) {
  var address = a
  var _ = this.broadcast("Init")
  ... // other fields
  def broadcast(m: String) = {
    ... // sends a message
  }
}
```
Design space

- Strict initialization
- Proof of program
- Expressive
- Modular
- Sound
- No access to uninitialized field
- Authorize controlled escaping
- Modular
- Class by class analysis
- Limited footprint
- Usable
- Understandable annotations
- Inference
Sound

- No access to uninitialized field
Design space

strict initialization  proof of program

Sound

- No access to uninitialized field
Design space

strict initialization \leftrightarrow \text{proof of program}

Sound

- No access to uninitialized field

Expressive

- Authorize controlled escaping
Design space

Expressive

strict initialization

proof of program

Sound

- No access to uninitialized field

Expressive

- Authorize controlled escaping
Design space

Expressive

Strict initialization ← proof of program

Sound
- No access to uninitialized field

Modular
- Class by class analysis
- Limited footprint

Expressive
- Authorize controlled escaping
Design space

Sound
- No access to uninitialized field

Modular
- Class by class analysis
- Limited footprint

Expressive
- Authorize controlled escaping
Design space

Sound
- No access to uninitialized field

Expressive
- Authorize controlled escaping

Modular
- Class by class analysis
- Limited footprint

Usable
- Understandable annotations
- Inference
Design space

Expressive
- No access to uninitialized field
- Authorize controlled escaping

Modular
- Class by class analysis
- Limited footprint

Usable
- Understandable annotations
- Inference
Plan

1. The Celsius model of initialization
2. The Core principles
   • High-level, language agnostic
   • Design choices for the minimal calculus
3. Local reasoning and soundness (overview)

In the paper
• The minimal calculus
• The semantic interpretation of the principles
• The typing system inspired by the principles
• The (modular) soundness proof
• The Coq artifact
Plan

In this presentation

1. The Celsius model of initialization
In this presentation

1. The Celsius model of initialization
2. The Core principles
Plan

In this presentation

1. The Celsius model of initialization
2. The Core principles
   - High-level, language agnostic
Plan

In this presentation

1. The Celsius model of initialization
2. The Core principles
   - High-level, language agnostic
   - Design choices for the minimal calculus

In the paper

- The minimal calculus
- The semantic interpretation of the principles
- The typing system inspired by the principles
- The (modular) soundness proof
- The Coq artifact
Plan

In this presentation

1. The Celsius model of initialization
2. The Core principles
   - High-level, language agnostic
   - Design choices for the minimal calculus
3. Local reasoning and soundness (overview)
Plan

In this presentation
1. The Celsius model of initialization
2. The Core principles
   - High-level, language agnostic
   - Design choices for the minimal calculus
3. Local reasoning and soundness (overview)
Plan

In this presentation
1. The Celsius model of initialization
2. The Core principles
   - High-level, language agnostic
   - Design choices for the minimal calculus
3. Local reasoning and soundness (overview)

In the paper
- The minimal calculus
In this presentation

1. The Celsius model of initialization
2. The Core principles
   - High-level, language agnostic
   - Design choices for the minimal calculus
3. Local reasoning and soundness (overview)

In the paper

- The minimal calculus
- The semantic interpretation of the principles
Plan

In this presentation
1. The Celsius model of initialization
2. The Core principles
   - High-level, language agnostic
   - Design choices for the minimal calculus
3. Local reasoning and soundness (overview)

In the paper
- The minimal calculus
- The semantic interpretation of the principles
- The typing system inspired by the principles
Plan

In this presentation
1. The Celsius model of initialization
2. The Core principles
   - High-level, language agnostic
   - Design choices for the minimal calculus
3. Local reasoning and soundness (overview)

In the paper
- The minimal calculus
- The semantic interpretation of the principles
- The typing system inspired by the principles
- The (modular) soundness proof
Plan

In this presentation
1. The Celsius model of initialization
2. The Core principles
   - High-level, language agnostic
   - Design choices for the minimal calculus
3. Local reasoning and soundness (overview)

In the paper
- The minimal calculus
- The semantic interpretation of the principles
- The typing system inspired by the principles
- The (modular) soundness proof
- The Coq artifact
The Celsius Model
The Celsius model

- no assumptions
- has some initialized fields
- all fields initialized
- transitively initialized

initialization state
The Celsius model

- no assumptions
- cold
- initialization state

All fields initialized transitively initialized
The Celsius model

cold

cool

no assumptions

has some initialized fields

initialization state
The Celsius model

- no assumptions
- has some initialized fields
- all fields initialized

- cold
- cool
- warm

initialization state
The Celsius model

- No assumptions
- Has some initialized fields
- All fields initialized
- Transitivity initialized

Initialization state:
- Cold
- Cool
- Warm
- Hot
The Celsius model

- no assumptions
- has some initialized fields
- all fields initialized
- transitively initialized

\[ \text{cold} \subseteq \text{cool} \subseteq \text{warm} \subseteq \text{hot} \]

initialization state
The core principles
Principle 1/4: Monotonicity

- cold
- cool
- warm
- hot

Design choices (for the calculus)
- No de-initialization
- Update fields only with hot values
Principle 1/4: Monotonicity

- **cold**
- **cool**
- **warm**
- **hot**

- Partial monotonicity
  - Fields cannot be un-initialized

- Perfect monotonicity
  - Initialization state of every field cannot decrease

- Design choices (for the calculus)
  - No de-initialization
  - Update fields only with hot values
Partial monotonicity \( \preceq \)

Fields cannot be un-initialized
Partial monotonicity \( \preceq \)
Fields cannot be un-initialized

Perfect monotonicity \( \preceq \)
Initialization state of every field
cannot decrease
Principle 1/4: Monotonicity

Partial monotonicity \( \preceq \)
Fields cannot be un-initialized

Perfect monotonicity \( \preceq \)
Initialization state of every field cannot decrease

Design choices (for the calculus)

- No de-initialization
- Update fields only with hot values
Principle 1/4: Monotonicity

**Partial monotonicity** ⊆
Fields cannot be un-initialized

**Perfect monotonicity** ≼
Initialization state of every field cannot decrease

**Design choices (for the calculus)**
- No de-initialization
**Principle 1/4: Monotonicity**

**Partial monotonicity** ≤
Fields cannot be un-initialized

**Perfect monotonicity** ≼
Initialization state *of every field* cannot decrease

**Design choices (for the calculus)**
- No de-initialization
- Update fields only with hot values
Principle 2/4: Authority

State updates are only authorized on a distinguished alias:

- **Cold**
- **Cool**
- **Warm**
- **Hot**

Design choices:

- Distinguish FST assignment / update
- Type updates (up to warm) only inside the constructor
Principle 2/4: Authority

Local vision of the initialization state might differ between aliases. Authority on state updates is only authorized on a distinguished alias:

- Design choices
  - Distinguish 1
  - Type updates (up to warm) only inside the constructor.
Principle 2/4: Authority

Local vision of the initialization state might differ between aliases. Authority state updates are only authorized on a distinguished alias: this

Design choices
- Distinguish 1
- Type updates (up to warm) only

8 (3) / 27
Local vision of the initialization state might differ between aliases
Local vision of the initialization state might differ between aliases
Principle 2/4: Authority

Local vision of the initialization state might differ between aliases
Local vision of the initialization state might differ between aliases

Authority
State updates are only authorized on a distinguished alias
Principle 2/4: Authority

Local vision of the initialization state might differ between aliases

Authority
State updates are only authorized on a distinguished alias: this
Local vision of the initialization state might differ between aliases

Authority
State updates are only authorized on a distinguished alias: this
**Principle 2/4: Authority**

*Local vision* of the initialization state might differ between aliases

**Authority**

State updates are only authorized on a distinguished alias: *this*

**Design choices**

- Distinguish 1\(^{\text{st}}\) assignment / update
Principle 2/4: Authority

Local vision of the initialization state might differ between aliases

Authority

State updates are only authorized on a distinguished alias: this

Design choices

- Distinguish $1^{\text{st}}$ assignment / update
- Type updates (up to warm) only inside the constructor
Principle 3/4: Stackability

All fields must be initialized at the end of their constructor → constructors form a call stack.

Design choices:
- Mandatory field initializers
- No control effects
Principle 3/4: Stackability

- cold
- cool
- warm
- hot

Design choices:
- Mandatory field initializers
- No control effects
Principle 3/4: Stackability

Stackability
Stackability

All fields must be initialized at the end of their constructor

→ constructors form a *call stack*
Principle 3/4: Stackability

Stackability
All fields must be initialized at the end of their constructor
→ constructors form a call stack

Design choices
- Mandatory field initializers
Principle 3/4: Stackability

Stackability
All fields must be initialized at the end of their constructor
→ constructors form a call stack

Design choices
- Mandatory field initializers
- No control effects
Principle 4/4: Scopability

- cold
- cool
- warm
- hot

Nested/parallel initializations → Control the accessible part of the heap

Scopability
Access to objects under initialization must go through controlled channels, i.e. be controlled by static scoping

Design choices
- No global variables (see Liu 2023)
- Over-approximate reachable objects
Principle 4/4: Scopability

- cold
- cool
- warm
- hot

Nested/parallel initializations → Control the accessible part of the heap

Access to objects under initialization must go through controlled channels, i.e., be controlled by static scoping.

Design choices:
- No global variables (see Liu 2023)
- Over-approximate reachable objects
Principle 4/4: Scopability

Nested/parallel initializations → Control the accessible part of the heap

Access to objects under initialization must go through controlled channels, i.e. be controlled by static scoping

Design choices:
• No global variables (see Liu 2023)
• Over-approximate reachable objects
Principle 4/4: Scopability

Nested/parallel initializations → Control the accessible part of the heap

Scopability
Access to objects under initialization must go through controlled channels, i.e. be controlled by static scoping

Design choices
• No global variables (see Liu 2023)
• Over-approximate reachable objects
Principle 4/4: Scopability

- cold
- cool
- warm
- hot

Nested/parallel initializations $\rightarrow$ Control the accessible part of the heap

Scopability

Access to objects under initialization must go through controlled channels, i.e. be controlled by static scoping.

Design choices:
- No global variables (see Liu 2023)
- Over-approximate reachable objects
Principle 4/4: Scopability

- cold
- cool
- warm
- hot

Nested/parallel initializations → Control the accessible part of the heap

Scopability
Access to objects under initialization must go through controlled channels, i.e., be controlled by static scoping.

Design choices:
- No global variables (see Liu 2023)
- Over-approximate reachable objects
Principle 4/4: Scopability

Nested/parallel initializations → Control the accessible part of the heap

Access to objects under initialization must go through controlled channels, i.e. be controlled by static scoping.

Design choices:
- No global variables (see Liu 2023)
- Over-approximate reachable objects
Principle 4/4: Scopability

Nested/parallel initializations → Control the accessible part of the heap

Scopability

Access to objects under initialization must go through controlled channels, i.e. be controlled by static scoping.

Design choices

• No global variables (see Liu 2023)
• Over-approximate reachable objects
Principle 4/4: Scopability

Nested/parallel initializations → Control the accessible part of the heap

Scopability
Access to objects under initialization must go through controlled channels, i.e. be controlled by static scoping

Design choices
• No global variables (see Liu 2023)
• Over-approximate reachable objects
Principle 4/4: Scopability

Nested/parallel initializations
Principle 4/4: Scopability

Nested/parallel initializations → Control the accessible part of the heap
Nested/parallel initializations → Control the accessible part of the heap

**Scopability**
Nested/parallel initializations $\rightarrow$ Control the accessible part of the heap

**Scopability**

Access to objects under initialization must go through controlled channels, i.e. be controlled by static scoping
Principle 4/4: Scopability

Nested/parallel initializations → Control the accessible part of the heap

**Scopability**

Access to objects under initialization must go through controlled channels, i.e. be controlled by static scoping

**Design choices**

- No global variables (see Liu 2023)
Principle 4/4: Scopability

Nested/parallel initializations $\rightarrow$ Control the accessible part of the heap

**Scopability**

Access to objects *under initialization* must go through controlled channels, i.e. be controlled by static scoping

**Design choices**

- No global variables (see Liu 2023)
- Over-approximate reachable objects
Local reasoning

Theorem (Local reasoning)

 Executing an expression in an hot environment results in an hot object

Proof.

 In the resulting memory, accessible objects are either

  • new and therefore warm (by stackability)
  • old, so already accessible in the execution environment (by scopability),

   and therefore still hot (by monotonicity)

→ gives rises to a typing system with hot-bypasses:

   you can safely ignore initialization issues when handling hot objects
Local reasoning

Theorem (Local reasoning)

*Executing an expression in an hot environment results in an hot object*
Local reasoning

Theorem (Local reasoning)

Executing an expression in an hot environment results in an hot object.

Proof.
Local reasoning

Theorem (Local reasoning)

Executing an expression in an hot environment results in an hot object

Proof.

In the resulting memory, accessible objects are either
Local reasoning

Theorem (Local reasoning)

*Executing an expression in an hot environment results in an hot object*

Proof.

In the resulting memory, accessible objects are either

- new and therefore warm (by stackability)
Local reasoning

**Theorem (Local reasoning)**

*Executing an expression in an hot environment results in an hot object*

**Proof.**

In the resulting memory, accessible objects are either

- new and therefore warm (by stackability)
- old, so already accessible in the execution environment (by scopability),
Theorem (Local reasoning)

Executing an expression in an hot environment results in an hot object

Proof.

In the resulting memory, accessible objects are either

- new and therefore warm (by stackability)
- old, so already accessible in the execution environment (by scopability),
Theorem (Local reasoning)

Executing an expression in an hot environment results in an hot object

Proof.

In the resulting memory, accessible objects are either

- new and therefore warm (by stackability)
- old, so already accessible in the execution environment (by scopability), and therefore still hot (by monotonicity)
Theorem (Local reasoning)

Executing an expression in an hot environment results in an hot object

Proof.

In the resulting memory, accessible objects are either

- new and therefore warm (by stackability)
- old, so already accessible in the execution environment (by scopability), and therefore still hot (by monotonicity)

→ gives rises to a typing system with hot-bypasses:
Theorem (Local reasoning)

Executing an expression in an hot environment results in an hot object

Proof.

In the resulting memory, accessible objects are either

- new and therefore warm (by stackability)
- old, so already accessible in the execution environment (by scopability), and therefore still hot (by monotonicity)

→ gives rises to a typing system with hot-bypasses: you can safely ignore initialization issues when handling hot objects
Take away

• A conceptual framework for safe initialization based on four principles
  • the Celsius model (cold, cool, warm, hot)
  • Four language agnostic principles

See the paper for precise definitions, typing system and soundness proof!
A conceptual framework for safe initialization based on four principles

- the Celsius model (cold, cool, warm, hot)
Take away

A conceptual framework for safe initialization based on four principles

- the Celsius model (cold, cool, warm, hot)
- Four language agnostic principles
Take away

A **conceptual framework** for safe initialization based on four principles

- the Celsius model \(\text{\textit{cold}}, \text{\textit{cool}}, \text{\textit{warm}}, \text{\textit{hot}}\)
- Four language agnostic principles
- **Local reasoning**
A conceptual framework for safe initialization based on four principles

- the Celsius model (cold, cool, warm, hot)
- Four language agnostic principles
- Local reasoning
Take away

A **conceptual framework** for safe initialization based on four principles

- the Celsius model (cold, cool, warm, hot)
- Four language agnostic principles
- Local reasoning

See the paper for precise definitions, typing system and soundness proof!
The Celsius calculus
Grammar

Expressions

\[ e ::= x \quad (\text{Local variable}) \]
\[ \text{this} \quad (\text{Self-reference}) \]
\[ e.f \quad (\text{Field access}) \]
\[ e.m(e) \quad (\text{Method call}) \]
\[ \text{new} \ C \quad (\text{Instance creation}) \]
\[ e.f \leftarrow e; \quad (\text{Assignment}) \]

Mode

\[ \mu ::= \text{cold} \quad | \quad \text{cool} \quad | \quad \text{warm} \quad | \quad \text{hot} \]

Type

\[ T ::= C \mu \]

Class

\[ C ::= \text{class} \ C (x : T) \{ \]
\[ \text{fields} = F, \quad \text{methods} = M \}

[\]

Method

\[ M ::= @\mu \text{def} \ m (x : T) : T = \{ \]
\[ e \}

[\]

Program

\[ P ::= \{ \]
\[ \text{ct} = C, \quad \text{entry} = C \}

[\]
Grammar

Expressions

\[ e ::= x \] (Local variable)
\[ \mid this \] (Self-reference)
\[ \mid e.f \] (Field access)
\[ \mid e.m(\bar{e}) \] (Method call)
\[ \mid \text{new } C(\bar{e}) \] (Instance creation)
\[ \mid e.f \leftarrow e; e \] (Assignment)

Mode

\[ \mu ::= \text{cold} \mid \text{cool } f \mid \text{warm} \mid \text{hot} \]
Grammar

**Expressions**

\[ e ::= x \quad \text{(Local variable)} \]

\[ \mid \text{this} \quad \text{(Self-reference)} \]

\[ \mid e.f \quad \text{(Field access)} \]

\[ \mid e.m(\bar{e}) \quad \text{(Method call)} \]

\[ \mid \text{new } C(\bar{e}) \quad \text{(Instance creation)} \]

\[ \mid e.f \leftarrow e; e \quad \text{(Assignment)} \]

**Type**

\[ T ::= C^{\mu} \]

**Class**

\[ C ::= \text{class } C(x: T) \{
\text{fields } = \overline{F}, \text{methods } = \overline{M}\} \]

**Field**

\[ F ::= \text{var } f : T = e \]

**Method**

\[ M ::= @\mu \text{def } m(x: T) : T = \{e\} \]

**Program**

\[ P ::= \{\text{ct } = \overline{C}, \text{entry } = C\} \]
Examples in Celsius syntax

```celsius
class A () {
    var b: B@warm = new B(this)
    var c: C@warm = this.b.c
}
```
class A () {
    var b: B@warm = new B(this)
    var c: C@warm = this.b.c
}
class B (arg: A@cold) {
    var a: A@cold = arg
    var c: C@warm = new C(this)
}
class Server (a: Address@hot) {
    var address : Address@hot = a
    var_ = this.broadcast("Init");
    ...  // other fields
    @cool(address)
    def broadcast(m: String) = {
    ...
    // sends a message
    }
    ...  // other methods
}

Examples in Celsius syntax
Examples in Celsius syntax

```
1 class A () {
2     var b: B@warm = new B(this)
3     var c: C@warm = this.b.c
4 }
5 class B (arg: A@cold) {
6     var a: A@cold = arg
7     var c: C@warm = new C(this)
8 }
9 class C (arg: B@cool(a)) {
10    var a: A@cold = arg.a
11    var b: B@cool(a) = arg
12 }
```
Examples in Celsius syntax

```celsius
class A () {
    var b: B@warm = new B(this)
    var c: C@warm = this.b.c
}
class B (arg: A@cold) {
    var a: A@cold = arg
    var c: C@warm = new C(this)
}
class C (arg: B@cool(a)) {
    var a: A@cold = arg.a
    var b: B@cool(a) = arg
}
class Server (a: Address@hot) {
    var address : Address@hot = a
    var _ = this.broadcast("Init");
    ...
    // other fields
    @cool(address)
    def broadcast(m: String) = {
        ...
        // sends a message
    }
}
```
Semantics - Big steps

\[ \sigma : l \mapsto (C, \omega) \]

Expressions

\[ J_\mathcal{E}_K(\sigma, \rho, \psi) \rightarrow (v, \sigma') \]

Initialization

\[ \text{init} \mathcal{C}(\psi, i, \rho, \sigma) \rightarrow \sigma' \]

\(\psi\) current object (this)

\(i\) number of initialized fields

\(\rho\) local environment (args)

\(\sigma\) store
Semantics - Big steps

Store

\( \sigma : I \mapsto (C, \omega) \)
Store

\[ \sigma : l \mapsto (C, \omega) \]

Expressions
Semantics - Big steps

Store

\[ \sigma : l \mapsto (C, \omega) \]

Expressions

\[ \llbracket e \rrbracket(\sigma, \rho, \psi) \rightarrow (v, \sigma') \]
Semantics - Big steps

Store

\[ \sigma : l \mapsto (C, \omega) \]

Expressions

\[ [e](\sigma, \rho, \psi) \rightarrow (v, \sigma') \]

- \( e \) expression
- \( \psi \) current object (this)
- \( i \) number of initialized fields
- \( \rho \) local environment (args)
- \( \sigma \) store
Semantics - Big steps

Store

\[ \sigma : l \mapsto (C, \omega) \]

Expressions

\[ \llbracket e \rrbracket(\sigma, \rho, \psi) \rightarrow (v, \sigma') \]

- \( e \) expression
- \( \sigma \) store
- \( \psi \) current object (this)
- \( \rho \) local environment (args)
- \( \sigma \) store
- \( l \) local environment (fields)
Semantics - Big steps

Store

$$\sigma : l \mapsto (C, \omega)$$

Expressions

$$\llbracket e \rrbracket(\sigma, \rho, \psi) \longrightarrow (v, \sigma')$$

- $e$ expression
- $\sigma$ store
- $\rho$ local environment (fields)
Store

\[ \sigma : l \mapsto (C, \omega) \]

Expressions

\[ \llbracket e \rrbracket (\sigma, \rho, \psi) \rightarrow (v, \sigma') \]

e expression

\( \sigma \) store

\( \rho \) local environment (fields)

\( \psi \) current object (\texttt{this})
Semantics - Big steps

Store

\[ \sigma : l \mapsto (C, \omega) \]

Expressions

\[ \llbracket e \rrbracket (\sigma, \rho, \psi) \rightarrow (v, \sigma') \]

- e expression
- \( \sigma \) store
- \( \rho \) local environment (fields)
- \( \psi \) current object (**this**)

Initialization

\[ \text{init } C (\psi, i, \rho, \sigma) \rightarrow \sigma' \]

- \( \psi \) current object (**this**)
- \( i \) number of initialized fields
- \( \rho \) local environment (args)
- \( \sigma \) store
Semantics - Big steps

Store

\[ \sigma : l \mapsto (C, \omega) \]

Expressions

\[ \llbracket e \rrbracket (\sigma, \rho, \psi) \rightarrow (v, \sigma') \]

e expression

\sigma store

\rho local environment (fields)

\psi current object (\textit{this})

Initialization
Semantics - Big steps

Store

\[ \sigma : l \mapsto (C, \omega) \]

Expressions

\[ [e](\sigma, \rho, \psi) \rightarrow (v, \sigma') \]

Initialization

\[ \text{init}_C(\psi, i, \rho, \sigma) \rightarrow \sigma' \]

e  expression

\[ \sigma \text{  store} \]

\[ \rho \text{  local environment (fields)} \]

\[ \psi \text{  current object (this)} \]
Semantics - Big steps

Store

\[ \sigma : l \mapsto (C, \omega) \]

Expressions

\[
\llbracket e \rrbracket (\sigma, \rho, \psi) \rightarrow (v, \sigma')
\]
e expression
\sigma store
\rho local environment (fields)
\psi current object (this)

Initialization

\[ \text{init}_C(\psi, i, \rho, \sigma) \rightarrow \sigma' \]
\psi current object (this)
Semantics - Big steps

**Store**

\[ \sigma : l \mapsto (C, \omega) \]

**Expressions**

\[ \llbracket e \rrbracket(\sigma, \rho, \psi) \rightarrow (v, \sigma') \]

- \( e \) expression
- \( \sigma \) store
- \( \rho \) local environment (fields)
- \( \psi \) current object (this)

**Initialization**

\[ \text{init}_C(\psi, i, \rho, \sigma) \rightarrow \sigma' \]

- \( \psi \) current object (this)
- \( i \) number of initialized fields
Semantics - Big steps

Store

\[ \sigma : l \mapsto (C, \omega) \]

Expressions

\[ \llbracket e \rrbracket(\sigma, \rho, \psi) \longrightarrow (v, \sigma') \]

Initialization

\[ \text{init}_C(\psi, i, \rho, \sigma) \longrightarrow \sigma' \]
Semantics - Big steps

Store

\[ \sigma : l \mapsto (C, \omega) \]

Expressions

\[ [e](\sigma, \rho, \psi) \rightarrow (v, \sigma') \]

- e: expression
- \(\sigma\): store
- \(\rho\): local environment (fields)
- \(\psi\): current object (this)

Initialization

\[ \text{init}_C(\psi, i, \rho, \sigma) \rightarrow \sigma' \]

- \(\psi\): current object (this)
- \(i\): number of initialized fields
- \(\rho\): local environment (args)
- \(\sigma\): store
E-New
E-New

\[\llbracket \text{new } C \left( \overline{e_a} \right) \rrbracket (\sigma, \rho, \psi) \rightarrow ( , )\]
**Semantic rules**

**E-New**

\[\llbracket e_a \rrbracket (\sigma, \rho, \psi) \rightarrow (l_a, \sigma_1)\]

\[\llbracket \text{new } C (\overline{e_a}) \rrbracket (\sigma, \rho, \psi) \rightarrow (\ , \ )\]
**Semantic rules**

**E-New**

\[ [\overline{e_a}] (\sigma, \rho, \psi) \rightarrow (l_a, \sigma_1) \quad l_{\text{fresh}} \notin \text{dom}(\sigma_1) \]

\[ \frac{[\text{new } C (\overline{e_a})] (\sigma, \rho, \psi) \rightarrow (\quad , \quad )}{\quad} \]
Semantic rules

E-New
\[
\llbracket e_a \rrbracket(\sigma, \rho, \psi) \rightarrow (l_a, \sigma_1) \quad l_{\text{fresh}} \notin \text{dom}(\sigma_1) \quad \text{args}(C) = (x : T)
\]

\[
\llbracket \text{new } C(e_a) \rrbracket(\sigma, \rho, \psi) \rightarrow (,)
\]
E-New

\[[e_a]\](\sigma, \rho, \psi) \rightarrow (\overline{l_a}, \sigma_1) \quad \text{if } l_{\text{fresh}} \notin \text{dom}(\sigma_1) \quad \text{args}(C) = (x : T)

\text{init}_C(l_{\text{fresh}}, 0, (x \mapsto l_a), \sigma_1 \cup \{l_{\text{fresh}} \mapsto (C, \emptyset)\}) \rightarrow \sigma_2

\\[\begin{align*}
\text{init}_C(l_{\text{fresh}}, 0, (x \mapsto l_a), \sigma_1 \cup \{l_{\text{fresh}} \mapsto (C, \emptyset)\}) & \rightarrow \sigma_2 \\
\end{align*}\]

\[[\text{new } C (e_a)]\](\sigma, \rho, \psi) \rightarrow (\text{init}_C(l_{\text{fresh}}, 0, (x \mapsto l_a), \sigma_1 \cup \{l_{\text{fresh}} \mapsto (C, \emptyset)\}), \sigma_2)\]
**Semantic rules**

\[
\text{E-New} \\
\llbracket e_a \rrbracket (\sigma, \rho, \psi) \rightarrow (l_a, \sigma_1) \quad l_{\text{fresh}} \notin \text{dom}(\sigma_1) \quad \text{args}(C) = (x : T) \\
\text{init}_C(l_{\text{fresh}}, 0, (x \mapsto l_a), \sigma_1 \cup \{l_{\text{fresh}} \mapsto (C, \emptyset)\}) \rightarrow \sigma_2 \\
\llbracket \text{new } C (e_a) \rrbracket (\sigma, \rho, \psi) \rightarrow (l_{\text{fresh}}, )
\]
**Semantic rules**

E-New

\[
\begin{align*}
\llbracket e_a \rrbracket (\sigma, \rho, \psi) & \rightarrow (l_a, \sigma_1) \quad l_{\text{fresh}} \notin \text{dom}(\sigma_1) \quad \text{args}(C) = (x : T) \\
\text{init}_C(l_{\text{fresh}}, 0, (x \mapsto l_a), \sigma_1 \cup \{l_{\text{fresh}} \mapsto (C, \emptyset)\}) & \rightarrow \sigma_2
\end{align*}
\]

\[
\llbracket \text{new } C (e_a) \rrbracket (\sigma, \rho, \psi) \rightarrow (l_{\text{fresh}}, \sigma_2)
\]
Semantic rules

**E-New**

\[
\llbracket \overline{e_a} \rrbracket (\sigma, \rho, \psi) \rightarrow (\overline{l_a}, \sigma_1) \quad \text{l}_{\text{fresh}} \notin \text{dom}(\sigma_1) \quad \text{args}(C) = (x : T) \\
\text{init}_C(l_{\text{fresh}}, 0, (x \mapsto l_a), \sigma_1 \cup \{l_{\text{fresh}} \mapsto (C, \emptyset)\}) \rightarrow \sigma_2
\]

\[
\llbracket \text{new } C \left(\overline{e_a}\right) \rrbracket (\sigma, \rho, \psi) \rightarrow (l_{\text{fresh}}, \sigma_2)
\]

**E-Init-Cons**

\[
\text{init}_C(\psi, i, \rho, \sigma) \rightarrow
\]
Semantic rules

E-New
\[
\llbracket e_a \rrbracket (\sigma, \rho, \psi) \rightarrow (\overline{I_a}, \sigma_1) \quad \text{l}_{\text{fresh}} \notin \text{dom}(\sigma_1) \quad \text{args}(C) = (\overline{x : T})
\]
\[
\text{init}_C(l_{\text{fresh}}, 0, (x \mapsto \overline{I_a}), \sigma_1 \cup \{l_{\text{fresh}} \mapsto (C, \emptyset)\}) \rightarrow \sigma_2
\]
\[
\llbracket \text{new } C (\overline{e_a}) \rrbracket (\sigma, \rho, \psi) \rightarrow (l_{\text{fresh}}, \sigma_2)
\]

E-Init-Cons

\[
\text{fields}(C)(i) = \text{var } f_i : T = e
\]

\[
\text{init}_C(\psi, i, \rho, \sigma) \rightarrow
\]
Semantic rules

E-New
\[
\sem{e_a}(\sigma, \rho, \psi) \rightarrow (l_a, \sigma_1) \quad l_{\text{fresh}} \notin \text{dom}(\sigma_1) \quad \text{args}(C) = (x : T)
\]
\[
\text{init}_C(l_{\text{fresh}}, 0, \langle x \mapsto l_a \rangle, \sigma_1 \cup \{l_{\text{fresh}} \mapsto (C, \emptyset)\}) \rightarrow \sigma_2
\]
\[
\sem{\text{new } C(\overline{e_a})}(\sigma, \rho, \psi) \rightarrow (l_{\text{fresh}}, \sigma_2)
\]

E-Init-Cons
\[
\text{fields}(C)(i) = \text{var } f_i : T = e \quad \sem{e}(\sigma, \rho, \psi) \rightarrow (l_1, \sigma_1)
\]
\[
\text{init}_C(\psi, i, \rho, \sigma) \rightarrow
\]
Semantic rules

**E-New**
\[
\llbracket e_a \rrbracket(\sigma, \rho, \psi) \rightarrow (l_a, \sigma_1) \quad l_{\text{fresh}} \notin \text{dom}(\sigma_1) \quad \text{args}(C) = (x : T)
\]
\[
\text{init}_C(l_{\text{fresh}}, 0, (x \mapsto l_a), \sigma_1 \cup \{l_{\text{fresh}} \mapsto (C, \emptyset)\}) \rightarrow \sigma_2
\]

\[
\llbracket \text{new } C (e_a) \rrbracket(\sigma, \rho, \psi) \rightarrow (l_{\text{fresh}}, \sigma_2)
\]

**E-Init-Cons**
\[
\text{fields}(C)(i) = \text{var } f_i : T = e \quad \llbracket e \rrbracket(\sigma, \rho, \psi) \rightarrow (l_1, \sigma_1)
\]
\[
\sigma_1(\psi) = (C, \omega)
\]

\[
\text{init}_C(\psi, i, \rho, \sigma) \rightarrow
\]
**Semantic rules**

**E-New**
\[
\left[ e_a \right](\sigma, \rho, \psi) \rightarrow (\overline{l_a}, \sigma_1) \quad \text{if } l_{\text{fresh}} \notin \text{dom}(\sigma_1) \quad \text{args}(C) = (x : T)
\]

\[
\text{init}_C(l_{\text{fresh}}, 0, (x \mapsto l_a), \sigma_1 \cup \{l_{\text{fresh}} \mapsto (C, \emptyset)\}) \rightarrow \sigma_2
\]

\[
\left[ \text{new } C(\overline{e_a}) \right](\sigma, \rho, \psi) \rightarrow (l_{\text{fresh}}, \sigma_2)
\]

**E-Init-Cons**

\[
\text{fields}(C)(i) = \text{var } f_i : T = e \quad \left[ e \right](\sigma, \rho, \psi) \rightarrow (l_1, \sigma_1)
\]

\[
\sigma_1(\psi) = (C, \omega) \quad \sigma_2 = [\psi \mapsto (C, \omega \cup (f_i \mapsto l_1))] \sigma_1
\]

\[
\text{init}_C(\psi, i, \rho, \sigma) \rightarrow
\]
Semantic rules

**E-New**

\[
\begin{align*}
\llbracket e_a \rrbracket (\sigma, \rho, \psi) & \rightarrow (\overline{l_a}, \sigma_1) & \text{if } l_{\text{fresh}} \notin \text{dom}(\sigma_1) & \text{args}(C) = (x : T) \\
\text{init}_C(l_{\text{fresh}}, 0, (x \mapsto l_a), \sigma_1 \cup \{l_{\text{fresh}} \mapsto (C, \emptyset)\}) & \rightarrow \sigma_2 \\
\llbracket \text{new } C (\overline{e_a}) \rrbracket (\sigma, \rho, \psi) & \rightarrow (l_{\text{fresh}}, \sigma_2)
\end{align*}
\]

**E-Init-Cons**

\[
\begin{align*}
\text{fields}(C)(i) &= \text{var } f_i : T = e & \llbracket e \rrbracket (\sigma, \rho, \psi) & \rightarrow (l_1, \sigma_1) \\
\sigma_1(\psi) &= (C, \omega) & \sigma_2 &= [\psi \mapsto (C, \omega \cup (f_i \mapsto l_1))] \sigma_1 \\
\text{init}_C(\psi, i + 1, \rho, \sigma_2) & \rightarrow \sigma_3 \\
\text{init}_C(\psi, i, \rho, \sigma) & \rightarrow
\end{align*}
\]
Semantic rules

**E-New**

\[
[e_a](\sigma, \rho, \psi) \rightarrow (l_a, \sigma_1) \quad l_{fresh} \notin \text{dom} (\sigma_1) \quad \text{args}(C) = (x : T)
\]

\[
\text{init}_C(l_{fresh}, 0, (x \mapsto l_a), \sigma_1 \cup \{l_{fresh} \mapsto (C, \emptyset)\}) \rightarrow \sigma_2
\]

\[
\left[\text{new } C (e_a)\right](\sigma, \rho, \psi) \rightarrow (l_{fresh}, \sigma_2)
\]

**E-Init-Cons**

\[
\text{fields}(C)(i) = \text{var } f_i : T = e \quad \left[e\right](\sigma, \rho, \psi) \rightarrow (l_1, \sigma_1)
\]

\[
\sigma_1(\psi) = (C, \omega) \quad \sigma_2 = [\psi \mapsto (C, \omega \cup (f_i \mapsto l_1))] \sigma_1
\]

\[
\text{init}_C(\psi, i + 1, \rho, \sigma_2) \rightarrow \sigma_3
\]

\[
\text{init}_C(\psi, i, \rho, \sigma) \rightarrow \sigma_3
\]
Semantic rules

E-New
\[
\llbracket e_a \rrbracket(\sigma, \rho, \psi) \rightarrow (l_a, \sigma_1) \quad l_{\text{fresh}} \notin \text{dom}(\sigma_1) \quad \text{args}(C) = (x : T)
\]
\[
\text{init}_C(l_{\text{fresh}}, 0, (x \mapsto l_a), \sigma_1 \cup \{l_{\text{fresh}} \mapsto (C, \emptyset)\}) \rightarrow \sigma_2
\]
\[
\llbracket \text{new } C (\llbracket e_a \rrbracket) \rrbracket(\sigma, \rho, \psi) \rightarrow (l_{\text{fresh}}, \sigma_2)
\]

E-Init-Cons
\[
\text{fields}(C)(i) = \text{var } f_i : T = e \quad \llbracket e \rrbracket(\sigma, \rho, \psi) \rightarrow (l_1, \sigma_1)
\]
\[
\sigma_1(\psi) = (C, \omega) \quad \sigma_2 = [\psi \mapsto (C, \omega \cup (f_i \mapsto l_1))] \sigma_1
\]
\[
\text{init}_C(\psi, i + 1, \rho, \sigma_2) \rightarrow \sigma_3
\]
\[
\text{init}_C(\psi, i, \rho, \sigma) \rightarrow \sigma_3
\]

E-Init-End
\[
\text{init}_C(\psi, \text{length(fields}(C)), \rho, \sigma) \rightarrow \sigma
\]
Typing and soundness
Definitional interpreter \cite{Amin and Rompf(2017)}

\[
\begin{align*}
J_{\mathcal{E}}(\sigma, \rho, \psi, n) &= r \text{ with } r := \text{success}(v, \sigma') \mid \text{error} \mid \text{timeout} \\
\text{init } \mathcal{C}(\psi, i, \rho, \sigma, n) &= r \text{ with } r := \text{success}(\sigma') \mid \text{error} \mid \text{timeout} \\
J_{\mathcal{E}}(\sigma, \rho, \psi) \rightarrow (v, \sigma') &\iff \exists n. J_{\mathcal{E}}(\sigma, \rho, \psi, n) = \text{success}(v, \sigma') \\
\text{init } \mathcal{C}(\psi, i, \rho, \sigma) \rightarrow \sigma' &\iff \exists n. \text{init } \mathcal{C}(\psi, i, \rho, \sigma, n) = \text{success}(\sigma')
\end{align*}
\]

Soundness invariant (structure)

\[
\begin{align*}
J_{\mathcal{E}}(\sigma, \rho, \psi, n) &= r \quad r \neq \text{timeout} \quad \vdash e : T \\
\end{align*}
\]

- Monotonicity
- Authority
- Stackability
- Scopability
Definitional interpreter [Amin and Rompf(2017)]
Definitional interpreter [Amin and Rompf(2017)]

\[
\begin{align*}
\llbracket e \rrbracket(\sigma, \rho, \psi, n) &= r \quad \text{with} \quad r := \text{success}(v, \sigma') \mid \text{error} \mid \text{timeout} \\
\text{init}_C(\psi, i, \rho, \sigma, n) &= r \quad \text{with} \quad r := \text{success}(\sigma') \mid \text{error} \mid \text{timeout}
\end{align*}
\]
Soundness (1/3)

Definitional interpreter [Amin and Rompf(2017)]

\[
\begin{align*}
\llbracket e \rrbracket(\sigma, \rho, \psi, n) &= r \quad \text{with} \quad r := \text{success}(v, \sigma') \mid \text{error} \mid \text{timeout} \\
\text{init}_C(\psi, i, \rho, \sigma, n) &= r \quad \text{with} \quad r := \text{success}(\sigma') \mid \text{error} \mid \text{timeout}
\end{align*}
\]

\[
\begin{align*}
\llbracket e \rrbracket(\sigma, \rho, \psi) \longrightarrow (v, \sigma') & \iff \exists n. \llbracket e \rrbracket(\sigma, \rho, \psi, n) = \text{success}(v, \sigma') \quad (1) \\
\text{init}_C(\psi, i, \rho, \sigma) \longrightarrow \sigma' & \iff \exists n. \text{init}_C(\psi, i, \rho, \sigma, n) = \text{success}(\sigma') \quad (2)
\end{align*}
\]
Soundness (1/3)

Definitional interpreter [Amin and Rompf(2017)]

\[
\begin{align*}
\llbracket e \rrbracket(\sigma, \rho, \psi, n) &= r \quad \text{with} \quad r := \text{success}(v, \sigma') \mid \text{error} \mid \text{timeout} \\
\text{init}_C(\psi, i, \rho, \sigma, n) &= r \quad \text{with} \quad r := \text{success}(\sigma') \mid \text{error} \mid \text{timeout}
\end{align*}
\]

\[
\llbracket e \rrbracket(\sigma, \rho, \psi) \rightarrow (v, \sigma') \iff \exists n. \llbracket e \rrbracket(\sigma, \rho, \psi, n) = \text{success}(v, \sigma') \tag{1}
\]

\[
\text{init}_C(\psi, i, \rho, \sigma) \rightarrow \sigma' \iff \exists n. \text{init}_C(\psi, i, \rho, \sigma, n) = \text{success}(\sigma') \tag{2}
\]

Soundness invariant (structure)
Soundness (1/3)

Definitional interpreter [Amin and Rompf(2017)]

\[
\begin{align*}
\lbrack e \rbrack(\sigma, \rho, \psi, n) &= r \quad \text{with} \quad r := \text{success}(v, \sigma') \mid \text{error} \mid \text{timeout} \\
\text{init}_C(\psi, i, \rho, \sigma, n) &= r \quad \text{with} \quad r := \text{success}(\sigma') \mid \text{error} \mid \text{timeout}
\end{align*}
\]

\[
\begin{align*}
\lbrack e \rbrack(\sigma, \rho, \psi) \longrightarrow (v, \sigma') &\iff \exists n. \lbrack e \rbrack(\sigma, \rho, \psi, n) = \text{success}(v, \sigma') \quad (1) \\
\text{init}_C(\psi, i, \rho, \sigma) \longrightarrow \sigma' &\iff \exists n. \text{init}_C(\psi, i, \rho, \sigma, n) = \text{success}(\sigma') \quad (2)
\end{align*}
\]

Soundness invariant (structure)

\[
\lbrack e \rbrack(\sigma, \rho, \psi, n) = r \quad \begin{cases} 
    r \neq \text{timeout} \\
\end{cases}
\]
Soundness (1/3)

Definitional interpreter [Amin and Rompf(2017)]

\[
\sem{e}(\sigma, \rho, \psi, n) = r \quad \text{with} \quad r := \text{success}(v, \sigma') \mid \text{error} \mid \text{timeout}
\]

\[
\text{init}_C(\psi, i, \rho, \sigma, n) = r \quad \text{with} \quad r := \text{success}(\sigma') \mid \text{error} \mid \text{timeout}
\]

\[
\sem{e}(\sigma, \rho, \psi) \rightarrow (v, \sigma') \iff \exists n. \sem{e}(\sigma, \rho, \psi, n) = \text{success}(v, \sigma') \quad (1)
\]

\[
\text{init}_C(\psi, i, \rho, \sigma) \rightarrow \sigma' \iff \exists n. \text{init}_C(\psi, i, \rho, \sigma, n) = \text{success}(\sigma') \quad (2)
\]

Soundness invariant (structure)

\[
\begin{align*}
\sem{e}(\sigma, \rho, \psi, n) &= r \\
r &\neq \text{timeout} \\
\vdash e : T
\end{align*}
\]
Soundness (1/3)

Definitional interpreter [Amin and Rompf(2017)]

\[ \llbracket e \rrbracket(\sigma, \rho, \psi, n) = r \quad \text{with} \quad r := \text{success}(v, \sigma') \mid \text{error} \mid \text{timeout} \]

\( \text{init}_C(\psi, i, \rho, \sigma, n) = r \quad \text{with} \quad r := \text{success}(\sigma') \mid \text{error} \mid \text{timeout} \)

\[ \llbracket e \rrbracket(\sigma, \rho, \psi) \rightarrow (v, \sigma') \iff \exists n. \llbracket e \rrbracket(\sigma, \rho, \psi, n) = \text{success}(v, \sigma') \quad (1) \]

\[ \text{init}_C(\psi, i, \rho, \sigma) \rightarrow \sigma' \iff \exists n. \text{init}_C(\psi, i, \rho, \sigma, n) = \text{success}(\sigma') \quad (2) \]

Soundness invariant (structure)

\[ \llbracket e \rrbracket(\sigma, \rho, \psi, n) = r \]

\( r \neq \text{timeout} \)

\( \vdash e : T \)

\( \implies \exists \sigma', v. \)
Definitional interpreter [Amin and Rompf(2017)]

\[
\begin{align*}
\llbracket e \rrbracket (\sigma, \rho, \psi, n) &= r \quad \text{with} \quad r := \text{success}(v, \sigma') \mid \text{error} \mid \text{timeout} \\
\text{init}_C(\psi, i, \rho, \sigma, n) &= r \quad \text{with} \quad r := \text{success}(\sigma') \mid \text{error} \mid \text{timeout}
\end{align*}
\]

\[
\begin{align*}
\llbracket e \rrbracket (\sigma, \rho, \psi) \to (v, \sigma') &\iff \exists n. \llbracket e \rrbracket (\sigma, \rho, \psi, n) = \text{success}(v, \sigma') \quad (1) \\
\text{init}_C(\psi, i, \rho, \sigma) \to \sigma' &\iff \exists n. \text{init}_C(\psi, i, \rho, \sigma, n) = \text{success}(\sigma') \quad (2)
\end{align*}
\]

Soundness invariant (structure)

\[
\begin{align*}
\llbracket e \rrbracket (\sigma, \rho, \psi, n) &= r \\
r &\neq \text{timeout} \\
\begin{array}{c}
\vdash e : T
\end{array}
\end{align*}
\]

\[
\begin{align*}
&\implies \exists \sigma', v.
\end{align*}
\]
Soundness (1/3)

Definitional interpreter [Amin and Rompf(2017)]

\[
\llbracket e \rrbracket(\sigma, \rho, \psi, n) = r \quad \text{with} \quad r := \text{success}(v, \sigma') | \text{error} | \text{timeout}
\]

\[
\text{init}_C(\psi, i, \rho, \sigma, n) = r \quad \text{with} \quad r := \text{success}(\sigma') | \text{error} | \text{timeout}
\]

\[
\llbracket e \rrbracket(\sigma, \rho, \psi) \rightarrow (v, \sigma') \iff \exists n. \llbracket e \rrbracket(\sigma, \rho, \psi, n) = \text{success}(v, \sigma') \tag{1}
\]

\[
\text{init}_C(\psi, i, \rho, \sigma) \rightarrow \sigma' \iff \exists n. \text{init}_C(\psi, i, \rho, \sigma, n) = \text{success}(\sigma') \tag{2}
\]

Soundness invariant (structure)

\[
\llbracket e \rrbracket(\sigma, \rho, \psi, n) = r \quad \text{if}
\]

\[
\begin{align*}
\text{r \neq timeout} & \quad \implies \exists \sigma', v. \ \llbracket e \rrbracket(\sigma, \rho, \psi, n) = \text{success}(v, \sigma') \\
\vdash e : T & \quad \implies v : T
\end{align*}
\]
Soundness (1/3)

Definitional interpreter [Amin and Rompf(2017)]

\[
\llbracket e \rrbracket(\sigma, \rho, \psi, n) = r \quad \text{with} \quad r := \text{success}(v, \sigma') \mid \text{error} \mid \text{timeout}
\]

\[
\text{init}_C(\psi, i, \rho, \sigma, n) = r \quad \text{with} \quad r := \text{success}(\sigma') \mid \text{error} \mid \text{timeout}
\]

\[
\llbracket e \rrbracket(\sigma, \rho, \psi) \longrightarrow (v, \sigma') \iff \exists n. \llbracket e \rrbracket(\sigma, \rho, \psi, n) = \text{success}(v, \sigma') \quad (1)
\]

\[
\text{init}_C(\psi, i, \rho, \sigma) \longrightarrow \sigma' \iff \exists n. \text{init}_C(\psi, i, \rho, \sigma, n) = \text{success}(\sigma') \quad (2)
\]

Soundness invariant (structure)

\[
\llbracket e \rrbracket(\sigma, \rho, \psi, n) = r \quad \begin{cases} \quad \text{r = success}(v, \sigma') \\ \vdash v : T \end{cases} \quad \text{Monotonicity}
\]
Soundness (1/3)

Definitional interpreter [Amin and Rompf(2017)]

\[ [e](\sigma, \rho, \psi, n) = r \quad \text{with} \quad r := \text{success}(v, \sigma') | \text{error} | \text{timeout} \]

\[ \text{init}_C(\psi, i, \rho, \sigma, n) = r \quad \text{with} \quad r := \text{success}(\sigma') | \text{error} | \text{timeout} \]

\[ [e](\sigma, \rho, \psi) \rightarrow (v, \sigma') \iff \exists n. [e](\sigma, \rho, \psi, n) = \text{success}(v, \sigma') \quad (1) \]

\[ \text{init}_C(\psi, i, \rho, \sigma) \rightarrow \sigma' \iff \exists n. \text{init}_C(\psi, i, \rho, \sigma, n) = \text{success}(\sigma') \quad (2) \]

Soundness invariant (structure)

\[ [e](\sigma, \rho, \psi, n) = r \quad r \neq \text{timeout} \quad \vdash e : T \quad \exists \sigma', v. \]

\[ r = \text{success}(v, \sigma') \]

Monotonicity

Authority
Soundness (1/3)

Definitional interpreter [Amin and Rompf(2017)]

\[ \llbracket e \rrbracket(\sigma, \rho, \psi, n) = r \quad \text{with} \quad r := \text{success}(v, \sigma') \mid \text{error} \mid \text{timeout} \]
\[ \text{init}_C(\psi, i, \rho, \sigma, n) = r \quad \text{with} \quad r := \text{success}(\sigma') \mid \text{error} \mid \text{timeout} \]

\[ \llbracket e \rrbracket(\sigma, \rho, \psi) \longrightarrow (v, \sigma') \iff \exists n. \llbracket e \rrbracket(\sigma, \rho, \psi, n) = \text{success}(v, \sigma') \tag{1} \]
\[ \text{init}_C(\psi, i, \rho, \sigma) \longrightarrow \sigma' \iff \exists n. \text{init}_C(\psi, i, \rho, \sigma, n) = \text{success}(\sigma') \tag{2} \]

Soundness invariant (structure)

\[ \llbracket e \rrbracket(\sigma, \rho, \psi, n) = r \]
\[ r \neq \text{timeout} \]
\[ \vdash e : T \]
\[ \vdash v : T \]

\[ \Rightarrow \exists \sigma', v. \]
\[ \left\{ \begin{array}{l}
\quad r = \text{success}(v, \sigma') \\
\quad \vdash v : T \\
\quad \text{Monotonicity} \\
\quad \text{Authority} \\
\quad \text{Stackability}
\end{array} \right. \]
Soundness (1/3)

Definitional interpreter [Amin and Rompf(2017)]

\[
\llbracket e \rrbracket(\sigma, \rho, \psi, n) = r \quad \text{with} \quad r := \text{success}(v, \sigma') | \text{error} | \text{timeout}
\]

\[
\text{init}_C(\psi, i, \rho, \sigma, n) = r \quad \text{with} \quad r := \text{success}(\sigma') | \text{error} | \text{timeout}
\]

\[
\llbracket e \rrbracket(\sigma, \rho, \psi) \rightarrow (v, \sigma') \iff \exists n. \llbracket e \rrbracket(\sigma, \rho, \psi, n) = \text{success}(v, \sigma') \tag{1}
\]

\[
\text{init}_C(\psi, i, \rho, \sigma) \rightarrow \sigma' \iff \exists n. \text{init}_C(\psi, i, \rho, \sigma, n) = \text{success}(\sigma') \tag{2}
\]

Soundness invariant (structure)

\[
\llbracket e \rrbracket(\sigma, \rho, \psi, n) = r \quad \begin{cases} \quad r = \text{success}(v, \sigma') \\ \vdash v : T \\ \text{Monotonicity} \\ \text{Authority} \\ \text{Stackability} \\ \text{Scopability} \end{cases} \Rightarrow \exists \sigma', v.
\]

\[
\llbracket e \rrbracket(\sigma, \rho, \psi, n) = r \\
\quad r \neq \text{timeout} \\
\quad \vdash e : T
\]
\[
\Rightarrow \exists \sigma', v.
\]
Reachability and Scopability

Reachability -
\[ \sigma \models l \Rightarrow l' \]

Transitive closure of field access

Scopability -
\[ (\sigma, L) \preceq (\sigma', L') \]

Every location reachable from \( L' \) in \( \sigma' \) is either new or already reachable from \( L \) in \( \sigma \):
\[ \forall l \in \text{dom}(\sigma), \sigma' \models L' \Rightarrow l \Rightarrow \sigma \models L \Rightarrow l \]

Scopability theorem

The heap reachable from the result location \( v \) is scoped in the result store \( \sigma' \) by the execution environment \( \text{codom}(\rho) \cup \{\psi\} \) in the starting store \( \sigma \):
\[ \text{Jeff} (\sigma, \rho, \psi) \rightarrow (v, \sigma') \Rightarrow (\sigma, \rho \cup \{\psi\}) \preceq (\sigma', \{v\}) \]
Reachability and Scopability

Reachability -
$\sigma \models l \Rightarrow l'\,$

Transitive closure of field access

Scopability -
$(\sigma, L) \preceq (\sigma', L')$

Every location reachable from $L'$ in $\sigma'$ is either new or already reachable from $L$ in $\sigma$:  
$\forall l \in \text{dom}(\sigma), \sigma' \models L' \Rightarrow l \Rightarrow \sigma \models L \Rightarrow l$

Scopability theorem

The heap reachable from the result location $v$ is scoped in the result store $\sigma'$ by the execution environment $(\text{codom}(\rho) \cup \{\psi\})$ in the starting store $\sigma$:  
$\exists \exists \sigma, \rho, \psi : J(\sigma, \rho, \psi) \Rightarrow (v, \sigma') \Rightarrow (\sigma, \rho \cup \{\psi\}) \preceq (\sigma', \{v\})$
Reachability and Scopability

Reachability - $\sigma \models l \leftrightsquigarrow l'$
Reachability and Scopability

**Reachability** - $\sigma \models l \leadsto l'$

Transitive closure of field access
Reachability and Scopability

Reachability - $\sigma \vdash l \leadsto l'$

Scopability - $(\sigma, L) \prec (\sigma', L')$

Transitive closure of field access
Reachability and Scopability

**Reachability** - $\sigma \models l \leadsto l'$
Transitive closure of field access

**Scopability** - $(\sigma, L) \prec (\sigma', L')$
Every location reachable from $L'$ in $\sigma'$ is either new or already reachable from $L$ in $\sigma$:

$$\forall l \in \text{dom}(\sigma), \sigma' \models L' \leadsto l \implies \sigma \models L \leadsto l$$
Reachability and Scopability

**Reachability** - \( \sigma \models l \rightsquigarrow l' \)

Transitive closure of field access

**Scopability** - \((\sigma, L) \lessdot (\sigma', L')\)

Every location reachable from \(L'\) in \(\sigma'\) is either new or already reachable from \(L\) in \(\sigma\):

\[
\forall l \in \text{dom}(\sigma), \sigma' \models L' \rightsquigarrow l \implies \sigma \models L \rightsquigarrow l
\]
Reachability and Scopability

**Reachability** - $\sigma \models l \leadsto l'$

Transitive closure of field access

**Scopability** - $(\sigma, L) \prec (\sigma', L')$

Every location reachable from $L'$ in $\sigma'$ is either new or already reachable from $L$ in $\sigma$:

$$\forall l \in \text{dom}(\sigma), \sigma' \models L' \leadsto l \implies \sigma \models L \leadsto l$$

**Scopability theorem**

The heap reachable from the result location $v$ is scoped in the result store $\sigma'$ by the *execution environment* $(\text{codom}(\rho) \cup \{\psi\})$ in the starting store $\sigma$:

$$\llbracket e \rrbracket(\sigma, \rho, \psi) \rightarrow (v, \sigma') \implies (\sigma, \rho \cup \{\psi\}) \prec (\sigma', \{v\})$$
Typing (1/2) - typing rules

Mode lattice - \( \mu \sqsubseteq \mu' \)

\[
\text{cold} \sqsubseteq \text{cool} \sqsubseteq \text{warm} \sqsubseteq \text{hot}
\]
Mode lattice - $\mu \sqsubseteq \mu'$

cold $\sqsubseteq$ cool $\overline{f} \sqsubseteq$ warm $\sqsubseteq$ hot

Typing - $(\Gamma, T_{\text{this}}) \vdash e : T$
Typing (1/2) - typing rules

Mode lattice - $\mu \sqsubseteq \mu'$

- cold $\sqsubseteq$ cool $\sqsubseteq$ warm $\sqsubseteq$ hot

Typing - $(\Gamma, T_\text{this}) \vdash e : T$

- Local environment

* T-E-VAR
  \[ \Gamma(x) = T \]
  \[
  (\Gamma, T_\text{this}) \vdash x : T
  \]

* T-E-THIS
  \[
  (\Gamma, T_\text{this}) \vdash \text{this} : T_\text{this}
  \]
Typing (1/2) - typing rules

Mode lattice - $\mu \sqsubseteq \mu'$

- cold $\sqsubseteq$ cool $\sqsubseteq$ warm $\sqsubseteq$ hot

Typing - $(\Gamma, T_{this}) \vdash e : T$

- Local environment
- Ambient subtyping
Typing (1/2) - typing rules

Mode lattice - $\mu \sqsubseteq \mu'$

- cold $\sqsubseteq$ cool $\sqsubseteq$ warm $\sqsubseteq$ hot

Typing - $(\Gamma, T_{\text{this}}) \vdash e : T$

- Local environment
- Ambient subtyping
- Stackability (T-E-New)

T-E-FLD

$(\Gamma, T_{\text{this}}) \vdash e : D^{\text{cool}} f \quad f \in \overline{f}$

fieldType($D, f$) = $T$

$(\Gamma, T_{\text{this}}) \vdash e.f : T$

T-E-CALL

$(\Gamma, T_{\text{this}}) \vdash e : C^\mu$

lookup($C, m$) = $\oplus \mu$ def $m : (x : T) \rightarrow T$

$(\Gamma, T_{\text{this}}) \vdash \overline{e_a} : \overline{T}$

$(\Gamma, T_{\text{this}}) \vdash e.m(\overline{e_a}) : T$

T-E-NEW

ct($C$) = class $C(x : T)$ { ... }

$(\Gamma, T_{\text{this}}) \vdash \overline{e} : \overline{T}$

$(\Gamma, T_{\text{this}}) \vdash \text{new } C(\overline{e}) : C^{\text{warm}}$
Typing (1/2) - typing rules

**Mode lattice** - $\mu \subseteq \mu'$

\[
\text{cold} \subseteq \text{cool} \subseteq \text{warm} \subseteq \text{hot}
\]

**Typing** - $(\Gamma, T_{\text{this}}) \vdash e : T$

- Local environment
- Ambient subtyping
- Stackability ($T\text{-E-New}$)
- Hot shortcuts

\[
\begin{align*}
T\text{-E-Fld-Hot} & \quad (\Gamma, T_{\text{this}}) \vdash e : D^{\text{hot}} \\
\text{fieldType} (D, f) & = C^\mu \\
(\Gamma, T_{\text{this}}) & \vdash e.f : C^{\text{hot}}
\end{align*}
\]

\[
\begin{align*}
T\text{-E-Call-Hot} & \quad (\Gamma, T_{\text{this}}) \vdash e : C_0^{\text{hot}} \\
\text{lookup} (C_0, m) & = @\mu \text{ def } m : (x : D^\mu) \rightarrow C^\mu \\
(\Gamma, T_{\text{this}}) & \vdash e_a : D^{\text{hot}} \\
(\Gamma, T_{\text{this}}) & \vdash e.m (e_a) : C^{\text{hot}}
\end{align*}
\]

\[
\begin{align*}
T\text{-E-New-Hot} & \quad \text{ct}(C) = \text{class } C(x : D^\mu) \{ \ldots \} \\
(\Gamma, T_{\text{this}}) & \vdash \overline{e} : D^{\text{hot}} \\
(\Gamma, T_{\text{this}}) & \vdash \text{new } C (\overline{e}) : C^{\text{hot}}
\end{align*}
\]
Typing (1/2) - typing rules

Mode lattice - $\mu \subseteq \mu'$

- cold $\subseteq$ cool $\subseteq$ warm $\subseteq$ hot

Typing - $(\Gamma, T_{\text{this}}) \vdash e : T$

- Local environment
- Ambient subtyping
- Stackability (T-E-New)
- Hot shortcuts
- Monotonicity (T-E-Block)

\[ T-E-B\text{lock} \]

\[
(\Gamma, T_{\text{this}}) \vdash e_1.f : C^\mu \\
(\Gamma, T_{\text{this}}) \vdash e_2 : C^{\text{hot}} \\
(\Gamma, T_{\text{this}}) \vdash e_3 : T \\
\]

\[
(\Gamma, T_{\text{this}}) \vdash e_1.f \leftarrow e_2; e_3 : T
\]
Typing (2/2) - Store typing

- Prevent cyclic dependencies

Object typing - $\Sigma \models (C, \omega) : (C, \mu)$

Lower bound of actual initialization state

Abstraction of the store - $\Sigma \models \sigma$

$\text{dom}(\sigma) = \text{dom}(\Sigma)$

$\forall l \in \text{dom}(\sigma)$

$\Sigma \models \sigma(l) : \Sigma(l)$

$\Sigma \models \sigma$

Environment typing - $\Sigma \models \rho : \Gamma$

$\Sigma \models \emptyset : \emptyset$

$\Sigma \models \rho : \Gamma$

$\Sigma \models (x \mapsto \rightarrow l) \cup \rho : (x \mapsto \rightarrow T) \cup \Gamma$
Store typing - $\Sigma : l \mapsto T$

Prevent cyclic dependencies
Typing (2/2) - Store typing

Store typing - $\Sigma : l \rightarrow T$
Prevent cyclic dependencies

Object typing - $\Sigma \vdash (C, \omega) : (C, \mu)$
Lower bound of actual initialization state
Typing (2/2) - Store typing

Store typing - $\Sigma : l \mapsto T$
Prevent cyclic dependencies

Abstraction of the store - $\Sigma \models \sigma$

Object typing - $\Sigma \models (C, \omega) : (C, \mu)$
Lower bound of actual initialization state
Typing (2/2) - Store typing

Store typing - $\Sigma : l \mapsto T$

Object typing - $\Sigma \models (C, \omega) : (C, \mu)$

Prevent cyclic dependencies

Lower bound of actual initialization state

Abstraction of the store - $\Sigma \models \sigma$

\[
\begin{align*}
\text{dom}(\sigma) &= \text{dom}(\Sigma) \\
\forall l \in \text{dom}(\sigma).\, \Sigma \models \sigma(l) : \Sigma(l)
\end{align*}
\]

$\Sigma \models \sigma$
Typing (2/2) - Store typing

Store typing - $\Sigma : l \mapsto T$

Object typing - $\Sigma \vdash (C, \omega) : (C, \mu)$

Prevent cyclic dependencies

Lower bound of actual initialization state

Abstraction of the store - $\Sigma \vDash \sigma$

$$
\begin{align*}
\text{dom}(\sigma) &= \text{dom}(\Sigma) \\
\forall l \in \text{dom}(\sigma). \Sigma \vDash \sigma(l) : \Sigma(l) \\
\Sigma \vDash \sigma
\end{align*}
$$

Environment typing - $\Sigma \vDash \rho : \Gamma$
Typing (2/2) - Store typing

**Store typing** - $\Sigma : l \mapsto T$

Prevent cyclic dependencies

**Abstraction of the store** - $\Sigma \models \sigma$

\[
\begin{align*}
\text{dom} (\sigma) &= \text{dom} (\Sigma) \\
\forall l \in \text{dom} (\sigma). \Sigma \models \sigma(l) : \Sigma(l)
\end{align*}
\]

$\Sigma \models \sigma$

**Object typing** - $\Sigma \models (C, \omega) : (C, \mu)$

Lower bound of actual initialization state

**Environment typing** - $\Sigma \models \rho : \Gamma$

$\Sigma \models \emptyset : \emptyset$

$\Sigma \models \rho : \Gamma \quad \Sigma \models l : T$

$\Sigma \models (x \mapsto l) \cup \rho : (x \mapsto T) \cup \Gamma$
Soundness (2/3)

\[
[e](\sigma, \rho, \psi, n) = r
\]

\[
r \neq \text{timeout}
\]

\[
\vdash e : T
\]

\[
\Rightarrow \exists \sigma', v
\]

\[
\begin{cases}
  r = \text{success}(v, \sigma') \\
  \vdash v : T
\end{cases}
\]

Monotonicity
Authority
Stackability
Scopability
\[ \lbrack e \rbrack(\sigma, \rho, \psi, n) = r \]
\[ r \neq \text{timeout} \]
\[ \models e : T \]

\[ \models \exists \sigma', v . \right \{ \begin{array}{l}
 r = \text{success}(v, \sigma') \\
 \models v : T \\
 \text{Monotonicity} \\
 \text{Authority} \\
 \text{Stackability}
\end{array} \]
Soundness (2/3)

\[ \llbracket e \rrbracket(\sigma, \rho, \psi, n) = r \]
\[ r \neq \text{timeout} \]
\[ \Sigma \models \sigma \]
\[ \vdash e : T \]

\[ \iff \exists \sigma', v \]

\[ r = \text{success}(v, \sigma') \]
\[ \models v : T \]

Monotonicity
Authority
Stackability
Soundness (2/3)

\[
\left[ e \right](\sigma, \rho, \psi, n) = r \quad \begin{cases} 
\sum \vdash \sigma \\
\sum \vdash \rho : \Gamma \\
\vdash e : T 
\end{cases} \quad \implies \quad \exists \sigma', v . \left\{ \begin{array}{l}
r = \text{success}(v, \sigma') \\
\vdash v : T \\
\text{Monotonicity} \\
\text{Authority} \\
\text{Stackability} 
\end{array} \right.
\]

r \neq \text{timeout}
\[ \llbracket e \rrbracket(\sigma, \rho, \psi, n) = r \]
\[ r \neq \text{timeout} \]
\[ \Sigma \models \sigma \]
\[ \Sigma \models \rho : \Gamma \]
\[ \Sigma \models \psi : T_{\text{this}} \]
\[ \models e : T \]
\[ \implies \exists \sigma', v \]
\[ r = \text{success}(v, \sigma') \]
\[ \models v : T \]

Monotonicity
Authority
Stackability
Soundness (2/3)

\[
\begin{align*}
\llbracket e \rrbracket(\sigma, \rho, \psi, n) &= r \\
r &\neq \text{timeout} \\
\Sigma \vdash \sigma \\
\Sigma \vdash \rho : \Gamma \\
\Sigma \vdash \psi : T_{\text{this}} \\
(\Gamma, T_{\text{this}}) \vdash e : T \\
\end{align*}
\]

\[
\Rightarrow \exists \sigma', v . \\
r = \text{success}(v, \sigma') \\
\vdash v : T \\
\text{Monotonicity} \\
\text{Authority} \\
\text{Stackability}
\]
\[
\begin{align*}
\llbracket e \rrbracket(\sigma, \rho, \psi, n) &= r \\
r &\neq \text{timeout} \\
\Sigma &\models \sigma \\
\Sigma &\models \rho : \Gamma \\
\Sigma &\models \psi : T_{\text{this}} \\
(\Gamma, T_{\text{this}}) &\vdash e : T \\
\end{align*}
\]

\[
\begin{align*}
\Rightarrow &\exists \sigma', v, \Sigma'. \\
&\{ \text{Monotonicity} \} \\
&\{ \text{Authority} \} \\
&\{ \text{Stackability} \}
\end{align*}
\]
Soundness (2/3)

\[
\begin{align*}
\llbracket e \rrbracket(\sigma, \rho, \psi, n) &= r \\
r &\neq \text{timeout} \\
\Sigma &\vdash \sigma \\
\Sigma &\vdash \rho : \Gamma \\
\Sigma &\vdash \psi : T_{\text{this}} \\
(\Gamma, T_{\text{this}}) &\vdash e : T \\
\end{align*}
\]

\[
\begin{align*}
\Rightarrow & \exists \sigma', v, \Sigma'. \\
r &= \text{success}(v, \sigma') \\
\Sigma' &\vdash v : T \\
\text{Monotonicity} \\
\text{Authority} \\
\text{Stackability}
\end{align*}
\]
\[ \llbracket e \rrbracket(\sigma, \rho, \psi, n) = r \]
\[ r \neq \text{timeout} \]
\[ \Sigma \vdash \sigma \]
\[ \Sigma \vdash \rho : \Gamma \]
\[ \Sigma \vdash \psi : T_{\text{this}} \]
\[ (\Gamma, T_{\text{this}}) \vdash e : T \]
\[ \implies \exists \sigma', v, \Sigma'. \]
\[ r = \text{success}(v, \sigma') \]
\[ \Sigma' \vdash v : T \]
\[ \Sigma' \vdash \sigma' \]

Monotonicity

Authority

Stackability
Core principles for typing

Monotonicity

\[ \Sigma \preceq \Sigma' : \forall l \in \text{dom} (\Sigma). \Sigma' (l) < \Sigma (l) \]

Stackability

\[ \Sigma \preceq \Sigma' : \forall l \in \text{dom} (\Sigma') \setminus \text{dom} (\Sigma). \Sigma' (l) \models l : \text{warm} \]

Authority

\[ \Sigma \colon \Sigma' : \forall l. \Sigma (l) = C \Rightarrow \Sigma' (l) = \Sigma (l) \]
Core principles for typing

Monotonicity

\[ \Sigma \preceq \Sigma' := \forall l \in \text{dom}(\Sigma). \, \Sigma'(l) \preceq \Sigma(l) \]
Core principles for typing

Monotonicity

\[ \Sigma \preceq \Sigma' := \forall l \in \text{dom}(\Sigma). \Sigma'(l) \ll : \Sigma(l) \]

Stackability

\[ \Sigma \preceq \Sigma' := \forall l \in \text{dom}(\Sigma') \setminus \text{dom}(\Sigma). \Sigma' \vdash l : \text{warm} \]
Core principles for typing

Monotonicity

\[ \Sigma \preceq \Sigma' := \forall l \in \text{dom}(\Sigma). \Sigma'(l) \lll \Sigma(l) \]

Stackability

\[ \Sigma \ll \Sigma' := \forall l \in \text{dom}(\Sigma') \setminus \text{dom}(\Sigma). \Sigma' \models l : \text{warm} \]

Authority

\[ \Sigma \triangleright \Sigma' := \forall l. \Sigma'(l) = C^{\text{cool} \bar{f}} \implies \Sigma'(l) = \Sigma(l) \]
\[ [e](\sigma, \rho, \psi, n) = r \]
\[ r \neq \text{timeout} \]
\[ \Sigma \vdash \sigma \]
\[ \Sigma \vdash \rho : \Gamma \]
\[ \Sigma \vdash \psi : T_{\text{this}} \]
\[ (\Gamma, T_{\text{this}}) \vdash e : T \]
\[ \Rightarrow \exists \sigma', v, \Sigma'. \]
\[ r = \text{success}(v, \sigma') \]
\[ \Sigma' \vdash \sigma' \]
\[ \Sigma' \vdash v : T \]
\[ \Sigma \preceq \Sigma' \]
\[ \Sigma \triangleright \Sigma' \]
\[ \Sigma \preccurlyeq \Sigma' \]
Soundness (3/3)

\[
\left[ e \right](\sigma, \rho, \psi, n) = r
\]
\[
\begin{align*}
\text{\( r \neq \text{timeout} \)}
\end{align*}
\]
\[
\Sigma \vdash \sigma
\]
\[
\Sigma \vdash \rho : \Gamma
\]
\[
\Sigma \vdash \psi : T_{\text{this}}
\]
\[
(\Gamma, T_{\text{this}}) \vdash e : T
\]
\[
\implies \exists \sigma', \nu, \Sigma'.
\]
\[
\left\{ \begin{array}{l}
\text{\( r = \text{success}(\nu, \sigma') \)} \\
\Sigma' \vdash \sigma'
\end{array} \right.
\]
\[
\left\{ \begin{array}{l}
\Sigma' \vdash \nu : T \\
\Sigma \preceq \Sigma'
\end{array} \right.
\]
\[
\left\{ \begin{array}{l}
\Sigma \triangleright \Sigma'
\end{array} \right.
\]
\[
\Sigma \ll \Sigma'
\]

\[
\text{init}_C(\psi, i, \rho, \sigma, n) = r
\]
\[
\text{\( r \neq \text{timeout} \)}
\]
\[
\Sigma \vdash \sigma
\]
\[
\Sigma \vdash \rho : \Gamma
\]
\[
\Sigma(\psi) = \text{cool} \{ f_0, \ldots, f_{i-1} \}
\]
\[
\Gamma <: \Gamma_a
\]
\[
\implies \exists \sigma', \Sigma'.
\]
\[
\left\{ \begin{array}{l}
\text{\( r = \text{success}(\sigma') \)} \\
\Sigma' \vdash \sigma'
\end{array} \right.
\]
\[
\left\{ \begin{array}{l}
\Sigma \preceq \Sigma'
\end{array} \right.
\]
\[
\left\{ \begin{array}{l}
\Sigma \ll \Sigma'
\end{array} \right.
\]
\[
\left[ \psi \mapsto C^{\text{cool(fields}(C))} \right] \Sigma \triangleright \Sigma'
\]

(3)

(4)
Program Soundness

Theorem (Program Soundness)

A well typed program cannot run into an error

\[ \vdash P \implies \forall n, [P](n) \neq \text{error} \]
Conclusion
• Four principles for the safe initialization of objects
• Four principles for the safe initialization of objects
  • Monotonicity (invariants progress)
Take-away

- Four principles for the safe initialization of objects
  - Monotonicity (invariants progress)
  - Authority (distinguished alias)
Take-away

- Four principles for the safe initialization of objects
  - Monotonicity (invariants progress)
  - Authority (distinguished alias)
  - Stackability (all fields initialized at the end of the constructor)

- A minimal calculus to illustrate the principles
- A modular proof, mechanized in Coq
Four principles for the safe initialization of objects

- Monotonicity (invariants progress)
- Authority (distinguished alias)
- Stackability (all fields initialized at the end of the constructor)
- Scopability (control the access to un-initialized objects)
Take-away

- Four principles for the safe initialization of objects
  - Monotonicity (invariants progress)
  - Authority (distinguished alias)
  - Stackability (all fields initialized at the end of the constructor)
  - Scopability (control the access to un-initialized objects)

- A minimal calculus to illustrate the principles
Take-away

- Four principles for the safe initialization of objects
  - Monotonicity (invariants progress)
  - Authority (distinguished alias)
  - Stackability (all fields initialized at the end of the constructor)
  - Scopability (control the access to un-initialized objects)

- A minimal calculus to illustrate the principles

- A modular proof, mechanized in Coq
N. Amin and T. Rompf.

Type soundness proofs with definitional interpreters, 2017.
Pages: 666–679 Publisher: ACM.