Type Systems for Dynamic Languages

PhD Seminar Presentation

Meetesh Kalpesh Mehta (23D0361)

Advisor: Dr Manas Thakur

Overview

- Introduction
    - Classification of types
- Inference of types
- Strength of type systems
- Uses of type systems
    - Optimization, safety
- Types for dynamic languages
- Types for higher level abstractions

What are Types?

```
int a = 123456 + "Hello" + &main; // Typecheck ERROR: binop...
```

What are Types?

```
int a = 123456 + "Hello" + &main; // Typecheck ERROR: binop...
```

Type rules can help prevent invalid operations

What are Types?

```
int a = 123456 + "Hello" + &main; // Typecheck ERROR: binop...
```

```
int a = 123456; // Some malicious virtual address
void (fun_ptr)(int) = (void ()(int))a;
fun_ptr(1);
```

What are Types?

```
int a = 123456 + "Hello" + &main; // Typecheck ERROR: binop...
```

```
int a = 123456; // Some malicious virtual address
void (fun_ptr)(int) = (void ()(int))a;
fun_ptr(1);
```

Not all rules can be verified during static analysis.

What are Types?

```c
int a = 123456 + "Hello" + &main; // Typecheck ERROR: binop...
```

```c
int a = 123456; // Some malicious virtual address
void (fun_ptr)(int) = (void ()(int))a;
fun_ptr(1);
```

In languages like C/C++, types act as annotations that allow the compiler to identify invalid code using type rules.

What are Types?

```
import java.util.Random;

class A {}
class B extends A {}
class C extends B {}

public class Main {
  public static void main(String[] args) {
    A obj1 = new A(); // Typecheck OK, dynamic OK
    A obj2 = new B(); // Typecheck OK, dynamic OK
    A obj3 = new C(); // Typecheck OK, dynamic OK
    Random rand = new Random();
    int rand_int = rand.nextInt(10);
    Object obj4 = new Object();
    if (rand_int > 5) {
      obj4 = new Object();
    } else {
      obj4 = new A();
    }
    A obj = (A)obj4; // Typecheck OK, dynamic OK
    System.out.println("obj is " + obj);
  }
}
```

What are Types?

```
import java.util.Random;

class A {}
class B extends A {}
class C extends B {}

public class Main {
  public static void main(String[] args) {
    A obj1 = new A(); // Typecheck OK, dynamic OK
    A obj2 = new B(); // Typecheck OK, dynamic OK
    A obj3 = new C(); // Typecheck OK, dynamic OK
    Random rand = new Random();
    int rand_int = rand.nextInt(10);
    Object obj4 = new Object();
    if (rand_int > 5) {
      obj4 = new Object();
    } else {
      obj4 = new A();
    }
    A obj = (A)obj4; // Typecheck OK, dynamic OK
    System.out.println("obj is " + obj);
  }
}
```

In Java, these assignments are always correct.

What are Types?

```java
import java.util.Random;

class A {}
class B extends A {}
class C extends B {}

public class Main {
  public static void main(String[] args) {
    A obj1 = new A(); // Typecheck OK, dynamic OK
    A obj2 = new B(); // Typecheck OK, dynamic OK
    A obj3 = new C(); // Typecheck OK, dynamic OK
    Random rand = new Random();
    int rand_int = rand.nextInt(10);
    Object obj4 = new Object();
    if (rand_int > 5) {
      obj4 = new Object();
    } else {
      obj4 = new A();
    }
    A obj = (A)obj4; // Typecheck OK, dynamic OK
    System.out.println("obj is " + obj);
  }
}
```

obj4 -> {Object, A}

What are Types?

```java
import java.util.Random;

class A {}
class B extends A {}
class C extends B {}

public class Main {
  public static void main(String[] args) {
    A obj1 = new A(); // Typecheck OK, dynamic OK
    A obj2 = new B(); // Typecheck OK, dynamic OK
    A obj3 = new C(); // Typecheck OK, dynamic OK
    Random rand = new Random();
    int rand_int = rand.nextInt(10);
    Object obj4 = new Object();
    if (rand_int > 5) {
      obj4 = new Object();
    } else {
      obj4 = new A();
    }
    A obj = (A)obj4; // Typecheck OK, dynamic OK
    System.out.println("obj is " + obj);
  }
}
```

obj4 -> {Object, A}

50% of time this code is correct, other times JVM prevents execution of this code.

What are Types?

```java
import java.util.Random;

class A {}
class B extends A {}
class C extends B {}

public class Main {
  public static void main(String[] args) {
    A obj1 = new A(); // Typecheck OK, dynamic OK
    A obj2 = new B(); // Typecheck OK, dynamic OK
    A obj3 = new C(); // Typecheck OK, dynamic OK
    Random rand = new Random();
    int rand_int = rand.nextInt(10);
    Object obj4 = new Object();
    if (rand_int > 5) {
      obj4 = new Object();
    } else {
      obj4 = new A();
    }
    A obj = (A)obj4; // Typecheck OK, dynamic OK
    System.out.println("obj is " + obj);
  }
}
```

In languages like Java, types are values at runtime. This allows the type system to provide guarantees about the system.

What are Types?

MODULE *Diehard*

EXTENDS *Integers*
VARIABLES *small*, *big*

$$Init \triangleq \land (small = 0) \quad \land (big = 0)$$

A type check ensures the following predicates hold true got all program states

$$TypeOK \triangleq \land (small \in 0 \,..\, 3) \quad \land (big \in 0 \,..\, 5)$$

$$FillSmall \triangleq (small' = 3) \land (big' = big)$$

$$FillBig \triangleq (small' = small) \land (big' = 5)$$

$$EmptySmall \triangleq (small' = 0) \land (big' = big)$$

$$EmptyBig \triangleq (small' = small) \land (big' = 0)$$

$$PourSmallToBig \triangleq \text{IF } (big + small \leq 5) \text{ THEN } \land big' = big + small \;\land small' = 0 \text{ ELSE } \land big' = 5 \;\land small' = small - (5 - big)$$

$$PourBigToSmall \triangleq \text{IF } (big + small \leq 3) \text{ THEN } \land small' = big + small \;\land big' = 0 \text{ ELSE } \land small' = 3 \;\land big' = big - (3 - small)$$

$$Next \triangleq \lor EmptySmall \lor EmptyBig \lor FillSmall \lor FillBig \lor PourSmallToBig \lor PourBigToSmall$$

[Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers - ‘02]

What are Types?

MODULE *Diehard*

EXTENDS *Integers*

VARIABLES *small*, *big*

$Init \triangleq \land (small = 0) \land (big = 0)$

A type check ensures the following predicates hold true got all program states

$TypeOK \triangleq \land (small \in 0 \mathrel{..} 3) \land (big \in 0 \mathrel{..} 5)$

$FillSmall \triangleq (small' = 3) \land (big' = big)$

$FillBig \triangleq (small' = small) \land (big' = 5)$

$EmptySmall \triangleq (small' = 0) \land (big' = big)$

$EmptyBig \triangleq (small' = small) \land (big' = 0)$

$PourSmallToBig \triangleq$ IF $(big + small \leq 5)$ THEN $\land\ big' = big + small \land small' = 0$ ELSE $\land\ big' = 5 \land small' = small - (5 - big)$

$PourBigToSmall \triangleq$ IF $(big + small \leq 3)$ THEN $\land\ small' = big + small \land big' = 0$ ELSE $\land\ small' = 3 \land big' = big - (3 - small)$

$Next \triangleq \lor EmptySmall \lor EmptyBig \lor FillSmall \lor FillBig \lor PourSmallToBig \lor PourBigToSmall$

Variable declaration and initialization.

What are Types?

```
---------------------------- MODULE Diehard ----------------------------
EXTENDS Integers
VARIABLES small, big

Init ≜ ∧ (small = 0)
       ∧ (big = 0)

A type check ensures the following predicates hold true got all program states

TypeOK ≜ ∧ (small ∈ 0 .. 3)
         ∧ (big ∈ 0 .. 5)

FillSmall ≜ (small' = 3) ∧ (big' = big)
FillBig ≜ (small' = small) ∧ (big' = 5)
EmptySmall ≜ (small' = 0) ∧ (big' = big)
EmptyBig ≜ (small' = small) ∧ (big' = 0)

PourSmallToBig ≜ IF (big + small  ≤ 5)
                   THEN ∧ big' = big + small
                        ∧ small' = 0
                   ELSE ∧ big' = 5
                        ∧ small' = small − (5 − big)

PourBigToSmall ≜ IF (big + small ≤ 3)
                   THEN ∧ small' = big + small
                        ∧ big' = 0
                   ELSE ∧ small' = 3
                        ∧ big' = big − (3 − small)

Next ≜ ∨ EmptySmall ∨ EmptyBig ∨ FillSmall
       ∨ FillBig  ∨ PourSmallToBig ∨ PourBigToSmall
=========================================================================
```

Valid steps that can be taken at each point.

What are Types?

MODULE *Diehard*

EXTENDS *Integers*
VARIABLES *small*, *big*

$$Init \triangleq \land (small = 0) \\ \land (big = 0)$$

A type check ensures the following predicates hold true got all program states

$$TypeOK \triangleq \land (small \in 0 \,..\, 3) \\ \land (big \in 0 \,..\, 5)$$

Type invariant

$$FillSmall \triangleq (small' = 3) \land (big' = big)$$
$$FillBig \triangleq (small' = small) \land (big' = 5)$$
$$EmptySmall \triangleq (small' = 0) \land (big' = big)$$
$$EmptyBig \triangleq (small' = small) \land (big' = 0)$$

$$PourSmallToBig \triangleq \text{IF } (big + small \leq 5) \\ \text{THEN } \land big' = big + small \\ \land small' = 0 \\ \text{ELSE } \land big' = 5 \\ \land small' = small - (5 - big)$$

$$PourBigToSmall \triangleq \text{IF } (big + small \leq 3) \\ \text{THEN } \land small' = big + small \\ \land big' = 0 \\ \text{ELSE } \land small' = 3 \\ \land big' = big - (3 - small)$$

$$Next \triangleq \lor EmptySmall \lor EmptyBig \lor FillSmall \\ \lor FillBig \lor PourSmallToBig \lor PourBigToSmall$$

What are Types?

```
---------------------------- MODULE Diehard ----------------------------

EXTENDS Integers
VARIABLES small, big

Init ≜ ∧ (small = 0)
       ∧ (big = 0)

A type check ensures the following predicates hold true got all program states

TypeOK ≜ ∧ (small ∈ 0 .. 3)
         ∧ (big ∈ 0 .. 5)

FillSmall ≜ (small′ = 3) ∧ (big′ = big)
FillBig ≜ (small′ = small) ∧ (big′ = 5)
EmptySmall ≜ (small′ = 0) ∧ (big′ = big)
EmptyBig ≜ (small′ = small) ∧ (big′ = 0)

PourSmallToBig ≜ IF (big + small  ≤ 5)
                  THEN ∧ big′ = big + small
                       ∧ small′ = 0
                  ELSE ∧ big′ = 5
                       ∧ small′ = small − (5 − big)

PourBigToSmall ≜ IF (big + small ≤ 3)
                  THEN ∧ small′ = big + small
                       ∧ big′ = 0
                  ELSE ∧ small′ = 3
                       ∧ big′ = big − (3 − small)

Next ≜ ∨ EmptySmall ∨ EmptyBig ∨ FillSmall
       ∨ FillBig  ∨ PourSmallToBig ∨ PourBigToSmall
```

In languages like TLA+, types are invariants on variables.

When implementing a specification, the type invariants may be asserted statically or dynamically.

What are Types?

```
let a = 10;
a = (a, b) => a < b;
a = [1,3,2].sort(a);
```

What are Types?

```
let a = 10;
a = (a, b) => a < b;
a = [1,3,2].sort(a);
```

Int

Closure

list

What are Types?

```
let a = (a, b) => a < b;
let b = (a, b) => a < b;
let c = a + b; // What does it even mean to add two functions ?
console.log(c);
let d = c(1,2);
```

What are Types?

```
let a = (a, b) => a < b;
let b = (a, b) => a < b;
let c = a + b; // What does it even mean to add two functions ?
console.log(c);
let d = c(1,2);
```

c is a string!

What are Types?

```
let a = (a, b) => a < b;
let b = (a, b) => a < b;
let c = a + b; // What does it even mean to add two functions ?
console.log(c);
let d = c(1,2);
```

Runtime assertion fails!

What are Types?

```
let a = (a, b) => a < b;
let b = (a, b) => a < b;
let c = a + b; // What does it even mean to add two functions ?
console.log(c);
let d = c(1,2);
```

In languages like JavaScript, types are runtime values, checked at runtime.

What are Types?

C/C++ ⇒ annotations**@static**

Java ⇒ annotations**@static** + values**@runtime**

TLA+ ⇒ invariants**@const**

JS ⇒ values**@runtime**

What are Types?

C/C++ ⇒ annotations@static

Java ⇒ annotations@static + values@runtime

TLA+ ⇒ invariants@const

JS ⇒ values@runtime

Tags

What are Types?

C/C++ ⇒ annotations@static

Java ⇒ annotations@static + values@runtime

TLA+ ⇒ invariants@const

JS ⇒ values@runtime

Tag-free variables ==> improved performance & complex GC/runtime

Tagged variable ==> slower performance & robust GC/runtime

What are Types?

C/C++ ⇒ annotations@static

Java ⇒ annotations@static + values@runtime

TLA+ ⇒ invariants@const

JS ⇒ values@runtime

Tag-free variables ==> improved performance & complex GC/runtime

Tagged variable ==> slower performance & robust GC/runtime

Overview

- Introduction
  - Classification of types
- Inference of types
- Strength of type systems
- Uses of type systems
  - Optimization, safety
- Types for dynamic languages
- Types for higher level abstractions

How are types created?

From existing types – *Using language primitives, other declared types.*

Completely new types – *Behave like new primitives.*

How are types created? Types from existing types

```
// Type String in Haskell
type String = [Char]

// One might view it as a typedef in C
typedef char* string;
```

How are types created? Types from existing types

```
// Type Distance described in terms of Type Point in Haskell
type Point = ( Int , Int )
type Distance = Point -> Point -> Int


// One might view it as the following in C ++.
typedef struct Point {
    int a,b;
}
typedef int (* Distance ) (Point,Point);
```

How are types created? **Types from existing types**

```
// Type Distance described in terms of Type Point in Haskell
type Point = ( Int , Int )
type Distance = Point -> Point -> Int


// One might view it as the following in C ++, but not really.
typedef struct Point {
    int a,b;
}
typedef int (* Distance ) (Point,Point);
```

How are types created? **Types from existing types**

```
// Type A and Type B in Haskell are synonyms
type A = (Int,Int)
type B = (Int,Int)

f :: A -> B
f (a,b) = (b,a)

g :: A -> Int
g (a,b) = a + b

// Executing function f and g
g ( f (1 ,2) )
```

How are types created? Types from existing types

```
// Type A and Type B in Haskell are synonyms
type Point1 = (Int,Int)
type Point2 = (Int,Int)

// In C they are not
typedef struct Point1 {
    int a,b;
}

typedef struct Point2 {
    int a,b;
}
```

How are types created? Types from existing types

Equality in Haskell for types declared using **‘type’** is **structural**.

Recursion is **now allowed** for such types.

```
// Invalid in Haskell
type Tree=(Int,[Tree])

// Haskell compiler does not infer recursion in types, instead the
programmer is responsible for explicitly marking it.
```

How are types created? **Completely new types**

```
// Type A and Type B in Haskell are synonyms
data Point1 = C1 Int Int
data Point2 = C2 Int Int
```

How are types created? **Completely new types**

```
// Type A and Type B in Haskell are synonyms
data Point1 = C1 Int Int
data Point2 = C2 Int Int
```

Constructor

Keyword

How are types created? Completely new types

```
// Type A and Type B in Haskell are synonyms
data Point1 = C1 Int Int
data Point2 = C2 Int Int

// Equivalent C code
typedef struct Point1 {
    int a,b;
}

typedef struct Point2 {
    int a,b;
}
```

Overview

- Introduction
  - Classification of types
- **Inference of types**
- Strength of type systems
- Uses of type systems
  - Optimization, safety
- Types for dynamic languages
- Types for higher level abstractions

Inference of types

```
letrec map f m = if (null m) then nil
                 else cons (f (car m)) (map f (cdr m))
```

A Theory of Type Polymorphism in Programming [JCSS-1978]

Inference of types

```
letrec map f m = if (null m) then nil
                 else cons (f (car m)) (map f (cdr m))

typeof (map) = (α → β) → α list → β list
```

Modeling Types as a System of Linear Equations

```
letrec map f m = if (null m) then nil
                 else cons (f (car m)) (map f (cdr m))
```

Free identifiers ⇒ Not defined as an argument of current/parent lambdas.

Modeling Types as a System of Linear Equations

```
letrec map f m = if (null m) then nil
                 else cons (f (car m)) (map f (cdr m))
```

$null : \alpha\ list \rightarrow bool$

$nil : \alpha\ list$

$car : \alpha\ list \rightarrow \alpha$

$cdr : \alpha\ list \rightarrow \alpha\ list$

$cons : \alpha \rightarrow \alpha\ list \rightarrow \alpha\ list$

Generic types of free identifiers

Modeling Types as a System of Linear Equations

```
letrec map f m = if (null m) then nil
                 else cons (f (car m)) (map f (cdr m))
```

$$\sigma_{null} = \tau_1 \; list \rightarrow bool$$

$$\sigma_{nil} = \tau_2 \; list$$

$$\sigma_{car} = \tau_3 \; list \rightarrow \tau_3$$

$$\sigma_{cdr} = \tau_4 \; list \rightarrow \tau_4 \; list$$

$$\sigma_{cons} = \tau_5 \rightarrow \tau_5 \; list \rightarrow \tau_5 \; list$$

Substituting Type Variables

Modeling Types as a System of Linear Equations

```
letrec map f m = if (null m) then nil
                 else cons (f (car m)) (map f (cdr m))
```

1 $\sigma_{map} = \sigma_f \to \sigma_m \to \rho_1$

2 $\sigma_{null} = \sigma_m \to bool$

3 $\sigma_{car} = \sigma_m \to \rho_2$

4 $\sigma_{cdr} = \sigma_m \to \rho_3$

5 $\sigma_f = \rho_2 \to \rho_4$

6 $\sigma_{map} = \sigma_f \to \rho_3 \to \rho_5$

7 $\sigma_{cons} = \rho_4 \to \rho_5 \to \rho_6$

8 $\rho_1 = \sigma_{nil} = \rho_6$

Modeling Types as a System of Linear Equations

```
letrec map f m = if (null m) then nil
                 else cons (f (car m)) (map f (cdr m))
```

$$\sigma_{map} = (\alpha \rightarrow \beta) \rightarrow \alpha \;\; list \rightarrow \rho_1$$

$$\sigma_{null} = \alpha \;\; list \rightarrow bool$$

$$\sigma_{car} = \alpha \;\; list \rightarrow \alpha$$

$$\sigma_{cdr} = \alpha \;\; list \rightarrow \rho_3$$

$$\sigma_f = (\alpha \rightarrow \beta)$$

$$\sigma_{map} = (\alpha \rightarrow \beta) \rightarrow \rho_3 \rightarrow \rho_5$$

$$\sigma_{cons} = \beta \rightarrow \rho_5 \rightarrow \rho_6$$

$$\rho_1 = \sigma_{nil} = \rho_6$$

Modeling Types as a System of Linear Equations

```
letrec map f m = if (null m) then nil
                 else cons (f (car m)) (map f (cdr m))
```

1. $\sigma_{map} = (\alpha \rightarrow \beta) \rightarrow \alpha \ list \rightarrow \rho_1$
2. $\sigma_{null} = \alpha \ list \rightarrow bool$
3. $\sigma_{car} = \alpha \ list \rightarrow \alpha$
4. $\sigma_{cdr} = \alpha \ list \rightarrow \alpha \ list$
5. $\sigma_f = (\alpha \rightarrow \beta)$
6. $\sigma_{map} = (\alpha \rightarrow \beta) \rightarrow \alpha \ list \rightarrow \rho_5$
7. $\sigma_{cons} = \beta \rightarrow \rho_5 \rightarrow \rho_6$
8. $\rho_1 = \sigma_{nil} = \rho_6$

Modeling Types as a System of Linear Equations

```
letrec map f m = if (null m) then nil
                 else cons (f (car m)) (map f (cdr m))
```

$$\sigma_{map} = (\alpha \to \beta) \to \alpha \ list \to \rho_1$$
$$\sigma_{null} = \alpha \ list \to bool$$
$$\sigma_{car} = \alpha \ list \to \alpha$$
$$\sigma_{cdr} = \alpha \ list \to \alpha \ list$$
$$\sigma_f = (\alpha \to \beta)$$
$$\sigma_{map} = (\alpha \to \beta) \to \alpha \ list \to \beta \ list$$
$$\sigma_{cons} = \beta \to \beta \ list \to \beta \ list$$
$$\rho_1 = \sigma_{nil} = \beta \ list$$

$$\sigma_{map} = (\alpha \to \beta) \to \alpha \ list \to \beta \ list$$

Modeling Types as a System of Linear Equations

```
If Robinson’s unification algorithm succeeds, typechecking OK,
otherwise fail.
```

```cpp
// Example1: auto keyword in C++

std::vector <int> foo (int a) {
    auto vec;
    return vec;
}
```

```cpp
// Example1: auto keyword in C++

std::vector <int> foo (int a) {
    auto vec;
    return vec;
}
```

$$\sigma_{foo} = \tau_1 \rightarrow \tau_2$$

$$\sigma_{vec} = \tau_3$$

$$\sigma_{ret} = \sigma_{foo} = \sigma_{vec}$$

```cpp
// Example1: auto keyword in C++

std::vector <int> foo (int a) {
    auto vec;
    return vec;
}
```

$$\sigma_{foo} = \sigma_a \rightarrow \sigma_{ret}$$
$$\sigma_{vec} = \tau_3$$
$$\sigma_{ret} = \sigma_{foo} = \sigma_{vec}$$

```
// Example1: auto keyword in C++

std::vector <int> foo (int a) {
    auto vec;
    return vec;
}
```

$$\sigma_{foo} = int \rightarrow int\ \ vector$$

$$\sigma_{vec} = int\ \ vector$$

$$\sigma_{ret} = \sigma_{foo} = \sigma_{vec}$$

```cpp
// Example1: auto keyword in C++

std::vector <int> foo (int a) {
    auto vec;
    return vec;
}
```

TYPECHECK FAIL

'auto' specifies that the type of the variable that is being declared will be automatically ***deduced from its initializer****.*

Some languages impose restrictions on their type inference systems.

```
// Example2: inference in TypeScript

function foo ( a : number ) : number [] {
    let vec;

    return vec;
}
```

Same example works fine in case of TypeScript!

```
// Example2: inference in TypeScript

function foo ( a : number ) : number
    let vec;
    // Successful unification
    return vec;
}
```

TYPECHECK OK

Same example works fine in case of TypeScript!

Limitations of this system

```
let reversepair (x, y) = (reverse(x),reverse(y))
```

Tuple of size two as input argument

Return a tuple with reversed tuple values.

Limitations of this system

```
let reversepair (x, y) = (reverse(x),reverse(y))
```

$$\sigma_{reversepair} = (\tau_1, \tau_2) \rightarrow \tau_3$$
$$\sigma_{reverse} = \tau_3 \rightarrow \tau_4$$
$$\sigma_{reverse} = \tau_5 \rightarrow \tau_6$$
$$\sigma_{reversepair} = (\tau_3, \tau_5)$$

$$\sigma_{reversepair} = (\sigma_x, \sigma_y) \rightarrow \tau_3$$
$$\sigma_{reverse} = \sigma_x \rightarrow \tau_4$$
$$\sigma_{reverse} = \sigma_y \rightarrow \tau_6$$
$$\sigma_{reversepair} = (\sigma_x, \sigma_y)$$

Limitations of this system

```
let reversepair (x, y) = (reverse(x),reverse(y))
```

$$\sigma_{reversepair} = (\alpha\ list, \beta\ list) \rightarrow \tau_3$$
$$\sigma_{reverse} = \alpha\ list \rightarrow \tau_4$$
$$\sigma_{reverse} = \beta\ list \rightarrow \tau_6$$
$$\sigma_{reversepair} = (\alpha\ list, \beta\ list)$$

Limitations of this system

```
let reversepair (x, y) = (reverse(x),reverse(y))
```

$$\sigma_{reversepair} = (\alpha\ list, \beta\ list) \rightarrow \tau_3$$
$$\sigma_{reverse} = \alpha\ list \rightarrow \alpha\ list$$
$$\sigma_{reverse} = \beta\ list \rightarrow \beta\ list$$
$$\sigma_{reversepair} = (\alpha\ list, \beta\ list)$$

$$\sigma_{reversepair} = (\alpha\ list, \alpha\ list) \rightarrow (\alpha\ list, \alpha\ list)$$
$$\sigma_{reverse} = \alpha\ list \rightarrow \alpha\ list$$
$$\sigma_{reversepair} = (\alpha\ list, \alpha\ list)$$

Limitations of this system: **Possible Solution**

```
let reversepair (x, y) = (reverse(x),reverse(y))
```

$$\sigma_{reversepair} = (\alpha\ list, \beta\ list) \rightarrow (\alpha\ list, \beta\ list)$$
$$\sigma_{reverse_a} = \alpha\ list \rightarrow \alpha\ list$$
$$\sigma_{reverse_b} = \beta\ list \rightarrow \beta\ list$$
$$\sigma_{reversepair} = (\alpha\ list, \beta\ list)$$

We could instantiate type variables.

Types that do not appear as part of any enclosing formal parameters are allowed to be instantiated.

Limitations of this system: Possible Solution

```
let Func f (a,b) = (f(a), f(b))
```

Types that do not appear as part of any enclosing formal parameters are allowed to be instantiated.

When f is an argument, even this fails.

Overview

- Introduction
    - Classification of types
- Inference of types
- **Strength of type systems**
- Uses of type systems
    - Optimization, safety
- Types for dynamic languages
- Types for higher level abstractions

Strength of Type Systems

Ideally we would want to allow **as much program behaviour as possible** and still be able to typecheck the program.

A very restrictive type system makes it **difficult to describe** certain kinds of programs.

A limited type system may also make performing **type preserving optimizations** difficult.

Strength of Type Systems

```
let l = [1,“Hello”,3.14] as const

let m = reverse(...l)

let n : inference = m.first()

console.log(n + 33.3)
```

Strength of Type Systems

`let l = [1,“Hello”,3.14] as const`	**[Int | String | Float]**
`let m = reverse(...l)`	**[Float | String | Int]**
`let n : inference = m.first()`	**[Float | String | Int]**
`console.log(n + 33.3)`	**floatAdd | StringAdd | IntAdd**

Strength of Type Systems

`let l = [1,“Hello”,3.14] as const`	`[Int, String, Float]`
`let m = reverse(...l)`	`[Float, String, Int]`
`let n : inference = m.first()`	`[Float]`
`console.log(n + 33.3)`	**`Call floatAdd`**

Strength of Type Systems: **Pattern Matching**

```
type Reverse <T extends any []> =
      T extends [infer T1 , ...infer Ts]
      ? [ ... Reverse < Ts > , T1 ]
      : T;

declare function reverse<T extends any []>(... ts : T) : Reverse<T>;
```

Strength of Type Systems: **Pattern Matching**

```
type Reverse <T extends any []> =
      T extends [infer T1 , ...infer Ts]
      ? [ ... Reverse < Ts > , T1 ]
      : T;

declare function reverse<T extends any []>(... ts : T) : Reverse<T>;
```

A **Type Variable T** can be a any list.

Strength of Type Systems: **Pattern Matching**

```
type Reverse <T extends any []> =
      T extends [infer T1 , ...infer Ts]
      ? [ ... Reverse < Ts > , T1 ]
      : T;

declare function reverse<T extends any []>(...ts : T) : Reverse<T>;
```

Return type of **reverse** if **Reverse<T>**

Strength of Type Systems: Pattern Matching

```
type Reverse <T extends any []> =
      T extends [infer T1 , ...infer Ts]
      ? [ ... Reverse < Ts > , T1 ]
      : T;

declare function reverse<T extends any []>(...ts : T) : Reverse<T>;
```

Pattern matching

Strength of Type Systems: Pattern Matching

```
type Reverse <T extends any []> =
      T extends [infer T1 , ...infer Ts]
      ? [ ... Reverse < Ts > , T1 ]
      : T;

declare function reverse<T extends any []>(...ts : T) : Reverse<T>;
```

Pattern matching, with inferred type variables **T1** and **Ts**

Even recursion is supported!!

Strength of Type Systems: Conditional Typing

```
type APIResult<T extends boolean> = T extends true ?
                                 ExtraInformation : BasicInformation;

function apiCall(extra: boolean) : ApiResult<typeof extra> {
  if (extra) { return new ExtraInformation(); }
  else { return new BasicInformation(); }
}

const extraInformation = apiCall(true); // ExtraInformation
const basicInformation = apiCall(false); // BasicInformation
```

Strength of Type Systems: Conditional Typing

```
type APIResult<T extends boolean> = T extends true ?
                                 ExtraInformation : BasicInformation;

function apiCall(extra: boolean) : ApiResult<typeof extra> {
  if (extra) { return new ExtraInformation(); }
  else { return new BasicInformation(); }
}

const extraInformation = apiCall(true); // ExtraInformation
const basicInformation = apiCall(false); // BasicInformation
```

Narrowing

Strength of Type Systems: **Defunctionalization**

Defunctionalization is an important optimization, it helps separate a functions body and data.

Typed Closure Conversion [POPL ‘96]
Defunctionalization with Dependent Types [PLDI ‘24]

Strength of Type Systems: **Defunctionalization**

```
let val x = 1
    val y = 2
    val z = 3
    val f = λw. x + y + w
in
    f 100
end
```

Strength of Type Systems: Defunctionalization

```
let val x = 1
    val y = 2
    val z = 3
    val f = (λenv. λw. (#x env) + (#y env) + w) { x=x, y=y }
in
    f 100
end
```

Strength of Type Systems: **Defunctionalization**

```
let val x = 1
    val y = 2
    val z = 3
    val code = λenv. λw. (#x env) + (#y env) + w
    val env = { x=x, y=y }
    val f = (code, env)
in
    (#1 f) (#2 f) 100
end
```

Strength of Type Systems: Defunctionalization

```
let val x = 1
    val y = 2
    val z = 3
    val code = λenv. λw. (#x env) + (#y env) + w
    val env = { x=x, y=y }
    val f = (code, env)
in
    (#1 f) (#2 f) 100
end
```

$\tau_{code} = \tau_{env} \rightarrow \tau_1 \rightarrow \tau_2$ $\qquad \tau_f = (\tau_{env} \rightarrow \tau_1 \rightarrow \tau_2) \times \tau_{env}$

Strength of Type Systems: Existential Types

```
let val y = 1
in
    if true then
        λx. x + y
    else
        λz. z
end
```

Strength of Type Systems: Existential Types

```
let val y = 1
in
    if true then
        (λenv. λx. x + #y(e), {y=y})
    else
        (λenv. λz. z, {})
end
```

({y: int}⟶int⟶int) X {y:int})

({}⟶int⟶int) X {})

Strength of Type Systems: Existential Types

```
let val y = 1
in
  if true then
    pack {y:int} with (λenv. λx. x+#y(env),{y=y})
    as ∃t_ve.(t_ve → int → int) × t_ve
  else
    pack {} with (λenv. λz. z, {})
    as ∃t_ve.(t_ve → int → int) × t_ve
end
```

Overview

- Introduction
  - Classification of types
- Inference of types
- Strength of type systems
- **Uses of type systems**
  - **Optimization, safety**
- Types for dynamic languages
- Types for higher level abstractions

Uses of Type Systems: Optimization [TIL- PLDI ‘96]

```
fun sub [α] ( x :α array , i : int ) =
    typecase α of
      int => intsub (x , i )
    | float => floatsub (x , i )
    | ptr (τ) => ptrsub (x , i )
```

```
fun sub [ float ] (x , 10)
```

```
floatsub (x , 10)
```

Intensional polymorphism, static analysis of types

Uses of Type Systems: GC [TIL- PLDI ‘96]

Locations of pointers and their liveness information can be encoded in the stack frame directly.

So no **tag’s** need to be maintained for stack variables and registers.

Only tags for heap-allocated objects are required.

A LOT OF MODERN LANGUAGES LIKE JAVA USE THESE PRINCIPLES.

Uses of Type Systems: **Safety**

```
package p ;
public class Table {
    private Bucket [] buckets;
    public Object[] get ( Object key ) { return buckets; }
}

class Bucket {
    Bucket next ;
    Object key , val ;
}
```

Type-based Confinement [JFP - ‘06]

Uses of Type Systems: **Safety**

C1: Must not appear in the type of a public/prot field or the return type of a public/prot method.

C2 : A confined type must not be public.

C3 : Method invoked on an expression of confined type must either be defined in a confined class or be anonymous.

C4 : Subtypes of a confined type must be confined.

C5 : Confined types can be widened only to other confined types.

C6 : Overriding must preserve anonymity of the methods.

A1: **“this”** is used only to select fields and be a receiver of other anonymous methods.

Uses of Type Systems: **Safety**

```
package p ;
public class Table {
    private Bucket [] buckets ;
    public Object[] get ( Object key ) { return buckets; }
}

conf class Bucket {
    Bucket next ;
    Object key , val ;
}
```

Uses of Type Systems: **Safety**

```
package p ;
public class Table {
    private Bucket [] buckets ;
    public Object[] get ( Object key
}

conf class Bucket {
    Bucket next ;
    Object key , val ;
}
```

TYPECHECK FAIL

Overview

- Introduction
  - Classification of types
- Inference of types
- Strength of type systems
- Uses of type systems
  - Optimization, safety
- **Types for dynamic languages**
- Types for higher level abstractions

Types for dynamic languages: **A case for R**

- Implemented a tool called **TypeTracer**, that traces types for methods at runtime.
- **ContractR** decorates function bodies with type assertions.

T	::=	`any`	*top type*
	|	`null`	*null type*
	|	`env`	*environment type*
	|	S	*scalar type*
	|	V	*vector type*
	|	$T \mid T$	*union type*
	|	$?\,T$	*nullable type*
	|	$\langle A_1, \ldots A_n \rangle \rightarrow T$	*function type*
	|	`list`$\langle T \rangle$	*list type*
	|	`class`$\langle ID_1, \ldots ID_n \rangle$	*class type*

A	::=	T	*arguments*
	|	...	*dots*
V	::=	S`[]`	*vector types*
	|	`^`S`[]`	*na vector types*
S	::=	`int`	*integer*
	|	`chr`	*character*
	|	`dbl`	*double*
	|	`lgl`	*logical*
	|	`clx`	*complex*
	|	`raw`	*raw*

Designing Types for R, Empirically [OOPSLA - ‘20]

Types for dynamic languages: **A case for R**

- Even though types as data frames and complex classes are used regularly, they found that the most popular types at runtime are vectors and matrices, etc.

- They found that 80% of functions are monomorphic or have only one polymorphic argument.

Types for dynamic languages: A case for Python

CLASS BASED TYPES

Approximate missing things using a missing field.

OBJECT BASED TYPES

Every object evolution is a new type.

What Types Are Needed for Typing Dynamic Objects? A Python-Based Empirical Study [APLAS - ‘23]

Types for dynamic languages: **A case for Python**

```
class Panel:
  def __init__(self,p1,p2):
    self.title = title
    self.width = width
    if self.width is not None:
      self.height = self.width

  def _title(self):
    if isinstance(self.title, str):
      return Text.from_markup(self.title)
    else
      return self.title.copy()

  def setheight(self,height):
    self.height = height

  def measure(self):
    return self.width * self.height

panel1 = Panel(Text(),42)
panel2 = Panel("Example Table",None)
panel2.width = 5 # modification
panel2.setHeight(42) # extension
```

Constructor Polymorphism

```
typeof (panel1) =
  { title: Text,
    width: int,
    height: int }

typeof (panel2) =
  { title: Str,
    width: NoneType }
```

Types for dynamic languages: **A case for Python**

```python
class Panel:
  def __init__(self,p1,p2):
    self.title = title
    self.width = width
    if self.width is not None:
      self.height = self.width

  def _title(self):
    if isinstance(self.title, str):
      return Text.from_markup(self.title)
    else
      return self.title.copy()

  def setheight(self,height):
    self.height = height

  def measure(self):
    return self.width * self.height

panel1 = Panel(Text(),42)
panel2 = Panel("Example Table",None)
panel2.width = 5 # modification
panel2.setHeight(42) # extension
```

Class Based Types

```
Panel@C1: {
     title : Text,
     width : Int
}

Panel@C2: {
     title : Str,
     width : None or
             Int
     height: None or
             Int
}
```

Types for dynamic languages: **A case for Python**

```
class Panel:
  def __init__(self,p1,p2):
    self.title = title
    self.width = width
    if self.width is not None:
      self.height = self.width

  def _title(self):
    if isinstance(self.title, str):
      return Text.from_markup(self.title)
    else
      return self.title.copy()

  def setheight(self,height):
    self.height = height

  def measure(self):
    return self.width * self.height

panel1 = Panel(Text(),42)
panel2 = Panel("Example Table",None)
panel2.width = 5 # modification
panel2.setHeight(42) # extension
```

Object Evolution

```
typeof (panel2@21) =
  { title: Str,
    width: NoneType}

typeof (panel2@22) =
  { title: Str,
    width: Int }

typeof (panel2@23) =
  { title: Str,
    width: Int,
    height: int }
```

Types for dynamic languages: **A case for Python**

```
class Panel:
  def __init__(self,p1,p2):
    self.title = title
    self.width = width
    if self.width is not None:
      self.height = self.width

  def _title(self):
    if isinstance(self.title, str):
      return Text.from_markup(self.title)
    else
      return self.title.copy()

  def setheight(self,height):
    self.height = height

  def measure(self):
    return self.width * self.height

panel1 = Panel(Text(),42)
panel2 = Panel("Example Table",None)
panel2.width = 5 # modification
panel2.setHeight(42) # extension
```

Object-Based Types

```
typeof (Panel@21) =
  { title: Str,
    width: None }
```

↓

```
typeof (Panel@21) =
  { title: Str,
    width: Int or None }
```

↓

```
typeof (Panel@21) =
  { title: Str,
    width: Int or None,
    height: Int or None
or Abs }
```

Types for dynamic languages: A case for Python

Constructor Polymorphism

Around 20% of constructors were polymorphic.

Most polymorphic constructors have a low degree (less than five)

- 87% are polymorphic on attribute types
- 6% differ on the attributes
- 7% exhibit both.

80% of times the output of a constructor's type was directly correlated with the arguments.

Types for dynamic languages: **A case for Python**

Object Evolution

27% of all runtime objects evolved (33% of all classes)

Evolution is largely **monotonic in nature**

- Addition of attributes (or)
- Types only change to their subtypes

Overview

- Introduction
  - Classification of types
- Inference of types
- Strength of type systems
- Uses of type systems
  - Optimization, safety
- Types for dynamic languages
- **Types for higher level abstractions**

Types for higher level abstractions: TypeStates

Associate a property with a class called as **“state”**.

Typestate-Oriented Programming [ONWARD - ‘09]

Types for higher level abstractions: **TypeStates**

Associate a property with a class called as **“state”**.

Each **“state”** describes the valid behaviour.

Typestate-Oriented Programming [ONWARD - ‘09]

Types for higher level abstractions: TypeStates

Associate a property with a class called as **“state”**.

Each **“state”** describes the valid behaviour.

Special methods can lead to transition in state, these are explicitly marked.

Typestate-Oriented Programming [ONWARD - ‘09]

Types for higher level abstractions: TypeStates

Associate a property with a class called as **“state”**.

Each **“state”** describes the valid behaviour.

Special methods can lead to transition in state, these are explicitly marked.

Type Checker is then used to identify operations that may be performed on an invalid state, for example, reading from a previously closed file.

Typestate-Oriented Programming [ONWARD - ‘09]

Types for higher level abstractions: **TypeStates**

```
state File {
    public final String filename;
}
state OpenFile extends File {
    private CFilePtr filePtr;
    public int read() { ... }
    public void close() [OpenFile>>ClosedFile] { ... }
}
state ClosedFile extends File {
    public void open() [ClosedFile>>OpenFile] { ... }
}
```

Types for higher level abstractions

When modeling types for higher level abstractions is the difficulty in mixing them with existing primitive types.

Most Javascript code in the wild is probably impure and doing static analysis is futile. But react code is almost always pure, writing impure code is just unnatural. However, when it gets compiled down to Javascript it becomes harder and harder to analyze.

Types for higher level abstractions

```
import React, { useState } from 'react';

const App = () => {
  // Define state to track which component to display
  const [selected,setSelect] = useState('A');
  return (
    <div>
      <h1>Select a Component</h1>
      <button onClick={() => setSelect('A')}>Select A</button>
      <button onClick={() => setSelect('B')}>Select B</button>
      <button onClick={() => setSelect('C')}>Select C</button>
      <button onClick={() => setSelect('D')}>Select D</button>
      <div> <MyComponent someProp={selected}/> </div>
    </div>
  );
};
```

Types for higher level abstractions

```
import React, { useState } from 'react';

const App = () => {
  // Define state to track which component to display
  const [selected,setSelect] = useState('A');
  return (
    <div>
      <h1>Select a Component</h1>
      <button onClick={() => setSelect('A')}>Select A</button>
      <button onClick={() => setSelect('B')}>Select B</button>
      <button onClick={() => setSelect('C')}>Select C</button>
      <button onClick={() => setSelect('D')}>Select D</button>
      <div> <MyComponent someProp={selected}/> </div>
    </div>
  );
};
```

State is a high level concept in React

Types for higher level abstractions

```
import React, { useState } from 'react';

const App = () => {
  // Define state to track which component to display
  const [selected,setSelect] = useState('A');
  return (
    <div>
      <h1>Select a Component</h1>
      <button onClick={() => setSelect('A')}>Select A</button>
      <button onClick={() => setSelect('B')}>Select B</button>
      <button onClick={() => setSelect('C')}>Select C</button>
      <button onClick={() => setSelect('D')}>Select D</button>
      <div> <MyComponent someProp={selected}/> </div>
    </div>
  );
};
```

setState

getState

State is a high level concept in React

Types for higher level abstractions

```
import React, { useState } from 'react';

const App = () => {
  // Define state to track which component to display
  const [selected,setSelect] = useState('A');
  return (
    <div>
      <h1>Select a Component</h1>
      <button onClick={() => setSelect('A')}>Select A</button>
      <button onClick={() => setSelect('B')}>Select B</button>
      <button onClick={() => setSelect('C')}>Select C</button>
      <button onClick={() => setSelect('D')}>Select D</button>
      <div> <MyComponent someProp={selected}/> </div>
    </div>
  );
};
```

Types for higher level abstractions

```
import React, { useState } from 'react';

const App = () => {
  // Define state to track which component to display
  const [selected,setSelect] = useState('A');
  return (
    <div>
      <h1>Select a Component</h1>
      <button onClick={() => setSelect('A')}>Select A</button>
      <button onClick={() => setSelect('B')}>Select B</button>
      <button onClick={() => setSelect('C')}>Select C</button>
      <button onClick={() => setSelect('D')}>Select D</button>
      <div> <MyComponent someProp={selected}/> </div>
    </div>
  );
};
```

Types for higher level abstractions

```
import React, { useState } from 'react';

const App = () => {
  // Define state to track which component to display
  const [selected,setSelect] = useState('A');
  return (
    <div>
      <h1>Select a Component</h1>
      <button onClick={() => setSelect('A')}>Select A</button>
      <button onClick={() => setSelect('B')}>Select B</button>
      <button onClick={() => setSelect('C')}>Select C</button>
      <button onClick={() => setSelect('D')}>Select D</button>
      <div> <MyComponent someProp={selected}/> </div>
    </div>
  );
};
```

Types for higher level abstractions

```
import React, { useState } from 'react';

const App = () => {
  // Define state to track which component to display
  const [selected,setSelect] = useState('A');
  return (
    <div>
      <h1>Select a Component</h1>
      <button onClick={() => setSelect('A')}>Select A</button>
      <button onClick={() => setSelect('B')}>Select B</button>
      <button onClick={() => setSelect('C')}>Select C</button>
      <button onClick={() => setSelect('D')}>Select D</button>
      <div> <MyComponent someProp={selected}/> </div>
    </div>
  );
};
```

Types for higher level abstractions

```
import React, { useState } from 'react';

const App = () => {
  // Define state to track which component to display
  const [selected,setSelect] = useState('A');
  return (
    <div>
      <h1>Select a Component</h1>
      <button onClick={() => setSelect('A')}>Select A</button>
      <button onClick={() => setSelect('B')}>Select B</button>
      <button onClick={() => setSelect('C')}>Select C</button>
      <button onClick={() => setSelect('D')}>Select D</button>
      <div> <MyComponent someProp={selected}/> </div>
    </div>
  );
};
```

Types for higher level abstractions

```
import React, { useState } from 'react';

const MyComponent = (p) => {

  return (
     <div>
       Hello My name is {p}
     <div>
  )
};
```

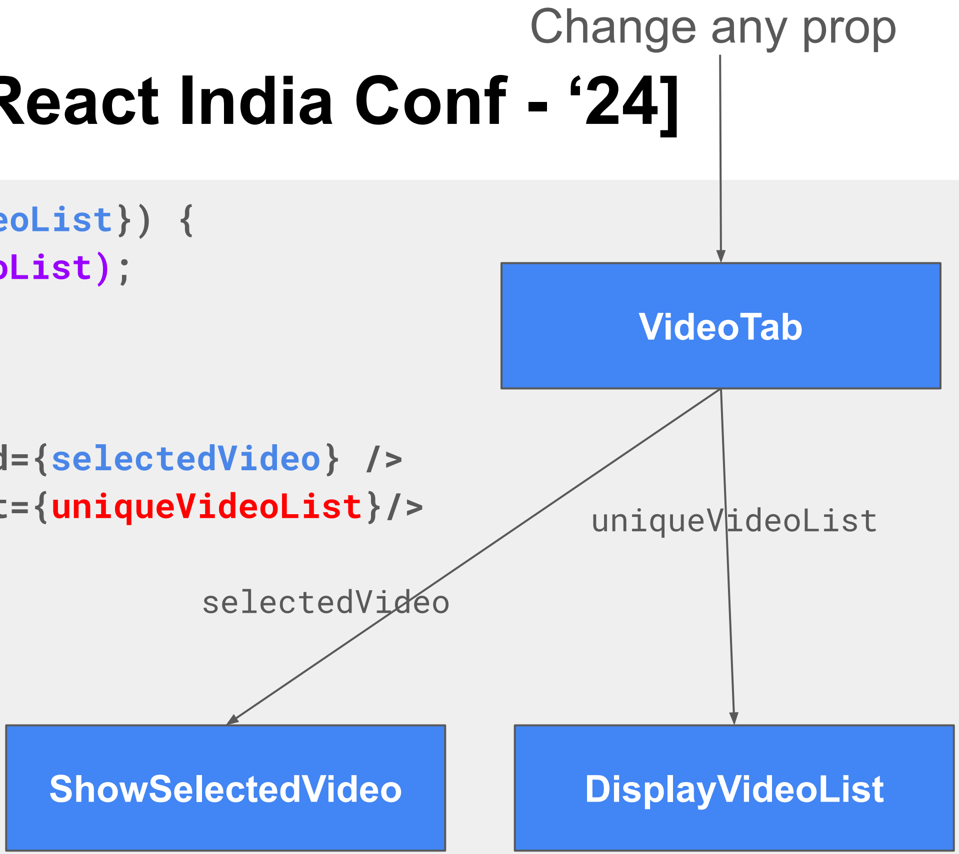
Whenever any prop changes we re-render the entire component.

Types for higher level abstractions

```
function videoTab({ selectedVideo, videoList}) {
    let uniqueVideoList = new Set(videoList);

    return (
        <div>
            <ShowSelectedVideo selected={selectedVideo} />
            <DisplayVideoList videoList={uniqueVideoList}/>
        </div>
    )
}
```

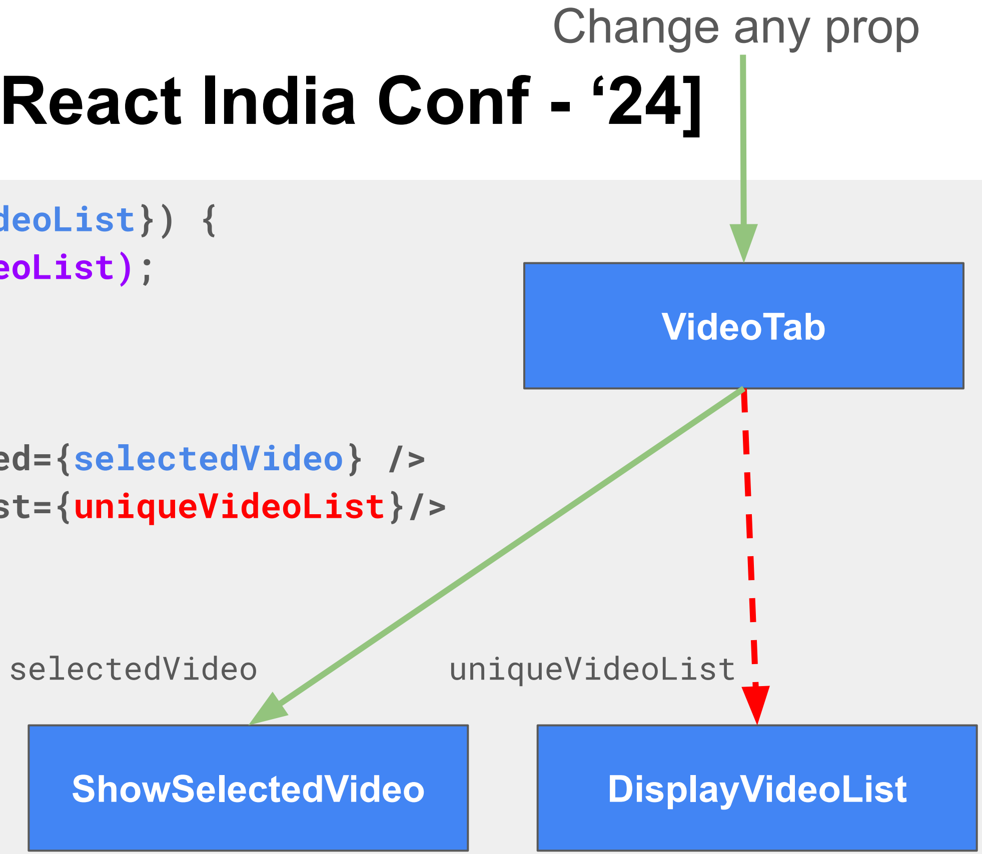
React Forget Compiler [React India Conf - ‘24]

```
function VideoTab({ selectedVideo, videoList}) {
    let uniqueVideoList = new Set(videoList);

    return (
        <div>
            <ShowSelectedVideo selected={selectedVideo} />
            <DisplayVideoList videoList={uniqueVideoList}/>
        </div>
    )
}
```



React Forget Compiler [React India Conf - ‘24]

```
function VideoTab({ selectedVideo, videoList}) {
    let uniqueVideoList = new Set(videoList);

    return (
        <div>
            <ShowSelectedVideo selected={selectedVideo} />
            <DisplayVideoList videoList={uniqueVideoList}/>
        </div>
    )
}
```



React Forget Compiler [React India Conf - ‘24]

```
function VideoTab({ selectedVideo, videoList}) {
    let uniqueVideoList = useMemo (() => new Set(videoList), [videoList]);

    return (
        <div>
            <ShowSelectedVideo selected={selectedVideo} />
            <DisplayVideoList videoList={uniqueVideoList}/>
        </div>
    )
}
```

More Possible Optimizations!!

A case for dependent/conditional types is react.

More Possible Optimizations?

```
function someFun({...}) {
    let [loadSuccess, setLoadSuccess] = useState(false);
    let [data, setData] = useState(undefined);
    const hotFunction = () => {
        // Operates heavily on data object
    }

    useEffect(() => {
        loadFromApi().then(() => {
          setData(new StringList()); setLoadSuccess(true);
        }
    }, [...]);

    return (...)
}
```

More Possible Optimizations?

```
function someFun({...}) {
    let [loadSuccess, setLoadSuccess] = useState(false);
    let [data, setData] = useState(undefined);
    const hotFunction = () => {
        // Operates heavily on data object
    }

    useEffect(() => {
        loadFromApi().then(() => {
          setData(new StringList()); setLoadSuccess(true);
        }
    }, [...]);

    return (...)
}
```

loadSuccess: bool

setLoadSuccess: udef

More Possible Optimizations?

```
function someFun({...}) {
    let [loadSuccess, setLoadSuccess] = useState(false);
    let [data, setData] = useState(undefined);
    const hotFunction = () => {
        // Operates heavily on data object
    }

    useEffect(() => {
        loadFromApi().then(() => {
            setData(new StringList()); setLoadSuccess(true);
        }
    }, [...]);

    return (...)
}
```

loadSuccess: bool

setLoadSuccess:

udef OR StringList

This doesn't help us much in any optimization. Things like null checks are needed

More Possible Optimizations?

```
function someFun({...}) {
    let [loadSuccess, setLoadSuccess] = useState(false);
    let [data, setData] = useState(undefined);
    const hotFunction = () => {
        // Operates heavily on data object
    }

    useEffect(() => {
        loadFromApi().then(() => {
            setData(new StringList()); setLoadSuccess(true);
        }
    }, [...]);

    return (...)
}
```

```
loadSuccess: bool

setLoadSuccess:

loadSuccess ? StringList
: udef
```

This doesn't help us much in any optimization. Things like null checks are needed

More Possible Optimizations!!

A case for dependent/conditional types is react.

An IR to reason about high level react components?

A way to model react states?

Empirical evaluation of slowdowns due to runtime checks

Improving GC behaviour?

Conclusion

Objects in dynamic languages exhibit a lot of different behaviours and typing under such constraints requires strong type systems.

Frameworks and programming abstractions can aid compilers infer high-level domain specific information that can be used to optimize code at lower levels.

Types are used to describe and enforce many different properties in a language, but describing type systems for dynamic languages is a non-trivial task.