Concrete Types for TypeScript

Abstract. TypeScript extends JavaScript with a set of optional type annotations that are, by design, unsound and, that the TypeScript compiler discards as it emits JavaScript code. This design preserves programming idioms developers are familiar with, and their legacy code, while offering a measure of static error checking. We present an alternative design for TypeScript, one where it is possible to support the same degree of dynamism, but where types can be strengthened to provide guarantees. We report on an implementation, called StrongScript, that improves runtime performance of typed programs when run on an optimizing JavaScript engine.

1 Introduction

Perhaps surprisingly, a number of modern computer programming languages have been designed with intentionally unsound type systems. Unsoundness may arise for pragmatic reasons, for instance, Java has a covariant array subtype rule designed to allow for a single `sort` implementation. More recently, industrial extensions to dynamic languages, such as Hack, Dart and TypeScript, have featured *optional type systems* [5] geared to accommodate dynamic programming idioms and preserve the behavior of legacy code. Type annotations are second class citizens intended to provide machine-checked documentation, and only slightly reduce the testing burden. Unsoundness, here, means that a variable annotated with some type `T` may, at runtime, hold a value of a type that is not a subtype of `T` due to unchecked casts, covariant subtyping, and untyped code. Implementations deal with this by ignoring annotations, emitting code where all types are erased. For example, TypeScript translates classes to JavaScript code without casts. Unsurprisingly, the generated code neither enjoys performance benefits nor strong safety guarantees.

A *gradual type system* [22, 20, 24] presents a safer alternative, as values that cross between typed and untyped parts of a program are tracked and a mechanism for assigning blame eases the debugging effort by pinpointing the origin of any offending value. But the added safety comes with runtime overhead, a price tag that, for object-oriented programs, can be steep. Also, gradual types affect the semantics of programs; which means that adding type annotations can cause runtime errors in otherwise correct programs.

We argue that programmers should be given the means to express how much type checking they want to take place in any part of their program. Depending on their choice, they should either be able to rely on the fact that type annotations will not introduce errors in well-tested and widely deployed dynamic code, or, if they select more stringent checks, they should have guarantees of the absence of type errors and improved performance.

This paper illustrates this idea with the design of a new type system for the TypeScript language. TypeScript is an extension to JavaScript from Microsoft that introduces classes, structural subtyping, and type annotations on properties, arguments and return types. Syntactially, our extension, which we call StrongScript, is minimal: a single type constructor for concrete types (written `!`) is added. Semantically the changes are more subtle. Our type system allows developers to choose between writing untyped code (i.e., all variables are of type `any` as in JavaScript), optionally typed code that does not affect the semantics of dynamic programs (i.e., no new dynamic errors), and concretely typed code that provides the traditional correctness guarantees but affects the semantics of dynamic code (i.e., types are retained by the compiler and used to optimize the program, new dynamic errors may show up). More specifically, the goals that guided design of StrongScript are:

- All JavaScript programs must be valid StrongScript programs and common programming idioms should be typeable.
- Optional types guarantee that variables are used consistently with their declarations; concrete types are sound up to down casts.
- Type information should improve performance in the context of a highly-optimizing virtual machine.
- TypeScript does not provide checked casts. As they are central to many object oriented idioms, we support them.

One of the more subtle departures between our proposal and TypeScript is that we had to switch to nominal subtyping for classes. The reasons for this change are pragmatic: generating efficient property access code for structural subtyping is not a solved problem, whereas it is well understood for nominal subtyping. Moreover, with nominal subtyping, we can reuse the existing JavaScript subtype test. Interfaces retain their structural subtyping rules and are erased at compile-time like in TypeScript. This yields a type system where any class name `C` can be used as an optional type, written `C`, or as a concrete type, written `!C`. While the former have a TypeScript-like semantics, variables typed with concrete types are guaranteed to refer to an object of class `C`, a class that extends `C`, or `null`. We exploit concrete type annotations and nominal subtyping to provide fast property access and efficient checked casts. Unannotated variables default to type `any`, ensuring that JavaScript programs are valid StrongScript programs.

The contribution of this paper are twofold:

- *Design.* We design a minimal extension of TypeScript that adds a single syntactic element and reinterpets the semantics according to a new type system. To validate our ideas we also implent StrongScript as an extension to the TypeScript compiler. All the TypeScript programs we have tried run without changes on our implementation. To get a measure of performance benefits we have extended the Truffle JavaScript implementation from Oracle labs [26] to provide fast access to properties through concretely typed variables. Truffle is a highly optimizing virtual machine that strives to match the performance of Google's V8. We obtained preliminary results on a small number of benchmarks showing speed ups between 2% and 32%. We also provide evidence that the type system is not overly restrictive, as it validates all the large TypeScript benchmarks from [16].

– *Formalization.* While work on gradual typing focused on blame, we propose *trace preservation* as a key property for the evolution of programs from untyped to typed. Informally adding a type annotation to a program is trace-preserving if the program's behavior is unaffected. More precisely, we prove a *trace preservation theorem* for optional types: if expression e is untyped, and e' only differs by the addition of optional types, then e and e' evaluate to the same value. We do this within a core calculus, in the style of λ_{JS} of [14], that captures the semantics of the two kinds of class types. A *safety* theorem states that terms can only get stuck when evaluating a cast or when accessing a property from a `any` or optionally typed variable. We also show that our design support program evolution by proving a *strengthening theorem*: when a fully optionally typed program is annotated with concrete types, the program will be trace preserving.

As with the formalization of Bierman et al. [1], we restrict ourselves to TypeScript 0.9.1, the last version before the addition of generics. Our implementation effort was done before these were stabilized in the language specification.

2 Background on Optional and Gradual Types

The divide between static and dynamic types has fascinated academics and practitioners for years. Academics come determined to "cure the dynamic" as the absence of types is viewed as a flaw. Practitioners, on the other hand, seek to supplement their testing practices with machine-checked documentation and some ahead-of-time error checking. Dynamic languages are appreciated by practitioners for their support of exploratoty programming, as any grammatically correct dynamic program, even a partial program or one with obvious errors, can be run, their productivity and their smaller learning curve. Decades of research were devoted to attempts to add static types to dynamic languages. In the 1980's, type

	TypeScript	Typed Racket	Reticulated Python	StrongScript
`x : C`	`any`	`W`	`W`	`any`
`x : !C`	–	–	–	`C`
Trace preserving	●	○	○	◐
Fast property access	○	○	○	●

Fig. 1. Optional and gradual type systems. This table's first line indicates possible values of variable declared of class `C`. This type is either `any` or `W` to denote the possibility of encountering a wrapper. The second line shows the possible value of variable declared `!C` in StrongScript, they are guaranteed to be unwrapped subtypes of that class. Trace preservation holds in TypeScript, in StrongScript developers can choose to forgo this property in exchange for stronger guarantees. The last line refers to the ability of a compiler to generate fast path code for property accesses.

inference and soft typing were proposed for Smalltalk and Scheme [21, 3, 7]. Inference based approaches turned out to be brittle as they required non-local analysis and were eventually abandoned.

Twenty years ago, while working at Animorphic on the virtual machine that would eventually become HotSpot, Bracha designed the first *optional type system* [6]. Subsequent work fleshed out the design [4] and detailed the philosophy behind optional types [5]. An optional type system is one that: (1) has no effect on the language's runtime semantics, and (2) does not mandate type annotations in the syntax. Strongtalk like Facebook's Hack, Google's Dart, and Microsoft's TypeScript was an industrial effort. In each case, a dynamic language is equipped with a static type system that is flexible enough to support backwards compatibility with untyped code. While optional types have benefits, they provide no guarantee of absence of type errors nor information that could be relied upon by an optimizing compiler.

Another important line of research is due to Felleisen and his collaborators. After investigating soft typing approaches for Scheme, Findler and Felleisen turned their attention to software contracts [9]. In [10], they proposed wrappers to enforce contracts for higher-order functions; these wrappers, higher-order functions themselves, were in charge of validating pre- and post-conditions and assigning blame in case of contract violations. Together with Flatt, they turned higher-order contracts into semantics casts [11]. A semantics cast consists of an argument (a value), a target type, and blame information. It evaluates to an object of the target type that delegates all behavior to its argument, and produces meaningful error messages in case the value fails to behave in a type appropriate manner. In 2006, Tobin-Hochstadt and Felleisen proposed a type system for, Typed Racket, a variant of Scheme that used higher-order contracts to enforce types at module boundaries [22]. Typed Racket has a robust implementation and is being used on large bodies of code [23]. The drawback of this approach is that contracts impose a runtime overhead which can be substantial in some programs.

In parallel with the development of Typed Racket, Siek and Taha defined *gradual typing* to refer to languages where type annotations can be added incrementally to untyped code [20, 18]. Like in Typed Racket, wrappers are used to enforce types but instead of focusing on module boundaries, any part of a program can be written in a mixture of typed and untyped code. The type system uses two relations, a subtyping relation and a consistency relation for assignment. Their work led to a flurry of research on issues such as bounding the space requirements for wrappers and how to precisely account for blame. In an imperative language their approach suffers from an obvious drawback: wrappers do not preserve object identity. One can thus observe the same object through a wrapper and through a direct reference at different types. Solutions are not appealing, either every property read must be checked or fairly severe restrictions must be imposed on writes. In a Python implementation, called Reticulated Python, both solutions cause slowdowns that are larger than 2x [19]. Another drawback of gradual type systems is that they are not trace preserving. Consider:

```
class C:
```

```
  b = 41
def id( x:Object{b:String} ) -> Object{b:String}: return x
id( C() ).b + 1
```

Without annotations the program evaluates to 42. When type annotations are taken into account it stops at the read of b. A type violation is reported as the required type for b is String while b holds an Int. Similar problems occur when developers put contracts that unnecessarily strong without understanding the range of types that can flow through a function.

IBM's Thorn programming language was an attempt to combine optional types (called *like types*) with concrete types [2]. The type system was formalized along with a proof that wrappers can be compiled away [25]. Preliminary performance results suggested that concrete types could yield performance improvements when compared to a naive implementation of the language, but it was not demonstrated that the results hold for an optimizing compiler.

SafeTypeScript [16] is a recent effort from Microsoft to modify TypeScript by making it safe: in a nutshell, all types are concrete, and type checks are inserted when dynamic values are cast to concrete types. This technique yields a safe language which allows dynamic types, but lacks optional types. Because type checks are always inserted, SafeTypeScript is not trace-preserving and it lacks support for evolving programs from typed to untyped. On the other hand, SafeTypeScript focused on ensuring safety within the browser which is not a goal of our work.

Figure 1 summarizes the main approaches to typing dynamic languages.

3 TypeScript: Unsound by design

Bierman et al. captured the key aspects of the design of TypeScript in [1]. TypeScript is a superset of JavaScript, with syntax for declaring classes, interfaces, and modules, and for optionally annotating variables, expressions and functions with types. Types are fully erased: errors not identified at compile-time will not be caught at runtime. The type system is structural rather than nominal, which causes some complications for subtyping. Type inference is performed to reduce the number of annotations. Some deliberate design decisions are the source of type holes, these include: unchecked casts, <String>obj is allowed if the type of obj is supertype of String, yet no check will be done at runtime; indexing with computed strings, obj[a+b] cannot be type checked as the value of string index is not known ahead of time; covariance of properties/arguments, this is similar to the Java array subtyping rule except that TypeScript does not have runtime checks for stores.

We will look more closely at the parts of the design that are relevant to our work, starting with subtyping. Consider the following well-typed program:

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

Interfaces include properties and methods. Extend declarations amongst interfaces are not required for other purposes than documenting intent, thus `Pt` is a subtype of both `P` and `T`. Classes can be defined as usual, and the extends clause there has a semantic meaning as it specifies inheritance of properties.

```
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... }
}
```

Both classes are subtypes of the interfaces declared above. Note that the `dist` method is overridden covariantly at argument `p` and that `CPoint.dist` in fact does require `p` to be an instance of `CPoint`.

```
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);
```

The first assignment implicitly casts `Point` to `Pt` which is allowed by structural subtyping. The function `pdist` will invoke `dist` at static types `Point`, yet it is invoked with a `CPoint` as first argument. The compiler allows the call, at runtime the access the `p.color` property will return the `undefined` value. Any type can be converted implicitly to `any`, and any method can be invoked on an `any` reference. More surprisingly, an `any` reference can be passed to all argument positions and be converted implicitly to any other type.

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

Our last example demonstrates a case of unchecked cast. Here `o` is declared of type `Pt` and we cast it to its subtype `CPoint`. The access will fail at runtime as variable `o` refers to an instance of `Point` which does not have color. The compiler does not emit a warning in any of the cases above.

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

Bierman et al. showed that the type system can be formalized as the combination of a sound calculus extended with unsound rules. For our purposes, the sound calculus is a system with records, equi-recursive types and structural subtyping. The resulting assignment compatibility relation can be defined coinductively using well-studied techniques [13]. We underline the critical choice of defining `any` as the supertype of all the types; since upcasts are well-typed, values of arbitrary types can be assigned to a variable of type `any` without the need of explicit casts. Type holes are introduced in three steps. First, a rule allows *downcasts* to subtypes. The second step is more interesting, as it changes the subtyping relation by stating that *all types are supertypes of* `any`. This implies arbitrary values can flow into typed variables without explicit casts. No syntactic construct identifies the boundaries between the dynamic and typed world. Thirdly, *covariant overloading* of class/interface members and methods is allowed.

Type inference is orthogonal to our proposal. As for generics, Bierman et al. describe decidability of subtyping as "challenging" [1]; will not consider them here. Lastly, we do not discuss TypeScript's liberal use of indexing. Our implementation supports it by explicitly inserting type casts (see Section 4).

4 StrongScript: Sound when needed

StrongScript builds on and extends TypeScript. Syntactically, the only addition is a new type constructor, written `!`. This yields three three kinds of type annotations:

Dynamic types, denoted by `any`, represent values manipulated with no static restrictions. Any object can be referenced by a `any` variable, all operations are allowed and any may fail at runtime.

Optional types, denoted by class names `C`, enable local type checking. All manipulations of optionally typed variables are verified statically against `C`'s interface. Optionally typed variables can reference arbitrary values, and so runtime checks are required to verify that those values conform to `C`'s interface.

Concrete types, denoted by `!C`, represent objects that are instance of the homonymous class or its subclasses. Static type checking is performed on these, and no dynamic checks are needed in the absence of downcasts.

Optional types have the same intent as TypeScript type annotations: they capture some type errors and enable IDE completion without reducing flexibility of dynamic programs. Concrete types behave exactly how programmers steeped in statically typed languages would expect. They restrict the values that can be bound to a variable and unlike other gradual type systems they do not support the notion of wrapped values or blame. No runtime error can arise from using a concretely typed variable and the compiler can rely on type informations to emit efficient code with optimizations such as unboxing and inlining.

To make good on the promise of concrete types, StrongScript has a sound type system. This forces some changes to TypeScript's overly permissive type rules and to the underlying implementation. The runtime thus distinguishes between *dynamic* objects, created with the JavaScript syntax `{ x:v .. }`, and objects which are instances of a class, created with the `new C(v..)` Java-like syntax. Casts are explicit and in many cases they require checks at runtime. Covariant subtyping, such as the array subtype rule, is checked at runtime as well. Moreover, class subtyping is nominal to ensure that the memory layout of parent classes is a prefix of child classes and thus that code to access properties is fast. Compared to TypeScript, subtyping is slightly simpler as we do not allow for `any` to be both the top and bottom of the type lattice. By design, any JavaScript program is a well-type StrongScript program, furthermore most TypeScript programs are also valid StrongScript programs – only in the rare cases discussed in Sec. 4.2 are TypeScript programs rejected by our type system (see also the evaluation in Sec. 6.2).

In what follows we introduce the main aspects of programming in our system. Code snippets should be read in sequence.

4.1 Programming with Concrete Types

We aim to let developers incrementally add types to their code, hardening parts that they feel need to be, while having the freedom to leave other parts dynamic. This is possible thanks to the interplay between the dynamic code, the flexible semantics of optionally typed variables, and the runtime guarantees of the concretely typed code. Consider the following program:

```
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
```

Without any loss of flexibility, programmers may choose to document their expectations about the argument of functions and data structures, and then annotate `p` and the argument of `f` with the optional type `Point`:

```
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 0 else
  return p.distance(p) } //Wrong
```

Arbitrary objects can still flow into optionally typed variables, preserving flexibility (and ensuring trace-preservation), while the annotation of the argument of `f` enables local type checking, catching type errors such as the call to `distance`. The programmer can also create instances of class `Point`, which are concretely typed as `!Point`, and pass them to `f`:

```
var s:!Point = new Point{5,6};
f(s);  // evaluates to 10
```

As function `f` has been type checked assuming that its argument is a `Point`, we known it's body will manipulate the argument as a `Point`. However, whenever an object which is an instance of a class is passed to an optionally or dynamically typed context, it protects its own abstractions at runtime. Consider a new class definition, where the `x` and `y` fields have been strenghtened as `!number` and as such can only refer to instances of class `number`:

```
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
```

Some flexibility is lost by this class but the compiler can exploit the type informations to compute property offsets, remove runtime type checks and unbox values. Observe that dynamic, optional and concrete types can be mixed seamlessly; above for instance we have left the argument of the `dist` function dynamically typed, so that it is correct to invoke it with an arbitrary object as in `t.dist({x=1;y=2})`.

Our strategy for program evolution is to first add optional types, catching and fixing unexpected local type errors; the programmer can then identify the parts of the code that obey to a stricter type discipline, and replace optional types with concrete types. Optional types act as a bridge to move values into the concrete world:

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

In the example, `p` has type `any`, and `u` points to a dynamic object with a method `dist` typed `any` $\rightarrow$ `any`. However, `u` has been typed as `TypedPoint`; the runtime will ensure that the method `dist` respects the `TypedPoint.dist` signature `any` $\rightarrow$ `!number` and will dynamically check that the returned value is an instance of class `number`. As a consequence, `fact(u.dist(p))` is well-typed (the concretely typed function `fact` is guaranteed to receive a value of type `!number`) and the programmer, by specifying just one optional type, can invoke the concretely-typed function `fact` with a value that has been computed from the dynamic world. The ability to have fine grained control over typing guarantees is one of the main benefits of StrongScript.

4.2 From TypeScript to StrongScript Types

A significant departure of our work is that we adopt nominal subtyping for classes and retain structural subtyping for interfaces. If a class `C` extends `D`, their concrete types are subtypes, denoted `!C` <: `!D`. Furthermore each concrete type is a subtype of the corresponding optional type, `!C` <: `C`, with an order on optional types that mirrors the one on concrete types: `!C` <: `!D` implies `C` <: `D`. `any` is an isolate with no super or subtype. Subtyping for interfaces follows [1] with the exception that an interface cannot extend a class.

Casts play a central role in the type system. Statically casts are always allowed to and from `any`, while casts to optional and concrete types are only permitted if one type is subtype of the other. At runtime, all programmer-inserted casts are checked, and additional casts are added by the implementation. Whenever a function with concretely typed arguments is injected in a dynamic context, the runtime adds a wrapper that uses casts to check the actual arguments. For instance, casting `fact` to `any` results in the following wrapper:

```
func(x) { <any>(fact( <!number>x)) }
```

To keep the syntax of the two languages in sync, several TypeScript dynamic features are rewritten as implicit casts. In particular, at function arguments and the right hand side of the assignment operator, casts to or from `any` and optional types are inserted automatically. For instance, the expression on the left is transformed into the one on the right:

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

If casts from `any` or optional types to concrete types are inserted, they are checked exactly like explicit casts. In addition, to support TypeScript's unsafe covariant subtyping, covariant overloading is implemented by injecting casts. Finally, casts are inserted in function calls to assure that if the function is called from an untyped context, its type annotations are honored. For instance, the class `CPoint` below extends `Point` and requires a concrete type for the argument of `dist`:

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

The overloading of `dist` is unsound, as `CPoint` is a subtype of `Point`. It is rewritten to perform a cast, and thus a check, on its argument `p`:

```
class CPoint extends Point {  ...
  dist(pa){var p:!CPoint = <!CPoint>pa; ...p.color...}}
```

Departing from TypeScript, the type of `this` is not `any`, but the concrete type of the surrounding class. This allows calls to methods of `this` to be statically type checked. But it creates an incompatibility with TypeScript code which uses "method stripping". It is possible to remove a method from the context of its object, and by using the builtin function `call`, to call the method with a different value for `this`. Consider, for instance, the following example:

```
class Animal {
  constructor(public nm:string) {}}
class Loud extends Animal {
  constructor(nm:string, public snd:string) { super(nm) }
  speak() { alert(this.nm+" says "+this.snd) }}

var a = new Animal("Snake");
var l = new Loud("Chris", "yo");
var m = l.speak;
m.call(a);
```

The `speak` method will be called with `this` referring to an `Animal`. This is plainly incorrect, but allowed, and will result in the string `"Chris says undefined"`. In StrongScript, `this` is concrete and the stripped method will include checks that cause the call to fail.

4.3 Backwards compatibility

JavaScript allows a range of highly dynamic features. StrongScript does not prevent any of these features from being used, but, since their type behavior is so unpredictable, it does not attempt to provide informative types for them. For instance, as objects are maps of string field names to values, it is possible to access members using computed strings. Thus `x[y]` accesses a member of `x` named by the string value of `y`, coercing it to a string if necessary; the type of the expression is always `any`. Assignment to `x[y]` may fail, if the member has a concrete type and the assigned value is not a subtype. Similarly, `eval` takes any string and executes it as code. StrongScript treats that code as JavaScript, not StrongScript. This is not an issue in practice as `eval`'s uses are mostly mundane [17]. The type of `eval(x)` is `any`.

Objects in JavaScript can be extended by adding new fields, and fields may be removed. An object's StrongScript type must be correct insofar as all fields and methods supported by its declared type must be present, but fields and methods *not* present in its type are unconstrained. As such, StrongScript protects its own fields from deletion or update to values of incorrect types, but does not prevent addition or deletion of new fields and methods. It is even possible to dynamically add new methods to classes, by updating an object prototype. None of this affects the soundness of the type system, and access to one of these in a value not typed `any` will result in a static type error.

4.4 Discussion

While our prototype implements an optional blame tracking mode similar to Typed Racket, we do not recommend it for production as it incurs performance overheads. Wrappers require, for instance, specialized field access code. We envision blame tracking as an optional command line switch like assertion checking.

The change to nominal subtyping is controversial but practical experience suggests that structural subtyping is rather brittle. In large systems, developed by different teams, the structural subtype relations are implicit and thus any small change in one part of the system could break the structural subtyping expected by another part of the system. We believe that having structural subtyping for optionally typed interface is an appropriate compromise. It should also be noted that Strongtalk started structural and switched to nominal [4].

StrongScript departs from Thorn inasmuch Thorn performed an optimized check on method invocation on optionally typed objects: rather than fully type checking the actual arguments against the method interface, it relied on the fact that this check had already been performed statically and simply compared the interface of the method invoked against the interface declared in the like type annotation. Thorn's type system is sound, but the simpler check introduces an asymmetry between optional and dynamic types at runtime which Thiemann exploited to prove that Thorn is not trace-preserving (personal communication).

5 Formal properties

We formalize StrongScript as an extension of the core language λ_{JS} of [14]; in particular we equip λ_{JS} with a nominal class-based type system à la Featherweight Java [15] and optional types. This treatment departs from Bierman et al. [1] in that they focused on typing interfaces and ignored classes, whereas we ignore interfaces and focus on classes. Thus our calculus does not include rules needed for structural subtyping of interface types; these rules would, assumedly, follow [1] but would add too much baggage to the formalization that is not directly relevant to our proposal. We also do not model method overloading (as discussed, StrongScript keeps covariant overloading sound by inserting appropriate casts) and references; our design enforces the runtime abstractions even in presence of aliasing.

Syntax. Class names are ranged over by C, D, the associated optional types are denoted by C and concrete types by $!C$, and the dynamic type is any. The function type $t_1 \mathinner{..} t_n \to t$ denotes explicitly typed functions while the type undefined is the type of the value `undefined`. The syntax of the language makes it easy to disambiguates class names from optional type annotations.

$$t ::= \; !C \;\mid\; C \;\mid\; \textsf{any} \;\mid\; t_1 \mathinner{..} t_n \to t \;\mid\; \texttt{undefined}$$

A program consists of a collection of class definitions plus an expression to be evaluated. A class definition $\texttt{class}\, C\, \texttt{extends}\, D\{s_1{:}t_1 \mathinner{..} s_k{:}t_k; md_1 \mathinner{..} md_n\}$ introduces a class named C with superclass D. The class has fields $f_1 \mathinner{..} f_k$ of types $t_1 \mathinner{..} t_k$ and methods $md_1 \mathinner{..} md_n$, where each method is defined by its name m, its signature, and the expression e it evaluates, denoted $m(x_1{:}t_1 \mathinner{..} x_k{:}t_k)\{\,\texttt{return}\, e{:}t\}$. Type annotations appearing in fields and method definitions in a class definition cannot contain `undefined` or function types. Rather than baking base types into the calculus, we assume that there is a class *String*; string constants are ranged over by s. Expressions are inherited from λ_{JS} with some modifications:

$$\begin{aligned} e ::= \;& x \;\mid\; \{\, s{:}e \mathinner{..} \mid t \,\} \;\mid\; e_{1\langle t\rangle}[e_2] \;\mid\; e_1[e_2] = e_3 \;\mid\; \texttt{delete}\, e_1[e_2] \;\mid\; \langle t\rangle e \\ \mid\;& \texttt{new}\, C\ (\, e_1 \mathinner{..}\,) \;\mid\; \texttt{let}\ (x{:}t = e_1)\ e_2 \;\mid\; \texttt{func}(x_1{:}t_1 \mathinner{..})\{\texttt{return}\ e : t\} \;\mid\; e(e_1 \mathinner{..}) \end{aligned}$$

Functions and let bindings are explicitly typed, expressions can be casted to arbitrary types, and the $\texttt{new}\, C\ (\, e_1 \mathinner{..}\,)$ expression creates a new instance of class C. More interestingly, objects, denoted $\{\, s{:}e \mathinner{..} \mid t \,\}$, in addition to the fields' values, carry a type tag t: this is any for usual dynamic JavaScript objects, while for objects created by instantiating a class it is the name of the class. This tag enables preserving the class-based object abstraction at runtime. Additionally, field access (and, in turn, method invocation) is annotated with the static type t of the callee e_1: this is used to choose the correct dispatcher or getter when executing method calls and field accesses. These annotations can be added via a simple elaboration pass on the core language performed by the type checker.

$$\frac{}{!C <: !\mathit{Object}}\,[\textsc{SObject}] \qquad \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\ ...\}}{!C <: !D}\,[\textsc{SClass}] \qquad \frac{}{\mathsf{undefined} <: t}\,[\textsc{SUndef}] \qquad \frac{}{!C <: C}\,[\textsc{SOptInj}] \qquad \frac{!C <: !D}{C <: D}\,[\textsc{SOptCov}]$$

$$\frac{t <: t' \quad t'_1 <: t_1\,..}{t_1.. \to t <: t'_1.. \to t'}\,[\textsc{SFunc}] \qquad \frac{\Gamma \vdash e : t_1 \quad t_1 <: t_2}{\Gamma \vdash e : t_2}\,[\textsc{TSub}] \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}\,[\textsc{TVar}] \qquad \frac{}{\Gamma \vdash \texttt{undefined} : \mathsf{undefined}}\,[\textsc{TUndefined}]$$

$$\frac{\begin{array}{l}\Gamma \vdash e : t_1 \\ t_1 = \mathsf{any} \vee t_2 = \mathsf{any} \vee t_1 <: t_2 \vee t_2 <: t_1\end{array}}{\Gamma \vdash \langle t_2 \rangle e : t_2}\,[\textsc{TCast}] \qquad \frac{x_1 : t_1.., \Gamma \vdash e : t}{\Gamma \vdash \texttt{func}(x_1{:}t_1..)\{\texttt{return}\ e : t\} : t_1.. \to t}\,[\textsc{TFunc}]$$

$$\frac{}{\Gamma \vdash \{..\ |\ t\} : t}\,[\textsc{TObj}] \qquad \frac{\begin{array}{l}\Gamma \vdash e_1 : \mathsf{any} \\ \Gamma \vdash e_2 : t\end{array}}{\Gamma \vdash \texttt{delete}\ e_1[e_2] : \mathsf{any}}\,[\textsc{TDelete}] \qquad \frac{\begin{array}{l}t = !C \vee C \\ \Gamma \vdash e : t\end{array}}{\Gamma \vdash e_{\langle t \rangle}[s] : C[s]}\,[\textsc{TGet}] \qquad \frac{\begin{array}{l}\Gamma \vdash e_1 : \mathsf{any} \\ \Gamma \vdash e_2 : t\end{array}}{\Gamma \vdash e_{1\,\langle \mathsf{any} \rangle}[e_2] : \mathsf{any}}\,[\textsc{TGetAny}]$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : t \\ t = !C \vee C \\ \mathsf{not\_function\_type}(C[s]) \\ \Gamma \vdash e_2 : C[s]\end{array}}{\Gamma \vdash e_1[s] = e_2 : t}\,[\textsc{TUpdate}] \qquad \frac{\begin{array}{l}\Gamma \vdash e_1 : \mathsf{any} \\ \Gamma \vdash e_2 : t_2 \\ \Gamma \vdash e_3 : t_3\end{array}}{\Gamma \vdash e_1[e_2] = e_3 : \mathsf{any}}\,[\textsc{TUpdateAny}] \qquad \frac{\begin{array}{l}\mathsf{fields}\,(C) = s_1{:}t_1\,..\,s_n{:}t_n \\ \Gamma \vdash e_1 : t_1 \quad .. \quad \Gamma \vdash e_n : t_n\end{array}}{\Gamma \vdash \texttt{new}\ C(e_1\,..\,e_n) : !C}\,[\textsc{TNew}]$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : t \\ x{:}t, \Gamma \vdash e_2 : t'\end{array}}{\Gamma \vdash \texttt{let}\ (x{:}t = e_1)\ e_2 : t'}\,[\textsc{TLet}] \qquad \frac{\begin{array}{l}\Gamma \vdash e : t_1\,..\,t_n \to t \\ \Gamma \vdash e_1 : t_1 \quad .. \quad \Gamma \vdash e_n : t_n\end{array}}{\Gamma \vdash e(e_1\,..\,e_n) : t}\,[\textsc{TApp}] \qquad \frac{\begin{array}{l}\Gamma \vdash e : \mathsf{any} \\ \Gamma \vdash e_1 : t_1 \quad .. \quad \Gamma \vdash e_n : t_n\end{array}}{\Gamma \vdash e(e_1\,..\,e_n) : \mathsf{any}}\,[\textsc{TAppAny}]$$

$$\frac{\begin{array}{l}\forall\, i.\ t_i \neq \texttt{undefined} \wedge t_i \neq t'_1..t'_{n'} \to t' \\ \forall\, i.\ \vdash\ md_i \\ (s_1..) \cap \mathsf{fields}(D) = \emptyset\ \wedge\ (md_1..) \cap \mathsf{methods}(D) = \emptyset\end{array}}{\vdash \texttt{class}\ C\ \texttt{extends}\ D\ \{\ s_1{:}t_1..;\ md_1..\ \}}\,[\textsc{TClass}] \qquad \frac{x_1 : t_1.. \vdash e : t}{\vdash m(x_1 : t_1..)\{\texttt{return}\ e : t\}}\,[\textsc{TMethod}]$$

Fig. 2. The type system.

Runtime abstractions. Two worlds coexist at runtime: fully dynamic objects, characterized by the any type tag, and instances of classes, characterized by the corresponding class name type tag. Dynamic objects can grow and shrink, with fields being added and removed at runtime, and additionally values of arbitrary types can be stored in any field, exactly as in JavaScript. The reduction rules confirm that on objects tagged any it is indeed possible to create and delete fields, and accessing or updating a field always succeeds.

[EGetProto]

$$\frac{s \notin \{\ s...\}}{\{\ \texttt{"\_\_proto\_\_"}{:}v, s{:}v\ ..\ |\ t\ \}_{\langle t' \rangle}[s] \longrightarrow v_{\langle t' \rangle}[s]}$$

[EGet]

$$\frac{s \in \mathsf{fields}(C)}{\{s{:}v\ ..\ |\ t\}_{\langle\ !C\rangle}[s] \longrightarrow v}$$

[EGetAny]

$$\frac{}{\{s{:}v\ ..\ |\ t\}_{\langle \mathsf{any} \rangle}[s] \longrightarrow \langle \mathsf{any} \rangle v}$$

[EGetOpt]

$$\frac{s \in \mathsf{fields}(C)}{\{s{:}v\ ..\ |\ t\}_{\langle C \rangle}[s] \longrightarrow \langle C[s] \rangle v}$$

[EUpdate]

$$\frac{\mathsf{tag}(v') <: C[s] \vee s \notin \mathsf{fields}(C)}{\{s{:}v\ ..\ |\ \ !C\}[s] = v' \longrightarrow \{s{:}v'\ ..\ |\ \ !C\}}$$

[EUpdateAny]

$$\frac{}{\{s{:}v\ ..\ |\ \mathsf{any}\}[s] = v' \longrightarrow \{s{:}v'\ ..\ |\ \mathsf{any}\}}$$

[ECreate]

$$\frac{s_1 \notin \{\ s...\}}{\{\ s{:}v\ ..\ |\ t\ \}[s_1] = v \longrightarrow \{\ s_1{:}v, s{:}v\ ..\ \ |\ t\ \}}$$

[EDelete]

$$\frac{t = \mathsf{any} \vee (t = !C \wedge s \notin \mathsf{fields}(C))}{\texttt{delete}\ \{s{:}v\ ..\ |\ t\}[s] \longrightarrow \{..\ |\ t\}}$$

[EGetNotFound]

$$\frac{s' \notin \{\ s...\} \qquad \texttt{"\_\_proto\_\_"} \notin \{\ s...\}}{\{\ s{:}v\ ..\ |\ t\ \}_{\langle t' \rangle}[s'] \longrightarrow \texttt{undefined}}$$

[EDeleteNotFound]

$$\frac{s \notin \{\ s_1...\} \vee (t = !C \wedge s \in \mathsf{fields}(C))}{\texttt{delete}\ \{\ s_1{:}v_1\ ..\ |\ t\ \}[s] \longrightarrow \{\ s_1{:}v_1\ ..\ |\ t\ \}}$$

[ELet]

$$\frac{}{\texttt{let}\ (x{:}t = v)\ e \longrightarrow e\{x/v\}}$$

[ECastObj]

$$\frac{(t' = !C \wedge t' <: t) \vee (t = \mathsf{any} \vee C)}{\langle t \rangle \{..\ |\ t'\} \longrightarrow \{..\ |\ t'\}}$$

[ECastFun]

$$\frac{t' = t_1'.. \rightarrow t'' \vee (t' = \mathsf{any} \wedge t_1' = \mathsf{any}\ .. \wedge t'' = \mathsf{any})}{\begin{array}{l}\langle t' \rangle (\texttt{func}(x_1{:}t_1..)\{\texttt{return}\ e : t\}) \longrightarrow \\ \qquad \texttt{func}(x_1{:}t_1'..)\{\texttt{return}\ \langle t'' \rangle ((\texttt{func}(x_1{:}t_1..)\{\texttt{return}\ e : t'\})(\langle t_1 \rangle x'..)) : t''\}\end{array}}$$

[EApp]

$$\frac{}{(\texttt{func}(x_1{:}t_1..)\{\texttt{return}\ e : t\})(v_1..) \longrightarrow e\{x_1/v_1..\}}$$

[ECtx]

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

[ENew]

$$\frac{}{\texttt{new}\ C\ (\ v_1..\ ) \longrightarrow \{\ \mathsf{gfields}\ C\ (v_1..);\ \mathsf{gmethods}\ C\ |\ !C\}}$$

where, for $\texttt{class}\ C\ \texttt{extends}\ D\{s_1{:}t_1\ ..\ s_k{:}t_k; md_1\ ..\ md_n\}$, we define:

$$\begin{array}{l}
\mathsf{gfields}\ C\ (v_1..v_n\, v'..) \triangleq s_1{:}v_1..s_k{:}v_k;\ \mathit{fields}\ D\ (v'..) \\
\mathsf{gmth}\,(m(x_1 : t_1..)\{\texttt{return}\ e : t\}) \triangleq \texttt{"}m\texttt{"} : \texttt{func}(x_1{:}t_1..)\{\texttt{return}\ e : t\} \\
\mathsf{gmethods}\ C \triangleq \texttt{"__proto__"} = \{\ \mathsf{gmth}\, md\ ..; \mathsf{gmethods}\ D\ |\ C_{proto}\ \}
\end{array}$$

Fig. 3. The dynamic semantics.

In our design, objects which are instances of classes benefit from static typing guarantees; for instance, runtime type checking of arguments on method invocation is not needed as the type of the arguments has already been checked statically. For this, we protect the class abstraction: all fields and methods specified in the class interface must always be defined and point to values of the expected type. To understand how this is done, it is instructive to follow the life of a class-based object. The ENew rule implements the class pattern [8] commonly used to express inheritance in JavaScript. This creates an object with properly initialized fields — the type of the initialization values was checked statically by the TNew rule — and the methods stored in an object reachable via the `"__proto__"` field — the conformance of the method bodies with their interfaces is checked when typechecking classes, rules TClass and TMethod. For each method m defined in the interface, a corresponding function is stored in the prototype. The following type rules for method invocation can thus be derived from the rules for reading a field and applying a function:

$$\frac{t = !C \vee C \quad \Gamma \vdash e : t \quad C[s] = t_1 \mathinner{..} t_n \to t' \quad \Gamma \vdash e_1 : t_1 \quad .. \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash e_{\langle t \rangle}[s](e_1 \mathinner{..} e_n) : t'} \qquad \frac{\Gamma \vdash e : \mathsf{any} \quad \Gamma \vdash e' : t' \quad \Gamma \vdash e_1 : t_1 \quad .. \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash e_{\langle \mathsf{any} \rangle}[e'](e_1 \mathinner{..} e_n) : \mathsf{any}}$$

The static view of the object controls the amount of type checking that must be performed at runtime. For this, field lookup $e_{\langle t \rangle}[e']$ is tagged at runtime with the static type t of e, as enforced by rules TGet and TGetAny. The absence of implicit subsumption to `any` guarantees that the tag is correct.

Suppose that the class *Num* implements integers and defines the method $+ : !\mathit{Num} \to\ !\mathit{Num}$. Let class C be:

$$\texttt{class}\ C\{m(x\colon !\mathit{Num})\{\ \texttt{return}\ x + 1\colon !\mathit{Num}\}\}$$

Invoking m in a statically typed context directly passes the arguments to the method body:[1]

$$\begin{array}{l}(\texttt{new}\ C(\,))_{\langle C \rangle}[\texttt{"}m\texttt{"}](1) \xrightarrow{\text{ENew}} \{\texttt{"\_\_proto\_\_"}\text{:}\{\texttt{"}m\texttt{"}\text{:}v \mid\ !C_{proto}\} \mid\ !C\}_{\langle\, !C \rangle}[\texttt{"}m\texttt{"}](1) \\ \xrightarrow{\text{EGetProto}} \{\texttt{"}m\texttt{"}\text{:}v \mid\ !C_{proto}\}_{\langle\, !C \rangle}[\texttt{"}m\texttt{"}](1) \xrightarrow{\text{EGet}} v(1)\end{array}$$

where $v = \texttt{func}\,(x\colon !\mathit{Num})\{\ \texttt{return}\ x + 1\colon !\mathit{Num}\}$. In a dynamic context, method invocation initially typechecks the arguments against the parameter type annotations of the method:

[1] For simplicity we ignore the *this* argument. A preliminary λ_{JS}-like desugarer would rewrite the class definition as $\texttt{class}\ C\{m(\mathit{this}\colon !C, x\colon\mathit{Num})\{\ \texttt{return}\ x{+}1\colon\mathit{Num}\}\}$ and the method invocation as $\texttt{let}\ (o\colon !C = \texttt{new}\ C(\,))\ o_{\langle\, !C \rangle}[\texttt{"}m\texttt{"}](o, 1)$.

$$
\begin{aligned}
&(\langle \mathsf{any}\rangle \mathtt{new}\, C(\,))_{\langle \mathsf{any}\rangle}[\texttt{"}m\texttt{"}](1) \xrightarrow{\text{ENew}} \\
&\quad (\langle \mathsf{any}\rangle\{\texttt{"\_\_proto\_\_"}{:}\{\texttt{"}m\texttt{"}{:}v \mid\ !C_{proto}\} \mid\ !C\})_{\langle \mathsf{any}\rangle}[\texttt{"}m\texttt{"}](1) \\
&\quad \xrightarrow{\text{ECast}}\ \{\texttt{"\_\_proto\_\_"}{:}\{\texttt{"}m\texttt{"}{:}v \mid\ !C_{proto}\} \mid\ !C\}_{\langle \mathsf{any}\rangle}[\texttt{"}m\texttt{"}](1) \\
&\quad \xrightarrow{\text{EGetProto}}\ \{\texttt{"}m\texttt{"}{:}v \mid\ !C_{proto}\}_{\langle \mathsf{any}\rangle}[\texttt{"}m\texttt{"}](1) \ \xrightarrow{\text{EGetAny}}\ ((\langle \mathsf{any}\rangle v)(1) \\
&\quad \xrightarrow{\text{ECastFun}}\ \langle \mathsf{any}\rangle(\mathtt{func}\,(x{:}\mathsf{any})\{\,\mathtt{return}\, v(\langle\ !Num\rangle x){:}\ !Num\}(1)) \\
&\quad \xrightarrow{\text{EApp}}\ \langle \mathsf{any}\rangle(v(\langle\ !Num\rangle 1)) \ \xrightarrow{\text{ECast}}\ \langle \mathsf{any}\rangle(v(1))
\end{aligned}
$$

The expression above dynamically checks that the method argument argument is a $!Num$ (last ECast reduction) via the cast introduced by the combination of EGetAny and ECastFun rule. Observe that the choice of the rule EGetAny was guided by the tag any of the field access. The return value is injected back into the dynamic world via a cast to any, thus matching the corresponding static type rule. Contrast this with an invocation at the optional type D for some class D that defines a method m with type $!Num \to t$:

$$
\begin{aligned}
&(\langle D\rangle \mathtt{new}\, C(\,))_{\langle \mathsf{any}\rangle}[\texttt{"}m\texttt{"}](1) \xrightarrow{\text{ENew}}\xrightarrow{\text{ECast}}\xrightarrow{\text{EGetProto}}\ \{\texttt{"}m\texttt{"}{:}v \mid\ !C_{proto}\}_{\langle D\rangle}[\texttt{"}m\texttt{"}](1) \\
&\quad \xrightarrow{\text{EGetOpt}}\ ((\langle\ !Num \to t\rangle v)(1) \\
&\quad \xrightarrow{\text{ECastFun}}\ \langle t\rangle(\mathtt{func}\,(x{:}\ !Num)\{\,\mathtt{return}\, v(\langle\ !Num\rangle x){:}\ !Num\}(1)) \ \xrightarrow{\cdots}
\end{aligned}
$$

In this case rule EGetOpt, selected via the D tag, inserts a cast to $!Num \to t$ that not only typechecks the actual arguments (as the caller can still an arbitrary object), but also casts the return value to the type t expected by the context.

Other invariants that preserve the class-based objects are enforced via the rule EDeleteNotFound, that turns deleting a field appearing in the interface of a class-based object into a no-op (which in static contexts is also forbidden by the TDelete rule), and rule EUpdate, that ensures that a field appearing in a class interface can only be updated if the type of the new value is compatible with the interface. For this, the auxiliary function $\mathsf{tag}(v)$ returns the type tag of an object, and is undefined on functions.

A quick inspection of the type rules shows that optionally-typed expressions — that is, expressions whose static type is C — are treated by the static semantics as objects of type $!C$, thus performing local type checking. At runtime, the reduction semantics highlights instead that optionally-typed objects are treated as dynamic objects except for the treatment of the return values. This ensures the third key property of optional types, namely that whenever field access or method invocation succeeds, the returned value is of the expected value and not any. We have seen how this is realized on method invocation; similarly for field accesses, let C be defined as $\mathtt{class}\ C\{\texttt{"}f\texttt{"}{:}\ !Num\}$ and compare the typing judgments $\{..\mid t\}_{\langle \mathsf{any}\rangle}[\texttt{"}f\texttt{"}] : \mathsf{any}$ and $\{..\mid t\}_{\langle C\rangle}[\texttt{"}f\texttt{"}] : !Num$. Field access on an object in a dynamic context invariably returns a value of type any. Instead if the object is accessed as C, then the rule TGet states that the type of the field access is $!Num$ (which is then enforced at runtime by the cast inserted around the return value by rule EGetOpt).

Formalization. Once the runtime invariants are understood, the static and dynamic semantics is unsurprising. As usual, in the typing judgment for expressions, denoted $\Gamma \vdash e : t$, the environment Γ records the types of the free variables accessed by e. *Object* is a distinguished class name and is also the root of the class hierarchy; for each class name C we have a distinguished class name C_{proto} used to tag the prototype of class-based objects at runtime. Function types are covariant on the return type, contravariant on the argument types: since the formalization does not support method overriding, it is sound for the *this* argument to be contravariant rather that invariant, which simplifies the presentation; the implementation supports overriding and imposes invariance of the *this* argument. Optional types are covariant and it is always safe to consider a variable of type $!C$ as a variable of type C. The type rule for an object simply extracts its type tag, which as discussed is any for dynamic javascript objects, and a class name for objects generated as instances of classes (possibly with the *proto* suffix). The notation $C[s]$ returns the type of field s in class definition C; it is undefined if s does not belong to the interface of C. Auxiliary functions $\mathsf{fields}(C)$ and $\mathsf{methods}(C)$ return the set of all the fields and methods defined in class C (and superclasses). The condition $\mathsf{not\_function\_type}(C[s])$ ensures that method updates in class-based objects are badly typed. Evaluation contexts are defined as follows:

$$\begin{aligned} E ::= {} & \bullet \mid \mathtt{let}\,(x{:}t = E)e_2 \mid E_{\langle t\rangle}[e] \mid v_{\langle t\rangle}[E] \mid E[e_2] = e_3 \mid v[E] = e_3 \\ \mid {} & v_1[v_2] = E \mid E(e_1 \mathinner{..} e_n) \mid v(v_1 \mathinner{..} v_n, E, e_1 \mathinner{..} e_k) \mid \mathtt{new}\, C(v_1 \mathinner{..} v_n\, E\, e_1\, e_k) \\ \mid {} & \{s_1{:}v_1 \mathinner{..} s_n{:}v_n\; s{:}E\; s_1{:}e_1 \mathinner{..} s_k{:}e_k \mid t\} \mid \mathtt{delete}\, E[e] \mid \mathtt{delete}\, v[E] \mid \langle t\rangle E \end{aligned}$$

As mentioned above, method invocation has higher priority than field access, and reduction under contexts (rule ECtx) should try to reduce $e_{\langle t\rangle}[e'](e_1)$ to $v_{\langle t\rangle}[v'](v_1)$ whenever possible.

Metatheory. In StrongScript, *values* are functions, and objects whose fields contain values. We say that an expression is *stuck* if it is not a value and no reduction rule applies; stuck expressions capture the state of computation just before a runtime error. The Safety theorem states that a well-typed expression can get stuck only on a downcast (as in Java) or on an optional-typed or dynamic expression.

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

This theorem relies on two lemmas, the Preservation lemma states that typings (but not types) are preserved across reductions, and the Progress lemma identifies the cases above as the states in which well-typed terms can be stuck. The Safety theorem has several interesting consequences. First, a program in which all type annotations are concrete types has no runtime errors (apart from those occurring on downcasts): the concretely typed subset of StrongScript behaves as Featherweight Java (and, in turn, Java) and execution can be optimized along the same lines. Second, optional-typed programs (that is, programs with no occurrences of the any type and no downcasts to like types), benefit from the same execution guarantee: static type checking is strong enough to prevent runtime errors on entirely optional-typed programs.

The Trace Preservation theorem captures instead the idea that given a dynamic program, it is possible to add optional type annotations without breaking its runtime behavior; more precisely, if the type checker does not complain about the optional type annotation, then the runtime guarantees that the program will have the same behavior of the unannotated version. This theorem holds trivially in TypeScript because of type erasure.

Theorem 2 (Trace Preservation). *Let e be an expression where all type annotations are* any *and $\Gamma \vdash e : \mathsf{any}$. Let v be a value such that $e \longrightarrow^* v$. Let e' be e in which some type annotations have been replaced by optional type annotations (e.g. C, for C a class with no concrete types in its interface). If $\Gamma \vdash e' : t$ for some t, then $e' \longrightarrow^* v$.*

The Strengthening theorem states that if optional type annotations are used extensively, then the type checking performed is analogous to the type checking that would be performed by a strong type system à la Java. A consequence is that it is possible to transform a fully optionally typed program into a concretely typed program with the same behavior just by strengthening the type annotations. This property crucially relies on the fact that all source of unsoundness in our system are identified with explicit cast to optional types (or to any).

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 rewriting all occurrences of optional types C into the corresponding concrete types $!C$. The expression e' is well-typed and reduces to the same value v.*

6 Evaluating StrongScript

Our implementation of StrongScript consists of two components: an extended version of the TypeScript 0.9.1 compiler and a JavaScript engine derived from Oracle's `TruffleJS` [26]. The compiler outputs portable JavaScript, so the resulting code can run on any stock virtual machine, but no performance improvement should be expected in that case. The compiler is extended with the following type related features: (a) support for concrete types and dynamic contracts at explicit downcasts, (b) checked downcasts where TypeScript does so implicitly and unsoundly (including covariant subtyping), and (c) function code suitable for both typed and untyped invocation (including dynamic contracts at untyped invocation). The compiler emits intrinsics that describe the layout of concretely-typed objects: we extended the `TruffleJS` runtime to understand and exploit these intrinsics to perform check-free property access in concrete types. The compiler, as an option, can generate blame tracking wrappers for interfaces.

6.1 Implementation

Supporting concrete types simply requires adding the type constructor (`!`) and typing rules: `!C <: C` and `!C <: !D` implies `C <: D`. Since we use nominal typing for classes, optional and concrete types are compatible in both optional and concrete contexts; it is thus possible to implement type checks, using JavaScript's builtin `instanceof` mechanism. Nominal types are retained at runtime. The compiler ensures that concrete types are always used soundly. For this we include a small (200-line) library functions necessary to implement sound type checking. These functions rely on ECMAScript 5 features to protect themselves from being replaced or accidentally circumvented. To ensure soundness the compiler inserts dynamic contracts wherever unsafe downcasts occur, whether explicit or implicit. This is accomplished by the `$$check` function, which asserts that a value is of a specified type. For instance:

```
var untyped:any = new A();
var typed:!A = <!A>untyped;
```

is compiled into:

```
var untyped = new A();
var typed = A.$$check(untyped);
```

The check function is simple and generic, and does not require a per-class checker. For compatibility with TypeScript, several forms of unsafe, implicit casts are allowed in the source program. Specifically, implicit unsafe casts are inserted when a value is of type `any` and is in the context of a function argument or the right-hand-side of an assignment expression. For instance, the following code:

```
var unsafe:!B = <any>new A();
```

implies this additional cast:

```
var unsafe:!B = <!B><any>new A();
```

which in turn generates the following JavaScript code:

```
var unsafe = B.$$check(new A());
```

The cast to `!B` fails at runtime if `B` is not a supertype of `A`. Were this code to be rewritten with `unsafe:B` rather than `!B`, the inserted cast to `B` would imply no check, and the code would succeed at runtime. If the cast to `any` were omitted, this example would be rejected by the type checker.

Covariant overloading is implemented as unsafe downcasting, as described in Sec. 4.2. We describe some aspects of our type system as automatically-generated downcasts where TypeScript describes them as type compatibility. This is a matter of descriptive clarity and does not affect compatibility. All semantically valid TypeScript 0.9.1 programs, and programs valid in TypeScript 1.0 and greater which use types nominally and do not use features introduced after our version was forked from TypeScript, are semantically valid StrongScript with no syntactic changes.

Efficient and sound implementation of function code. Functions with type annotations may be called from typed or untyped contexts. If they have only optional types or `any`, this requires no checks. However, methods of classes don not fit that description, as the `this` argument is always concretely typed. One option would be to check all concretely typed arguments at runtime, but this would entail unnecessary dynamic checks when types of arguments are known. Our implementation generates both an unchecked and a checked function. The checked function simply verifies its arguments and then calls the unchecked function. Calls are redirected appropriately by a compilation step. For instance, the following code:

```
class Animal {
  constructor(name:String) {}
  eat(x:!Animal) {
   console.log(this.name+" eats "+x.name); }}

var a:!Animal = new Animal("Alice");
var b:any = a;
a.eat(new Animal("Bob"));
b.eat(new Animal("Bob"));
```

is translated by an intermediary stage to:

```
class Animal {
  constructor(name:String) {}
  $$unchecked$$eat(x:!Animal) {
    console.log(this.name+" eats "+x.name); }
  eat(x) {
    (<!Animal>this).$$unchecked$$eat(<!Animal>x); }}

var a:!Animal = new Animal("Alice");
var b:any = a;
a.$$unchecked$$eat(new Animal("Bob"));
b.eat(new Animal("Bob"));
```

Code is generated to assure that the `$$unchecked` versions of functions are unenumerable and irreplaceable. This prevents accidental damage, but is not safe against intentionally malicious code.

Intrinsics. With concrete types, it is possible to lay out objects at compile time, and to access fields and methods by their statically-known location in the object layout, obviating the need for hash table lookups. JavaScript, however, provides no way to explicitly specify the layout of objects. Therefore, to take advantage of known concrete objects, JavaScript code generated by StrongScript includes calls to several intrinsic operations which access fields by explicit offset within objects. On non-supporting engines, these intrinsics are implemented as no-ops. On `TruffleSS`, the only supporting engine, they are implemented as direct accesses. The intrinsics are `$$direct` and `$$directWrite`, and support direct reading and writing to offsets within an object, respectively. If an object is built with repeated `$$directWrite` calls, then fields are accessed with `$$direct` calls. For instance, the following StrongScript code:

```
class A { constructor(x:string); }
var a:!A = new A("foo");
alert(a.x);
```

compiles into the following JavaScript code:

```
function A(x) { this.$$directWrite(0).x = x; }
var a = new A("foo");
alert(a.$$direct(0).x);
```

Blame tracking. Our compiler support blame tracking to associate errors with the location of the responsible (structural) type cast. Unsafe downcasts to structural types are implemented by wrapping the objects with field getters and setters which validate their types [12]. The wrapper object additionally stores the location where it was created. When one of its type checks fail, both locations are reported. This strategy causes substantial runtime overhead, and blame tracking is disabled by default.

6.2 Evaluating Performance

We measure the performance of our implementation to demonstrate that adding type information to dynamic code can yield performance benefits. For this experiment, we modified a small number of programs to give them concrete types and compared the result of running those on the Truffle optimizing virtual machine against an untyped baseline. Truffle is a highly optimizing, type-specializing compiler. Many of its optimizations are redundant with our intrinsics and we expect the relative speedups to reflect this fact.

As there are no established TypeScript benchmarks, we adapted a number of programs from the Programming Language Benchmarks Game[2] and Octane[3]. We focused on programs which use classes as our optimizations rely on their presence. We chose `bg-binarytrees`, `bg-nbody`, `oct-deltablue`, `oct-navier-stokes`, and `oct-splay`. Moreover, we also used benchmarks provided with SafeTypeScript, namely `sts-crypto`, `sts-navier-stokes`, `sts-raytrace`, `sts-richards` and `sts-splay`. `sts-navier-stokes` and `sts-splay` overlap with `oct-navier-stokes` and `oct-splay`, but are slightly different ports. The SafeTypeScript benchmarks were unchanged (i.e. no added concrete types).

For each benchmark, a type erased and a typed form were compiled, called the "TypeScript" and "StrongScript" forms. Each benchmark times long-running iterative processes; several thousand iterations are performed before timing begins to allow the JIT a warmup period. We compare the runtime between the two forms on the same engine. i.e., the only change is the inclusion of intrinsics and type protection. Each benchmark was run in each form 10 times, interleaved to

[2] http://benchmarksgame.alioth.debian.org/
[3] https://developers.google.com/octane/

reduce the possibility of outside interaction. For the Benchmarks Game benchmarks, the reported result is runtime in milliseconds, so lower values represent better performance. For the Octane benchmarks, the reported result is speedup over a reference runtime, so higher values represent better performance. We report the arithmetic means of the results in each form, as well as the speedup or slowdown. Benchmarks were run an 8-core 64-bit Intel Xeon E5410 with 8GB of RAM, running Gentoo Linux. Our modification of Truffle is based on a snapshot dated October 15th, 2013. The SafeTypeScript benchmarks were compared against a snapshot of SafeTypeScript dated October 24, 2014.

Results. Figure 4 shows that all benchmarks were sped up when the virtual machine can rely on concrete types. Three of the benchmarks had speed up large enough to be statistically significant. The performance benefits come from type-specialization intrinsics and direct access to fields in class instances. `bg-nbody` uses large objects with typed members, and our type-specialized intrinsics allow us to build these objects efficiently. Truffle has similar optimizations, but they are heuristic and less effective. `oct-deltablue` and `oct-splay` both use subclasses and polymorphism. Our member access intrinsics are not affected by subclass polymorphism, and therefore are reliably faster. Figure 4 also indicates the number of expressions, properties, and method arguments that had type annotations attached, this range from 7 to 186.

Benchmark	Annotations	TypeScript runtime	StrongScript runtime	Speedup
`bg-binarytrees`	7	5750	5627	2.1%
`bg-nbody`	22	898	715	20.4%
		Ref. speedup	**Ref. speedup**	
`oct-deltablue`	186	1701	2518	32.5%
`oct-navier-stokes`	94	9170	9492	3.4%
`oct-splay`	55	890	1092	18.4%

Fig. 4. Performance comparison on the Truffle VM. For `bg-` lower is better, for `oct-` smaller is better. Times are in milliseconds.

Figure 5 compares StrongScript with SafeTypeScript. For these benchmark, we use the SafeTypeScript code as such as it is valid StrongScript (in particular we do not add concrete types). On Truffle, we are slightly faster overall because SafeTypeScript performs more runtime checks. We also compared the speed of the two implementation on Node.js (using the V8 engine). On Node, two of the benchmarks exhibited slightly slower performance, we presume, because the virtual machine spent more time constructing self-protecting classes. The speedups due to fewer runtime checks were more pronounced on Node.

Threats to validity. The number of programs available and their nature makes it difficult to generalize form our results. At least they point to the potential

Benchmark	Annotations	SafeTypeScript runtime (Truffle)	StrongScript runtime (Truffle)	Speedup
`sts-crypto`	263	1967	1900	3.5%
`sts-navier-stokes`	104	1175	1157	1.5%
`sts-raytrace`	110	839	807	3.9%
`sts-richards`	67	449	419	7.1%
`sts-splay`	13	1888	1801	4.8%

Benchmark	Annotations	SafeTypeScript runtime (Node)	StrongScript runtime (Node)	Speedup
`sts-crypto`	263	9.7	8.7	12.3%
`sts-navier-stokes`	104	5.1	5.1	-0.1%
`sts-raytrace`	110	3.5	3.5	-1.4%
`sts-richards`	67	0.3	0.2	28.6%
`sts-splay`	13	162.5	136.9	18.7%

Fig. 5. Performance comparison on Tuffle and Node.js. StrongScript against SafeTypeScript. Lower is better. Times are in millis, rounded to the nearest tenth. Execution times across virtual machines are incomparable as they used different harnesses to measure runtimes.

for performance improvements with concrete types. Also, it is worthy of note that conventional wisdom amongst virtual machine designers is that type annotations are not needed to get performance for JavaScript. Our result suggest that this may not be the case. Of course, this should be validated on other engines. Because our intrinsics are unchecked JavaScript, it is possible to use them to circumvent security properties of the engine. Although this problem would be resolved by implementing StrongScript directly rather than through a translation layer, the performance characteristics of such a system may vary somewhat from what is achieved with a JavaScript system. Similar changes would be expected if StrongScript's specialized functions (e.g. `$$check` and `$$unchecked`) were made secure from malicious code. Our measured benchmark code has no unsafe downcasts, and thus no runtime type checking. The overall benefit of our intrinsics depends on the underlying engine, and specifically the precision of its speculation. Our intrinsics would be expected to show narrower advantages over an engine with better object layout speculation; however our intrinsics ensure *predictable* benefits, while layout speculation relies on complex heuristics that might be invalidated with program evolution.

7 Conclusion

StrongScript is a natural evolution of the TypeScript design. Optional type annotations have already proven to be useful in practice despite their lack of runtime guarantees or performance benefits. With a modicum effort from the programmer, StrongScript can provide stronger runtime guarantees and predictable performance while allowing idiomatic JavaScript code and flexible program evolution. The type systems of TypeScript and StrongScript are fundamentally different, the former being intrinsically unsound for the stated goal of typing as many JavaScript programs as possible, and the latter being sound to ensure stronger invariants when needed. In practice, we have found that StrongScript type system does not limit expressiveness as our compiler silently inserts all the needed casts to optional types or `any` needed to mimic the unsound behaviors of TypeScript. The only incompatibilities between the two are due to structural vs. nominal subtyping on optional class types. However all programs well-typed in versions of TypeScript up-to 0.9.1 – which relied on nominal subtyping – are well-typed StrongScript programs, and the large benchmarks of [16] suggest that this is not a problem in practice. Compared to SafeTypeScript, our design delivers the flexibility offered by the optional types and the predictable performance given by intrinsics. In particular, in our design, optional types are not only useful for program evolution but can also durably play the role of interfaces between the dynamic and concretely typed parts of a program, avoiding the need for extra casts to concrete types.

The fact that we are able to achieve performance gains on a highly optimizing virtual machine gives one more reason for developers to adopt concrete types.

Artifact Availability. StrongScript is an open source project. The implementation is hidden during the double blind review period as it cannot easily be anonymized, it will be released to Artifact Evaluation Committee.

References

1. Bierman, G., Abadi, M., Torgersen, M.: Understanding TypeScript. In: ECOOP (2014)
2. Bloom, B., Field, J., Nystrom, N., Östlund, J., Richards, G., Strnisa, R., Vitek, J., Wrigstad, T.: Thorn—robust, concurrent, extensible scripting on the JVM. In: OOPSLA (2009)
3. Borning, A.H., Ingalls, D.H.H.: A type declaration and inference system for Smalltalk. In: POPL (1982)
4. Bracha, G.: The Strongtalk type system for Smalltalk. In: OOPSLA Workshop on Extending the Smalltalk Language (1996)
5. Bracha, G.: Pluggable type systems. OOPSLA Workshop on Revival of Dynamic Languages (2004)
6. Bracha, G., Griswold, D.: Strongtalk: Typechecking Smalltalk in a production environment. In: OOPSLA (1993)
7. Cartwright, R., Fagan, M.: Soft Typing. In: PLDI (1991)
8. Crockford, D.: Classical inheritance in JavaScript. `http://www.crockford.com/javascript/inheritance.html`
9. Findler, R.B., Felleisen, M.: Contract soundness for object-oriented languages. In: OOPSLA (2001)
10. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: ICFP (2002)
11. Findler, R.B., Flatt, M., Felleisen, M.: Semantic casts: Contracts and structural subtyping in a nominal world. In: ECOOP (2004)
12. Flatt, M., PLT: Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Design Inc. (2010), `http://racket-lang.org/tr1/`

13. Gapeyev, V., Levin, M., Pierce, B.: Recursive subtyping revealed. Journal of Functional Programming 12(6), 511–548 (2002)
14. Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of JavaScript. In: ECOOP (2010)
15. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. 23(3) (2001)
16. Rastogi, A., Swamy, N., Fournet, C., Bierman, G., Vekris, P.: Safe and efficient gradual typing for TypeScript. In: POPL (to appear) (2015)
17. Richards, G., Hammer, C., Burg, B., Vitek, J.: The eval that men do: A large-scale study of the use of eval in JavaScript applications. In: ECOOP (2011)
18. Siek, J., Taha, W.: Gradual typing for objects. In: ECOOP (2007)
19. Siek, J., Vitousek, M., Kent, A., Baker, J.: Design and evaluation of gradual typing for Python. Tech. rep., Indiana University (2014), `http://wphomes.soic.indiana.edu/jsiek/files/2014/03/retic-python.pdf`
20. Siek, J.G.: Gradual Typing for Functional Languages. In: Scheme and Functional Programming Workshop (2006)
21. Suzuki, N.: Inferring types in smalltalk. In: POPL (1981)
22. Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: from scripts to programs. In: DLS (2006)
23. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed Scheme. In: POPL (2008)
24. Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: ESOP (2009)
25. Wrigstad, T., Zappa Nardelli, F., Lebresne, S., Östlund, J., Vitek, J.: Integrating typed and untyped code in a scripting language. In: POPL (2010)
26. Würthinger, T., Wimmer, C., Wöss, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M.: One VM to rule them all. In: Onwards! (2013)