StrongScript:
concrete types for TypeScript

Gregor Richards

Jan Vitek

and myself

StrongScript:
...the language Andreas
has been dreaming about...

Gregor Richards

Jan Vitek

and myself

Scripting languages are not statically typed

They are successful

Can we improve them?

optional type systems

Hack, Dart, TypeScript, …

Surprising properties:
  types do not affect execution
  types are unsound

Pragmatic benefits:
  IDE support

optional type systems

Hack, Dart, TypeScript, …

Surprising properties:
- types do not affect execution
- types are unsound

Pragmatic benefits:
- IDE support

20 million lines of PHP refactored to use Hack at Facebook

optional type systems

Hack, Dart, TypeScript, …

Surprising properties:
- types do not affect execution
- types are unsound

Pragmatic benefits:
- IDE support

gradual type systems

Racket, StrongTalk, …

Types affect execution
Types are sound

Values are wrapped at
dynamic/static boundaries:
- statically-typed code can fail
- runtime overhead

20 million lines of PHP refactored to use Hack at Facebook

Why do I like types

1. Help with refactoring, IDE support, ...

2. In Java, at runtime `x.f` will always succeed if `x` is not null

3. In Java, `x.f` is compiled to a couple of machine instructions

Why do I like types

1. Help with refactoring, IDE support, ...

2. In Java, at runtime `x.f` will always succeed if `x` is not null

safety

3. In Java, `x.f` is compiled to a couple of machine instructions

speed

predictable execution time

Why do I like types

1. Help with refactoring, IDE support, ...

2. In Java, at runtime `x.f` will always succeed if `x` is not null

   safety

3. In Java, `x.f` is compiled to a couple of machine instructions

   speed

   predictable execution time

both optional and gradual type system fall short

Our plan

Allow programmers to chose:

lightweight type annotations should not introduce dynamic errors in well-tested dynamic programs

if programmers select more stringent checks, they should have the usual type-errors and improved performance properties.

Background: TypeScript

TypeScript

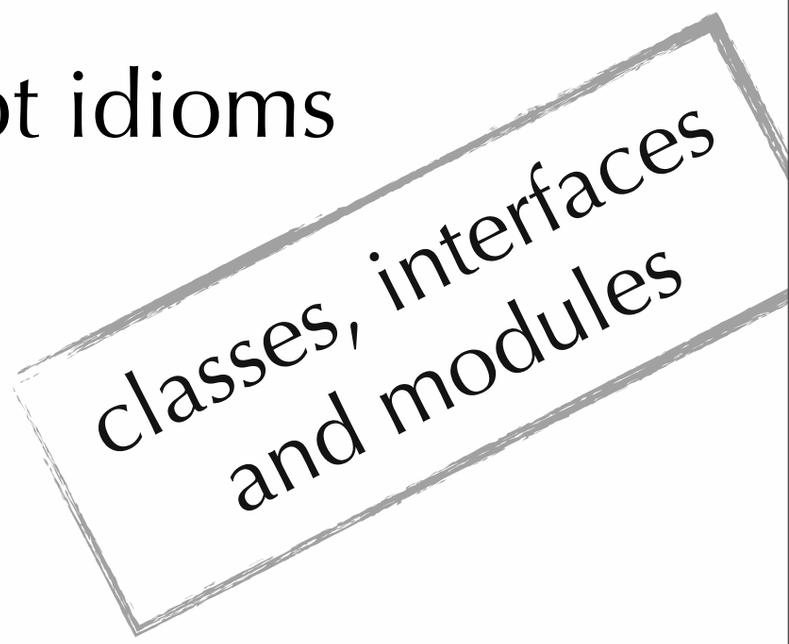
- Types are hints to the programming environment, not to the code generator

- Enables some type-checking, support IDE features
  name completion, semi-automated refactorings

- Lack of type preservation seen as necessary tradeoff

- Support only a small subset of JavaScript idioms

- No attempt to provide guarantees on other dynamic behaviours



```
interface P { x: number; }
interface T { y: number; }
interface Pt extends P { y: number; dist(p: Pt); }

class Point {
  constructor (public x:number, public y:number){}
  dist(p: Point) { ... }
}
class CPoint extends Point {
  constructor (public color:String, x:number, y:number){ super(x,y); }
  dist(p: CPoint) { ...p.color... }
}

var o:Pt = new Point(0,0);
var c:CPoint = new CPoint("Red",1,1);
function pdist(x:Point, y:Point) { x.dist(y); }
pdist( c, o );

var q:any = new CPoint("Red",1,1);
var c = q.dist( o );
var b = o.dist( q );

function getc(x:CPoint) { return x.color };
getc( <CPoint>o );
```

this code compiles with no warnings

```
interface P { x: number; }
interface T { y: number; }
interface Pt extends P { y: number; dist(p: Pt); }

class Point {
  constructor (public x:number, public y:number){}
  dist(p: Point) { ... }
}
class CPoint extends Point {
  constructor (public color:String, x:number, y:number){ super(x,y); }
  dist(p: CPoint) { ...p.color... }
}

var o:Pt = new Point(0,0);
var c:CPoint = new CPoint("Red",1,1);
function pdist(x:Point, y:Point) { x.dist(y); }
pdist( c, o );

var q:any = new CPoint("Red",1,1);
var c = q.dist( o );
var b = o.dist( q );

function getc(x:CPoint) { return x.color };
getc( <CPoint>o );
```

```
interface P { x: number; }
interface T { y: number; }
interface Pt extends P { y: number; dist(p: Pt); }

class Point {
  constructor (public x:number, public y:number){}
  dist(p: Point) { ... p.x ... p.y ... }
}
class CPoint extends Point {
  constructor (public color:string, x:number, y:number){ super(x,y); }
  dist(p: CPoint) { ... p.color ... }
}
```

interfaces declare properties and methods
extends has only documentation purposes
subtyping is structural

```
var o:Pt = new Point(0,0);
var c:CPoint = new CPoint("Red",1,1);
function pdist(x:Point, y:Point) { x.dist(y); }
pdist( c, o );

var q:any = new CPoint("Red",1,1);
var c = q.dist( o );
var b = o.dist( q );

function getc(x:CPoint) { return x.color };
getc( <CPoint>o );
```

```
interface P { x: number; }
interface T { y: number; }
interface Pt extends P { y: number; dist(p: Pt); }

class Point {
  constructor (public x:number, public y:number){}
  dist(p: Point) { ... }
}
class CPoint extends Point {
  constructor (public color:String, x:number, y:number){ super(x,y); }
  dist(p: CPoint) { ...p.color... }
}

var o:Pt = new Point(0,0);
var c:CPoint = new CPoint("Red",1,1);
function pdist( c, o );

var q:any = new CPoint("Red",1,1);
var c = q.dist( o );
var b = o.dist( q );

function getc(x:CPoint) { return x.color };
getc( <CPoint>o );
```

both classes are subtype of the interfaces above
dist is overridden covariantly

```
interface P { x: number; }
interface T { y: number; }
interface Pt extends P { y: number; dist(p: Pt); }
```

Implicit cast of Point to Pt, allowed by structural subtyping.
pdist calls dist at static type Point, but is invoked with a CPoint:
the compiler allows the call
at runtime the access of p.color returns undefined.

```
}

var o:Pt = new Point(0,0);
var c:CPoint = new CPoint("Red",1,1);
function pdist(x:Point, y:Point) { x.dist(y); }
pdist( c, o );

var q:any = new CPoint("Red",1,1);
var c = q.dist( o );
var b = o.dist( q );

function getc(x:CPoint) { return x.color };
getc( <CPoint>o );
```

```
interface P { x: number; }
interface T { y: number; }
interface Pt extends P { y: number; dist(p: Pt); }

class Point {
  constructor (public x:number, public y:number){}
  dist(p: Point) { ... }
}
class CPoint extends Point {
  constructor (public color:String, x:number, y:number){ super(x,y); }
  dist(p: CPoint) { ...p.color... }
}
var o:Pt = new Point(0,0);
var c:Pt = new CPoint("Red",1,1);
function pdist(x:Point, y:Point) { x.dist(y); }
pdist( c, o );

var q:any = new CPoint("Red",1,1);
var c = q.dist( o );
var b = o.dist( q );

function getc(x:CPoint) { return x.color };
getc( <CPoint>o );
```

Any type can be converted implicitly to any.
Any method can be invoked on an any reference.
An any reference can be converted implicitly to any other type.

```
interface P { x: number; }
interface T { y: number; }
interface Pt extends P { y: number; dist(p: Pt); }

class Point {
  constructor (public x:number, public y:number){}
  dist(p: Point) { ... }
}
class CPoint extends Point {
  constructor (public color:String, x:number, y:number){ super(x,y); }
  dist(p: CPoint) { ...p.color... }
}

var o:Pt = new Point(0,0);
var c:CPoint = new CPoint("Red",1,1);
function pdist(x:Point, y:Point) { x.dist(y); }
pdist( c, o );

var q:any = new CPoint("Red",1,1);
var c = q.dist( o );
var b = o.dist( q );

function getc(x:CPoint) { return x.color };
getc( <CPoint>o );
```

An unchecked explicit cast.

Summary + plan

Them...

	any	**C**	trace preserving?
Gradual	**any, W(any)**	**C, W(any)**	no
TypeScript	**any**	**any**	yes

...and Us

	any	C	trace preserving?	!C
Gradual	**any**, **W(any)**	C, **W(any)**	no	
TypeScript	**any**	**any**	yes	
StrongScript	**any**	**any**	at optional types	**C**

enable efficient compilation

Programming with concrete types

StrongScript

1. Start from TypeScript

2. Add the !C type for each class name C

3. Three kinds of type annotations:

	concrete types	dynamic type	optional types
denoted	!C	**any**	C
typechecks as	C	**any**	C
points to objects of type	C	**any**	**any**

```
var p:any = { x=3; z=4 }

var f:any = func (p) {
  if (p.x < 10) return 10 else return p.distance()
}

f(p)
```

```
var p:any = { x=3; z=4 }

var f:any = func (p) {
  if (p.x < 10) return 10 else return p.distance()
}

f(p)
```

Evaluates to 10.

```
var p:any = { x=3; z=4 }

var f:any = func (p) {
  if (p.x < 10) return 10 else return p.distance()
}

f(p)
```

Evaluates to 10.

now the programmer documents his expectations about the argument of f...

```
class Point {
  constructor(public x, public y){}
  dist(p) { return ... }
}

var p:Point = { x=3; z=4 }
```

```
var f:any = func (p:Point) {
  if (p.x < 10) return 10 else return p.distance()
}
```

```
class Point {
  constructor(public x, public y){}
  dist(p) { return ... }
}

var p:Point = { x=3; z=4 }
```

Correct.

```
var f:any = func (p:Point) {
  if (p.x < 10) return 10 else return p.distance()
}
```

```
class Point {
  constructor(public x, public y){}
  dist(p) { return ... }
}

var p:Point = { x=3; z=4 }
```

Correct.

```
var f:any = func (p:Point) {
  if (p.x < 10) return 10 else return p.distance()
}
```

Static type error.

```
class Point {
  constructor(public x, public y){}
  dist(p) { return ... }
}

var p:Point = { x=3; z=4 }
```

Correct.

```
var f:any = func (p:Point) {
  if (p.x < 10) return 10 else return p.distance()
}
```

Static type error.

still flexible

get local type checking, IDE completion, ...

```
class Point {
  constructor(public x, public y){}
  dist(p) { return ... }
}

var p:Point = { x=3; z=4 }
```

```
var f:any = func (p:Point) {
  if (p.x < 10) return 10 else return p.dist()
}

var s:!Point = new Point(5,6);
f(s);
```

f was typechecked against Point, so we can safely pass a real Point

```
class Point {
  constructor(public x, public y){}
  dist(p) { return ... }
}

var p:Point = { x=3; z=4 }
```

```
var f:any = func (p:Point) {
  if (p.x < 10) return 10 else return p.dist()
}

var s:!Point = new Point(5,6);
f(s);
```

Correct.

Evaluates to 10.

```
class TypedPoint {
  constructor(public x:!number, public y:!number){}
  dist(p) { return ... :!number }
}

var t:!TypedPoint = new TypedPoint (1,2)
(<any>t).x = "o"
```

```
class TypedPoint {
  constructor(public x:!number, public y:!number){}
  dist(p) { return ... :!number }
}

var t:!TypedPoint = new TypedPoint (1,2)
(<any>t).x = "o"
```

DYNAMIC ERR: type mismatch

```
class TypedPoint {
  constructor(public x:!number, public y:!number){}
  dist(p) { return ... :!number }
}

var t:!TypedPoint = new TypedPoint (1,2)
(<any>t).x = "o"
```

DYNAMIC ERR: type mismatch

instances of classes protect themselves

```
class TypedPoint {
  constructor(public x:!number, public y:!number){}
  dist(p) { return ... :!number }
}

var t:!TypedPoint = new TypedPoint (1,2)
(<any>t).x = "o"
t.dist({x=1;y=2});
```

```
var fact = func(x:!number) {return ...:!number}
var u:TypedPoint = { dist = function(p) {...} }
var n:!number = fact(u.dist(p))
```

```
class TypedPoint {
  constructor(public x:!number, public y:!number){}
  dist(p) { return ... :!number }
}

var t:!TypedPoint = new TypedPoint (1,2)
(<any>t).x = "o"
t.dist({x=1;y=2});
```

Correct

```
var fact = func(x:!number) {return ...:!number}
var u:TypedPoint = { dist = function(p) {...} }
var n:!number = fact(u.dist(p))
```

```
class TypedPoint {
  constructor(public x:!number, public y:!number){}
  dist(p) { return ... :!number }
}

var t:!TypedPoint = new TypedPoint (1,2)
(<any>t).x = "o"
t.dist({x=1;y=2});
```

Correct

```
var fact = func(x:!number) {return ...:!number}
var u:TypedPoint = { dist = function(p) {...} }
var n:!number = fact(u.dist(p))
```

u.dist(p) has type !number

Guarantees

Theorem 1 (Safety). *Given a well-typed program $\Gamma \vdash e : t$, if $e \longrightarrow^* e'$ and e' is stuck, then $e' = E[\langle t' \rightarrow t'' \rangle v'']$ and $\Gamma \vdash v'' : t''$ with $t'' \not<: !C$, or $e' = E[\{.. \mid t\}v[l]]$ and $t = C$, or $e' = E[\langle t' \rightarrow t'' \rangle v'']$ and $\Gamma \vdash v'' : any$ and v'' is not a function, or $e' =$ **undefined**.*

run-time errors arise only executing dynamically or optionally typed code

Theorem 2 (Trace Preservation). *Let e be an expression in which type annotations are* any *and $\Gamma \vdash e : any$. Let e be such that $e \longrightarrow^* v$. Let e' be an expression in which some type annotations have been replaced by optional type annotations (e.g. C, for C a class with concrete types in its interface). If $\Gamma \vdash e' : t$ for some t, then $e' \longrightarrow^* v$.*

adding an optional type annotation does not break a correct dynamic program

Theorem 3 (Strengthening). *Let e be a well-typed cast-free expression where all type annotations are of the form C or $!C$. Suppose that e reduces to the value v. Let e' be the expression obtained by replacing all occurrences of optional types C in the corresponding concrete types. The expression e' is well-typed and reduces to the value v.*

a completely optionally typed program, can be strengthened by replacing all option types with concrete types

Implementation and evaluation

StrongScript compilation strategy

StrongScript is compiled to plain JS code

Dynamic type-checking code is compiled into JS code

Intrinsics improve performance on supporting interpreters

Compiling classes

As in TypeScript, classes are compiled with the class pattern:

```
class C {
  public x = 42;
  public getX() {
    return this.x;
  }
}
```

```
C = (function () {
  function C() {
    this.x = 42;
  }
  C.prototype.getX =
  function() {
    return this.x;
  }
  return C;
})();
```

Also support for modules, as in TypeScript.

Dynamic type checks

StrongScript inserts dynamic checks at downcasts:

- primitive types are checked with the `typeof` operator;
- classes are checked with the `instanceof` operator.

Contrast with TypeScript, which erases all type and performs no checks at run-time.

By definition, optional types require no checks. If optional *blame tracking* is enabled, values at optional types are wrapped and checked lazily.

Type protection for classes

Fields are hidden, an accessor allows read and write access mediated by a checking function.

Fields accessed in an object with a statically-declared type, the accessor is bypassed.

Each method in a class is generated as two methods: one to be used by typed clients, one to be used by untyped clients.

Target

StrongScript output is idiomatic JavaScript, extended with

- checks for static types

- *intrinsics* to explicitly specify the memory layout (can also guide boxing/unboxing)

supported by our extension of TruffleJS (V8 impl in progress)
speedup observed on benchmarks heavily using objects

optional type systems

Hack, Dart, TypeScript, …

Surprising properties:
  types do not affect execution
  types are unsound

Pragmatic benefits:
  IDE support

gradual type systems

Racket, SmallTalk, …

Types affect execution
Types are sound

Values are wrapped at
dynamic/static boundaries:
- statically-typed code can fail
- runtime overhead

ype systems

TypeScript, …

operties:
t affect execution
nsound

enefits:
t

StrongScript

Static types are preserved

Dynamism of JavaScript preserved

Optional types are trace preserving
(enabling program evolution)

Efficient and predictable implementation for static types.

…implentation on top of V8 in progress…

gradual typ

Racket, Small

Types affect ex
Types are sour

Values are wra
dynamic/st
- statically-typ
- runtime over