Refinement Types for TypeScript

Panagiotis Vekris Benjamin Cosman Ranjit Jhala

University of California, San Diego

PLDI'16

Thursday, June 16

Extensible static analyses for modern scripting languages

Extensible static analyses for modern scripting languages

TypeScript

Wide scale

Looks like

Compiles to

JS

Extensible static analyses for modern scripting languages

TypeScript

Wide scale

Looks like JS

Compiles to JS

PL Interest

Higher Order Functions

Object Oriented

Optionally Typed

Generics

Extensible static analyses for modern scripting languages

```
class Greeter<T> {
    greeting: T;
    construct (property) Greeter<T>.greeting: T
        this.greeting = message;
    }
    greet() {
        return this.greeting;
    }
}
```

Verification

Documentation

No runtime overhead

Extensible static analyses for modern scripting languages

Fixed type tests

```
typeof x === 'string'

x === null
```

Extensible static analyses for modern scripting languages

Fixed type tests

```
typeof x === 'string'

x === null
```

+

User specified invariants

Extensible static analyses for modern scripting languages

Fixed type tests

```
typeof x === 'string'

x === null
```

+

User specified invariants

```
assert (shape.tag & Circle)
```

Extensible static analyses for modern scripting languages

Fixed type tests

```
typeof x === 'string'

x === null
```

+

User specified invariants

```
assert (shape.tag & Circle)

assert (user.auth())
```

Extensible static analyses for modern scripting languages

Fixed type tests

```
typeof x === 'string'

x === null
```

+

User specified invariants

```
assert (shape.tag & Circle)

assert (user.auth())

assert (i < a.length)
```

Example

Compute the index of the minimum element of an array

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

`reduce` folds over the elements of an array

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

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

Calls `reduce` with an appropriate `step` function and initialization

Example

Compute the index of the minimum element of an array

Verification goal

Prove that all array accesses are within bounds

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

Array bounds analysis:

$0 \leq \mathtt{min} < \mathtt{len}\ a$

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

Array bounds analysis:

$0 \le \mathtt{min} < \mathtt{len}\ a$

⇩

Constraint between **two values**

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

Array bounds analysis:

$0 \le \mathtt{min} < \mathtt{len}\ \mathtt{a}$

⇩

Constraint between

two values

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

Array bounds analysis:

$0 \leq min < \mathrm{len}\ a$

⇩

Constraint between **two values**

⇩

Constraint between **value** and **closure**

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

Constraint between **value** and **closure**

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

Constraint carries over through call to **function parameters**

Constraint between **value** and **closure**

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

Constraint carries over through call to **function parameters**

Constraint **checked** on invocation

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

Constraint carries over through call to **function parameters**

Constraint **checked** on invocation

Problem

To check array access we must track **relations** between **closures** and **values**

Problem

To check array access we must track **relations** between **closures** and **values**

Solution

Refinement types

Refinement Types

$$\{\, \underbrace{v}_{\text{Value variable}} : \underbrace{b}_{\text{Base type}} \mid \underbrace{p}_{\text{Logical predicate}} \,\}$$

Refinement Types

$$\{\ \underbrace{v}_{\text{Value variable}} : \underbrace{b}_{\text{Base type}} \mid \underbrace{p}_{\text{Logical predicate}}\ \}$$

“Set of values v of type b such that formula p is true”

Refinement Types

$$\{\, v : b \mid p \,\}$$

Value variable

Base type

Logical predicate

"Set of values v of type b such that formula p is true"

E.g.: $\{\, v : \texttt{number} \mid 0 \leq v \wedge v < \texttt{len}\ a \,\}$

"Set of valid indexes for an array a"

How can we type `reduce`?

```
function reduce(a, f, x) { ... }
```

How can we type `reduce`?

```
function reduce(a, f, x) { ... }
```

TypeScript type

```
function reduce<A,B>(a: A[], f: (B, A, number) => B, x: B): B { ... }
```

How can we type reduce?

```
function reduce(a, f, x) { ... }
```

TypeScript type

Acc

```
function reduce<A,B>(a: A[], f: (B, A, number) => B, x: B): B { ... }
```

Basic typing offers **some** guarantees

How can we type `reduce`?

```
function reduce(a, f, x) { ... }
```

TypeScript type

Acc

```
function reduce<A,B>(a: A[], f: (B, A, number) => B, x: B): B { ... }
```

Accepts Acc

Basic typing offers **some** guarantees

How can we type `reduce`?

```
function reduce(a, f, x) { ... }
```

TypeScript type

Acc

```
function reduce<A,B>(a: A[], f: (B, A, number) => B, x: B): B { ... }
```

Accepts Acc

Returns Acc

Basic typing offers **some** guarantees

How can we type `reduce`?

```
function reduce(a, f, x) { ... }
```

TypeScript type

```
function reduce<A,B>(a: A[], f: (B, A, number) => B, x: B): B { ... }
```

Does not capture:
"valid index of *a*"

Basic typing offers **some** guarantees
but **not value** related ones

How can we type `reduce` to account for valid indexes?

```
function reduce(a, f, x) { ... }
```

TypeScript type

```
function reduce<A,B>(a: A[], f: (B, A, number) => B, x: B): B { ... }
```

Refinement type

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

How can we type `reduce` to account for valid indexes?

```
function reduce(a, f, x) { ... }
```

TypeScript type

```
function reduce<A,B>(a: A[], f: (B, A, number) => B, x: B): B { ... }
```

Refinement type

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

```
type idx<a> = {v: number | 0 ≤ v ∧ v < len a}
```

How can we type `reduce` to account for valid indexes?

```
function reduce(a, f, x) { ... }
```

TypeScript type

```
function reduce<A,B>(a: A[], f: (B, A, number) => B, x: B): B { ... }
```

Refinement type

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

Captures the relation between **closure** and **value**

Our contribution

Design a refinement type system for TypeScript

Challenges
Assignments `while (i < n) { i++; }`

Mutability
`var x = { f: 1 };`
`x.f = 2;`

Overloading
`foo(x: number): number`
`foo(x: boolean): boolean`

Annotation Overhead

Challenges	Solutions we used
Assignments `while (i < n) { i++; }`	SSA Transformation

Mutability
`var x = { f: 1 };`
`x.f = 2;`
	Extend type system with immutability guarantees

Overloading
`foo(x: number): number`
`foo(x: boolean): boolean`
	Two-phased typing

Annotation Overhead	Liquid Types

Challenges	Solutions we used
Assignments `while (i < n) { i++; }`	SSA Transformation

Mutability
`var x = { f: 1 };`
`x.f = 2;`
	Extend type system with immutability guarantees

Overloading
`foo(x: number): number`
`foo(x: boolean): boolean`
	Two-phased typing

Annotation Overhead	Liquid Types

Assignments

```
while (i < n) { i++; }
```

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

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

```diff
function reduce(a, f, x) {
  var r = x;
+ var i = 0;
+ while (i < a.length) {
+   r = f(r, a[i], i);
+   i = i + 1;
+ }
  return r;
}
```

What is the type of `i`?

```
function reduce(a, f, x) {
    var r = x;
    var i = 0;
    while (i < a.length) {
        r = f(r, a[i], i);
        i = i + 1;
    }
    return r;
}
```

What is the type of `i`?

```
function reduce(a, f, x) {
    var r = x;
    var i = 0;
    while (i < a.length) {
        r = f(r, a[i], i);
        i = i + 1;
    }
    return r;
}
```

Types for `i`

{ number | v = 0 }

What is the type of `i`?

```
function reduce(a, f, x) {
    var r = x;
    var i = 0;
    while (i < a.length) {
        r = f(r, a[i], i);
        i = i + 1;
    }
    return r;
}
```

Types for `i`

$\{\ \texttt{number} \mid v = 0\ \}$

$\{\ \texttt{number} \mid 0 \le v \le \texttt{len}\ a\ \}$

What is the type of `i`?

```
function reduce(a, f, x) {
    var r = x;
    var i = 0;
    while (i < a.length) {
        r = f(r, a[i], i);
        i = i + 1;
    }
    return r;
}
```

Types for `i`

$\{ \mathtt{number} \mid v = 0 \}$

$\{ \mathtt{number} \mid 0 \leq v \leq \mathtt{len}\ a \}$

$\{ \mathtt{number} \mid v = \mathtt{i} + 1 \}$

What is the type of `i`?

```
function reduce(a, f, x) {
    var r = x;
    var i = 0;
    while (i < a.length) {
        r = f(r, a[i], i);
        i = i + 1;
    } // i
    return r;
}
```

Types for `i`

{ number | v = 0 }

{ number | 0 ≤ v ≤ len a }

{ number | v = i + 1 }

{ number | v = len a }

No single type for `i`

```
function reduce(a, f, x) {
    var r = x;
    var i = 0;
    while (i < a.length) {
        r = f(r, a[i], i);
        i = i + 1;
    } // i
    return r;
}
```

Types for `i`

{ number | v = 0 }
{ number | 0 ≤ v ≤ len a }
{ number | v = i + 1 }
{ number | v = len a }

No single type for `i`

Joining types of `i` causes **loss of precision**

No single type for `i`

Joining types of `i` causes **loss of precision**

Use different versions of `i`

Use different versions of `i`

Use different versions of `i`

```
function reduce(a, f, x) {
    var r = x;
    var i = 0;
    while (i < a.length) {
        r = f(r, a[i], i);
        i = i + 1;
    } // i
    return r;
}
```

Types for `i`

{ number | v = 0 }

{ number | 0 ≤ v ≤ len a }

{ number | v = i + 1 }

{ number | v = len a }

Use different versions of `i`

```
function reduce(a, f, x) {
    var r = x;
    var i1 = 0;
    while (i2 < a.length) {
        r = f(r, a[i2], i2);
        i3 = i2 + 1;
    } // i4
    return r;
}
```

Types for i_1-i_4

i_1: { number | $v = 0$ }

i_2: { number | $0 \le v \le$ len a }

i_3: { number | $v = i_2 + 1$ }

i_1: { number | $v =$ len a }

Each version of `i` has a single precise type & gets assigned once

```
function reduce(a, f, x) {
    var r = x;
    var i₁ = 0;
    while (i₂ < a.length) {
        r = f(r, a[i₂], i₂);
        i₃ = i₂ + 1;
    } // i₄
    return r;
}
```

Types for i_1-i_4

i_1: { number | $v = 0$ }
i_2: { number | $0 \leq v \leq \text{len } a$ }
i_3: { number | $v = i_2 + 1$ }
i_4: { number | $v = \text{len } a$ }

Static Single Assignment (SSA)

```
function reduce(a, f, x) {
    var r = x;
    var i₁ = 0;
    while (i₂ < a.length) {
        r = f(r, a[i₂], i₂);
        i₃ = i₂ + 1;
    } // i₄
    return r;
}
```

Types for i_1-i_4

i_1: { number | $v = 0$ }
i_2: { number | $0 \leq v \leq \text{len } a$ }
i_3: { number | $v = i_2 + 1$ }
i_4: { number | $v = \text{len } a$ }

Static Single Assignment (SSA)

```
function reduce(a, f, x) {
    var r = x;
    var i1 = 0;
    while (i2 < a.length) {
        r = f(r, a[i2], i2);
        i3 = i2 + 1;
    } // i4
    return r;
}
```

Types for i_1-i_4

i_1: { number | $v = 0$ }

i_2: { number | $0 \leq v \leq \mathrm{len}\ a$ }

i_3: { number | $v = i_2 + 1$ }

i_4: { number | $v = \mathrm{len}\ a$ }

How do we check these types?

Reminder

Assignment

`x = e`

generates subtyping constraint

`Type(e) <: Type(x)`

Subtyping Constraints

```
function reduce(a, f, x) {
    var r = x;
    var i₁ = 0;
    while (i₂ < a.length) {
        r = f(r, a[i₂], i₂);
        i₃ = i₂ + 1;
    } // i₄
    return r;
}
```

Generated constraints

$\text{Type}(0) \quad <: \quad \text{Type}(i_1)$

Subtyping Constraints

```
function reduce(a, f, x) {
    var r = x;
    var i₁ = 0;
    while (i₂ < a.length) {
        r = f(r, a[i₂], i₂);
        i₃ = i₂ + 1;
    } // i₄
    return r;
}
```

i: loop induction variable

$i_2 = \phi(i_1, i_3)$

Generated constraints

Type(0) <: Type(i_1)

Subtyping Constraints

```
function reduce(a, f, x) {
    var r = x;
    var i₁ = 0;
    while (i₂ < a.length) {
        r = f(r, a[i₂], i₂);
        i₃ = i₂ + 1;
    } // i₄
    return r;
}
```

i: loop induction variable

$i_2 = \phi(i_1, i_3)$

Generated constraints

Type(0)	<:	Type(i_1)
Type(i_1)	<:	Type(i_2)

Subtyping Constraints

```
function reduce(a, f, x) {
    var r = x;
    var i₁ = 0;
    while (i₂ < a.length) {
        r = f(r, a[i₂], i₂);
        i₃ = i₂ + 1;
    } // i₄
    return r;
}
```

i: loop induction variable

$$i_2 = \phi(i_1, i_3)$$

Generated constraints

	Type(0)	<:	$\mathrm{Type}(i_1)$
	$\mathrm{Type}(i_1)$	<:	$\mathrm{Type}(i_2)$
loop_cond ⊢	$\mathrm{Type}(i_3)$	<:	$\mathrm{Type}(i_2)$

Subtyping Constraints

```
function reduce(a, f, x) {
    var r = x;
    var i₁ = 0;
    while (i₂ < a.length) {
        r = f(r, a[i₂], i₂);
        i₃ = i₂ + 1;
    } // i₄
    return r;
}
```

i: loop induction variable

$i_2 = \phi(i_1, i_3)$

Loop condition

$i_2 < \mathrm{len}\ a$

Generated constraints

$\mathrm{Type}(0) <: \mathrm{Type}(i_1)$

$\mathrm{Type}(i_1) <: \mathrm{Type}(i_2)$

$\mathrm{loop\_cond} \vdash \mathrm{Type}(i_3) <: \mathrm{Type}(i_2)$

Path Sensitivity

Subtyping Constraints

```
function reduce(a, f, x) {
    var r = x;
    var i₁ = 0;
    while (i₂ < a.length) {
        r = f(r, a[i₂], i₂);
        i₃ = i₂ + 1;
    } // i₄
    return r;
}
```

Loop condition

$i_2 < \mathrm{len}\; a$

Generated constraints

$$\mathrm{Type}(0) <: \mathrm{Type}(i_1)$$
$$\mathrm{Type}(i_1) <: \mathrm{Type}(i_2)$$
$$\mathrm{loop\_cond} \vdash \mathrm{Type}(i_3) <: \mathrm{Type}(i_2)$$
$$\mathrm{loop\_cond} \vdash \mathrm{Type}(i_2 + 1) <: \mathrm{Type}(i_3)$$

Subtyping Constraints

```
function reduce(a, f, x) {
    var r = x;
    var i₁ = 0;
    while (i₂ < a.length) {
        r = f(r, a[i₂], i₂);
        i₃ = i₂ + 1;
    } // i₄
    return r;
}
```

Safe Array Access

Loop condition

$i_2 < \mathrm{len}\ a$

Generated constraints

$$\mathrm{Type}(0) <: \mathrm{Type}(i_1)$$

$$\mathrm{Type}(i_1) <: \mathrm{Type}(i_2)$$

$$\mathrm{loop\_cond} \vdash \mathrm{Type}(i_3) <: \mathrm{Type}(i_2)$$

$$\mathrm{loop\_cond} \vdash \mathrm{Type}(i_2 + 1) <: \mathrm{Type}(i_3)$$

$$\mathrm{loop\_cond} \vdash \mathrm{Type}(i_2) <: \mathrm{idx}\langle a \rangle$$

Subtyping Constraints

i_1: { number | v = 0 }
i_2: { number | 0 ≤ v ≤ len a }
i_3: { number | v = i_2 + 1 }
i_4: { number | v = len a }

Loop condition

i_2 < len a

Substitute

Generated constraints

	Type(0)	<:	Type(i_1)
	Type(i_1)	<:	Type(i_2)
loop_cond ⊢	Type(i_3)	<:	Type(i_2)
loop_cond ⊢	Type(i_2 + 1)	<:	Type(i_3)
loop_cond ⊢	Type(i_2)	<:	idx<a>

Subtyping Constraints

After substitution

$$\{\, num \mid v = 0 \,\} <: \{\, num \mid v = 0 \,\}$$

$$\{\, num \mid v = 0 \,\} <: \{\, num \mid 0 \leq v \leq \mathtt{len}\ a \,\}$$

$$i_2 < \mathtt{len}\ a \vdash \{\, num \mid v = i_2 + 1 \,\} <: \{\, num \mid 0 \leq v \leq \mathtt{len}\ a \,\}$$

$$i_2 < \mathtt{len}\ a \vdash \{\, num \mid v = i_2 + 1 \,\} <: \{\, num \mid v = i_2 + 1 \,\}$$

$$i_2 < \mathtt{len}\ a \vdash \{\, num \mid 0 \leq v \leq \mathtt{len}\ a \,\} <: \{\, num \mid 0 \leq v < \mathtt{len}\ a \,\}$$

Subtyping Constraints

Convert to logical implications

$$
\begin{array}{rcrcr}
 & & v = 0 & \Rightarrow & v = 0 \\
 & & v = 0 & \Rightarrow & 0 \leq v \leq \mathtt{len}\ a \\
i_2 < \mathtt{len}\ a & \Rightarrow & v = i_2 + 1 & \Rightarrow & 0 \leq v \leq \mathtt{len}\ a \\
i_2 < \mathtt{len}\ a & \Rightarrow & v = i_2 + 1 & \Rightarrow & v = i_2 + 1 \\
i_2 < \mathtt{len}\ a & \Rightarrow & 0 \leq v \leq \mathtt{len}\ a & \Rightarrow & 0 \leq v < \mathtt{len}\ a
\end{array}
$$

Subtyping Constraints

Convert to logical implications

Solved via SMT

$$
\begin{array}{rcrcr}
 & & v = 0 & \Rightarrow & v = 0 \\
 & & v = 0 & \Rightarrow & 0 \leq v \leq \mathtt{len}\ a \\
i_2 < \mathtt{len}\ a & \Rightarrow & v = i_2 + 1 & \Rightarrow & 0 \leq v \leq \mathtt{len}\ a \\
i_2 < \mathtt{len}\ a & \Rightarrow & v = i_2 + 1 & \Rightarrow & v = i_2 + 1 \\
i_2 < \mathtt{len}\ a & \Rightarrow & 0 \leq v \leq \mathtt{len}\ a & \Rightarrow & 0 \leq v < \mathtt{len}\ a
\end{array}
$$

Challenges	Solutions we used
Assignments `while (i < n) { i++; }`	SSA Transformation

Mutability
`var x = { f: 1 };`
`x.f = 2;`
	Extend type system with immutability guarantees

Overloading
`foo(x: number): number`
`foo(x: boolean): boolean`
	Two-phased typing

Annotation Overhead	Liquid Types

Mutability

```
var x = { f: 1 };
x.f = 2;
```

Why is the access `a[i]` safe?

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

Why is the access `a[i]` safe?

```
function reduce(a, f, x) {
  var res = x; 1
  for (var i = 0; i < a.length; i++)
    res = f(res, a[i], i);
  return res;
}
```

1. `i` is initialized to `0`

Why is the access `a[i]` safe?

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

1. `i` is initialized to `0`
2. `i` is bounded by `a`'s length

Why is the access `a[i]` safe?

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

1 2 3

1. `i` is initialized to `0`
2. `i` is bounded by `a`'s length
3. `i` increases only

Why is the access `a[i]` safe?

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

1. `i` is initialized to `0`
2. `i` is bounded by `a`'s length
3. `i` increases only
4. **Length of `a` does not mutate** in loop

What if array’s length mutates in loop?

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

lib.d.ts

```
interface Array<T> {
  /**
   *  Removes the last element from an array and returns it.
   */
  pop(): T;
}
```

What if array's length mutates in loop?

Silently updates `a.length`

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

lib.d.ts

```
interface Array<T> {
  /**
   *   Removes the last element from an array and returns it.
   */
  pop(): T:
}
```

What if array's length mutates in loop?

Check becomes stale

Silently updates a.length

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

lib.d.ts

```
interface Array<T> {
  /**
   *  Removes the last element from an array and returns it.
   */
  pop(): T:
}
```

What if array's length mutates in loop?

Check becomes stale

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

Silently updates a.length

Unsafe access!

```
interface Array<T> {
  /**
   *  Removes the last element from an array and returns it.
   */
  pop(): T:
}
```

lib.d.ts

Problem: stale checks break value reasoning

Check becomes stale

Silently updates a.length

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

Unsafe access!

lib.d.ts

```
interface Array<T> {
  /**
   *  Removes the last element from an array and returns it.
   */
  pop(): T;
}
```

Extend type system to enforce immutability constraints

Literature in Object & Reference Immutability

M. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. OOPSLA, 2005.

Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kiezun, and M. D. Ernst. Object and Reference Immutability using Java Generics. ESEC/FSE, 2007.

Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and Immutability in Generic Java. OOPSLA, 2010.

C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness & Reference Immutability for Safe Parallelism. OOPSLA, 2012.

C. S. Gordon, M. D. Ernst, and D. Grossman. Rely-Guarantee References for Refinement Types over Aliased Mutable Data. PLDI, 2013.

F. Militão, J. Aldrich, and L. Caires. Rely-Guarantee Protocols. ECOOP, 2014.

Literature in Object & Reference Immutability

Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kiezun, and M. D. Ernst. Object and Reference Immutability using Java Generics. ESEC/FSE, 2007.

- ✓ Simple extension to type system
- ✓ Encoded in base types – refinements leverage immutability guarantees

Immutability Generic Java [Zibin'07]

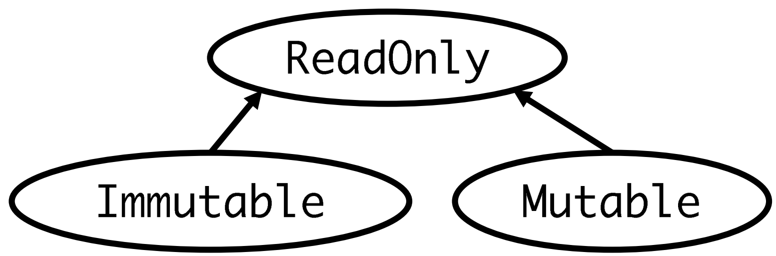
```
function reduce<A,B>(a: Array<Immutable,A>,
                     f: (B,A,idx<a>) => B,
                     x: B): B {
  var res = x;
  for (let i = 0; i < a.length; i++)
    res = f(res, a[i], i);
  return res;
}
```

Mutability as type parameter

Immutability Generic Java [Zibin'07]

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

Mutability as type parameter



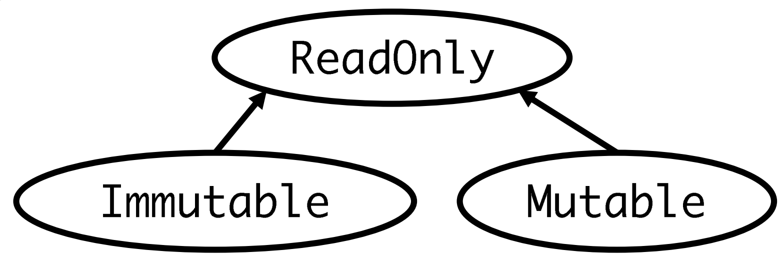
	This can mutate?	Others can mutate?
ReadOnly	✗	✓
Mutable	✓	✓
Immutable	✗	✗

Immutability Generic Java [Zibin'07]

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

Mutability as type parameter

Only immutable portions in refinement



	This can mutate?	Others can mutate?
ReadOnly	✗	✓
Mutable	✓	✓
Immutable	✗	✗

Immutability Generic Java [Zibin'07]

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

lib-IGJ.d.ts

```
interface Array<M extends ReadOnly, T> {
  /**
   * Removes the last element from an array and returns it.
   */
  /*@ Mutable */ pop(): T;
}
```

Immutability Generic Java [Zibin'07]

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

Call to pop is flagged as an error, because pop may only be applied to Mutable receivers

lib-IGJ.d.ts

```
interface Array<M extends ReadOnly, T> {
  /**
   * Removes the last element from an array and returns it.
   */
  /*@ Mutable */ pop(): T;
}
```

Challenges	Solutions we used
Assignments `while (i < n) { i++; }`	SSA Transformation

Mutability
`var x = { f: 1 };`
`x.f = 2;`
	Extend type system with immutability guarantees

Overloading
`foo(x: number): number`
`foo(x: boolean): boolean`
	Two-phased typing

Annotation Overhead	Liquid Types

Value Based

Overloading

```
foo(x: number): number
foo(x: boolean): boolean
```

Value Based Overloading

Function reflects upon and behaves according to types of its arguments

Value Based Overloading

Function reflects upon and behaves according to types of its arguments

```
function $reduce(a, f, x?) {
  if (arguments.length === 3)
    return reduce(a, f, x);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

#args	Signature
2	
3	

Value Based Overloading

Function reflects upon and behaves according to types of its arguments

```
function $reduce(a, f, x?) {
  if (arguments.length === 3)
    return reduce(a, f, x);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

1st behavior – 3 args:
x is of type B

#args	Signature
2	
3	`<A,B>(a: A[] , f: (B,A,idx<a>) => B, x: B): B`

Value Based Overloading

Function reflects upon and behaves according to types of its arguments

```
function $reduce(a, f, x?) {
  if (arguments.length === 3)
    return reduce(a, f, x);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

2nd behavior – 2 args:
x is of type
undefined

#args	Signature
2	`<A> (a: A[]⁺, f: (A,A,idx<a>) => A ): A`
3	`<A,B>(a: A[] , f: (B,A,idx<a>) => B, x: B): B`

Value Based Overloading

Function reflects upon and behaves according to types of its arguments

Q1: What makes it challenging?

Q2: How pervasive is it?

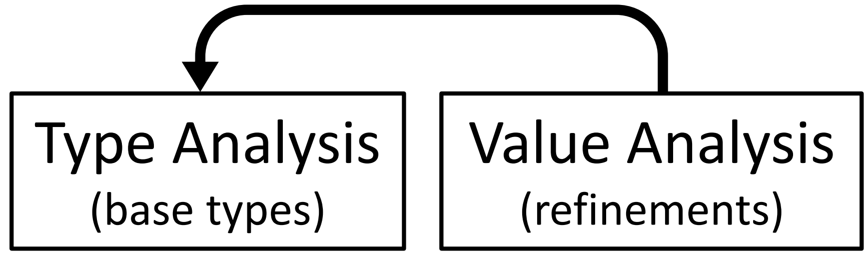
Q1: What makes it challenging?

```
function $reduce(a, f, x?) {
  if (arguments.length === 3)
    return reduce(a, f, x);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

Type Analysis (base types)	Value Analysis (refinements)

Q1: What makes it challenging?

```
function $reduce(a, f, x?) {
  if (arguments.length === 3)
    return reduce(a, f, x);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

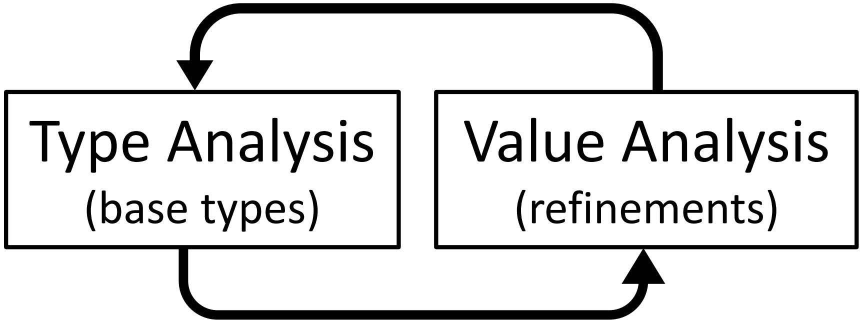


Refinements use invariants established by base types

E.g. tracking the `.length` access requires `arguments` to be array

Q1: What makes it challenging?

```
function $reduce(a, f, x?) {
  if (arguments.length === 3)
    return reduce(a, f, x);
  else
    return reduce(a.slice(1), f, a[0]);
}
```



Refinements use invariants established by base types

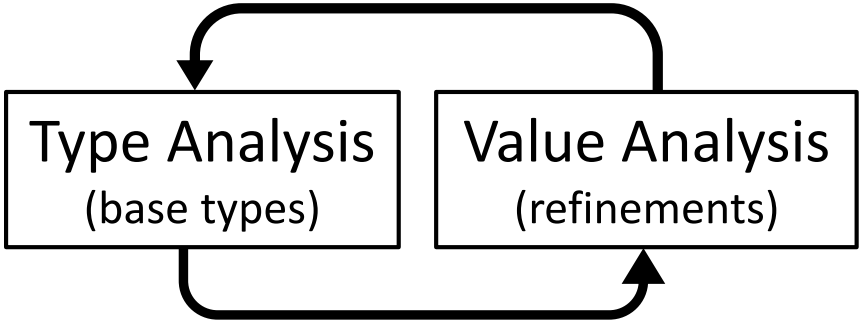
E.g. tracking the `.length` access requires `arguments` to be array

Type reasoning requires tracking logical relationships

E.g. base type of `x` depends on the value of `arguments.length`

Q1: What makes it challenging?

```
function $reduce(a, f, x?) {
  if (arguments.length === 3)
    return reduce(a, f, x);
  else
    return reduce(a.slice(1), f, a[0]);
}
```



Refinements use invariants established by base types

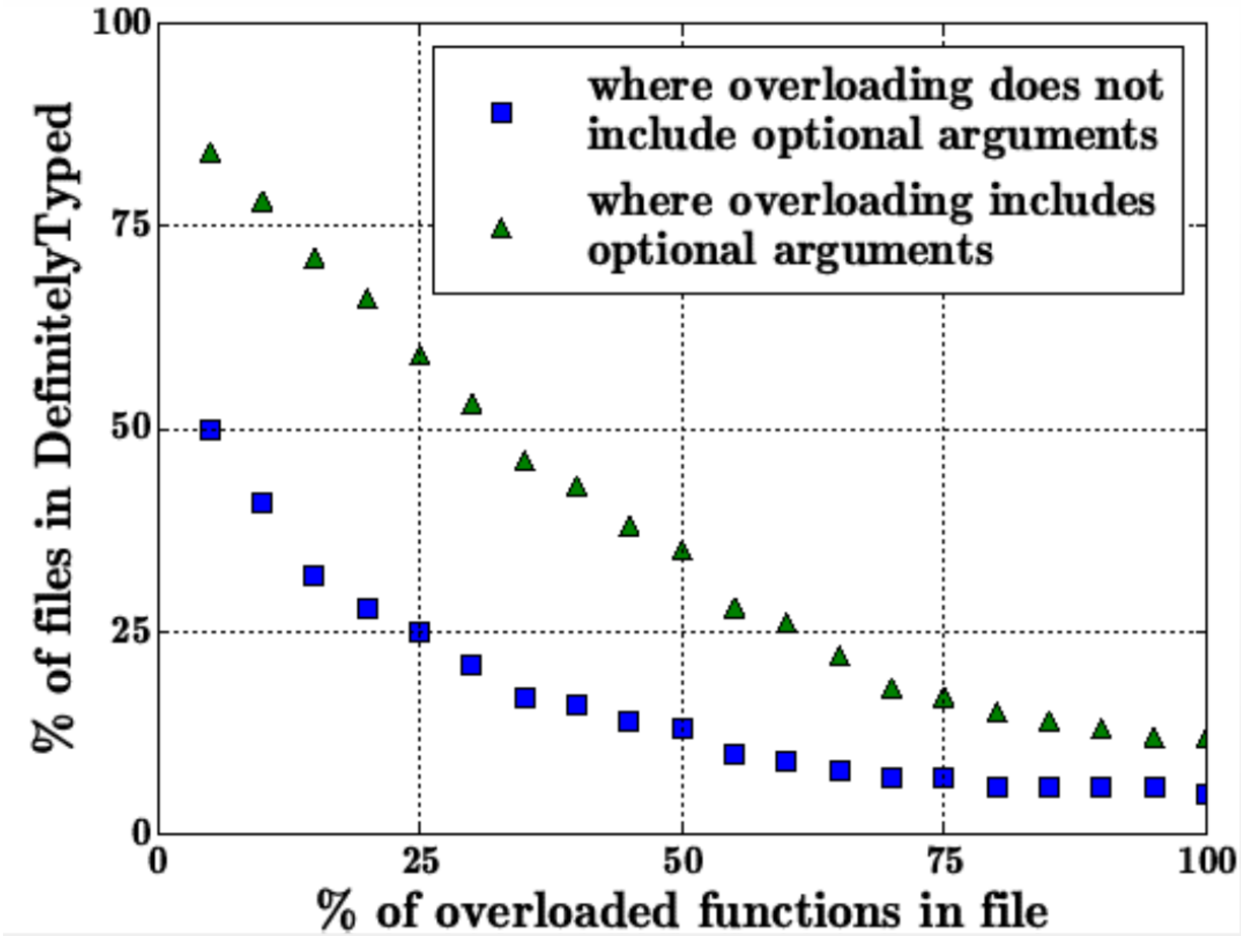
E.g. tracking the `.length` access requires `arguments` to be array

Type reasoning requires tracking logical relationships

E.g. base type of `x` depends on the value of `arguments.length`

Circular dependency complicates **formal reasoning** & **implementation**

Q2: How pervasive is it?

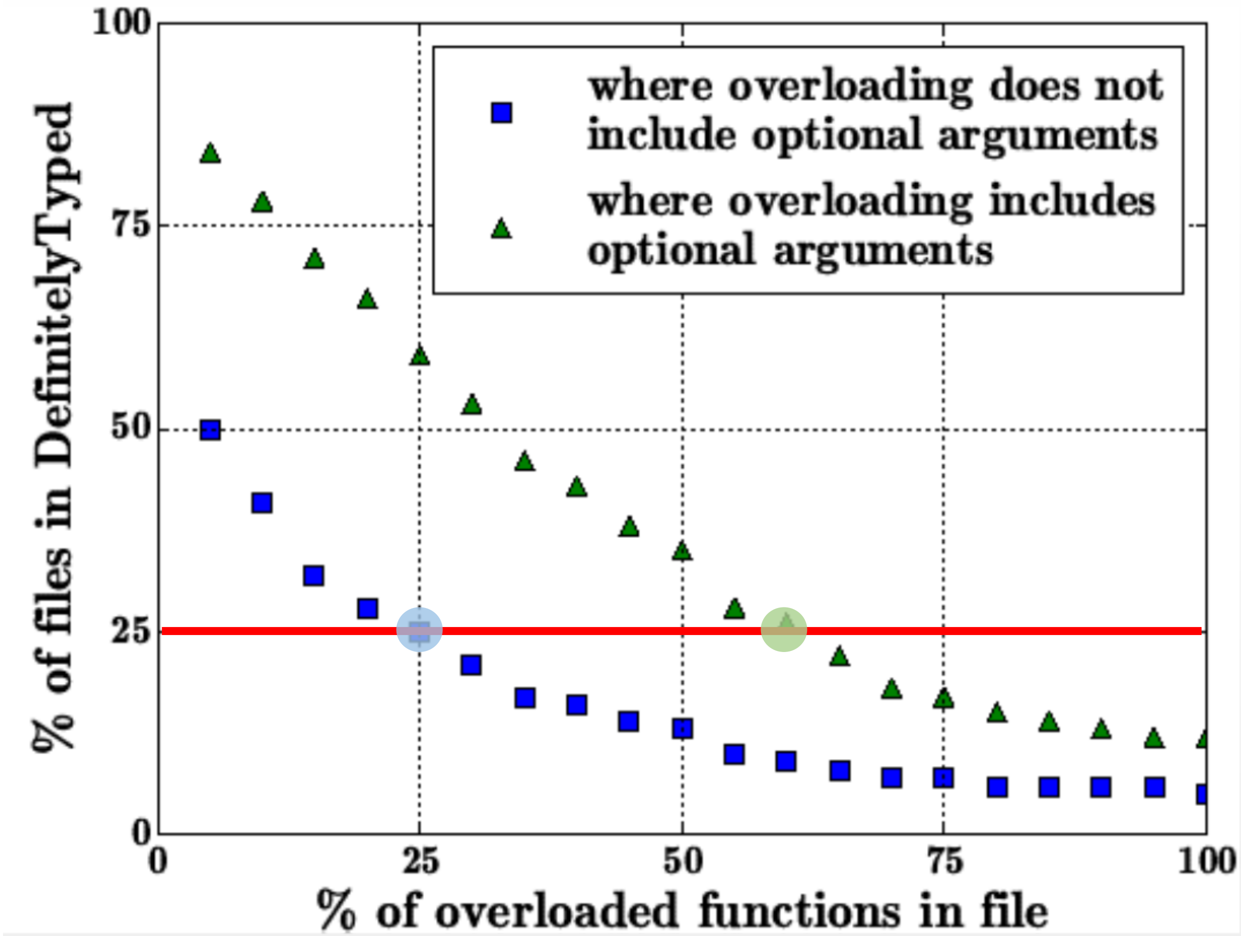


Study set:

DefinitelyTyped: The repository for high quality TypeScript type definitions

`http://definitelytyped.org/`

Q2: How pervasive is it?



Study set:

DefinitelyTyped: The repository for high quality TypeScript type definitions

`http://definitelytyped.org/`

How do we check overloaded functions?

Two-Phased Typing [ECOOP'15]

How do we check overloaded functions?

Two-Phased Typing [ECOOP'15]

```
function $reduce<A>  (a: A[]+, f: (A,A,idx<a>) => A      ): A
function $reduce<A,B>(a: A[] , f: (B,A,idx<a>) => B, x: B): B
function $reduce(a, f, x?) {
  if (arguments.length === 3)
    return reduce(a, f, x);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

How do we check overloaded functions?

Two-Phased Typing [ECOOP'15]

```
function $reduce<A>  (a: A[]+, f: (A,A,idx<a>) => A      ): A
function $reduce<A,B>(a: A[] , f: (B,A,idx<a>) => B, x: B): B
function $reduce(a, f, x?) {
  if (arguments.length === 3)
    return reduce(a, f, x);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

```
function $reduce<A>(a: A[]+, f: (A,A,idx<a>) => A): A {
  if (arguments.length === 3)
    return reduce(a, f, x);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

```
function $reduce<A,B>(a: A[] , f: (B,A,idx<a>) => B, x: B): B {
  if (arguments.length === 3)
    return reduce(a, f, x);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

Phase 1a. Make clones of body for each overload

How do we check overloaded functions?

Two-Phased Typing [ECOOP'15]

```
function $reduce<A>  (a: A[]+, f: (A,A,idx<a>) => A      ): A
function $reduce<A,B>(a: A[] , f: (B,A,idx<a>) => B, x: B): B
function $reduce(a, f, x?) {
  if (arguments.length === 3)
    return reduce(a, f, x);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

```
function $reduce<A>(a: A[]+, f: (A,A,idx<a>) => A): A {
  if (arguments.length === 3)
    return reduce(a, f, x);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

```
function $reduce<A,B>(a: A[] , f: (B,A,idx<a>) => B, x: B): B {
  if (arguments.length === 3)
    return reduce(a, f, x);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

Phase 1a. Make clones of body for each overload

How do we check overloaded functions?

Two-Phased Typing [ECOOP'15]

```
function $reduce#1<A>(a: A[]⁺, f: (A,A,idx<a>) => A): A {
  if (arguments.length === 3)
    return reduce(a, f, x);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

Phase 1b. Check body under clone signature

How do we check overloaded functions?

Two-Phased Typing [ECOOP'15]

```
function $reduce#1<A>(a: A[]⁺, f: (A,A,idx<a>) => A): A {
  if (arguments.length === 3)
    return reduce(a, f, x);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

Error: expecting type A, passed x of type undefined

Value- and path-insensitive type-checking

Phase 1b. Check body under clone signature

How do we check overloaded functions?

Two-Phased Typing [ECOOP’15]

```
function $reduce#1<A>(a: A[]+, f: (A,A,idx<a>) => A): A {
  if (arguments.length === 3)
    return assert(false);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

Replace errors with `assert(false)`, trusting they are indeed dead-code

Phase 1b. Check body under clone signature

How do we check overloaded functions?

Two-Phased Typing [ECOOP'15]

```
function $reduce#1<A>(a: A[]⁺, f: (A,A,idx<a>) => A): A {
  if (arguments.length === 3)
    return assert(false);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

Prove dead-code with flow- and path-sensitive analysis

Phase 2. Refinement Type Checking

How do we check overloaded functions?

Two-Phased Typing [ECOOP'15]

Signature implies: `arguments.length = 2`

```
function $reduce#1<A>(a: A[]⁺, f: (A,A,idx<a>) => A): A {
  if (arguments.length === 3)
    return assert(false);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

Condition makes branch's environment inconsistent

Prove dead-code with flow- and path-sensitive analysis

Phase 2. Refinement Type Checking

How do we check overloaded functions?

Two-Phased Typing [ECOOP’15]

Signature implies: `arguments.length = 2`

```
function $reduce#1<A>(a: A[]+, f: (A,A,idx<a>) => A): A {
  if (arguments.length === 3)
    return assert(false);
  else
    return reduce(a.slice(1), f, a[0]);
}
```

Condition makes branch’s environment inconsistent

Prove dead-code with flow- and path-sensitive analysis

Phase 2. Refinement Type Checking

Also in the paper…

Scaling to TypeScript

- Type features

  Object literal types
  Interface types
  Primitive types
  Unsound features
  Undefined & null types
  Co- & Contra-variant subtyping
  Unchecked overloads
  `any` *type*

- Array support

- Flexible object initialization

  Internal: Constructors
  External: Unique references

Also in the paper…

Scaling to TypeScript

- Type features

  Object literal types
  Interface types
  Primitive types
  Unsound features
  Undefined & null types
  Co- & Contra-variant subtyping
  Unchecked overloads
  any *type*

- Array support

- Flexible object initialization

  Internal: Constructors
  External: Unique references

Formal Results

Refinement type safety for core language

Experimental Evaluation

Benchmark suite

File	LOC
`navier-stokes`	366
`splay`	206
`richards`	304
`raytrace`	576
`transducers`	588
`d3-arrays`	189
`tsc-checker`	293
Total	**2522**

Benchmark suite

File	LOC
`navier-stokes`	366
`splay`	206
`richards`	304
`raytrace`	576
`transducers`	588
`d3-arrays`	189
`tsc-checker`	293
Total	**2522**

Octane
- NavierStokes: 2D fluid motion simulator
- Splay: splay tree implementation
- Richards: OS kernel simulator
- Raytrace: ray trace renderer

Benchmark suite

File	LOC
`navier-stokes`	366
`splay`	206
`richards`	304
`raytrace`	576
`transducers`	588
`d3-arrays`	189
`tsc-checker`	293
Total	**2522**

Octane
- NavierStokes: 2D fluid motion simulator
- Splay: splay tree implementation
- Richards: OS kernel simulator
- Raytrace: ray trace renderer

Transducers
Composable algorithmic transformations

Benchmark suite

File	LOC
`navier-stokes`	366
`splay`	206
`richards`	304
`raytrace`	576
`transducers`	588
`d3-arrays`	189
`tsc-checker`	293
Total	**2522**

Octane
- NavierStokes: 2D fluid motion simulator
- Splay: splay tree implementation
- Richards: OS kernel simulator
- Raytrace: ray trace renderer

Transducers
Composable algorithmic transformations

D3: A JavaScript visualization library
- Array operations

Benchmark suite

File	LOC
`navier-stokes`	366
`splay`	206
`richards`	304
`raytrace`	576
`transducers`	588
`d3-arrays`	189
`tsc-checker`	293
Total	**2522**

Octane

- NavierStokes: 2D fluid motion simulator
- Splay: splay tree implementation
- Richards: OS kernel simulator
- Raytrace: ray trace renderer

Transducers

Composable algorithmic transformations

D3: A JavaScript visualization library

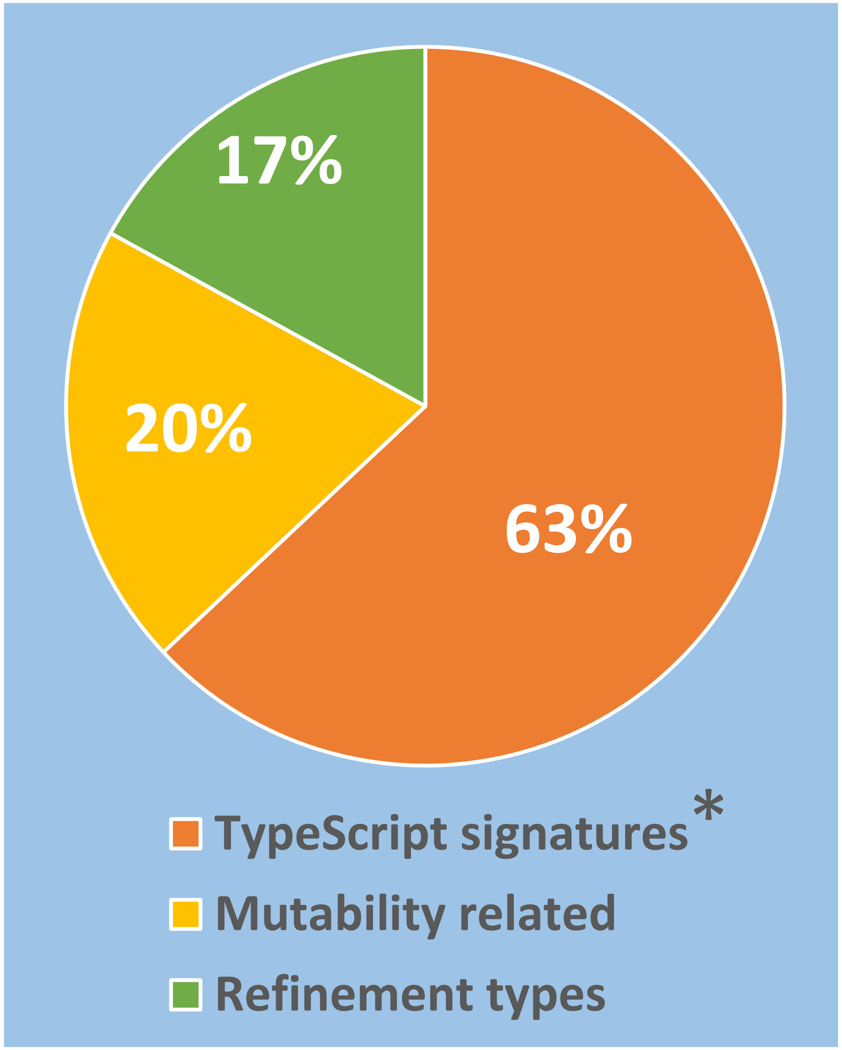
- Array operations

Microsoft's TypeScript compiler

- Parts of `core.ts` and `checker.ts`

Annotation Overhead

File	LOC	Annots (% LOC)
navier-stokes	366	24.6
splay	206	9.7
richards	304	27.3
raytrace	576	14.6
transducers	588	27.6
d3-arrays	189	26.5
tsc-checker	293	23.4
Total	**2522**	**21.4**



* Programs need to be fully typed
– no any type in signatures

Performance

File	LOC	Annots (% LOC)	Time (sec)
`navier-stokes`	366	24.6	473
`splay`	206	9.7	6
`richards`	304	27.3	7
`raytrace`	576	14.6	15
`transducers`	588	27.6	12
`d3-arrays`	189	26.5	37
`tsc-checker`	293	23.4	62
Total	**2522**	**21.4**	

Performance

File	LOC	Annots (% LOC)	Time (sec)
`navier-stokes`	366	24.6	473
`splay`	206	9.7	6
`richards`	304	27.3	7
`raytrace`	576	14.6	15
`transducers`	588	27.6	12
`d3-arrays`	189	26.5	37
`tsc-checker`	293	23.4	62
Total	**2522**	**21.4**	

More than 100 static array access sites with dynamically computed indexes

Properties Tested

- Property accesses
- Array bounds checks
- Overloads
- Safe Downcasts
  - Class based
  - Ad hoc type hierarchies
- User specified value properties. E.g. a function:
  - returns a positive number
  - accepts non-empty arrays

Properties Tested

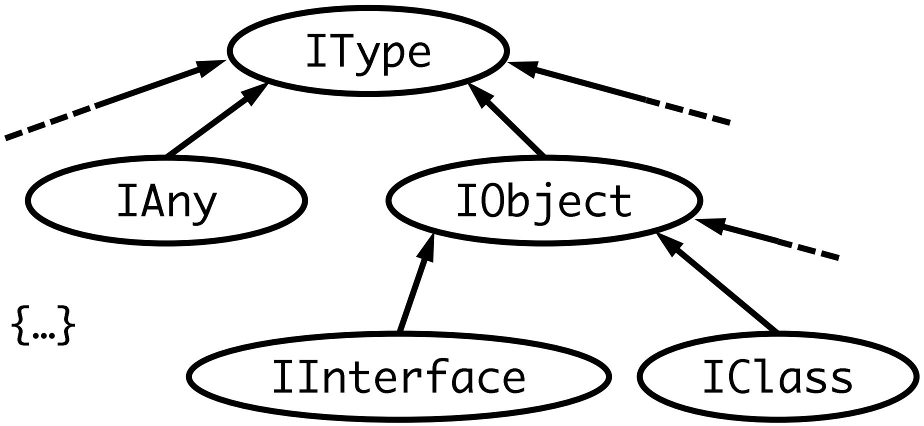
Safe Downcasts

Ad hoc type hierarchies

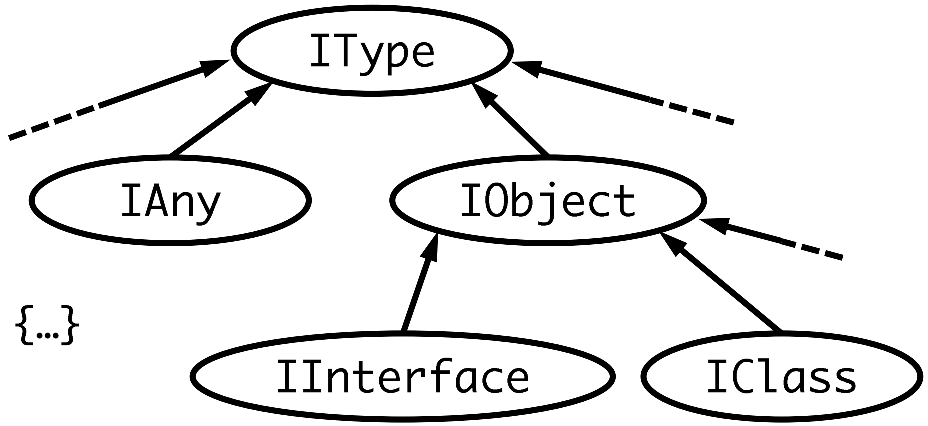
Example taken from:

`TypeScript compiler - v1.0.1.0 - src/compiler/types.ts`

```
interface IType {…}
interface IClass extends IType {…}
interface IAny extends IType {…}
interface IObject extends IType {…}
interface IInterface extends IObject {…}
```



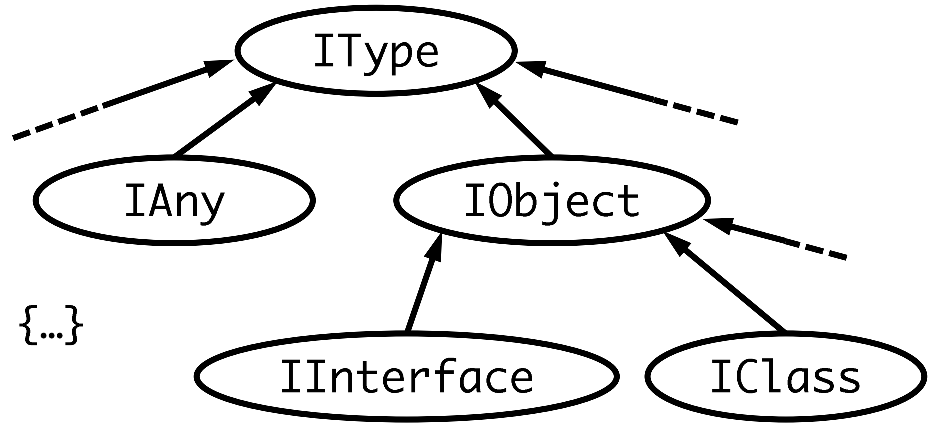
```
interface IType {…}
interface IClass extends IType {…}
interface IAny extends IType {…}
interface IObject extends IType {…}
interface IInterface extends IObject {…}
```



TypeScript interfaces are plain JavaScript objects

no type information at runtime

```
interface IType { flags:TypeFlags; }
interface IClass extends IType {…}
interface IAny extends IType {…}
interface IObject extends IType {…}
interface IInterface extends IObject {…}
```




```
const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```

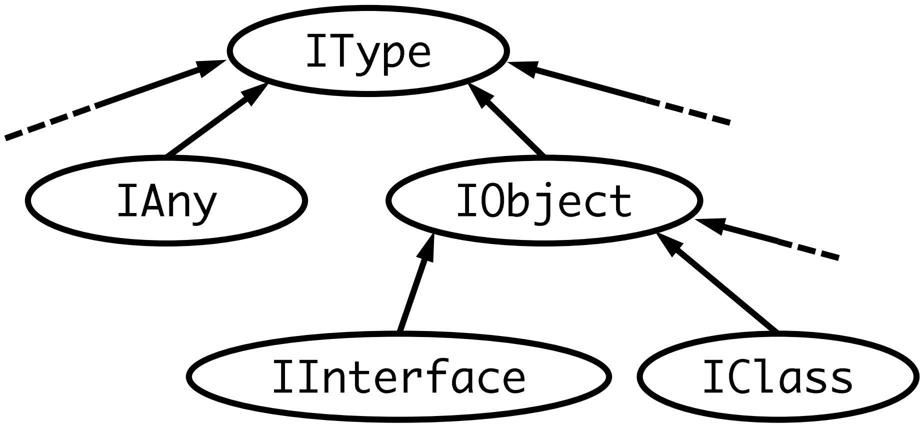
TypeScript interfaces are plain JavaScript objects

no type information at runtime

Explicit field (`flags`) to encode type info

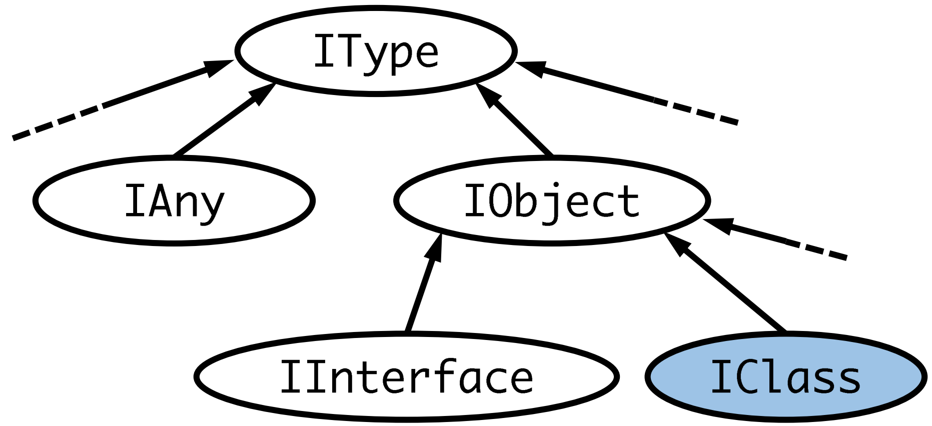
needed for dynamic tests

```
interface IType { flags:TypeFlags; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```



Invariants

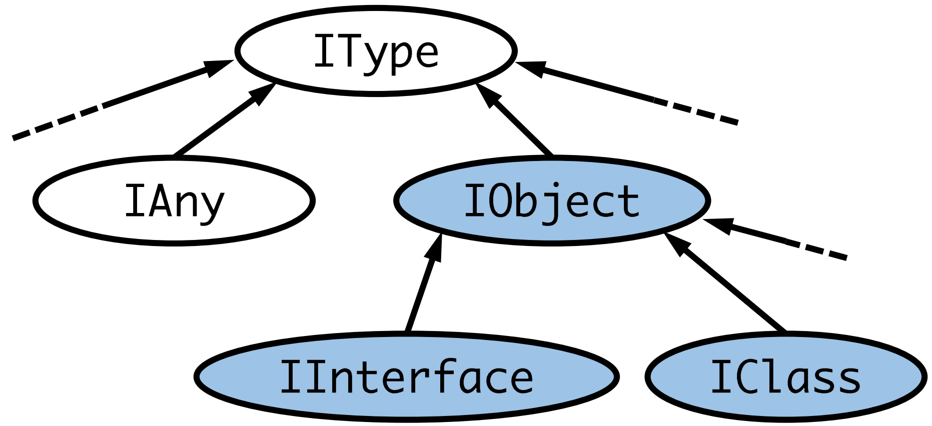
```
interface IType { flags:TypeFlags; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```



Invariants

t.flags & 0x0400 ≠ 0 ⇒ t: IClass

```
interface IType { flags:TypeFlags; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```



Invariants

$$\texttt{t.flags \& 0x0400} \neq 0 \Rightarrow \texttt{t: IClass}$$

$$\texttt{t.flags \& (0x0400|0x0800|...)} \neq 0 \Rightarrow \texttt{t: IObject}$$

...

```
interface IType { flags:TypeFlags; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```

Problem

Unchecked invariants

```
t.flags & 0x0400                 ≠ 0 ⇒ t: IClass

t.flags & (0x0400|0x0800|…) ≠ 0 ⇒ t: IObject

...
```

```
interface IType { flags:TypeFlags; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```

```
var t: IType = …
if (t.flags & TypeFlags.Class) {
  var o = <IClass> t;
}
```

Problem

Unchecked invariants

```
t.flags & 0x0400              ≠ 0 ⇒ t: IClass
t.flags & (0x0400|0x0800|…) ≠ 0 ⇒ t: IObject
                ...
```

```
interface IType { flags:TypeFlags; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```

```
var t: IType = …
if (t.flags & TypeFlags.Class) {
  var o = <IClass> t;
}
```

Problem

Unchecked invariants

```
t.flags & 0x0400              ≠ 0 ⇒ t: IClass
t.flags & (0x0400|0x0800|…) ≠ 0 ⇒ t: IObject
...
```

```
interface IType { flags:TypeFlags; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```

```
var t: IType = …
if (t.flags & TypeFlags.Class) {
  var o = <IAny> t;
}
```

No static or dynamic error

Problem

Unchecked invariants

$$\texttt{t.flags \& 0x0400} \neq 0 \Rightarrow \texttt{t: IClass}$$

$$\texttt{t.flags \& (0x0400|0x0800|…)} \neq 0 \Rightarrow \texttt{t: IObject}$$

...

```
interface IType { flags:TypeFlags; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```

```
var t: IType = …
if (t.flags & TypeFlags.Class) {
  var o = <IAny> t;
}
```

No static or dynamic error

Problem

Unchecked invariants

Solution

Encode invariants in refinement types

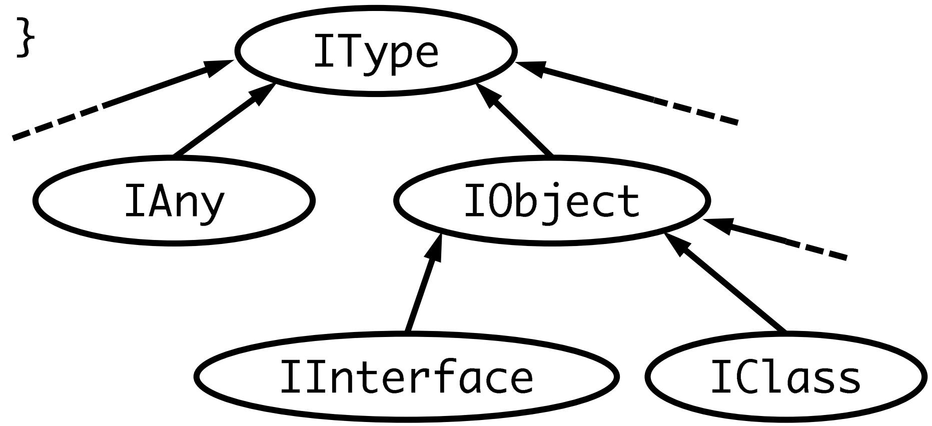
Encode type information in logic

```
interface S {…}
```

$\Downarrow$

```
x: S ⇔ implements(x, 'S')
```

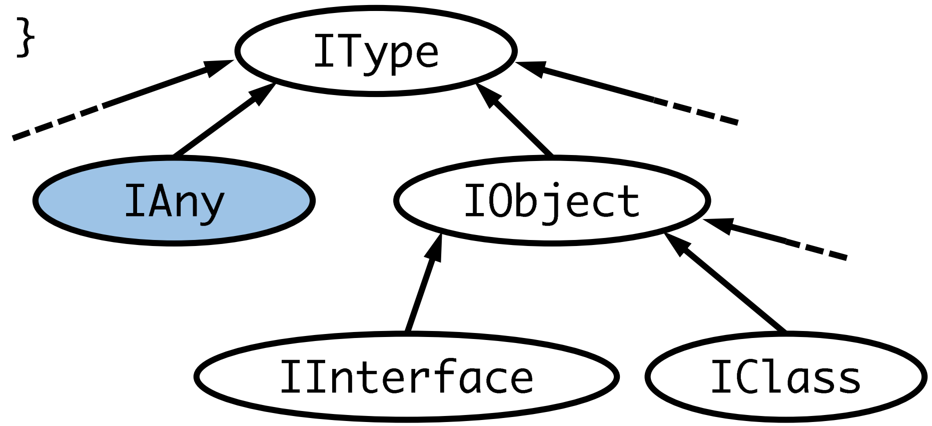
```
interface IType { flags:TypeFlagInv; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```



Type for `flags` accounts for possible sub-interfaces

```
type TypeFlagInv  =   TypeFlags
```

```
interface IType { flags:TypeFlagInv; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```



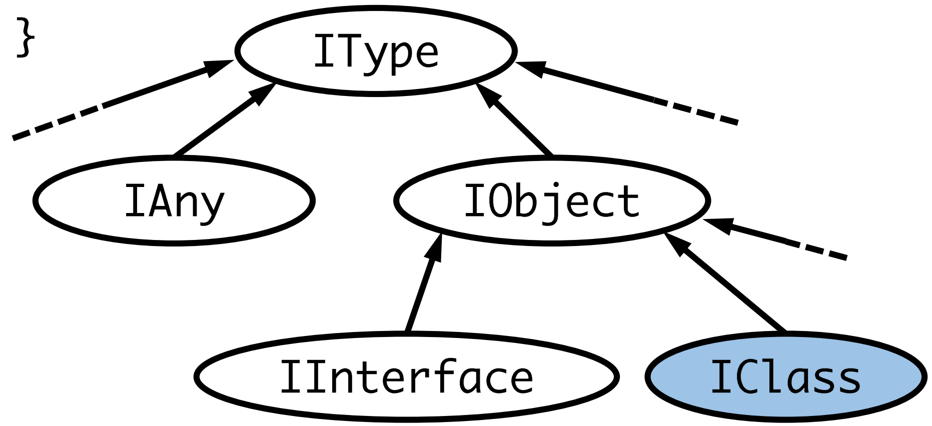
Type for `flags` accounts for possible sub-interfaces

```
type TypeFlagInv  = { TypeFlags |
    mask(v,0x0001) ⇒ implements(this, 'IAny')
}
```

Bitwise AND

The containing object

```
interface IType { flags:TypeFlagInv; }
...

const enum TypeFlags {
   Any       = 0x0001,
   Class     = 0x0400,
   Interface = 0x0800,
   ObjType   = Class
             | Interface
             | ...
}
```



Type for `flags` accounts for possible sub-interfaces

```
type TypeFlagInv  = { TypeFlags |
    mask(v,0x0001) ⇒ implements(this, 'IAny')
  ∧ mask(v,0x0400) ⇒ implements(this, 'IClass')

}
```

```
interface IType { flags:TypeFlagInv; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```

```
type TypeFlagInv  = { TypeFlags |
    mask(v,0x0001) ⇒ implements(this, 'IAny')
  ∧ mask(v,0x0400) ⇒ implements(this, 'IClass')
  ∧ mask(v,0x0800) ⇒ implements(this, 'IInterface')
  ∧ … }
```

```
interface IType { flags:TypeFlagInv; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```

```
var t: IType = …
if (t.flags & TypeFlags.Class) {
  var o = <IClass> t;
}
```

```
type TypeFlagInv  = { TypeFlags |
    mask(v,0x0001) ⇒ implements(this, 'IAny')
  ∧ mask(v,0x0400) ⇒ implements(this, 'IClass')
  ∧ mask(v,0x0800) ⇒ implements(this, 'IInterface')
  ∧ … }
```

```
interface IType { flags:TypeFlagInv; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```

```
var t: IType = …
if (t.flags & TypeFlags.Class) {
  var o = <IClass> t;
}
```

Check downcast

```
type TypeFlagInv  = { TypeFlags |
    mask(v,0x0001) ⇒ implements(this, 'IAny')
  ∧ mask(v,0x0400) ⇒ implements(this, 'IClass')
  ∧ mask(v,0x0800) ⇒ implements(this, 'IInterface')
  ∧ … }
```

```
interface IType { flags:TypeFlagInv; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```

```
var t: IType = …
if (t.flags & TypeFlags.Class) {
  var o = <IClass> t;
}
```

Check downcast

Invariant for
IType

```
type TypeFlagInv  = { TypeFlags |
    mask(v,0x0001) ⇒ implements(this, 'IAny')
  ∧ mask(v,0x0400) ⇒ implements(this, 'IClass')
  ∧ mask(v,0x0800) ⇒ implements(this, 'IInterface')
  ∧ … }
```

```
interface IType { flags:TypeFlagInv; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```

```
var t: IType = …
if (t.flags & TypeFlags.Class) {
  var o = <IClass> t;
}
```

Check downcast

Invariant for IType	∧	Path condition: t.flags & 0x0400 ≠ 0

```
type TypeFlagInv  = { TypeFlags |
    mask(v,0x0001) ⇒ implements(this, 'IAny')
  ∧ mask(v,0x0400) ⇒ implements(this, 'IClass')
  ∧ mask(v,0x0800) ⇒ implements(this, 'IInterface')
  ∧ … }
```

```
interface IType { flags:TypeFlagInv; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```

```
var t: IType = …
if (t.flags & TypeFlags.Class) {
  var o = <IClass> t;
}
```

Check downcast

Invariant for IType	∧	Path condition: t.flags & 0x0400 ≠ 0	⇒	implements(t, 'IClass')

```
type TypeFlagInv  = { TypeFlags |
     mask(v,0x0001) ⇒ implements(this, 'IAny')
   ∧ mask(v,0x0400) ⇒ implements(this, 'IClass')
   ∧ mask(v,0x0800) ⇒ implements(this, 'IInterface')
   ∧ … }
```

```
interface IType { flags:TypeFlagInv; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```

```
var t: IType = …
if (t.flags & TypeFlags.Class) {
  var o = <IClass> t;
}
```

Encode type information in logic

Check downcast

Invariant for IType	∧	Path condition: t.flags & 0x0400 ≠ 0	⇒	t: IClass

```
type TypeFlagInv  = { TypeFlags |
    mask(v,0x0001) ⇒ implements(this, 'IAny')
  ∧ mask(v,0x0400) ⇒ implements(this, 'IClass')
  ∧ mask(v,0x0800) ⇒ implements(this, 'IInterface')
  ∧ … }
```

```
interface IType { flags:TypeFlagInv; }
...

const enum TypeFlags {
  Any       = 0x0001,
  Class     = 0x0400,
  Interface = 0x0800,
  ObjType   = Class
            | Interface
            | ...
}
```

```
var t: IType = …
if (t.flags & TypeFlags.Class) {
  var o = <IClass> t;
}
```

Encode type information in logic

Check downcast

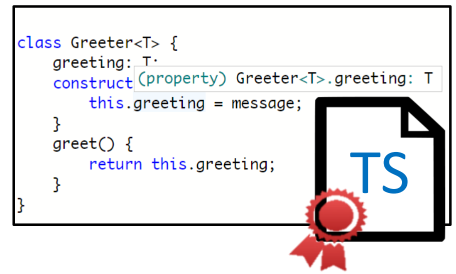
Invariant for IType	∧	Path condition: $t.flags \,\&\, 0x0400 \neq 0$	⇒	t: IClass

```
type TypeFlagInv  = { TypeFlags |
    mask(v,0x0001) ⇒ implements(this, 'IAny')
  ∧ mask(v,0x0400) ⇒ implements(this, 'IClass')
  ∧ mask(v,0x0800) ⇒ implements(this, 'IInterface')
  ∧ … }
```

Refinement Types for TypeScript

Extensible static analysis for a modern scripting language

- ✓ Fixed type tests
- ✓ User specified invariants



Challenges	Solutions
Assignments	SSA Transformation
Mutability	Extend type system with immutability guarantees
Overloading	Two-phased typing
Annotation Overhead	Liquid Types

Source: `github.com/UCSD-PL/refscript`

Demo: `goto.ucsd.edu/~pvekris/refscript`

Thanks!