Concrete Types for TypeScript

Abstract

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

1. Introduction

Unsound type systems seem to be all the rage. What is the attraction of type systems in which well-typed programs may experience type errors? Sometimes unsoundness arises for pragmatic reasons. Java in its early days lacked generics, thus array subtyping was made covariant to allow for a single implementation of the sort function. This choice implied that a store of a reference into an array had to be runtime-checked and could throw an exception. More interestingly, a number of industrial extensions to dynamic languages[1] have been designed with optional type systems [5] that provide few correctness guarantees. Languages such as StrongTalk [6], Hack [27], Dart [23] and TypeScript [16] extend Smalltalk, PHP, and JavaScript, following a "dynamic first" philosophy. Their design is geared to accommodate dynamic programming idioms and preserve the behavior of legacy code. Type annotations are second class citizens, tolerated as long they can reduce the testing burden and improve the user experience while editing code, but they should not get in the way.

Unsoundness means that a variable x annotated with type T may, at runtime, have a value of any type. This is due to unchecked casts, covariant subtyping, or the presence of untyped code. Implementations deal with unsoundness by ignoring type annotations. Thus, code emitted by the respective compilers fully erases all annotations. In TypeScript, constructs such as classes are translated to JavaScript code that preserves the semantics of the source program, but where all casts have been erased and no constraints are enforced on the values that can be stored in properties or passed as arguments. This also implies that no performance benefits are obtained from type annotations as the compiler must retain all runtime tests and cannot generate optimal code for field accesses.

The origins of this approach to typing can be traced to Bracha's work on optional types for StrongTalk [6] and pluggable types for Java [5]. In both cases, the goal was to superimpose a static type system on an existing language without affecting the behavior of programs. In this paper, we refer to this property as *trace preservation*. Informally, a type system is trace-preserving if adding type annotations to a program does not change its meaning. In contrast, recent proposals for gradual type systems [20, 25, 28] forgo trace-preservation in favor of improved error reporting. The pragmatic justification for optional types is that developers should have confidence that adding type annotation to a well-tested dynamic program will neither cause the program to get stuck nor degrade its performance.

This paper presents a design for a gradual type system for TypeScript that lets developers choose between writing code that has no type annotations (implicitly variables are of type any), with optional type annotations that are trace-preserving (types are erased), and finally with concrete type annotations that provide correctness guarantees but are not trace-preserving (types are retained and are used by the compiler). We name the language StrongScript in homage to StrongTalk. Its type system is inspired by ideas developed in Thorn [2] but terminology and syntax are aligned to TypeScript. Our design goals were as follows:

1. *Backwards compatibility:* All JavaScript programs must be valid StrongScript programs. Common 'dynamic' programming idioms should be easy typeable StrongScript.
2. *Performance model:* The StrongScript type system should not create runtime overheads; the only acceptable cost is for developer-inserted concrete casts.
3. *Correctness guarantees:* Optional type annotations should provide local guarantees that uses of a variable are consistent with declarations; concrete types should ensure soundness up to down casts.
4. *Runtime improvements:* It should be possible to demonstrate that type information can improve performance, this even in the context of a highly-optimizing JavaScript virtual machine.
5. *Addressing limitations:* TypeScript has a number of limitation. We focus on providing efficient checked casts as they central to common object oriented idioms.

StrongScript departs from TypeScript in significant ways. While TypeScript sports a structural type system with both classes and interfaces, StrongScript combines structural and nominal subtyping. Class subtyping is nominal; interface subtyping remains structural. Nominal subtyping allows for fast property access and efficient checked casts. Any class name C can be used as an optional type, written C, or as a concrete type, written !C. The difference between the two is that a variable of a concrete type is guaranteed to refer to an object of class C, a class that extends C or null. Interfaces are erased at compile time, as in TypeScript. Unannotated variables default to type any, the top of the type hierarchy. Our design ensures that JavaScript programs as valid StrongScript programs. Our contributions are as follows:

- *Type system design.* We exhibit a type system for TypeScript that integrates optional types with concrete types and that satisfies the above mentioned requirements.
- *Formalization.* A core calculus, in the style of λ_{JS} of [14], capture 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.
- *Trace preservation theorem.* We prove trace preservation for optional types. More precisely if expressions e is untyped, and e' only differs by the addition of optional types, then e and e' evaluate to the same value.

[1] Here, "dynamic language" refers to strongly typed programming languages where type checking happens at runtime. Examples include Perl, PHP, Ruby, JavaScript and Smalltalk. "Untyped code" refers to code without type annotations where variables are implicitly type any.

- *Concretization theorem.* We prove that when a fully optionally typed program is annotated with concrete types, the program will be trace preserving.
- *Implementation and evaluation.* StrongScript was implemented as an extension to version 0.9.1 of the TypeScript compiler. We have extended Truffle JavaScript implementation from Oracle labs [30] to provide fast access to properties access through concretely typed variables. Truffle/JS 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%.

As with the formalization of Bierman, Abadi and Torgersen [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 many years. Academics, traditionally, come with the intent of "curing the dynamic" as the absence of static types is clearly a major language design flaw. Practitioners, on the other hand, seek to supplement their exhaustive testing practices with additional machine-checked documentation and as much ahead-of-time error checking as they can get away with. They appreciate that, in a dynamic language, any grammatically correct program, even a partial program or one with obvious errors, can be run. This is particularly handy for exploratory programming. They also consider these language easier to teach to non-programmers and more productive for experts. Decades research were devoted to attempts to add static typing to dynamic languages. In the 1980's, type inference and soft-typing were proposed for Smalltalk and Scheme [3, 7, 22]. Inference based approaches turned out to be brittle as they required non-local analysis and were eventually abandoned.

Twenty years ago, while working at Anamorphic on the virtual machine that would eventually become known as HotSpot, Gilad Bracha designed the first optional type system [6]. Subsequent papers 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 run-time semantics, and
2. does not mandate type annotations in the syntax.

It is worth noting that StrongTalk was an industrial project, just like Facebook's Hack, Google's Dart, and Microsoft's TypeScript. In each case, a dynamic language is equipped with a static type system that is loose enough to support backwards compatibility with untyped code, and type annotations provided no guarantee of absence of type errors.

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 Scheme that used higher-order contracts to enforce types at module boundaries [25]. Typed Scheme has a robust implementation and is being used on large bodies of code [26]. But Typed Scheme, unlike TypeScript does not support mutation [24].

In parallel with the development of Typed Scheme, Siek and Taha defined *gradual typing* to refer to languages where type annotations can be added incrementally to untyped code [18, 20]. Like in Typed Scheme, wrappers are used to enforce types, 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 [21]. In a Python implementation, called Reticulated Python, both solutions cause slowdowns that are larger than 2x [19]. Even without mutation, the approach is not trace preserving. Consider the following Reticulated Python code:

```
class C:
  b = 41

def id(x:Object{b:String})->Object{b:String}:
  return x

f( C() ).b + 1
```

Without type annotations the program will evaluate to the number 42. When the type annotations are taken into account the program will stop at the read of property b. A type violation is reported as the required type for b is String while b holds an Int. While this case is rather trivial, similar problem can occur when developers put contracts that unnecessarily strong without understanding the range of types that can flow through a function. With optional types, no error would be reported.

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 [29]. Preliminary performance results suggested that concrete types could yield performance improvements when compared to a naive implementation of the language, but it is unclear if the results would hold for an optimizing compiler. Unlike Typed Scheme, Thorn did not support blame tracking to avoid having to pay for its overheads.

Figure 1 compares some of the approaches to typing dynamic languages. First we give the possible values of a variable annotated with some class type C. In TypeScript, there are no restrictions, and similarly for StrongScript. Typed Scheme and Reticular Python both allow for wrappers around any value. For concrete types, StrongScript restricts possible values to subtypes of C. TypeScript is the only fully trace preserving system. StrongScript is trace preserving up to concrete type casts. Mutable state is supported by all languages except Typed Scheme. Lastly, StrongScript is the only language to guarantee fast access to properties of concrete typed variables.

	Typ.Script	Typ.Scheme	Ret.Python	Str.Script
x : C	any	W	W	any
x : !C	–	–	–	C
Trace pres.	●	○	○	◐
Mut. state	●	○	●	●
Fast access	◐	○	○	●

Figure 1. Comparing optional and gradual type systems.

3. TypeScript: Unsound by design

Bierman, Abadi and Torgersen captured the key aspects of the design of TypeScript in [1], we recall it here. 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 to blame for 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] can not 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 StrongScript. Starting with subtyping. Consider the following well-typed TypeScript example:

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

Interfaces can include properties and methods. Extends declaration amongst interface are not required for other purposes than documenting programmer intent. Above, interface Pt is a structural subtype of both P and T.

```
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 ... }
}
```

Classes can be defined as one would expect, here the extends clause has a semantic meaning as it specifies inheritance of properties. Both of the classes are subtypes of the interfaces declared above. Note that the dist method is being overridden in covariantly at the argument p and that CPoint.dist in fact does require the argument to be an instance of the CPoint class.

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

The first assignment implicitly cast Point to Pt which is allowed by structural subtyping.

```
function pdist(x:Point, y:Point) { x.dist(y); }
pdist(c,o);
```

The function pdist will invoke dist at static types Point, yet it is invoked with a CPoint as first argument. The TypeScript compiler allows the call, at runtime the attempt to access the p.color property will return the undefined value.

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

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 in all argument positions and be converted implicitly to any other type.

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

Finally, we demonstrate 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.

In none of the cases above the TypeScript compiler emit a warning.

Bierman et al. showed that the core TypeScript type system can be formalized as the combination of a sound calculus extended with few unsound rules. For our purposes, the sound calculus can be seen as a system with records, equi-recursive types and structural subtyping. The resulting assignment compatibility relation can be defined coinductively using well-studied techniques along the lines of [13]. We underline the critical choice of defining any as the super-type of all the types; since up-casts 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 then introduced via three changes. The first add a type-check rule that allows *down-casts* to subtypes (but not to arbitrary types, for which the compiler emits a warning). The second change is more interesting, as it changes quite drastically the subtyping relation by stating that *all types are super-types of* any. This implies that arbitrary dynamic values can flow into typed variables without the need of explicit casts: in TypeScript no syntactic construct identifies the boundaries between the dynamic and typed world. The third change enables *covariant overloading* of class/interface members and method parameters.

In this treatment we will ignore type inference as it is not crucial to our proposal, and, like Bierman et al., omit generics – for which decidability of subtyping is describing as "challenging" [1]. We also we omit the discussion of the liberal use of indexing allowed by TypeScript. Our implementation supports the same liberal indexing by explicitly inserting type casts at compilation (see Section 4).

4. StrongScript: Sound when needed

StrongScript is a backwards compatible extension to JavaScript: any JavaScript program is a valid and well-typed StrongScript program. StrongScript builds on TypeScript but changes its semantics in subtle ways: though most of the time the two will agree, in some cases well-typed TypeScript programs will be rejected by the StrongScript type-system.[2] We focus here on the departures from TypeScript, everything else can be assumed to behave identically.

Syntactically, the only difference is the presence of a new type constructor, written !. StrongScript thus has three three classes of type annotations:

Dynamic types Denoted by any, represent values that are manipulated with no static checks. Any value can be referenced by a variable that has type any, all operations are allowed and may fail at runtime.

Optional types Denoted by class names C, capture the same intent of TypeScript type annotations. They enable local type-checking, because all manipulations of optionally typed variables are checked statically against C's interface, while, at run-

[2] In some cases involving method-stripping, i.e. when a method is separated from the object it belongs to, well-typed TypeScript programs can be well-typed in StrongScript but will raise a dynamic error (see Section 4.2).

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

class PtList extends Point {
 next = this;
 constructor(x,
             y) {
  super(x, y)
 }
 add(p) {
  var l=new PtList(p.x,p.y)
  l.next = this
  return l
 }
}
```

(a) Dynamic

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

class PtList extends Point {
 next: PtList = this;
 constructor(x: number,
             y: number) {
  super(x, y)
 }
 add(p: Point): PtList {
  var l = new PtList(p.x,p.y)
  l.next = this
  return l
 }
}
```

(b) Optional

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

class PtList extends Point {
 next: !PtList = this;
 constructor(x: !number,
             y: !number) {
  super(x, y)
 }
 add(p: Point): !PtList {
  var l = new PtList(p.x,p.y)
  l.next = this
  return l
 }
}
```

(c) Concrete

Figure 2. StrongScript allows developers to gradually, and selectively, increase their confidence in their code. Program (a) is entirely dynamic; (b) has optional type annotations to enable local type checking; (c) has some concrete types: properties are concrete and arguments to constructors as well, but the arguments to dist and add remain optionally typed.

time. At the same time, optionally typed variables can reference arbitrary values and the run-time will still check all their uses.

Concrete types At the other extreme, concrete types, denoted by !C where C is a class name, represent only objects that are instance of the homonymous class or its subclasses. Static type-checking la Java is performed on these, and the above run-time invariants ensures that no dynamic checks are needed.

Optional types have the same intent as TypeScript type annotations: they capture 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. No run-time error can arise from using a concretely typed variable and the compiler can rely on these static type information to emit efficient code with optimizations such as unboxing and inlining.

To make good on the promise of concrete types, StrongScript is built on a sound type system. This forces some changes to TypeScript's overly permissive type rules as well as to the underlying implementation. In StrongScript, casts between optional and concrete types are explicit and are checked at runtime. Covariant subtyping, such as the array subtype rule, involves runtime checks as well. Moreover, to deliver performance improvements, class subtyping is nominal.[3] Subtyping is slightly simpler as we do not allow for any to be both the top and bottom of the type lattice. The null value also looses its special status.

4.1 Programming with Concrete Types

The purpose of concrete types is 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. To illustrate the discussion consider the, admittedly simple, example of Figure 2(a). It shows classes Point and PtList with no type annotations and is, as matter of fact, also valid in TypeScript. Programmers may, without any loss of flexibility, choose to document their expectations about the argument of functions and properties of these classes. Figure 2(b) shows code with optional type annotations. In StrongScript, a class declaration introduces a new type name and a constructor for objects. Classes can extend one another, forming a single inheritance hierarchy and inducing a subtype relation. Classes can also implement interfaces; this is not strictly required as interface subtyping is structural, but often helpful to keep declarations in sync. The benefit of optional types is that the following code will be flagged as incorrect:

```
var p: Point = new Point(1,"o") //Err: 2nd arg
var q = p.distance(p) //Err: Method undefined
```

In the first case the type of one of the arguments is not consistent with the declaration of the method, and in second case the name of the method is wrong. Consider further:

```
var p: !Point = new Point(1,2)
var q:  Point = p
var s:  Point = <Point> { x=10; val=3 }
var r:  any   =  q
var t:  any   = { x=3; y=4 }
```

Here we show that an instance of Point can be assigned to a concretely typed variable (p); that concrete types can flow into optional types (q), but cast are required when trying to assign an arbitrary object to an optionally typed variable (s); values can flow into variables of the dynamic type any without casts (r and t).

Finally, a developer may choose to harden his abstractions by adding concrete types as shown in Figure 2(c). For class Point, properties x and y are required to be numbers.[4] Class PtList is similarly hardened. In order to retain compatibility with hypothetical clients of these classes, the developer may choose to leave arguments to dist and add optionally typed. Thus the following code will execute correctly:

```
var p = new Point(4,2)
var q = p.dist( <Point> { x=4;y=2 } )
```

The argument to dist is of course not a Point, but it is an object literal that behaves like a point as far as the method is concerned.[5] The cast <Point> is unchecked, as Point is not a concrete type, but is

[3] Nominal subtyping is easier to compile efficiently as the memory layout of parent classes is a prefix of child classes, code to access properties is fast. Similarly, subtype tests can be highly optimized.

[4] The keyword public in the constructor causes the properties to be implicitly added to the class and initialized with the values of the constructor's arguments.

[5] One could argue that the programmer could have used a structural type to document the fact that dist only requires an object with properties x and y. While true, this would be cumbersome as the number of types would multiply.

required by the type system. The ability to have fine grained control over typing guarantees is one of the main benefits of StrongScript.

4.2 The Type System

In StrongScript, subtyping is nominal for classes and structural for interfaces. Thus if class C extends class D, we have !C <: !D . A concrete type is considered a subtype of the corresponding optional type, thus we have !C <: C . The order on optional type mirrors concrete types: !C <: !D implies C <: D . any is a type isolate with no super or subtype. Subtyping for interfaces follows [1] and TypeScript, with one exception, namely an interface is not allowed to extend a class.

The type system allows arbitrary casts, relaxing the TypeScript stricter rule. However because of the distinction between optional types, concrete types and any, each form of cast is different. Casts to any are allowed and unchecked, and similarly for casts to optional types. As a result, optional types have no soundness guarantees. Downcasts and casts to concrete types are checked, using normal runtime type information and JavaScript's instanceof mechanism.

Several TypeScript dynamic features are rewritten as implicit casts in StrongScript, to keep the syntax of the two languages in synch. 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. In the case that these are casts from any or optional types to concrete types, they are checked exactly like an explicit cast.

In addition, to support unsafe covariant subtyping as in TypeScript, 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, using a variation of the Point and CPoint example with a concrete type for the argument of dist:

```
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 ... }
}
```

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

When functions are called from a concrete context, no runtime type checking of their arguments are necessary. However, when they are called from a non-concrete context, whether optional or any, soundness is not guaranteed, and so their types must be checked at runtime. For instance, in the following three calls to dist, only the first does not check its argument:

```
var p : !Point = new Point(0, 0);
var op : Point = p;
var ap : any = op;
p.dist(p);
op.dist(p);
ap.dist(p);
```

Departing form 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. This does, however, create an incompatibility with TypeScript code which uses a feature we shall call "method stripping": it is possible to remove a method from the context of its object, and by using the JavaScript 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, the stripped method will include checks that cause the call to fail with a dynamic type error.

4.3 Backwards compatibility

JavaScript allows a range of highly dynamic features. StrongScript does not prevent any of these features from being used; however, because their type behavior is so unpredictable, they type to any. As JavaScript objects are maps of string field names to values, it is possible to access members using a string generated at runtime. For instance, the code x[y] will access the member of x named by the string value of y, coercing it to a string if necessary. Because the value of y cannot be predicted, the type of x[y] is always any. Additionally, assignments to x[y] may fail at runtime, if the member happens to have a concrete type and the assigned value is not a subtype. Similarly, eval takes any string, possibly generated at runtime, and executes it as code. StrongScript provides no special extensions to eval, and thus eval code is interpreted as JavaScript, not StrongScript. This isn't 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 at any time, and fields may be removed. An object's StrongScript type must be correct insofar as all fields and methods supported by the StrongScript type must be present in the JavaScript type, but fields and methods *not* present in the StrongScript 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 from objects of StrongScript classes. It is even possible to dynamically add new methods to classes, by updating an object prototype. None of this affects the correctness of StrongScript's types, and access to one of these fields or methods in a value not typed any will result in a static type error.

4.4 Discussion

Unlike Typed Scheme, StrongScript does not support blame. While the prototype implements an optional blame tracking mode, we do not recommend it for production as it incurs performance overheads. Wrappers require, for instance, specialized field access

code. We can envision blame tracking as a command line switch like assertion checking.

The switch to nominal subtyping is somewhat controversial. But our practical experience is that structural subtyping is rather brittle. This because 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. Although the type system is sound, the simpler check introduces an asymmetry between optional and dynamic types at run-time which Thiemann exploited to prove that Thorn is not trace-preserving.

5. Formal properties

We formalize StrongScript as an extension of the core language λ_{JS} of [14]; in particular we extend 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 will not include rules needed for structural subtyping of interface types; these rules would, assumedly, follow [1] but would add 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 generics. Lastly, to simplify the formalization we ignore references; our semantics would preserve the run-time 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 by any. The function type $t_1 .. t_n \to t$ denotes explicitly typed functions, while the type undefined is the type of the value `undefined`. The syntax of the language make easy to disambiguates class names from optional type annotations.

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

A program consists of a collection of class definitions plus an expression to be evaluated. A class definition:

$$\texttt{class}\ C\ \texttt{extends}\ D\{s_1{:}t_1 .. s_k{:}t_k; md_1 .. md_n\}$$

introduces a class named C with superclass D. The class has fields $f_1..f_k$ of types $t_1..t_k$ and methods $md_1..md_n$, where each method is defined by its name m, its signature, and the expression e it evaluates:

$$m(x_1{:}t_1 .. x_k{:}t_k)\{\ \texttt{return}\ e{:}t\}$$

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 will be ranged over by s.

Expressions are inherited from λ_{JS} with some modifications:

e	::=	expression
	x	variable
	$\{s_1{:}e_1 .. s_n{:}e_n \mid t\}$	object
	$e_{1\,\langle t\rangle}[e_2]$	field access
	$e_1[e_2] = e_3$	field update
	$\texttt{delete}\ e_1[e_2]$	field delete
	$\texttt{new}\ C(e_1 .. e_n)$	instance of class
	$\texttt{let}\ (x{:}t = e_1)\ e_2$	let
	$\texttt{func}\,(x_1{:}t_1 .. x_n{:}t_n)\{\ \texttt{return}\ e{:}t\}$	function
	$e(e_1 .. e_n)$	application
	$\langle t\rangle e$	cast

Function abstractions and let binding are explicitly typed, expressions can be casted to arbitrary types, and the `new` $C(e_1 .. e_n)$ expression creates a new instance of class C. More interestingly, objects, denoted $\{\ s{:}e .. \mid t\ \}$, 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 enables preserving the class-based object abstraction at run-time, as discussed below. Additionally, field access (and, in turn, method invocation) is annotated with the static type t of the callee e_1: as discussed later this is used to choose the correct dispatcher or getter when executing method calls and field accesses. These annotation can be added via a simple elaboration pass on the core language performed by the type-checker.

Run-time abstractions Two worlds coexist in a StrongScript run-time: 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. A quick look at the reduction rules confirms that on objects of type any it is indeed possible to create and delete fields, and accessing or updating a field always succeeds.

In our design, objects which are instances of classes benefit from static-typing guarantees; for instance, run-time type-checking of arguments on method invocation is not needed as the type of the arguments has already been checked statically. For this, the runtime must enforce the protection of the class abstraction: in objects which are instances of classes, all fields and methods specified in the class interface must always be defined and point to values of the expected type. To understand how StrongScript guarantees this, 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 (again, the conformance of the method bodies with their interfaces is checked when type-checking 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:

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

$$\frac{\Gamma \vdash e : \texttt{any} \quad \Gamma \vdash e' : t' \quad \Gamma \vdash e_1 : t_1 \;\; .. \;\; \Gamma \vdash e_n : t_n}{\Gamma \vdash e_{\langle \texttt{any}\rangle}[e'](e_1 .. e_n) : \texttt{any}}$$

Although the class pattern traditionally implements method invocation via a combination of field lookup (to recover the corresponding function stored in the prototype object) and function application (to pass the actual arguments), our semantics uses ad-hoc reduction rules for method invocation. These are needed to correctly type-

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

$$\frac{t <: t' \quad t'_1 <: t_1 \quad .. \quad t'_n <: t_n}{t_1\,..\,t_n \to t <: t'_1\,..\,t'_n \to t'}\ [\textsc{SFunc}] \qquad \frac{}{!C <: C}\ [\textsc{SOptInj}] \qquad \frac{!C <: !D}{C <: D}\ [\textsc{SOptCov}]$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ [\textsc{TVar}] \qquad \frac{\Gamma \vdash e : t_1 \quad t_1 <: t_2}{\Gamma \vdash e : t_2}\ [\textsc{TSub}] \qquad \frac{\Gamma \vdash e : t_1}{\Gamma \vdash \langle t_2 \rangle e : t_2}\ [\textsc{TCast}]$$

$$\frac{}{\Gamma \vdash \mathsf{undefined} : \mathsf{undefined}}\ [\textsc{TUndefined}] \qquad \frac{}{\Gamma \vdash \{..\mid t\} : t}\ [\textsc{TObj}] \qquad \frac{\Gamma \vdash e_1 : \mathsf{any} \quad \Gamma \vdash e_2 : t}{\Gamma \vdash \texttt{delete}\ e_1[e_2] : \mathsf{any}}\ [\textsc{TDelete}]$$

$$\frac{t = !C \vee C \quad \Gamma \vdash e : t}{\Gamma \vdash e_{\langle t \rangle}[s] : C[s]}\ [\textsc{TGet}] \qquad \frac{\Gamma \vdash e_1 : \mathsf{any} \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_{1\,\langle \mathsf{any} \rangle}[e_2] : \mathsf{any}}\ [\textsc{TGetAny}]$$

$$\frac{\Gamma \vdash e_1 : t \quad t = !C \vee C \quad \mathsf{not\_function\_type}(C[s]) \quad \Gamma \vdash e_2 : C[s]}{\Gamma \vdash e_1[s] = e_2 : t}\ [\textsc{TUpdate}]$$

$$\frac{\Gamma \vdash e_1 : \mathsf{any} \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash e_3 : t_3}{\Gamma \vdash e_1[e_2] = e_3 : \mathsf{any}}\ [\textsc{TUpdateAny}]$$

$$\frac{\Gamma \vdash e_1 : t \quad x{:}t, \Gamma \vdash e_2 : t'}{\Gamma \vdash \texttt{let}\ (x{:}t = e_1)\ e_2 : t'}\ [\textsc{TLet}] \qquad \frac{\mathsf{fields}\,(C) = s_1{:}t_1\,..\,s_n{:}t_n \quad \Gamma \vdash e_1 : t_1 \quad .. \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash \texttt{new}\ C(e_1\,..\,e_n) : !C}\ [\textsc{TNew}]$$

$$\frac{\Gamma \vdash e : t_1\,..\,t_n \to t \quad \Gamma \vdash e_1 : t_1 \quad .. \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash e(e_1\,..\,e_n) : t}\ [\textsc{TApp}] \qquad \frac{\Gamma \vdash e : \mathsf{any} \quad \Gamma \vdash e_1 : t_1 \quad .. \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash e(e_1\,..\,e_n) : \mathsf{any}}\ [\textsc{TAppAny}]$$

$$\frac{x_1 : t_1.., \Gamma \vdash e : t}{\Gamma \vdash \texttt{func}(x_1{:}t_1..)\{\texttt{return}\ e : t\} : t_1.. \to t}\ [\textsc{TFunc}]$$

$$\frac{\forall i.\ t_i \neq \mathsf{undefined} \wedge t_i \neq t'_1..t'_{n'} \to t' \quad \forall i.\ \vdash md_i \quad (s_1..) \cap \mathsf{fields}(D) = \emptyset \wedge (md_1..) \cap \mathsf{methods}(D) = \emptyset}{\vdash \texttt{class}\ C\ \texttt{extends}\ D\ \{\ s_1{:}t_1..;\ md_1..\ \}}\ [\textsc{TClass}]$$

$$\frac{x_1 : t_1.. \vdash e : t}{\vdash m(x_1 : t_1..)\{\texttt{return}\ e : t\}}\ [\textsc{TMethod}]$$

Figure 3. The type system

check at run-time the actual parameters and the return value. The critical pair between rules for field access and method invocation should always be solved in favor of the latter (similarly for reductions under an evaluation context).

The static view of the object controls the amount of type-checking that must be performed at run-time. For this, field lookup $e_{\langle t \rangle}[e']$ and method invocation $e_{\langle t \rangle}[e'](e_1\,..\,e_n)$ are tagged at run-time with the static type t of e, as enforced by rules TGET and TGETANY or by the derived rules above. The absence of implicit subsumption to any guarantees that the tag is correct.

Suppose that a class C defines the method m :

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

where the class Num implements integers. Let $v =$ $\texttt{func}\,(x{:}\ !Num)\{\texttt{return}\ x + 1{:}\ !Num\}$. Invoking m in a statically typed context directly passes the arguments to the method body:[6]

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

In a dynamic context method invocation initially typechecks the arguments against the parameter type annotations of the method:

$$\begin{array}{l}(\langle \mathsf{any} \rangle \texttt{new}\ C(\,))_{\langle \mathsf{any} \rangle}[\texttt{"}m\texttt{"}](1) \xrightarrow{\textsc{ENew}}\\ (\langle \mathsf{any} \rangle\{\texttt{"\_\_proto\_\_"}{:}\{\texttt{"}m\texttt{"}{:}v \mid\ !C_{proto}\} \mid\ !C\})_{\langle \mathsf{any} \rangle}[\texttt{"}m\texttt{"}](1)\\ \xrightarrow{\textsc{ECast}}\ \{\texttt{"\_\_proto\_\_"}{:}\{\texttt{"}m\texttt{"}{:}v \mid\ !C_{proto}\} \mid\ !C\}_{\langle \mathsf{any} \rangle}[\texttt{"}m\texttt{"}](1)\\ \xrightarrow{\textsc{EGetProto}}\ \{\texttt{"}m\texttt{"}{:}v \mid\ !C_{proto}\}_{\langle \mathsf{any} \rangle}[\texttt{"}m\texttt{"}](1)\\ \xrightarrow{\textsc{EMethAppCheck}}\ \langle \mathsf{any} \rangle(\texttt{func}\,(x{:}\mathsf{any})\{\texttt{return}\ v(\langle Num \rangle x){:}Num\}(1))\end{array}$$

The expression above dynamically checks that the method argument argument is a Num via a cast and injects the return value back into the dynamic world via a cast to any, thus matching the corresponding static type rule. Contrast this with an invocation at type D for some class D that defines a method m with type $!Num \to t$:

$$\begin{array}{l}\{\texttt{"}m\texttt{"}{:}v \mid\ !C\}_{\langle D \rangle}[\texttt{"}m\texttt{"}](1)\\ \xrightarrow{\textsc{EMethAppCheck}}\ \langle t \rangle(\texttt{func}\,(x{:}\mathsf{any})\{\texttt{return}\ v(\langle\ !Num \rangle x){:}\ !Num\}(1))\end{array}$$

In this case rule EMETHAPPCHECK not only typechecks the actual arguments (as the caller can be 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 and reduction 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.

[6] For simplicity we ignore the *this* argument. In reality the desugarer would have rewritten the class definition as

$$\texttt{class}\ C\{m(this{:}\ !C, x{:}Num)\{\texttt{return}\ x + 1{:}Num\}\}$$

and the method invocation as $\texttt{let}\ (o{:}\ !C = \texttt{new}\ C(\,))\ o_{\langle\ !C \rangle}[\texttt{"}m\texttt{"}](o, 1)$.

$$\frac{v = \texttt{func}(x_1{:}t_1..)\{\texttt{return}\ e : t'''\} \quad (t' = \mathsf{any} \wedge t'' = \mathsf{any}) \vee (t' = C \wedge C[s] = t_1'' .. t_n'' \to t'')}{\{s{:}v .. \mid t\}_{\langle t'\rangle}[s](v_1..) \longrightarrow \langle t''\rangle(\texttt{func}(x_1{:}\mathsf{any}..)\{\texttt{return}\ v(\langle t_1\rangle x_1..) : t'''\}(v_1..))}\ [\textsc{EMethAppCheck}]$$

$$\frac{!D <: !C}{\{s{:}v .. \mid\ !D\}_{\langle\ !C\rangle}[s](v_1..) \longrightarrow v(v_1..)}\ [\textsc{EMethAppNoCheck}]$$

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

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

$$\frac{s \notin \{\ s...\}}{\{\ \texttt{"__proto__"}{:}v, s{:}v .. \mid t\ \}_{\langle t'\rangle}[s] \longrightarrow v_{\langle t'\rangle}[s]}\ [\textsc{EGetProto}]$$

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

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

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

$$\frac{s' \notin \{\ s...\} \quad \texttt{"__proto__"} \notin \{\ s...\}}{\{\ s{:}v .. \mid t\ \}_{\langle t'\rangle}[s'] \longrightarrow \texttt{undefined}}\ [\textsc{EGetNotFound}]$$

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

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

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

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

$$\frac{!D <: !C}{\langle\ !C\rangle\{.. \mid\ !D\} \longrightarrow \{.. \mid\ !D\}}\ [\textsc{ECast}]$$

$$\frac{t = \mathsf{any} \vee C}{\langle t\rangle\{.. \mid t'\} \longrightarrow \{.. \mid t'\}}\ [\textsc{ECastAny}]$$

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

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

$$\frac{}{\texttt{new}\ C\ (\overline{v}\,) \longrightarrow \{\ \mathsf{gfields}\ C\ (v_1..);\ \mathsf{gmethods}\ C\ |\ C\}}\ [\textsc{ENew}]$$

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

$\mathsf{gfields}\ C\ (v_1..v_n\ v'..) \triangleq s_1{:}v_1..s_k{:}v_k;$ *fields* $D\ (v'..)$

$\mathsf{gmth}\ (m(x_1 : t_1..)\{\texttt{return}\ e : t\}) \triangleq \texttt{"}m\texttt{"} : \texttt{func}(x_1{:}t_1..)\{\texttt{return}\ e : t\}$

$\mathsf{gmethods}\ C \triangleq \texttt{"__proto__"} = \{\ \mathsf{gmth}\ md\ ..; \mathsf{gmethods}\ D \mid C_{proto}\ \}$

Figure 4. The dynamic semantics

At run-time, the reduction semantics highlights instead that like-typed objects are treated as dynamic objects except for the treatment of the return values. This is captured by 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 $\texttt{class}\ C\{\texttt{"}f\texttt{"}{:}Num\}$ and compare the typing judgments below:

$$\{.. \mid t\}_{\langle \mathsf{any}\rangle}[\texttt{"}f\texttt{"}] : \mathsf{any} \qquad \{.. \mid t\}_{\langle C\rangle}[\texttt{"}f\texttt{"}] : Num$$

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 number (which is enforced at run-time by the cast inserted around the return value by rule EGETOPT).

Formalization. Once the run-time invariants are understood, the static and dynamic semantics of StrongScript 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:

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

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 run-time error.

The "safety" theorem below states that a well-typed expression can get stuck only on a down-cast (as in Java) or on evaluating 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 $e' = E[\langle\ !C\rangle e'']$ and*

$\Gamma \vdash e'' : t$ *with* $t \not<: !C$, *or* $e' = E[\{.. \mid t\}_{\langle t' \rangle}[v]]$ *and* $t' =$ any *or* $t' = C$, *or* $e' =$ undefined.

The proof of this theorem relies on two lemmas, the "preservation" lemma states that typing (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 run-time errors (apart from those occurring on down-casts): 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 down-casts to like types), benefit from the same execution guarantee: static type-checking is strong enough to prevent run-time 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 run-time behavior; more precisely, if the type-checker does not complain about the optional-type annotation, then the run-time 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* $\Gamma \vdash e :$ any. *Let* v *be a value such that* $e \longrightarrow^* v$. *Let* e' *be* e *in which some type annotations have been replaced by like 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 "types 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 does not hold in the original TypeScript semantics, and crucially exploits the fact that all source of unsoundness in our system are identified with explicit cast to optional types (or to any).

THEOREM 3 (Optional types erasure). *Let* e *be a well-typed cast-free expression where all type annotations are of the form* C *or* $!C$. *Suppose* e *reduces to* 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* v.

6. Evaluating StrongScript

The implementation of StrongScript consists of two components: An extended implementation of TypeScript 0.9.1, and a JavaScript engine, derived from Oracle's TruffleJS [30], specialized to optimize StrongScript code. StrongScript compiles to portable JavaScript, so generated 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:

1. Support for concrete types.
2. Dynamic contracts at explicit downcasts.
3. Checked downcasts where TypeScript does so implicitly and unsoundly, including covariant subtyping.
4. Function code suitable for both typed and untyped invocation, with dynamic contracts at untyped invocation.
5. (*Optional*) Blame-tracking wrappers for structural types (interfaces), which cannot be checked eagerly.
6. (*In TruffleJS*) Intrinsics which allow check-free property access in concrete types.

6.1 Implementation

Unlike TypeScript, StrongScript uses nominal typing for classes, in both the optional and concrete settings. This makes optional and concrete types compatible, as well as allowing simple, eager type checks, using JavaScript's builtin instanceof mechanism. While TypeScript erases its types, StrongScript keeps nominal runtime type information. By judicious use of this simple type-checking strategy, the implementation assures that concrete types are always used soundly. StrongScript includes a small (200-line) library of JavaScript functions necessary to implement sound type checking. These functions are implemented using ECMAScript 5 features which prevent them from being replaced or accidentally circumvented.

Differences with TypeScript. 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. In StrongScript, this is typed as concrete instead of any. This affects only methods which are stripped of their context. 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.

Concrete types. Concrete type simply require the addition of the concrete type constructor (!) and the concrete typing rules: !C <: C and !C <: !D implies C <: D. The remainder of the complexity is described below.

Casts. The TypeScript language allows unsafe casts with no runtime penalty and no guarantee of correctness. In StrongScript concrete types are dynamically checked. These dynamic contracts are inserted wherever unsafe downcasts occur, whether explicit or implicit. This is accomplished by the implementation of the $$check function, which asserts that a value is of a specified type. For instance, the following StrongScript code:

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

generates the following JavaScript code:

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

The check function is simple and generic, and does not require a per-class type checker.

For compatibility with TypeScript's typing rules, several forms of unsafe, implicit casts are allowed. 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 StrongScript 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 is of course certain to fail at runtime if B is not a supertype of A. Were this code to be rewritten with unsafe as type B, the cast would imply no check, and the code would succeed at runtime. If the cast to any were excluded, this example would be rejected by the type checker.

Additionally, to allow the syntax of unsafe covariant overloading as in TypeScript, covariant overloading is implemented as unsafe downcasting. For instance, the following StrongScript code:

```
class Animal { eat(x : !Animal) {} }
class Cannibal extends Animal {
  eat(x: !Cannibal) {}
}
```

is rewritten as follows:

```
class Animal { eat(x: !Animal) {} }
class Cannibal extends Animal {
  eat($$unchecked$$x: !Animal) {
    var x: !Cannibal = <!Cannibal>
        $$unchecked$$x;
  }
}
```

This allows covariant overloading, and does not affect optionally-typed code, but allows covariantly-overloaded functions to fail at runtime.

Duplicate function code. Typed functions may be called from typed or untyped contexts. In the case that they are declared with only optional types or any, this requires no checks and is not guaranteed sound. Notably, however, no methods of classes fit that description, as this is always concretely typed. A valid option would be to type-check all concretely typed arguments at runtime. While correct, this would be inefficient, requiring unnecessary dynamic checks even when the types are known. Instead, an unchecked function is called when the types are known, and a checked function otherwise. The checked function simply checks its arguments then calls the unchecked function. Calls are redirected 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 the following StrongScript code:

```
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.

Blame tracking. The StrongScript compiler support optional blame tracking to associates errors with the location of the responsible (structural) type cast. StrongScript implements blame tracking by wrapping [12]. Unsafe downcasts to structural types are implemented by wrapping the objects with field getters and setters which validate their types. The wrapper object additionally stores the location where it was created. When one of its type checks fails, both locations are reported. Because wrapping causes substantial runtime overhead, blame tracking is disabled by default. Classes may explicitly implement interfaces. StrongScript stores this implementation relationship in the runtime class description, so even with blame tracking on, no wrappers need to be created for casts from classes to interfaces they explicitly implement.

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. 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);
```

6.2 Empirical Evaluation

StrongScript's output is idiomatic code compatible with any JavaScript interpreter. As such, the performance of fully-static code generated by StrongScript is expected to be no worse than comparable JavaScript. To test this, we measure a selection of benchmarks translated to fully-static StrongScript code against their equivalent compiled by TypeScript, in both cases on `TruffleSS` . Because all types were statically known in our benchmarks, the only difference between versions translated by StrongScript and versions translated by TypeScript are the presence of our intrinsics.

Benchmark selection. There is no major suite of benchmarks implemented in TypeScript, so we opted to translate a selection of benchmarks from various JavaScript suites. The benchmarks were selected from the Programming Language Benchmarks Game[7] and Octane[8] benchmark suites, and translated to StrongScript code. Benchmarks which cannot be rewritten to use classes cannot take advantage of our intrinsics, and so produce identical code whether compiled by StrongScript or TypeScript. For this reason, they are excluded from measurement. Our final benchmarks are `bg-binarytrees`, `bg-nbody`, `octane-deltablue`, `octane-navier-stokes`, and `octane-splay`.

[7] http://benchmarksgame.alioth.debian.org/

[8] https://developers.google.com/octane/

Evaluation technique. For each benchmark, a type-erased and 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 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 from using StrongScript.

Truffle is a highly optimizing, type-specializing compiler. Many of its optimizations are redundant with our own intrinsics, and we expect the relative speedups to reflect this fact. The machine used to run the benchmarks was 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.

6.3 Performance

Figure 5 shows that of our five benchmarks, three showed marked improvements when using StrongScript, and two showed small improvements. None were slower, although the small improvements, 2.1% and 3.4%, are not statistically significant. We were able to improve benchmarks using our type-specialization intrinsics and direct access to fields in instances of classes. `bg-nbody` uses large objects with typed members, and our type-specialized intrinsics allow us to build these objects very efficiently. Truffle has similar optimizations, but they are heuristic and less effective. `octane-deltablue` and `octane-splay` both use subclasses and polymorphism, and our member access intrinsics are not affected by subclass polymorphism, and therefore are reliably faster.

Benchmark	TypeScript runtime	StrongScript runtime	Speedup
`bg-binarytrees`	5750	5627.8	2.1%
`bg-nbody`	898.8	715.1	20.4%
	Ref. speedup	**Ref. speedup**	
`octane-deltablue`	1701	2518.5	32.5%
`octane-navier-stokes`	9170	9492.4	3.4%
`octane-splay`	890.9	1092.2	18.4%

Figure 5. Performance comparison.

Threats to validity. The number of programs available and their nature makes it difficult to generalize form our the above results. At least this points to the potential 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 validate 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.

7. Conclusion

StrongScript is a natural evolution of the TypeScript design. Optional type annotations have proven to be useful in practice despite their lack of run-time guarantees or performance benefits. With a modicum effort from the programmer, StrongScript can provide stronger run-time guarantees and predictable performance while allowing idiomatic JavaScript code. 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 when stronger invariants are 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. Indeed all programs well-typed in versions of TypeScript up-to 0.9.1 – which relied on nominal subtyping – are well-typed StrongScript programs.

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 can't easily be anonymized, it will be released in time for artifact evaluation.

References

[1] Gavin Bierman, Martin Abadi, and Mads Torgersen. Understanding TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2014.

[2] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strnisa, Jan Vitek, and Tobias Wrigstad. Thorn—robust, concurrent, extensible scripting on the JVM. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, October 2009.

[3] Alan H. Borning and Daniel H. H. Ingalls. A type declaration and inference system for Smalltalk. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 133–141, 1982.

[4] Gilad Bracha. The strongtalk type system for Smalltalk. 1996.

[5] Gilad Bracha. Pluggable type systems. *OOPSLA04, Workshop on Revival of Dynamic Languages*, 2004.

[6] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA*, 1993.

[7] Robert Cartwright and Mike Fagan. Soft Typing. In *Conference on Programming language design and implementation (PLDI)*, 1991.

[8] Douglas Crockford. Classical inheritance in JavaScript. `http://www.crockford.com/javascript/inheritance.html`.

[9] Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2001.

[10] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Conference on Functional Programming (ICFP)*, 2002.

[11] Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *European Conference on Object-Oriented Programming (ECOOP)*, 2004.

[12] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. `http://racket-lang.org/tr1/`.

[13] Vladimir Gapeyev, Michael Levin, and Benjamin Pierce. Recursive subtyping revealed. 12(6):511–548, 2002.

[14] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 126–150, 2010.

[15] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3), 2001.

[16] Microsoft. Typescript – language specification version 0.9.1. Technical report, August 2013.

[17] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *European Conference on Object-Oriented Programming (ECOOP)*, 2011.

[18] Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object Oriented Programming (ECOOP)*, 2007.

[19] Jeremy Siek, Michael Vitousek, Andrew Kent, and Jim Baker. Design and evaluation of gradual typing for Python. Technical report, Indiana University, 2014.

[20] Jeremy G. Siek. Gradual Typing for Functional Languages. In *In Scheme and Functional Programming Workshop*, 2006.

[21] Jeremy G. Siek, Michael Vitousek, and Shashank Bharadwaj. Gradual typing for mutable objects. Technical report, Indiana University, 2013.

[22] Norihisa Suzuki. Inferring types in smalltalk. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 187–199, 1981.

[23] The Dart Team. Dart programming language specification – draft version 0.8. Technical report, November 2013.

[24] Sam Tobin-Hochstadt. *Typed Scheme: From Scripts to Programs*. PhD thesis, 2010.

[25] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Dynamic Language Symposium (DLS)*, 2006.

[26] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed Scheme. In *Symposium on Principles of Programming Languages (POPL)*, 2008.

[27] Julien Verlaguet. Hack for HipHop, September 2013. CUFP, 2013, `http://tinyurl.com/lk8fy9q`.

[28] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*, 2009.

[29] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *Symposium on Principles of Programming Languages (POPL)*, pages 377–388, 2010.

[30] Thomas Würthinger, Christian Wimmer, Andreas Wöss, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Symposium on New ideas, new paradigms, and reflections on programming & software (Onwards!)*, 2013.