Trust, but Verify

Two-Phase Typing for Dynamic Languages

Panagiotis Vekris, *Benjamin Cosman, Ranjit Jhala*

University of California, San Diego

Scripting Languages – Then

"Perl is the duct tape of the Internet."

Hassan Schroeder, Sun's first webmaster

"Scripting languages are designed for **'gluing'** applications; they use **typeless** approaches to achieve a higher level of programming and more **rapid** application development than system programming languages."

John K. Ousterhout

Scripting Languages – Now

Front-end

Server-side

Tooling

Complex reasoning calls for stronger guarantees

Talk Outline

- **Goal: Static Verification of Scripting Languages**

Program:

Compute index of smallest array element

Goal:

Verify that array accesses within bounds

Program #1: First-order

```
function minIndexFO(a) {




}
```

Program #1: First-order

```
function minIndexFO(a) {


  var res = 0;




  return res;
}
```

Program #1: First-order

```
function minIndexFO(a) {


  var res = 0;
  for (var i = 0; i < a.length; i++) {


  }
  return res;
}
```

Program #1: First-order

```
function minIndexFO(a) {


  var res = 0;
  for (var i = 0; i < a.length; i++) {
    if (a[i] < a[res])
      res = i;
  }
  return res;
}
```

Program #1: First-order

```
function minIndexFO(a) {
  if (a.length <= 0)
    return -1;
  var res = 0;
  for (var i = 0; i < a.length; i++) {
    if (a[i] < a[res])
      res = i;
  }
  return res;
}
```

Program #1: First-order

```
function minIndexFO(a) {
  if (a.length <= 0)
    return -1;
  var res = 0;
  for (var i = 0; i < a.length; i++) {
    if (a[i] < a[res])
      res = i;
  }
  return res;
}
```

Array bound checks:
easy

Dataflow Analysis: $0 \leq \texttt{res} \leq \texttt{i} < \texttt{a.length}$

Program #2: Higher-Order

```
function $reduce(a, f, x) {
  var res = x;
  for (var i = 0; i < a.length; i++) {
    res = f(res, a[i], i);
  }
  return res;
}
```

Program #2: Higher-Order

```
function $reduce(a, f, x) {
  var res = x;
  for (var i = 0; i < a.length; i++) {
    res = f(res, a[i], i);
  }
  return res;
}
```

```
function minIndexHO(a) {
  if (a.length <= 0) return -1;
  function step(min, cur, i) {
    return (cur < a[min]) ? i : min;
  }
  return $reduce(a, step, 0);
}
```

Challenge: Verify array access

```
function $reduce(a, f, x) {
  var res = x;
  for (var i = 0; i < a.length; i++) {
    res = f(res, a[i], i);
  }
  return res;
}
```

```
function minIndexHO(a) {
  if (a.length <= 0) return -1;
  function step(min, cur, i) {
    return (cur < a[min]) ? i : min;
```

Hard to track **relational facts** among values and **closures** with DataFlow analyses

Easy to track **relational facts** among values and **closures** with **Refinement Types**

Talk Outline

- Goal: Static Verification of Scripting Languages
- **Approach: Refinement Types**

```
type idx<a> = {v: number | 0 <= v && v < a.length}
```

“Set of valid indices for array **a**”

```
type idx<a> = {v: number | 0 <= v && v < a.length}
```

Base Type

“Set of valid indices for array **a**”

```
type idx<a> = {v: number | 0 <= v && v < a.length}
```

Logical Predicate

“Set of valid indices for array a”

```
type idx<a> = {v: number | 0 <= v && v < a.length}
```

Value Variable

“Set of valid indices for array **a**”

```
type idx<a> = {v: number | 0 <= v && v < a.length}
```

Higher-Order Example

```
function $reduce(a, f, x) {
  var res = x;
  for (var i = 0; i < a.length; i++)
    res = f(res, a[i], i);
  return res;
}
```

Higher-Order Type Checking

```
function $reduce<A,B>(a: A[], f: (B,A,number) => B, x: B): B {
  var res = x;
  for (var i = 0; i < a.length; i++)
    res = f(res, a[i], i);
  return res;
}
```

TypeScript

Higher-Order Value Checking

```
type idx<a> = { v: number | 0 <= v && v < a.length }
```

```
function $reduce<A,B>(a: A[], f: (B,A,idx<a>) => B, x: B): B {
  var res = x;
  for (var i = 0; i < a.length; i++)
    res = f(res, a[i], i);
  return res;
}
```

TypeScript + Refinements [Xi'99]

Type Analysis

Needs

Value Analysis

But there is a tricky problem …

Array.prototype.reduce()

Summary

The `reduce()` method applies a function against an accumulator and each value of the array (from left-to-right) has to reduce it to a single value.

Syntax

```
arr.reduce(callback[, initialValue])
```

Parameters

`callback`

Function to execute on each value in the array, taking four arguments:

`initialValue`

Optional. Object to use as the first argument to the first call of the `callback`.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce

Array.prototype.reduce()

Summary

The `reduce()` method applies a function against an accumulator and each value of the array (from left-to-right) has to reduce it to a single value.

Syntax

Problem: "Value Based" Overloading

Parameters

`callback`

Function to execute on each value in the array, taking four arguments:

`initialValue`

Optional. Object to use as the first argument to the first call of the `callback`.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce

Talk Outline

- Goal: Static Verification of Scripting Languages
- Approach: Refinement Types
- **Problem: Value Based Overloading**

Value Based Overloading

```
function reduce(a, f, x) {
  if (arguments.length === 3)
    return $reduce(a, f, x);
  else
    return $reduce(a, f, a[0]);
}
```

Value Based Overloading

```
function reduce(a, f, x) {
  if (arguments.length === 3)
    return $reduce(a, f, x);
  else
    return $reduce(a, f, a[0]);
}
```

Type when called with 3 values:

```
function reduce<A,B>(a: A[], f: (B,A,idx<a>) => B, x: B): B
```

Value Based Overloading

```
function reduce(a, f, x) {
  if (arguments.length === 3)
    return $reduce(a, f, x);
  else
    return $reduce(a, f, a[0]);
}
```

Type when called with 3 values:

```
function reduce<A,B>(a: A[], f: (B,A,idx<a>) => B, x: B): B
```

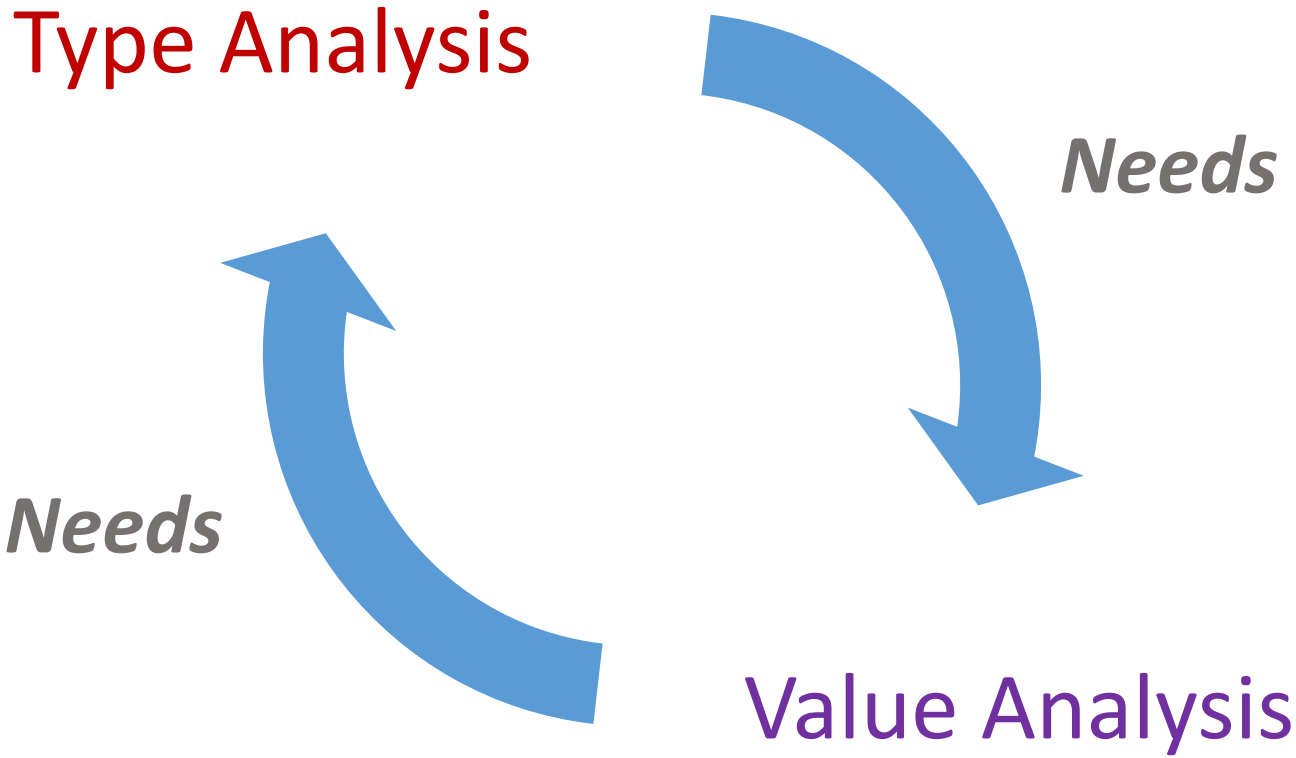
Type when called with 2 values:

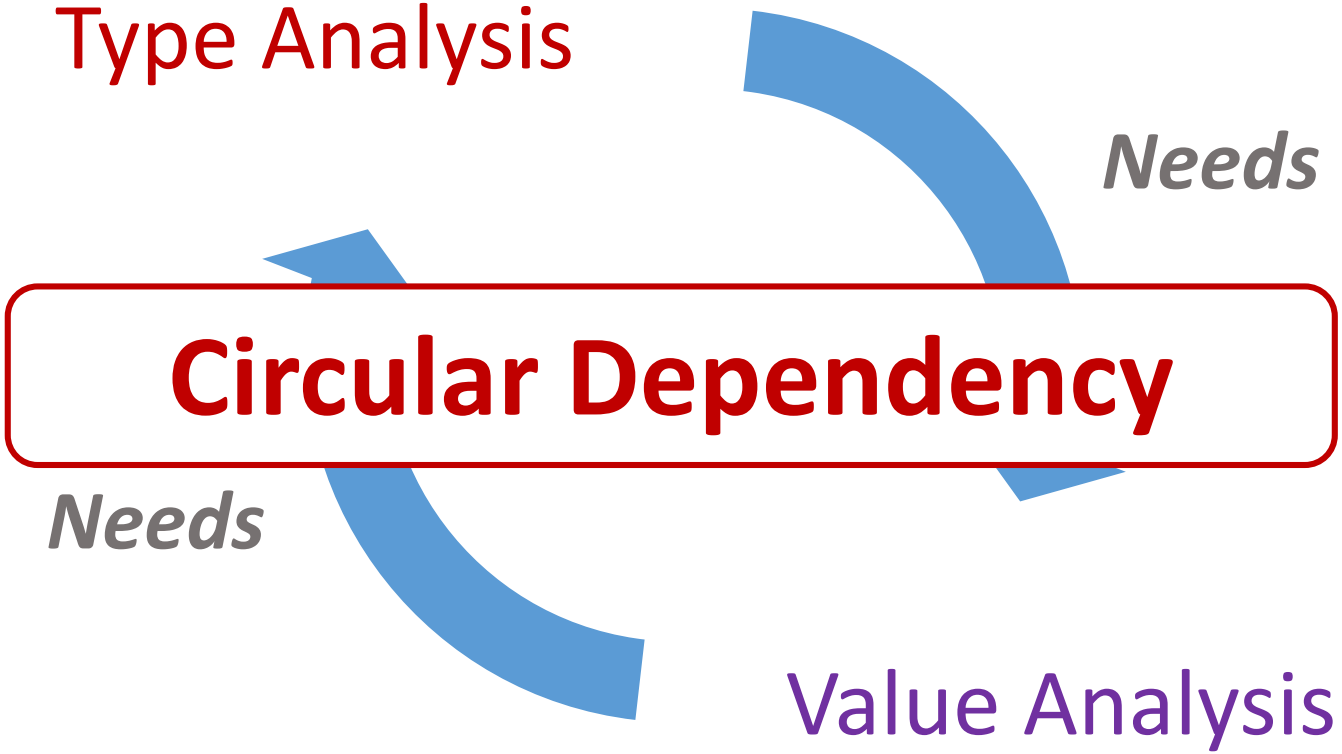
```
function reduce<A>(a: A[]+, f: (A,A,idx<a>) => A): A
```

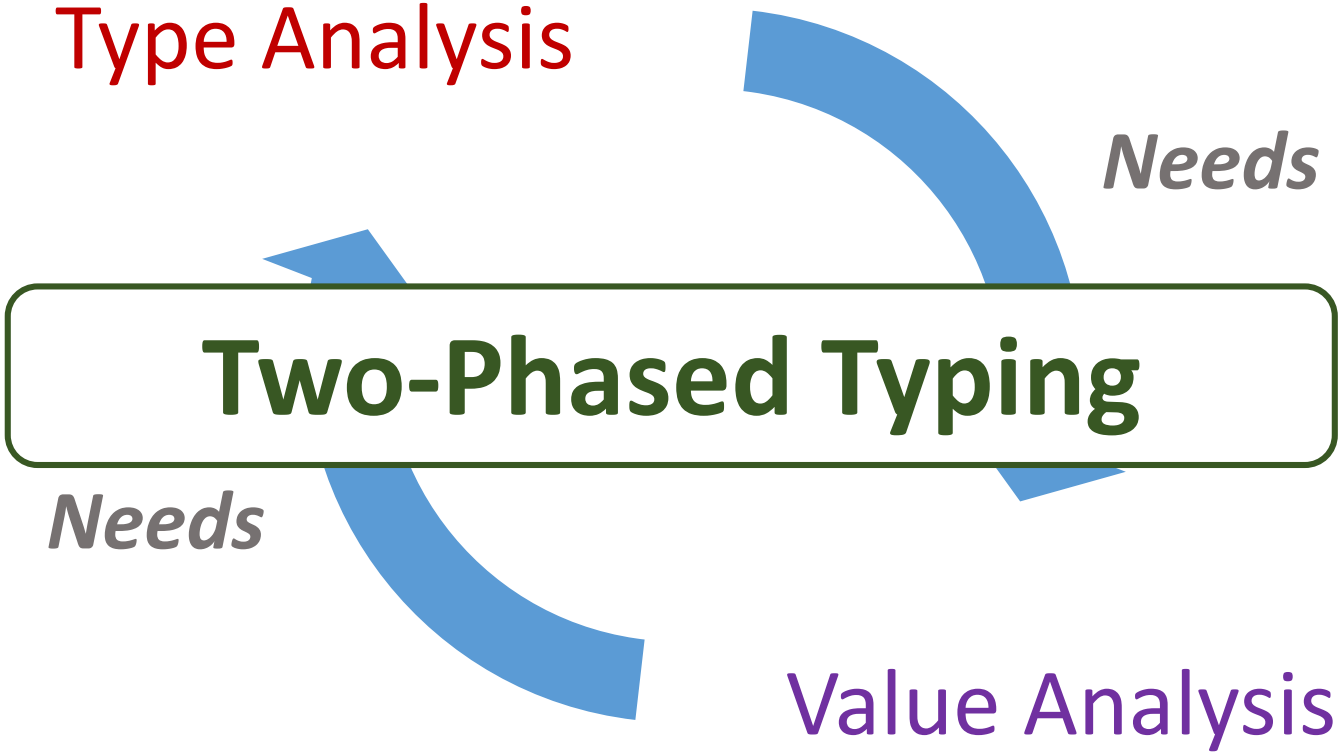
Type Analysis

Needs

Value Analysis


Type Analysis
Needs
Needs
Value Analysis


Type Analysis
Needs
Circular Dependency
Needs
Value Analysis


Type Analysis
Needs
Two-Phased Typing
Needs
Value Analysis

Talk Outline

- Goal: Static Verification of Scripting Languages
- Approach: Refinement Types
- Problem: Overloading
- **Solution: Two-Phased Typing**

Two-Phased Typing – Goal

```
function negate(flag, x) {
  if (flag) return 0 - x;
  else return !x;
}
```

(1) If `flag ≠ 0` then `x: number`
(2) If `flag = 0` then `x: boolean`

Two-Phased Typing – Goal

```
function negate(flag, x) {
  if (flag) return 0 - x;
  else return !x;
}
```

(1) If `flag ≠ 0` then `x: number`
(2) If `flag = 0` then `x: boolean`

✓
✓

```
var ok1  = negate(1, 1);
var ok2  = negate(0, true);
```

Two-Phased Typing – Goal

```
function negate(flag, x) {
  if (flag) return 0 - x;
  else return !x;
}
```

(1) If `flag ≠ 0` then `x: number`
(2) If `flag = 0` then `x: boolean`

✓ ✓ ✗ ✗

```
var ok1  = negate(1, 1);
var ok2  = negate(0, true);
var bad1 = negate(0, 1);
var bad2 = negate(1, true);
```

Two-Phased Typing – Goal

```
function negate(flag, x) {
  if (flag) return 0 - x;
  else return !x;
}
```

(1) If `flag` ≠ `0` then `x: number`
(2) If `flag` = `0` then `x: boolean`

Specification

```
type tt = {v: number | v ≠ 0}    // "truthy" numbers
type ff = {v: number | v = 0}    // "falsy" numbers
```

```
function negate(flag, x) {
  if (flag) return 0 - x;
  else return !x;
}
```

(1) If `flag ≠ 0` then `x: number`
(2) If `flag = 0` then `x: boolean`

Specification

```
type tt = {v: number | v ≠ 0}    // "truthy" numbers
type ff = {v: number | v = 0}    // "falsy" numbers
```

```
function negate(flag, x) {
  if (flag) return 0 - x;
  else return !x;
}
```

(1) If `flag: tt` then `x: number`
(2) If `flag: ff` then `x: boolean`

Specification

```
type tt = {v: number | v ≠ 0}    // "truthy" numbers
type ff = {v: number | v = 0}    // "falsy" numbers
```

```
function negate(flag: tt, x: number): number;
function negate(flag: ff, x: boolean): boolean;
function negate(flag, x) {
  if (flag) return 0 - x;
  else return !x;
}
```

Incorporate in negate's type

Verification

```
type tt = {v: number | v ≠ 0}    // "truthy" numbers
type ff = {v: number | v = 0}    // "falsy" numbers
```

```
function negate(flag: tt, x: number): number;
function negate(flag: ff, x: boolean): boolean;
function negate(flag, x) {
  if (flag) return 0 - x;
  else return !x;
}
```

(A) Statically check `negate`'s body

Verification

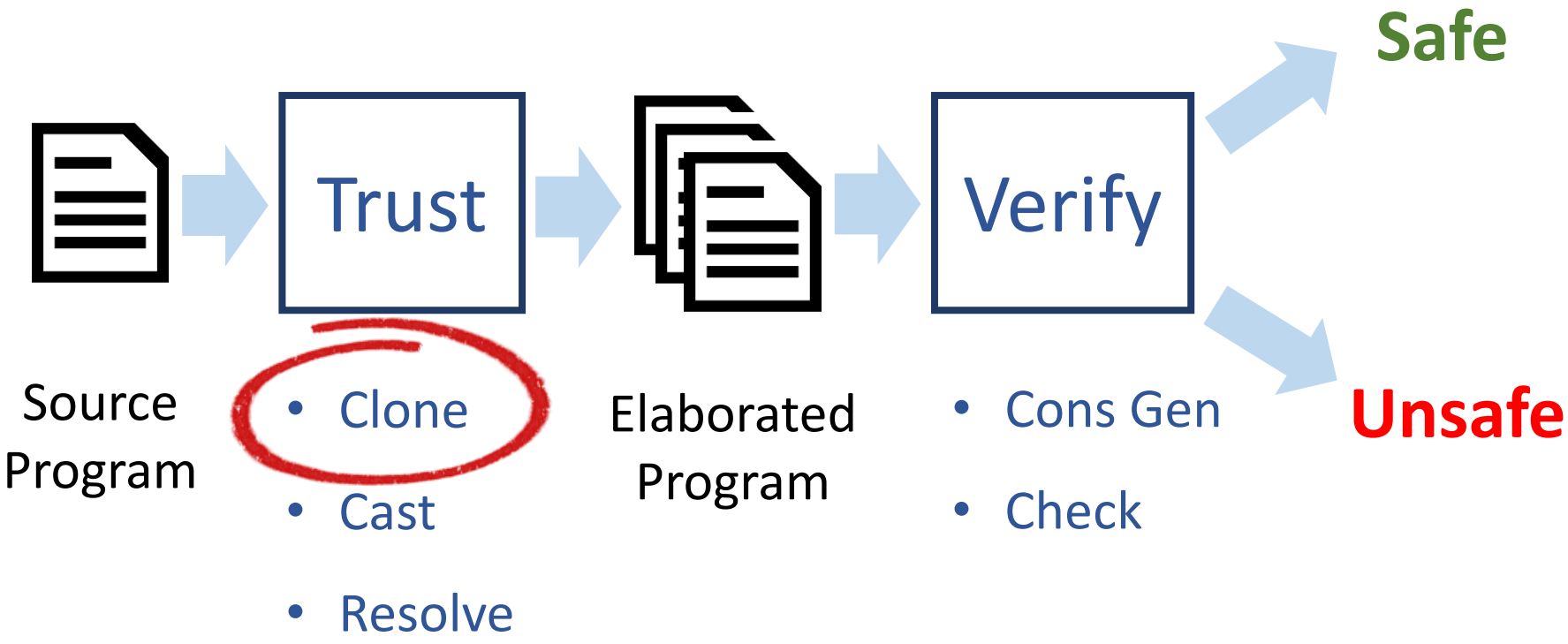
```
function negate(flag: tt, x: number): number;
function negate(flag: ff, x: boolean): boolean;
function negate(flag, x) {
  if (flag) return 0 - x;
  else return !x;
}
```

(A) Statically check negate's body

```
var ok1  = negate(1, 1);
var ok2  = negate(0, true);
var bad1 = negate(0, 1);
var bad2 = negate(1, true);
```

(B) Statically check call-sites


Two-Phased Typing
Safe
Unsafe
Source
Program


Trust
Verify
Safe
Unsafe
Source Program
• Clone
• Cast
• Resolve
Elaborated Program
• Cons Gen
• Check

1st Phase (Trust) – Clone

```
function negate(flag: tt, x: number): number;
function negate(flag: ff, x: boolean): boolean;
function negate(flag, x) {
  if (flag) return 0 - x;
  else return !x;
}
```

Split overloaded functions

1st Phase (Trust) – Clone

```
function negate(flag: tt, x: number): number;
function negate(flag: ff, x: boolean): boolean;
function negate(flag, x) {
  if (flag) return 0 - x;
  else return !x;
}
```

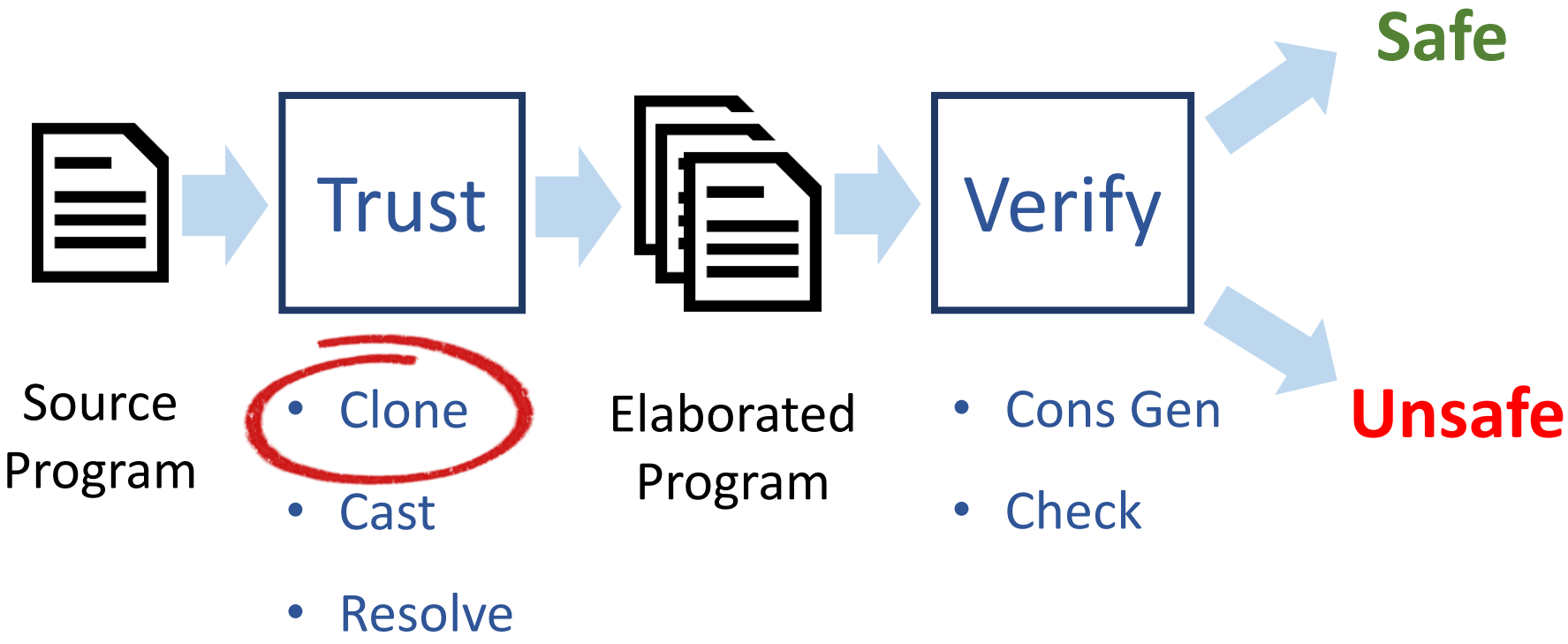
One clone for each conjunct

1st Phase (Trust) – Clone

```
function negate#1(flag: tt, x: number): number {
  if (flag) return 0 - x;
  else return !x;
}
```

```
function negate#2(flag: ff, x: boolean): boolean {
  if (flag) return 0 - x;
  else return !x;
}
```

One clone for each conjunct


Trust
Verify
Safe
Unsafe
Source Program
• Clone
• Cast
• Resolve
Elaborated Program
• Cons Gen
• Check

1st Phase (Trust) – Cast

```
function negate#1(flag: tt, x: number): number {
  if (flag) return 0 - x;
  else return !x;
}
```

```
function negate#2(flag: ff, x: boolean): boolean {
  if (flag) return 0 - x;
  else return !x;
}
```

Check each clone separately

1st Phase (Trust) – Cast

```
function negate#1(flag: number, x: number): number {
  if (flag) return 0 - x;
  else return !x;
}
```

This phase is agnostic to refinements

1st Phase (Trust) – Cast

```
function negate#1(flag: number, x: number): number {
  if (flag) return 0 - x;
  else return !x;
}
```

Path- and value-insensitive checking

TypeScript

1st Phase (Trust) – Cast

```
function negate#1(flag: number, x: number): number {
  if (flag) return 0 - x;
  else return !x;
}
```

```
Argument of type 'number' is not
assignable to parameter of type 'boolean'.
(parameter) x: number
```

We do **not** reject the program

TypeScript

1st Phase (Trust) – Cast

```
function negate#1(flag: number, x: number): number {
  if (flag) return 0 - x;
  else return !DEAD(x);
}
```

Wrap erroneous expression in DEAD-cast

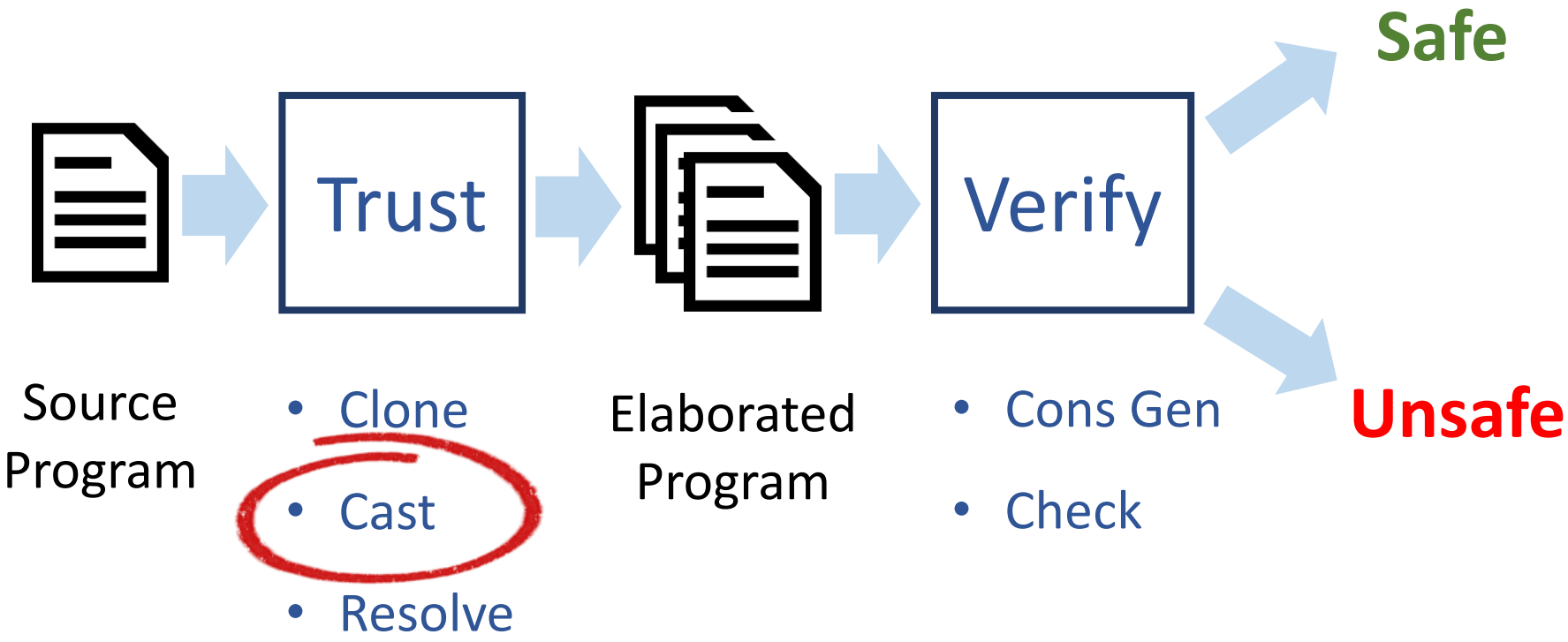
```
declare function DEAD<A,B>({v: A | false}): B;
```

Similar to `assert(false)`

1st Phase (Trust) – Cast

```
function negate#1(flag: number, x: number): number {
  if (flag) return 0 - x;
  else return !DEAD(x);
}
```

```
function negate#2(flag: number, x: boolean): boolean {
  if (flag) return 0 - DEAD(x);
  else return !x;
} declare function DEAD<A,B>({v: A | false}): B;
```


Trust
Verify
Safe
Unsafe
Source Program
• Clone
• Cast
• Resolve
Elaborated Program
• Cons Gen
• Check

1st Phase (Trust) – Overload Resolution

```
function negate#1(flag: tt, x: number): number;
function negate#2(flag: ff, x: boolean): boolean;
```

```
var ok1  = negate(1, 1);
var ok2  = negate(0, true);
var bad1 = negate(0, 1);
var bad2 = negate(1, true);
```

Which overload should be used?

1st Phase (Trust) – Overload Resolution

```
function negate#1(flag: tt, x: number): number;
function negate#2(flag: ff, x: boolean): boolean;
```

```
var ok1  = negate(1, 1);
var ok2  = negate(0, true);
var bad1 = negate(0, 1);
var bad2 = negate(1, true);
```

Resolve based on base type

1st Phase (Trust) – Overload Resolution

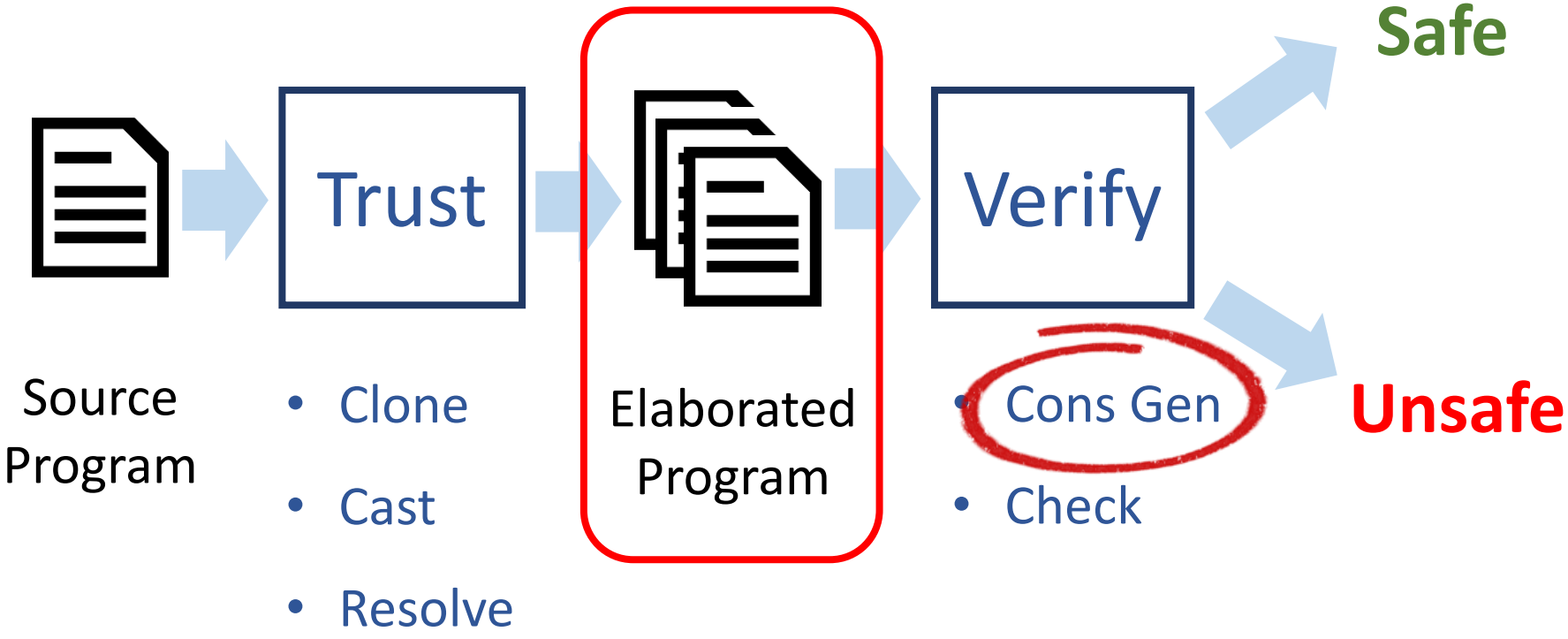
```
function negate#1(flag: tt, x: number): number;
function negate#2(flag: ff, x: boolean): boolean;
```

```
var ok1  = negate#1(1, 1);
var ok2  = negate(0, true);
var bad1 = negate#1(0, 1);
var bad2 = negate(1, true);
```

1st Phase (Trust) – Overload Resolution

```
function negate#1(flag: tt, x: number): number;
function negate#2(flag: ff, x: boolean): boolean;
```

```
var ok1  = negate#1(1, 1);
var ok2  = negate#2(0, true);
var bad1 = negate#1(0, 1);
var bad2 = negate#2(1, true);
```


Safe
Trust
Verify
Source Program
• Clone
• Cast
• Resolve
Elaborated Program
• Cons Gen
• Check
Unsafe

2nd Phase (Verify) – Constraint Generation

Goals:

1) CG for every clone of `negate`

2) CG for call-sites

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  if (flag) return 0 - x;
  else      return !DEAD(x);
}
```

Environment (Γ)	Guards (g)

Flow- and Path-sensitive

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  if (flag) return 0 - x;
  else      return !DEAD(x);
}
```

Environment (Γ)	Guards (g)
flag : {v: number | v ≠ 0} x : number	

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  if (flag) return 0 - x;
  else      return !DEAD(x);
}
```

Environment (Γ)	Guards (g)
flag : {v: number | v ≠ 0} x : number	flag ≠ 0

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  if (flag) return 0 - x;
  else      return !DEAD(x);
}
```

Environment (Γ)	Guards (g)
flag : {v: number | v ≠ 0} x : number	flag = 0

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  if (flag) return 0 - x;
  else      return !DEAD(x);
}
```

Environment (Γ)	Guards (g)
flag : {v: number | v ≠ 0} x : number	flag = 0

```
declare function DEAD<A,B>({v: A | false}): B;
```

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  if (flag) return 0 - x;
  else      return !DEAD(x);
}
```

Environment (Γ)	Guards (g)
flag : {v: number | v ≠ 0} x : number	flag = 0

$$\Gamma, g \vdash \{v: \texttt{number} \mid v = x\} \leq \{v: \texttt{number} \mid \texttt{false}\}$$

```
declare function DEAD<A,B>({v: A | false}): B;
```

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  if (flag) return 0 - x;
  else      return !DEAD(x);
}
```

Environment (Γ)	Guards (g)
flag : {v: number | v ≠ 0} x : number	flag = 0

$$\Gamma, g \vdash \{v: \texttt{number} \mid v = x\} \leq \{v: \texttt{number} \mid \texttt{false}\}$$

To verification condition

Environment and Guards ⇒ LHS ⇒ RHS

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  if (flag) return 0 - x;
  else      return !DEAD(x);
}
```

Environment (Γ)	Guards (g)
flag : {v: number | v ≠ 0} x : number	flag = 0

Γ, g ⊢ {v: number | v = x} ≤ {v: number | false}

flag ≠ 0 ⇒ LHS ⇒ RHS

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  if (flag) return 0 - x;
  else      return !DEAD(x);
}
```

Environment (Γ)	Guards (g)
flag : {v: number | v ≠ 0} x : number	flag = 0

Γ, g ⊢ {v: number | v = x} ≤ {v: number | false}

flag ≠ 0 ∧ true ⇒ LHS ⇒ RHS

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  if (flag) return 0 - x;
  else      return !DEAD(x);
}
```

Environment (Γ)	Guards (g)
flag : {v: number | v ≠ 0} x : number	flag = 0

$$\Gamma, g \vdash \{v: \texttt{number} \mid v = x\} \leq \{v: \texttt{number} \mid \texttt{false}\}$$

$$\texttt{flag} \neq 0 \wedge \texttt{true} \wedge \texttt{flag} = 0 \Rightarrow \boxed{\text{LHS}} \Rightarrow \boxed{\text{RHS}}$$

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  if (flag) return 0 - x;
  else      return !DEAD(x);
}
```

Environment (Γ)	Guards (g)
flag : {v: number | v ≠ 0} x : number	flag = 0

$$\Gamma, g \vdash \{v: \texttt{number} \mid v = x\} \leq \{v: \texttt{number} \mid \texttt{false}\}$$

$$\texttt{flag} \neq 0 \wedge \texttt{true} \wedge \texttt{flag} = 0 \Rightarrow \texttt{LHS} \Rightarrow \texttt{RHS}$$

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  if (flag) return 0 - x;
  else      return !DEAD(x);
}
```

Environment (Γ)	Guards (g)
flag : {v: number | v ≠ 0} x : number	flag = 0

$$\Gamma, g \vdash \{v: \text{number} \mid v = x\} \leq \{v: \text{number} \mid \text{false}\}$$

$$\text{flag} \neq 0 \wedge \text{true} \wedge \text{flag} = 0 \Rightarrow v = x \Rightarrow \text{RHS}$$

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  if (flag) return 0 - x;
  else      return !DEAD(x);
}
```

Environment (Γ)	Guards (g)
flag : {v: number | v ≠ 0} x : number	flag = 0

$$\Gamma, g \vdash \{v: \texttt{number} \mid v = x\} \leq \{v: \texttt{number} \mid \texttt{false}\}$$

$$\texttt{flag} \neq 0 \wedge \texttt{true} \wedge \texttt{flag} = 0 \Rightarrow v = x \Rightarrow \texttt{false}$$

2nd Phase (Verify) – Constraint Generation

Goals:

1) CG for every clone of `negate`

2) CG for call-sites

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  ...
}
var ok1  = negate#1(1, 1);
var bad1 = negate#1(0, 1);
```

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  ...
}
var ok1  = negate#1(1, 1);
var bad1 = negate#1(0, 1);
```

$\vdash$ {v: number | v = 1} $\leq$ {v: number | v $\neq$ 0}

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  ...
}
var ok1  = negate#1(1, 1);
var bad1 = negate#1(0, 1);
```

$\vdash$ {v: number | v = 1} $\leq$ {v: number | v $\neq$ 0}

$\vdash$ {v: number | v = 1} $\leq$ {v: number | true }

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  ...
}
var ok1  = negate#1(1, 1);
var bad1 = negate#1(0, 1);
```

$\vdash$ {v: number | v = 1} $\leq$ {v: number | v $\neq$ 0}

$\vdash$ {v: number | v = 1} $\leq$ {v: number | true }

$\vdash$ {v: number | v = 0} $\leq$ {v: number | v $\neq$ 0}

$\vdash$ {v: number | v = 1} $\leq$ {v: number | true }

2nd Phase (Verify) – Constraint Generation

```
function negate#1(flag: tt, x: number): number {
  ...
}
var ok1  = negate#1(1, 1);
var bad1 = negate#1(0, 1);
```

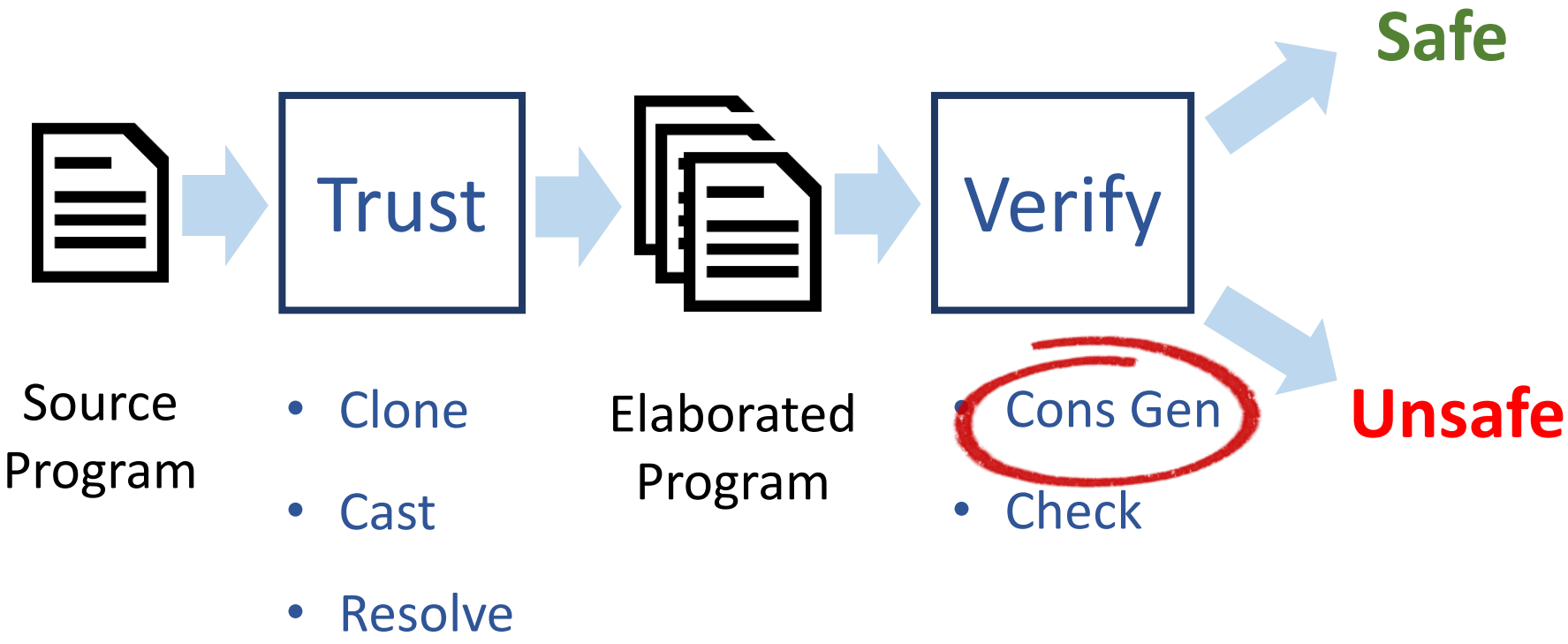
$$v = 1 \Rightarrow v \neq 0$$

$$v = 1 \Rightarrow \texttt{true}$$

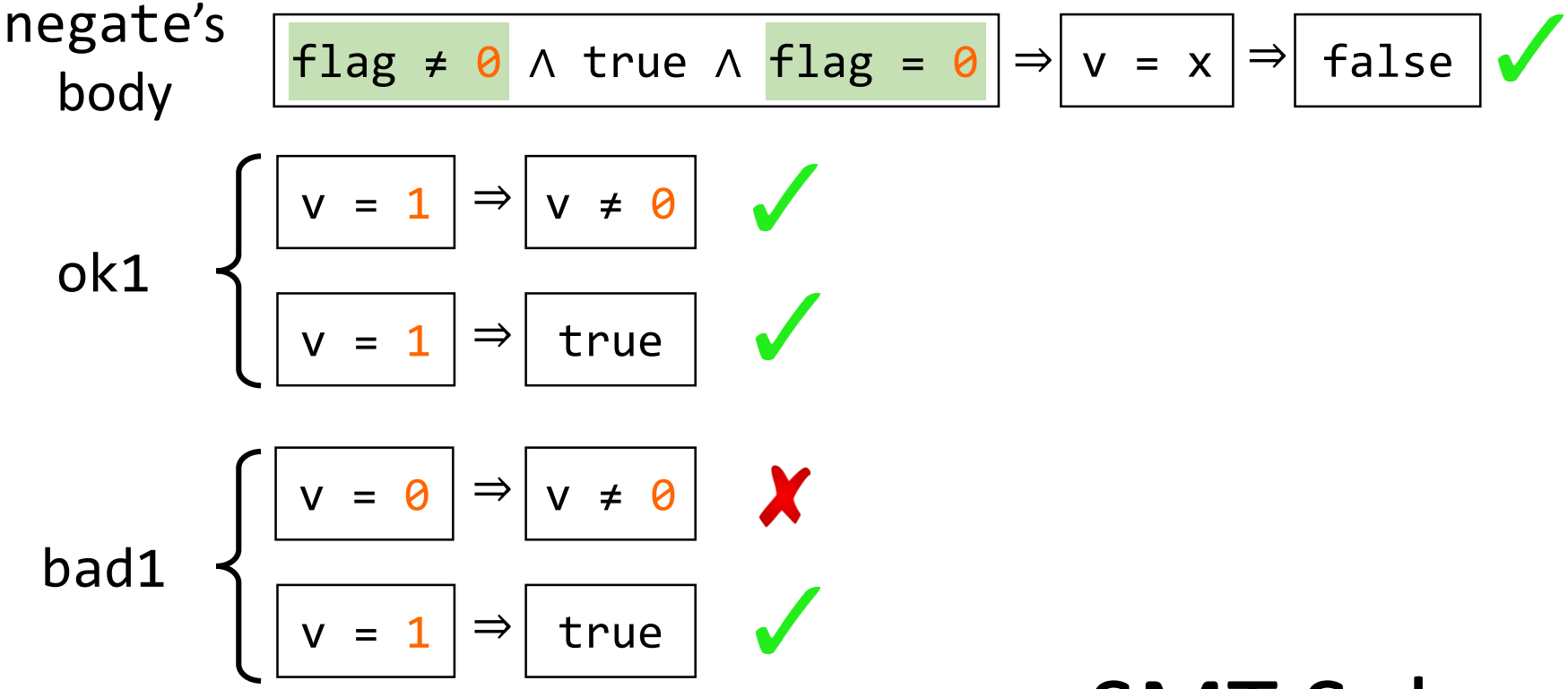
$$v = 0 \Rightarrow v \neq 0$$

$$v = 1 \Rightarrow \texttt{true}$$

To logical implications


Safe
Trust
Verify
Source Program
• Clone
• Cast
• Resolve
Elaborated Program
• Cons Gen
• Check
Unsafe

2nd Phase (Verify) – Check



SMT Solver

Benefits over previous work

Expressive Specification (Flow Typing, Typed Scheme)
Refinements capture complex value relationships

Automated Verification (Dependent JavaScript)
Compose simple type-checkers with program logics

End-To-End Soundness
Semantics preserving elaboration

Source

Value-based overloading:

- Intersections
- Untagged unions

Source

Value-based overloading:
- Intersections
- Untagged unions

$$\Gamma \vdash e :: A$$

Base type-checking (Trust)

Source

$\lambda^{\wedge}_{\vee}$

Value-based overloading:
- Intersections
- Untagged unions

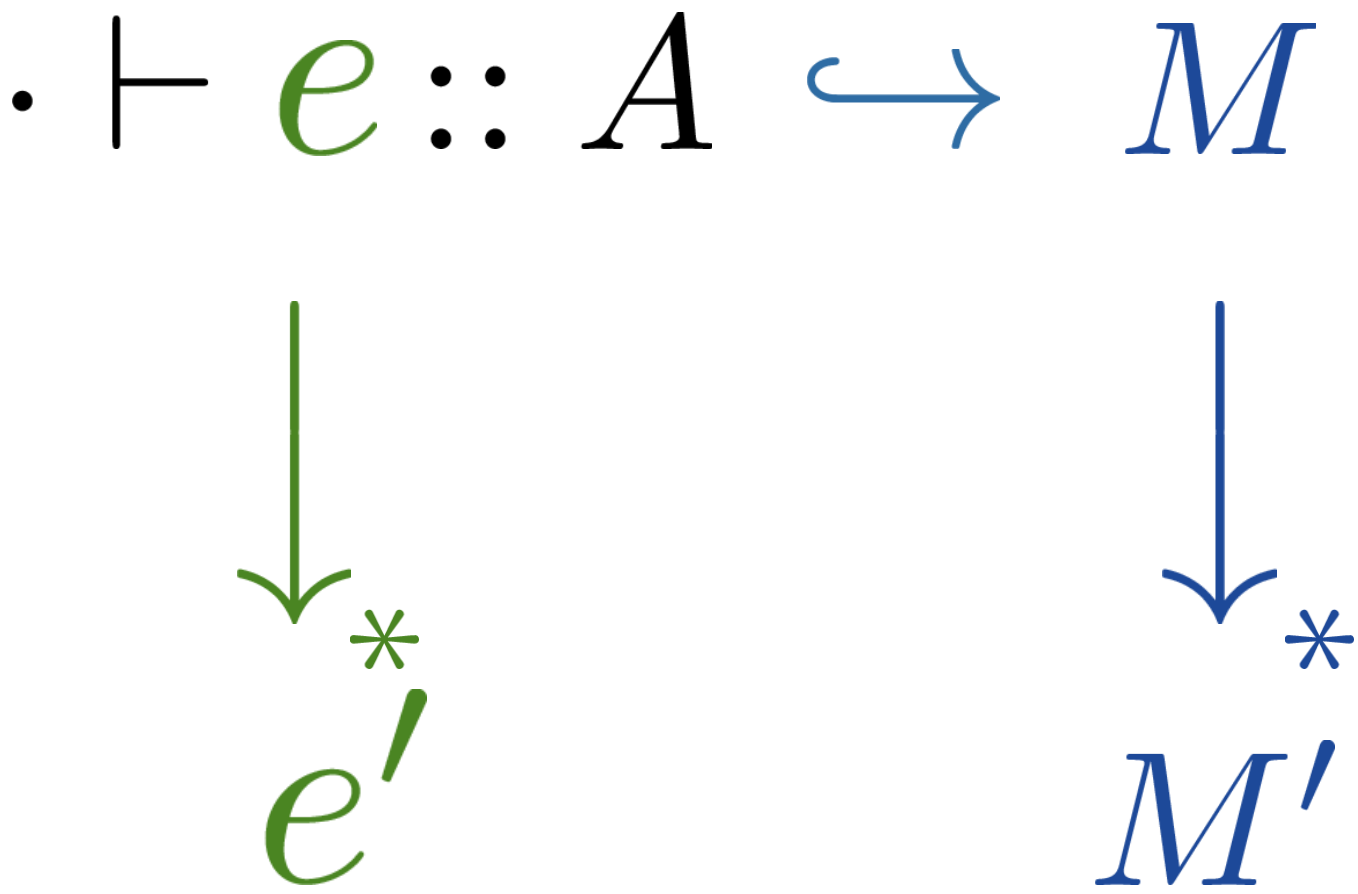
Target

$\lambda^{\times}_{+}$

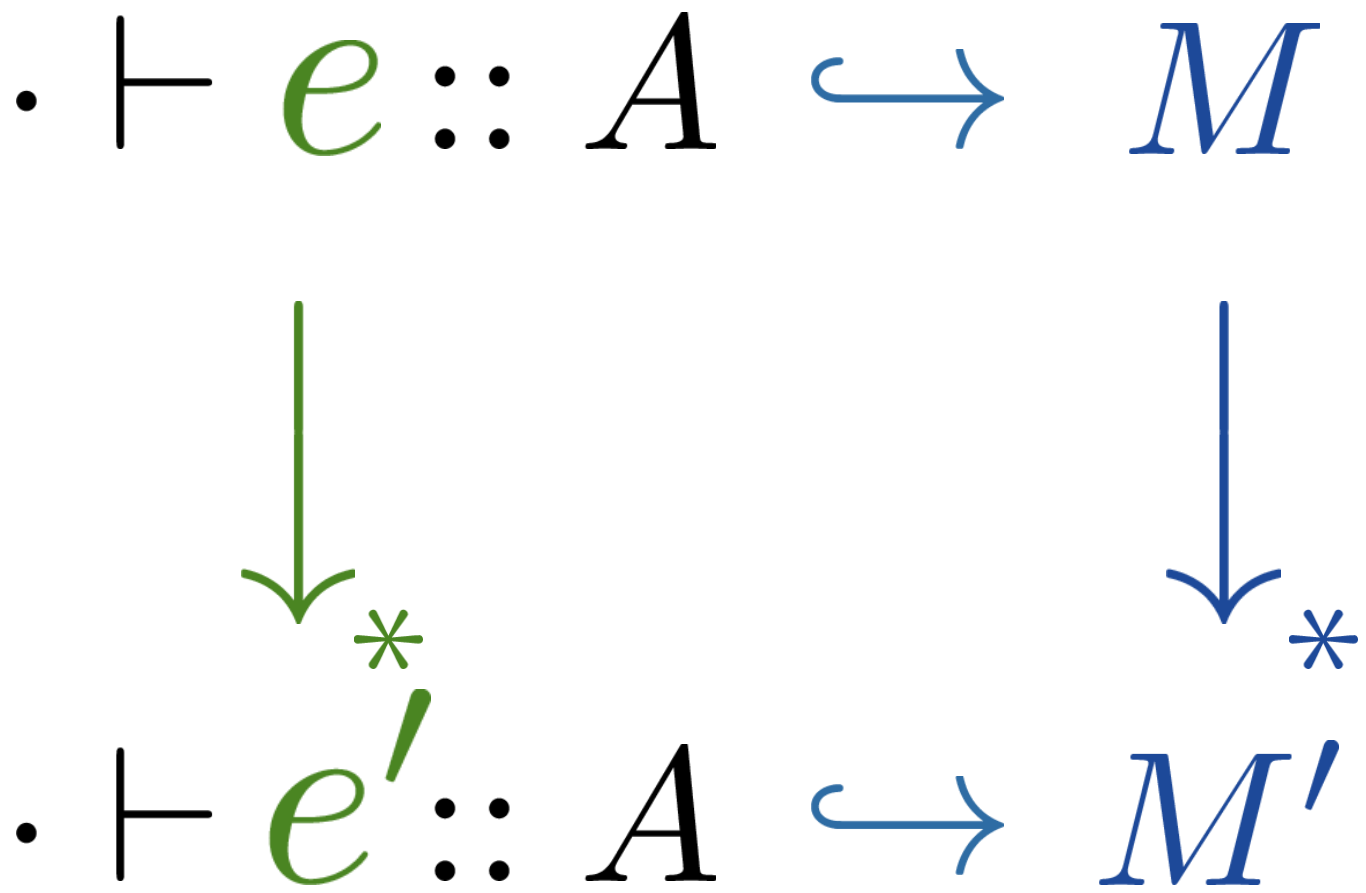
- Products
- Tagged unions
- DEAD-casts

$$\Gamma \vdash e :: A \hookrightarrow M$$

Elaboration
[Dunfield'12]


$\cdot \vdash e :: A \hookrightarrow M$
$*$
e'
$*$
M'

Elaboration Preserves Semantics



End-to-End Type Safety

Base type-checking (Trust)

$$\cdot \vdash e :: A \hookrightarrow M$$

End-to-End Type Safety

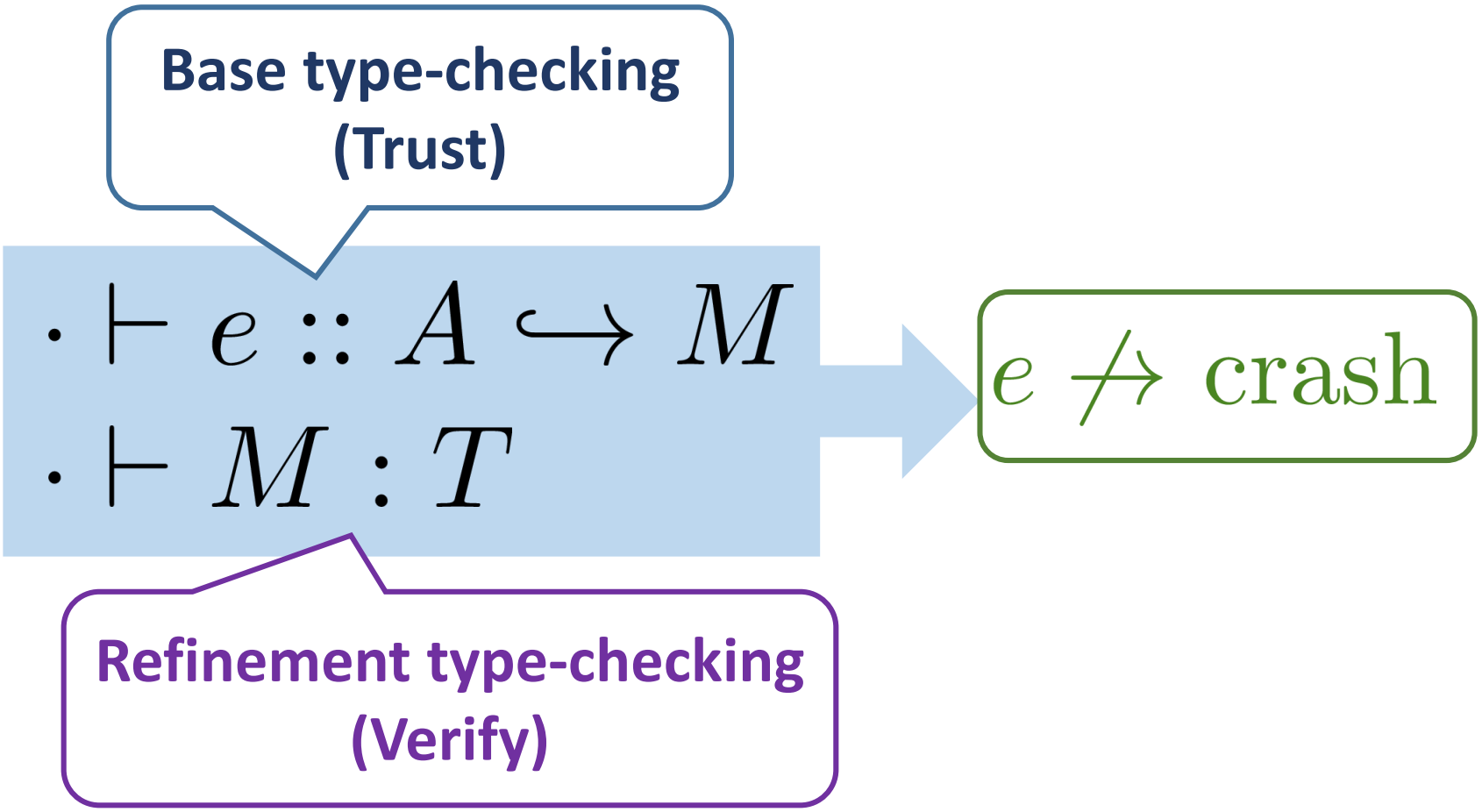
Base type-checking (Trust)

$$\cdot \vdash e :: A \hookrightarrow M$$

$$\cdot \vdash M : T$$

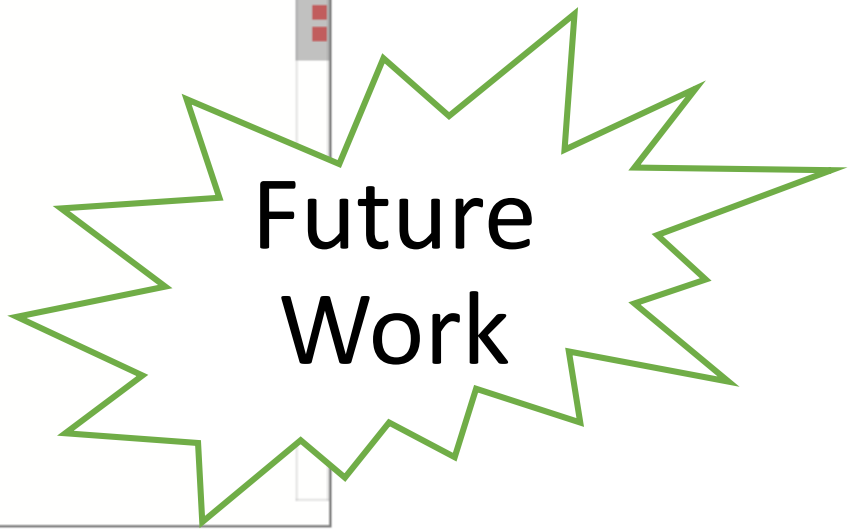
Refinement type-checking (Verify)

End-to-End Type Safety



RefScript negate Share Unsafe

```
1
2  /*@ alias tt = { v: number | v != 0 } */
3  /*@ alias ff = { v: number | v  = 0 } */
4
5  /*@ negate :: /\ (flag: tt, x: number)  => number
6              /\ (flag: ff, x: boolean) => boolean
7  */
8  function negate(flag, x): any {
9    if (flag) {
10     return 0 - x;
11   }
12   else {
13     return !x;
14   }
15 }
16
17 var ok1  = negate(1, 1);
18 var ok2  = negate(0, true);
19 var bad1 = negate(0, 1);
20 var bad2 = negate(1, true);
```



Foundation of refinement type-checker for TypeScript

- OO Features (imperative code, mutation etc)

Trust, but Verify

- Goal: Static Verification of Scripting Languages
- Approach: Refinement Types
- Problem: Overloading
- Solution: Two-Phased Typing
- Results: Elaboration Equivalence & Soundness

Thank you! Questions?