Types you can count on

Like types for JavaScript

Abstract

Scripting languages support exploratory development by eschewing static type annotations but fall short when small scripts turn into large programs. Academic researchers have proposed type systems that integrate dynamic and static types, providing robust error reports when the contract at the boundary between the two worlds is violated. Safety comes at the price of pervasive runtime checks. On the other hand, practitioners working on massive code bases have come up with type systems that are unsound and that are ignored at runtime, but which have proven useful in practice. This paper proposes a new design for the JavaScript language, combining dynamic types, concrete types and like types to let developers pick the level of guarantee that is appropriate for their code. We have implemented our type system and report on performance and software engineering benefits.

1. Introduction

Scripting languages, such as JavaScript, PHP and Perl, emphasize flexibility and exploratory development; these are key characteristics that make them compelling for rapid software deployment. The absence of type checking allows for execution of any grammatically correct program written in these languages, even partial programs or programs with obvious errors. In many cases, the language will attempt to compensate for programming mistakes by, *e.g.*, coercing data to its expected type or returning undefined value for illegal operations [8].

While permissive semantics shine early on, as software systems mature, developers may be pressed to focus on correctness, then scripting languages fall short. The lack of static typing and the presence of widely used features such as eval [12] limit the scalability of static analysis techniques [10]. Thus the only hope for any degree of assurance is testing. As exhaustive testing of large, dynamic, software systems is not practical, ruling out the presence of type errors remains challenging.

Decades of academic research were invested in attempts to add static typing to scripting languages. In the 1980's, type inference and soft-typing were proposed for Smalltalk and Scheme [2, 5, 15]. Inference based approaches turned out to be brittle as they required non-local analysis and were eventually abandoned. More recently, *gradual types* [14, 17, 21] have been introduced in Racket [18] and prototyped in Smalltalk [1]. In languages with gradual types, the boundary between typed and untyped code is annotated with type declarations that are interpreted as contracts by which the values crossing the boundary must abide. These contracts are enforced by wrappers that check the conformance of values to their declared types. A variable of type T may thus refer to an object of that type or a wrapper around an object that, hopefully, behaves as T. When a contract violation arises, blame can be assigned by the wrapper to the line of code where it was created. Gradual typing has a runtime cost: every wrapper requires allocation and any expression x.f must check if x is a wrapper before accessing the field. Since wrappers can flow anywhere, any expression x.f can yield a contract violation. A possibly worrisome feature of gradual types is that execution traces of a program that does have type errors are not preserved under addition of types. This can happen when overly constraining type annotations are added to a correct program.

As the need for error reporting was felt in industry, developers adopted a "dynamic first" philosophy, which led to the design of *optional type systems* for languages such as Smalltalk, PHP and JavaScript [3, 4, 11, 16, 20]. Optional type systems have two rather surprising properties: they do not affect the program's execution and they are unsound. These are design choices motivated by the need to be backwards compatible and to capture popular programming idioms used in scripting languages. Rather than limiting expressiveness of the language to what can shown to be type safe, this design instead limits the type system. For example, Hack [20], Facebook's optionally typed PHP dialect, has an UNSAFE directive that turns off type checking for the remainder of a block. Unsafe code may merrily violate any type rules. Dart [16] and Typescript [11] also have unsound type systems that are ignored by the underlying JavaScript runtime. In both cases, types are erased by the compiler and plain JavaScript code is emitted instead. It would be reasonable to wonder about the benefits of type systems that can not be used by the compiler to generate better code, and that do not provide programmers any guarantees about the absence of errors. The benefits are pragmatic but real. Discussion with the designers of Hack and TypeScript suggest that seemingly small things such as providing better IDE support for name completion or refactoring are worth the effort. This because they have tangible benefits for developers; name completion is something that helps in every keystroke. Moreover, the fact that not all type errors can be discovered does not decrease the value of finding some of them. As of this writing, close to 20 million lines of PHP code have been refactored to use Hack at Facebook.

We introduce LikeScript, a language that stakes a middle ground between gradual and optional types. LikeScript aims to provide guarantees in statically typed code while supporting popular untyped programming idioms. We target the JavaScript programming language and use TypeScript as a starting point. Our type system has the following annotations: any to denote the dynamic type, T denotes a concrete type, and like T a, so-called, like type. A variable of type T always refers to an instance of T while a variable of type like T can refer to any object. Concrete types give the traditional benefits of absence of errors and support for efficient code. Whereas, like types support untyped idioms. When manipulating like types or any, the compiler emits the same code that would be emitted for normal JavaScript with no overheads. Adding only like types is trace preserving, a correct untype program, will not report a type violation when like type annotations are added to it. Figure 1 summarizes the differences between the type systems.

	any	T	like T	Trace preserve
Gradual	any, W(any)	T, W(any)		○
LikeScript	any	T	any	◐
TypeScript	any	any		●

Figure 1. With gradual types a variable of type T may be a T or a wrapper any. Typescript ignores types at runtime. In LikeScript, a T variable always points to a T, and a like T variable may refer to any object. TypeScript preserves the traces of correct programs, LikeScript preserves them at like types.

2. Types for JavaScript?

JavaScript is a dynamic, object-oriented, prototype-based programming language [7] derived in part from SELF [19]. Objects are simple mappings of strings to values. While it may be possible to infer types for object literals, JavaScript does not have a static type system:

```
pt = {x: 0, y: 0}
```

Fields may be added dynamically, and typing objects statically is thus not computable:

```
pt = {};   pt.x = 0;   if (rand()) pt.y = 0
```

For inheritance, objects have a, single, immutable, *prototype* field which refers to an object in which missing properties are looked up. Any object can play the role of prototype if passed as argument to a call to create:

```
a = {x:"lo"}
b = Object.create(a);  print("Hel"+b.x)
```

Methods are function-valued fields. Calling functions implicitly passes an additional this argument referring to the object from which the method was extracted:

```
pt = {x: 0, y: 0, toString:
      function(){return this.x +","+this.y} }
```

Functions act as constructors when prefixed with the new keyword; thus new Point() creates an empty object with Point.prototype as prototype:

```
function Point(x, y) { this.x = x; this.y = y }
Point.prototype.toString = function() {
  return this.x + "," + this.y }
```

The above is a programming idiom that could be written in a class-based language. To support this style of programming, TypeScript extends JavaScript with classes and interfaces, reminiscent of languages such as Java or C#, and type annotations on variables and functions:

```
class Point {  x: number; y : number;
  constructor(x,y) { this.x = x; this.y = y }
  toString():string {return this.x +","+this.y}
}
```

Note that type annotations were omitted on constructor arguments above, yet the compiler did not complain about this program. No inference is being done here: the compiler simply accepted a program in which an instance of Point can be created with values that do not match their declared field types. In fact, TypeScript erases annotations, and compiles programs to plain JavaScript. For multiple subtyping, TypeScript also supports interfaces:

```
interface Pt { x: number; y : number }
```

TypeScript facilitates the task of creating class hierarchies. While subclassing in JavaScript can be achieved by disciplined use of prototyping, or through libraries such as jQuery[1] and Prototype[2], the TypeScript compiler takes care of the tedious details of inheritance. One can test that a variable holds an instance of Point using instanceof, but there is no such test for Pt; the symbol is undefined at runtime. This is because instanceof is limited to traversing the prototype chain. Interfaces are entirely erased. Subtyping between classes and interfaces is structural:

```
var pt : Pt = new Point(1,2)
var point : Point = pt  // compile error
```

The type checker validates that Point is structurally compatible with Pt, and correctly complains that Pt is not a subtype of Point.

[1] http://jquery.com/

[2] http://prototypejs.org/

Discussion. The TypeScript design philosophy emphasizes simplicity and expressivity. Types are not intended to help with code generation, but rather to be hints to the programming environment; they enable a modicum of type checking with very local guarantees, and they support IDE features such as name completion and semi-automated refactorings. The fact that fundamental theorems such as type preservation do not hold is not seen as a limitation of the approach but rather a necessary tradeoff to preserve backwards compatibility. Another interesting design choice is to only support a small subset of the programming idioms used by JavaScript programmers. Basically, TypeScript retrofits a class-based programming model on top of JavaScript. Of course, JavaScript is also used in highly dynamic ways. Objects may be created by other patterns, objects may be extended arbitrarily, reflection can be used in wild and unpredictable ways [12, 13]. These dynamic behaviors are vital for expressivity, and for this reason, they are allowed with no attempt from TypeScript to provide guarantees.

3. Designing a type system you can count on

One appealing feature of statically typed languages is that they provide guarantees. Simple correctness guarantees such as, in Java, that the expression x.f will succeed if x is not null. Similarly, developers can rely on predictable performance characteristics, in Java x.f is compiled to a couple of machine instructions. On the other hand, in JavaScript, x.f may always return undefined as the property f cannot be guaranteed to (still) exist in x, and, while JavaScript VMs will usually generate fast code for property access, these rely on heuristics that can be fooled by highly dynamic programs. Thus, from the programmer's view point, any change in their code may lead to x.f either failing or starting to exhibit degraded performance.

The TypeScript type system does not provide guarantees as it is unsound. Two other approaches were proposed in the literature: *gradual types* [14, 17] and *like types* [22]. Gradual type systems allow values to flow from dynamically typed code to statically typed code, but add contracts around those values to validate that they conform to their expected types. Consider the following example:

```
function add1( p : Point ) : number {
   return p.getX() + 1
}
var untyped = {getX(): function(){return 0}}
add1( untyped )
```

The argument of the add1 function is wrapped with a contract that ensures that when getX is called it does return a number. In this example, the argument, while untyped, happens to behave in a conforming way, so this program will not report a type error. If a violation is encountered, contracts provide debugging information by reporting the line of code where they were created as well as where the violation happened. This is called *assigning blame* and is useful for developers to track the source of type errors. With gradual types, any expression may report a contract violation. Thus, in add1, developers must assume that p.getX() may fail. To rule this out would require ensuring that all calls are well-typed. This cannot be determined statically, especially in the presence of eval. Gradual types are not sufficient to guarantee that a compiler will generate efficient code as the compiler must contend with dynamic contracts possibly showing up anywhere.

There is one more subtle issue; adding type annotations can cause a correct program to report an error. Consider:

```
function size( x, v ) { x["size"]=v; return x }
x = size( Point(1,1),  42 )
```

If the call site were representative of all uses, one could type it as size(x: Point, v: number). Now, imagine elsewhere in the code:

```
y = size( Person("Joe"),  "XXL" )
```

With the annotated size, variable y will refer to a Person wrapped in a Point contract and a string expected to behave as a number. A contract violation is likely as these are rather different types. We refer to this as *trace preservation*: any execution trace that does not end in an error should not be affected by the addition of type annotations. TypeScript is trivially trace preserving as types are discarded by the compiler. With gradual types, any overly strict typing has the potential to introduce false positives that will only be caught at runtime.

The LikeScript type system differentiates between type any, concrete types, and like types. One can think of it as supporting fully dynamic programming with any, statically typed programming with concrete types, and optionally typed programming with like types. The add1 function could have any of the following typings:

```
add1( p )                {...p.getX()...}
add1( p : like Point )   {...p.getX()...}
add1( p : Point )        {...p.getX()...}
```

The first two cases are identical operationally as they both permit any value to flow into the function and thus p.getX may fail. Both are trace preserving and, with the like type annotation, the compiler is able to check the body of add1 and verify that Point indeed has a getX method. The last case is different: there we use the concrete type Point. Operationally, this means that when add1 executes, the call to getX will not fail. Moreover, the developer can rely on the compiler's emitting fast code for the dispatch. These different typings have implications for client code. Consider a call:

```
add1( x )
```

For the first two typings, the implementation will directly call the function. If add1 is declared with a concrete type and if x is of type any or a like type, then a runtime check will be inserted to verify that the argument is a subtype of Point, otherwise a direct call will be emitted.

Discussion. Like types were initially designed to provide the flexibility of dynamic features to static languages; in this paper we show that they can be used to add assurance to an otherwise highly dynamic language. A design with concrete, dynamic, and like types, retains the simplicity and expressivity of TypeScript while adding some guarantees. To rule out type errors and ensure that the compiler will generate efficient code, developers can use concrete types. But if they are only interested in backwards compatibilty and trace preservation, they can use like types everywhere. It is tempting when defining a type system to define sophisticated types in an effort to precisely capture every popular programming idiom. We contend that this would be intrusive, as it requires that programmers accustomed to dynamic typing learn complex rules and syntax, when most code that is written can be encapsulated by simpler type systems, and that which can't can be left untyped. Our proposal eschews generics, dependent types, union or intersection types (even such obvious types as "nullable"), and type refinement in preference of a simple, classical, nominal class-based type hierarchy.

4. LikeScript: a typed dialect of JavaScript

LikeScript is a backwards compatible extension to JavaScript that builds on TypeScript but gives it different semantics. We focus here on the new features and the departures from TypeScript. For developers, the key novelty is how we mix optional typing with sound typing. One interesting side effect of our design is that we can get rid of interfaces, as they are subsumed by like types.

4.1 Classes

A class declaration introduces a new type name and a constructor for objects. As we compile down to plain JavaScript, classes must be encoded in a way that does not require language extensions and uses only the features of ECMAScript 5. As a running example, we use the following simple linked list:

```
class List {
  next = this;
  constructor(public data) {}
  add(v)  {
    var l  = new List(v)
    l.next = this
    return l
  }
  ...
}
```

This class has two fields. next, declared in the body of the class definition and initialized to this, and data, implicitly defined by the keyword public and initialized by an auto-generated constructor. Classes can extend one another, forming a single inheritance hierarchy.

```
class ColoredList extends List {
  constructor(public color, data){super(data)}
}
```

LikeScript ensures that fields specified by classes cannot be deleted through reflection. This means that objects have a more fixed shape than in plain JavaScript. However, extensions and deletions of fields not specified in the class are still allowed.

4.2 Types

Classes give rise to types with a subtype relationship defined as the closure of extends clauses. Object is the implicit parent of all classes. The type any denotes dynamically typed values. A variable or field of type any can store any value, and casts are implicitly added at assignments and arguments to functions and methods to ease such flows. Casts *from* any must always be explicit. Object forms the top of the class hierarchy, and has the smallest API of any class, supporting only built-in methods such as toString. For any property f and variable x of type any, x.f is allowed (and of type any), whereas if x is of type Object, the type checker will report that the property does not exist. Similarly, if x is a variable of another class, x.f is allowed only if that class defines a field f. Unlike JavaScript, methods are distinguished from fields. So if f is a method with no arguments, then y=x.f is not allowed, but x.f() is. This is to prevent methods from being stripped of their receiver. The type of an object can be checked through JavaScript's instanceof operator. If a value x was created by the constructor of class C or the constructor of a class that extends C, x instanceof C will evaluate to true.

Type annotations can occur on variables, fields, and method declarations. So, a typed version of List could be:

```
class List {
  next : List;
  constructor(public data: DataPoint) {}
  add(v : Object) : List  {
    var l : List  = new List(v)
    ...
    return l
  }
}
```

Type annotations are not required. Types are inferred for variables and field declarations from their initializers, and for functions, from the type they return. This allows for a slightly less verbose definition for add:

```
  add(v : DataPoint) {
    var l  = new List(v)
    ...
    return l
  }
```

When a type cannot be inferred, variables default to any. Of course, these defaults may always be overridden by explicit type declarations.

The type constructor like creates a type like C for every class C. Type like C is a supertype of C. Values of type C may freely flow into variables of type like C, but the opposite requires an explicit cast. Type like C has the same fields and methods as C, but is unchecked. At runtime, any value may be stored in a variable of type like C, and downcasts to like C are never checked. However, as a like-typed value is expected to have the same interface as its concretely-typed counterpart, values obtained by field lookup or method calls from like-typed values *are* checked. For instance, a call x.add(v) with x as a like List will incur a check of the return type of add to ensure that it is a List. Like types are similar to TypeScript's optional types, and so are familiar to programmers of optionally-typed languages such as TypeScript or Dart. For instance, we can make our List example above use like DataPoint:

```
class List {
  next : List;
  constructor(public data: like DataPoint) {}
  add(v : like DataPoint) : like List {
    var l  = new List(v)
    ...
  }
}
```

This allows the List type to locally guarantee that it is using the DataPoint API correctly, and provides information for the IDE, but does not place any restrictions on clients. Any value is allowed to flow into the like DataPoint fields, so annotating data and v with this type does not incur extra checks or allow for failures beyond what would occur with an any annotation. One idiom for programming with like types is to create abstractions that export like types at their interfaces, and internally use concrete types. For instance, the add method above is defined as returning like List rather than the more precise List. The motivation for this is that it makes the List class more flexible; it can serve as an interface for a family of similar implementations.

Class-typed values may flow to any-typed variables, and as such, may be used in ways that contradict the class description. For instance, the following snippet compiles with no errors:

```
var y : any = new List(dataPoint)
y.next = 42
```

However, allowing the assignment to y.next would violate the field's type annotation. Classes therefore protect themselves against untyped clients. All typed fields are hidden behind an accessor that verifies at runtime that the type specification is not violated. However, if the field is accessed through a reference of a concrete type, the accessor is bypassed and the field is accessed directly. Methods check that they are never passed incorrect arguments at runtime. Calls through a concretely typed reference need not peform runtime checks. Thus:

```
class HashMap { add( s : String, v ) { ... } }
var xC : HashMap = ...
var xL : like HashMap = xC
xC.add( 12 , 21 )  // Compile time error
xL.add( 12 , 21 )  // Runtime error
```

The first call to add can be checked statically as xC is of a concrete type. On the other hand, the second call is allowed by the compiler but will fail at runtime as the first argument is not a string.

4.3 Interfaces

TypeScript provides interfaces albeit with slightly unusual semantics. Instead of a nominal subtyping relationship, interfaces give rise to a structural subtype relation. Moreover, due to implementation considerations, TypeScript does not provide runtime structural subtype tests.

Like types already have structural flavor. So, add(v : like DataPoint) will work correctly if passed an argument of any type that behaves as a DataPoint. But the default is not to check client code, so any value can flow into this argument without checks. In LikeScript one could also define a stub class and have it play the role of an interface. So, consider:

```
class IList {
  next :  like IList
  add(v : like DataPoint) : like IList {}
}
```

One can define a variable of type like IList and have List values flow into it, or any other value that happens to have a next field and an add method. Semantically, the only difference between a like IList and a like List is their subtyping rules. like List is a supertype of List by construction, and another class with an identical API to List would not be a subtype. IList is a supertype of List by structural subtyping, so another class with an identical API to List would also be a subtype. We have investigated changing the subtyping relationship of like types to unify like and structural types, but neither option is clearly superior. Our implementation supports a structural option as a tunable flag, but defaults to nominal.

4.4 Blame

Type errors may be distant from the assignment where the erroneous value originated. LikeScript has support for tracking blame [21]. When enabled, casting to a like type wraps the value being cast with a dynamically allocated structure that combines the value itself with the code location where the cast was performed. The fields and values defined by the like type are wrapped and checked by the wrapper, and if a field contains a value of the wrong type, a method returns a value of the wrong type, a requested field does not exist or a requested method does not exist or is not a method, then both the type error and the location of the like-typed cast is reported. If a field or return type is itself a like type, then its value is wrapped and annotated with the same location as the outer wrapper. When a like-typed value is casted—implicitly or explicitly—to a non-like type, such as any or Object, this wrapper is removed and its location information discarded. This wrapping mechanism is, of course, expensive, and blame tracking is thus disabled by default. It is considered a debugging mechanism, and not a fundamental feature of the language: non-like types never require wrapping or blame tracking, as all checking can be performed eagerly. We also propose to support a more fine grained blame tracking mechanism:

```
function add( v :! like DataPoint )
```

The proposed operator :! denotes that the argument to the add function should be wrapped at the call site. Similarly, one could declare a variable or field with a type :! like C and all assignments would be wrapped.

5. Formalization

We formalise LikeScript as an extension of the core language λ_{JS} of [8]; in particular we extend λ_{JS} with a nominal class-based type system à la Featherweight Java [9] and like types.

Syntax Class names are ranged over by $C, D, ...$, the associated like types are denoted by like $C, ...$, and the dynamic type by any.

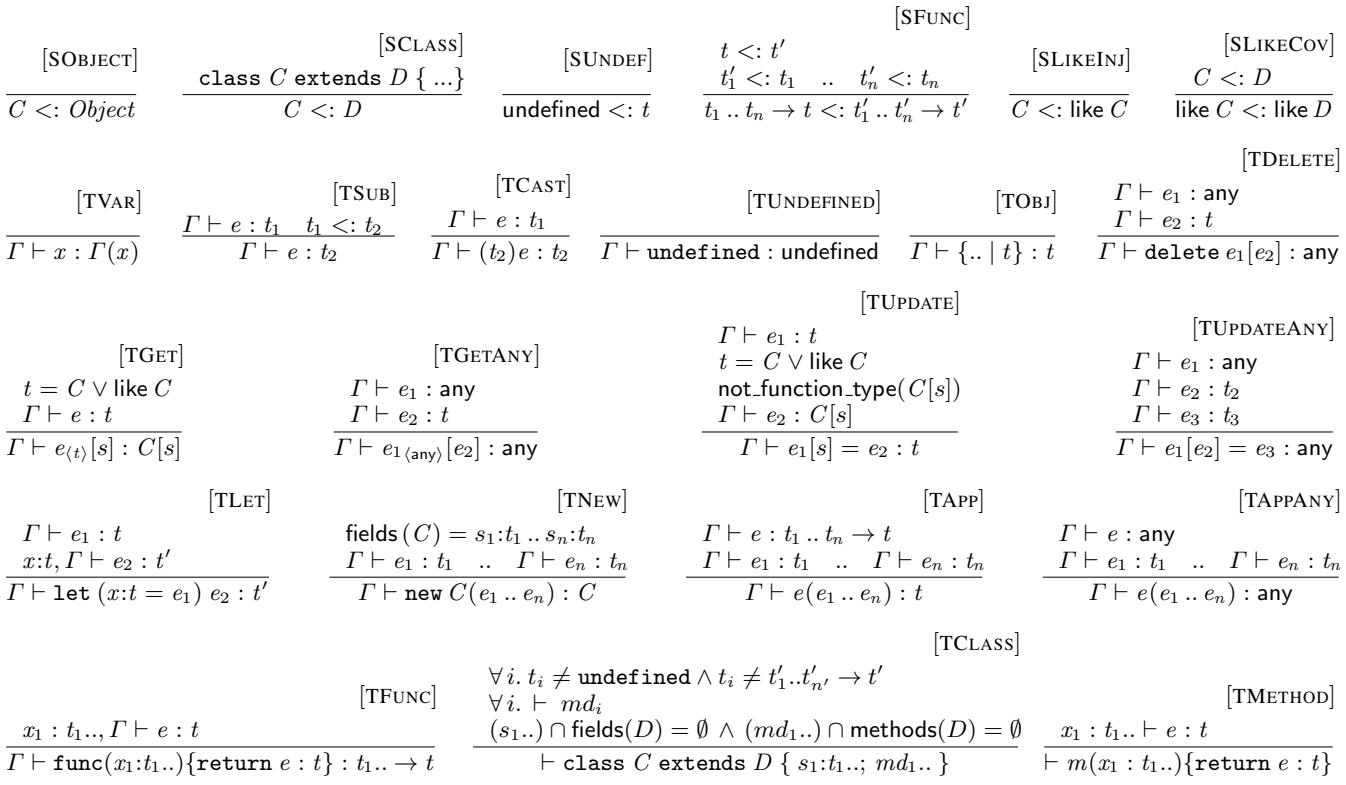

Figure 2. The type system

The function type $t_1\ ..\ t_n \to t$ denotes explicitly typed functions, while the type undefined is the type of the value `undefined`.

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

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
|	$(t)e$	cast

Function abstractions and let binding are explicitly typed, expressions can be casted to arbitrary types, and the $\texttt{new}\ C(e_1\ ..\ e_n)$ 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 e_1: as discussed later this is used to choose the correct dispatcher or getter when executing method calls and field accesses. Annotating field accesses with the static type of the callee can be done via a simple elaboration pass on the core language performed by the type-checker. To simplify the formalization we ignore references; our semantics would preserve the run-time abstractions even in presence of aliasing.

Run-time abstractions Two worlds coexist in a LikeScript 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. 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 succeds.

In our design, objects which are instances of classes must benefit from static-typing guarantees, as in Java; 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 run-time 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

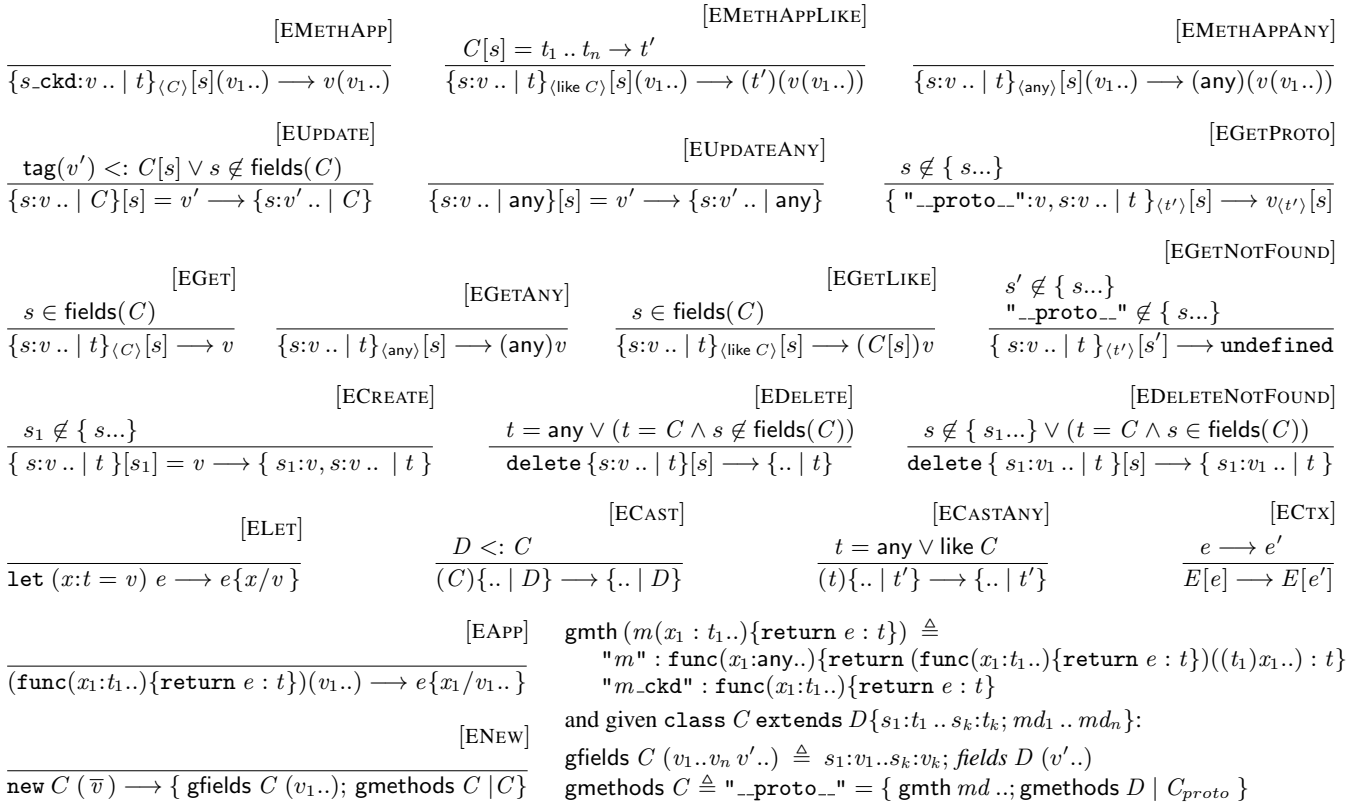

Figure 3. The dynamic semantics

to values of the expected type. To understand how LikeScript guarantees this, it is instructive to follow the life of a class based object. The ENEW rule implements the class pattern [6] commonly used to express inheritance in JavaScript. This creates an object with properly intialized 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, two versions (called `m` and `m_ckd`) are stored in the prototype. The former performs run-time typechecking of the actual arguments via type casts: this is called whenever the method, whose body has been compiled exploiting the type information declared in the interface, is invoked in a dynamic context that might pass arbitrary values (as allowed by the combination of rules TGETANY and TAPPANY). The latter does not type-check the arguments, which are simply passed to the method body, and should only be invoked in contexts where type checking of the arguments has been performed statically (via rules TGET and TAPP). The following type rules for method invocation can thus be derived from the rules for reading a field and applying a function:

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

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 insert run-time type checks around 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 method invocation rules are responsible for picking the correct method definition according to the static view of the object. 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. Invoking m in a statically typed context correctly invokes the `ckd` version of the method:[3]

$$\begin{array}{l}(\texttt{new}\ C())_{\langle C\rangle}[\texttt{"}m\texttt{"}](1) \xrightarrow{\text{ENEW}}\\ \{\texttt{"__proto__"}{:}\{\texttt{"}m\texttt{"}{:}v_1\ \texttt{"}m\_ckd\texttt{"}{:}v_2 \mid C\} \mid C\}_{\langle C\rangle}[\texttt{"}m\texttt{"}](1)\\ \xrightarrow{\text{EGETPROTO}}\ \{\texttt{"}m\texttt{"}{:}v_1\ \texttt{"}m\_ckd\texttt{"}{:}v_2 \mid C\}_{\langle C\rangle}[\texttt{"}m\texttt{"}](1)\\ \xrightarrow{\text{EGETMETHOD}}\ v_2(1)\end{array}$$

while in a dynamic context, the type-checking implementation e_1 is selected:

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

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

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

$$((\mathsf{any})\texttt{new}\ C())_{\langle \mathsf{any}\rangle}[\texttt{"}m\texttt{"}](1) \xrightarrow{\text{ENEW}}$$
$$((\mathsf{any})\{\texttt{"\_\_proto\_\_":}\{\texttt{"}m\texttt{":}v_1\ \texttt{"}m\_ckd\texttt{":}v_2 \mid C\} \mid C\})_{\langle \mathsf{any}\rangle}[\texttt{"}m\texttt{"}](1)$$
$$\xrightarrow{\text{ECAST}} \{\texttt{"\_\_proto\_\_":}\{\texttt{"}m\texttt{":}v_1\ \texttt{"}m\_ckd\texttt{":}v_2 \mid C\} \mid C\}_{\langle \mathsf{any}\rangle}[\texttt{"}m\texttt{"}](1)$$
$$\xrightarrow{\text{EGETPROTO}} \{\texttt{"}m\texttt{":}v_1\ \texttt{"}m\_ckd\texttt{":}v_2 \mid C\}_{\langle \mathsf{any}\rangle}[\texttt{"}m\texttt{"}](1)$$
$$\xrightarrow{\text{EMETHAPPANY}} (\mathsf{any})(e_1(1))$$

Observe that e_1 dynamically checks that its 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 like D for some class D that defines a method m with type $Num \to t$:

$$\{\texttt{"}m\texttt{":}v_1\ \texttt{"}m\_ckd\texttt{":}v_2 \mid C\}_{\langle \mathsf{like}\ D\rangle}[\texttt{"}m\texttt{"}](1) \xrightarrow{\text{EMETHAPPLIKE}} (t)(e_1(1))$$

In this case rule EMETHAPPLIKE invokes the type-checking version of the method (as the callee can be an arbitrary object), but 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 like-typed expressions—that is, expressions whose static type is like C for some class-name C—are treated by the static semantics as objects of type C, thus performing local type-checking.

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 like 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:

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

Field access on an object in a dynamic context invariably returns a value of type any. Instead if the object is accessed as like 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 EGETLIKE).

Formalization Once the run-time invariants are well understood, the static and dynamic semantics of LikeScript 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. Like types are covariant and it is always safe to consider a variable of type C as a variable of type like 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 LikeScript, *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 a like-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 $e' = E[(C)e'']$ and $\Gamma \vdash e'' : t$ with $t \not<: C$, or $e' = E[\{.. \mid t\}_{\langle t'\rangle}[v]]$ and $t' = \mathsf{any}$ or $t' = \mathsf{like}\ C$, or $e' = \texttt{undefined}$.*

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 occuring on down-casts): the concretely typed subset of LikeScript behaves as Featherweight Java (and, in turn, Java) and execution can be optimized along the same lines. Second, like-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 like-typed programs.

The "trace preservation" theorem captures instead the idea that given a dynamic program, it is possible to add like type annotations without breaking its run-time behavior; more precisely, if the type-checker does not complain about the like-type annotation, then the run-time guarantees that the program will have the same behavior of the unannotated version.

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 like type annotations (e.g. $\mathsf{like}\ C$, for C a class with no concrete types in its interface). If $\Gamma \vdash e' : t$ for some t, then $e' \longrightarrow^* v$.*

6. Compilation Strategy

TypeScript compiles to pure JavaScript code, runnable on any ECMAScript-5-capable JavaScript engine, including every modern browser. It integrates with the DOM and, optionally, the popular jQuery library. We extend upon this strategy in LikeScript to create correct, checked output code which is still pure, readable JavaScript requiring no extensions in the engine. All dynamic type checking code is compiled into JavaScript functions, and hidden LikeScript data is encapsulated in hidden properties. A small support library is included in compiled code as well.

6.1 Classes and modules

LikeScript is derived from TypeScript. Contrary to LikeScript, TypeScript's types are entirely erased, with no type-checking code ever generated in the output. As such, after type checking, the only compilation done by the TypeScript compiler is rewriting classes and modules as prototypes and functions, respectively. TypeScript attempts to remain as unintrusive as possible; the output code strongly resembles the input code, and is understandable and debuggable in its own right. Only classes, modules and types are compiled away; other structures remain almost entirely unchanged. For instance, Figure 4 shows the compilation of a simple function, with the only difference between the source and target code being the removal of types.

```
function f(): number {
  return 42;
}
```

```
function f() {
  return 42;
}
```

Figure 4. TypeScript code and resulting JavaScript code.

Classes are compiled to the class pattern, which is commonly used to express classical inheritance in JavaScript. Figure 5 shows a simple TypeScript class and its generated JavaScript code.

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

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

Figure 5. TypeScript code and resulting JavaScript code.

Modules, similarly, are compiled into functions. Consider the following TypeScript code that usings modules to enable the use of multiple versions of the same library in the same program.

```
///<reference path='libA.ts' />
///<reference path='libB.ts' />
module NeedsLibA{
  var lib = libA;
  ...
}
module NeedsLibB{
  var lib = libB;
  ...
}
```

As the following compiled code shows, JavaScript functions reproduce the functionality needed for modules.

```
var NeedsLibA;
(function (NeedsLibA) {
  var lib = libA;
  ...
})(NeedsLibA || (NeedsLibA = {}));

var NeedsLibB;
(function (NeedsLibB) {
  var lib = libB;
  ...
})(NeedsLibB || (NeedsLibB = {}));
```

6.2 Dynamic Type Checks

Types are strictly erased. This has the implication that when a value is downcasted, the cast is unchecked, and as such, it is possible to write unsafe code. This not only violates type safety as dynamic checks remain, but undermines any potential performance improvements from type guarantees. LikeScript inserts dynamic checks at downcasts in the generated code:

- Primitive types are checked with `typeof` operator.
- Classes are checked with the `instanceof` operator.
- If checking is desired, like types must be wrapped and checked lazily.
- Otherwise, by definition, like types require no checks.

Primitive types and classes are relatively inexpensive to check, requiring only a call to an automatically generated checking function. All classes share the same type-checking function, so little extraneous code is created with additional checks.

6.3 Wrapping and Blame

Like types cannot be checked eagerly for a variety of reasons:

- The checks are expensive (proportionally to the size of the data) and can be recursive.
- Due to JavaScript's support of transparent field getters, eagerly checking a field can in fact introduce unwanted side-effects.
- Aliasing is legal and like types may be aliased as `dyn`. If fields change values, eager checks may be wrong.

If blame is desired over like types, LikeScript checks them lazily by wrapping the value to be checked in a new object. This object performs type checks on all field accesses if the field returned is of a concrete type. If the value returned is of a like type, the object wraps the returned value to propagate blame information. This is accomplished by creating transparent getter functions for each field specified in the base type, which simply perform a standard type check on that field in the wrapped object.

The presence of wrappers in the system creates new complications, particularly with equality and identity comparisons. In order to reliably check for the equality of two values, they must both be unwrapped, as they may have been wrapped separately or have different types requiring different wrappers. This is far too expensive to be performed at every comparison in the generated code. Instead, we guarantee that wrappers are *only* present for values which have like types; all points at which like-typed values flow to nominal or any types are guarded to remove any wrappers. Values are explicitly unwrapped in comparison operations when they are of a like type. As such, the expense of like typing is highly localized: If code uses like types and desires blame tracking, it will suffer slowdown for those types, but if it does not, it will be unaffected.

6.4 Type protection for classes

All fields are generated as a hidden field of a reliably-known type and an accessor which allows read and write access to the value mediated by a checking function. The checking functions simply performs a runtime type check to verify that clients cannot violate the type specification. When a field is accessed in an object with a statically-declared type, the generated code bypasses the accessor and directly accesses the field. Field access through untyped references is generated as normal JavaScript code, and hence does not use the hidden fields, but is protected by the accessors.

Similarly, methods are protected by wrappers. Each method in a class is generated as two methods: One to be used by typed clients, one to be used by untyped clients. The version to be used by typed

clients is a direct compilation, requiring no further checking. Typed clients will never provide incorrect arguments. The version to be used by untyped clients checks all of its arguments, guaranteeing type safety.

6.5 Intrinsics

Compiled LikeScript code does not require a specialized JavaScript interpreter. When a standard interpreter is used, there is no performance benefit to using LikeScript; it is impossible through standard JavaScript to communicate our static types to an unmodified interpreter. However, LikeScript additionally supports generating *interpreter intrinsics*, which improve the performance on supporting interpreters by avoiding certain JavaScript costs. Currently the only supporting interpreter is `TruffleLS` , based on `Truffle`'s JavaScript implementation [23].

6.5.1 Direct access

LikeScript classes have a known set of fields, very much like in a conventional, static language. As such, it is possible for LikeScript to prescribe a particular memory layout to instances of classes. `TruffleLS` provides intrinsics to specify this memory layout. Since member access through fully typed expressions is type-safe, the generated JavaScript code can utilize intrinsics that read and write directly to known addresses within the object, rather than caching these locations and incurring the price of a cache check or falling back to expensive hash-table lookup.

6.5.2 Primitive typing

Some types correspond to low-level datatypes. In particular, LikeScript numbers correspond to IEEE-754 double-precision floating point values. `TruffleLS` 's memory layout and access intrinsics allow for fields to be specified as generic or as doubles. Its code generator can then take advantage of this information to specialize for statically-known types.

7. Empirical Evaluation

LikeScript's output is idiomatic JavaScript compatible with any JavaScript interpreter, extended with intrinsics to explicitly specify memory layout on supporting interpreters and checks for static types. As such, the performance of fully-static code generated by LikeScript is expected to be no worse than comparable JavaScript, and sometimes better. To test this, we measure a selection of benchmarks translated to fully-static LikeScript code against their equivalent compiled by TypeScript, in both cases on `TruffleLS` . `TruffleLS` was extended from `TruffleJS`, an in-development JavaScript interpreter; i.e., our intrinsics are a late addition, not a fundamental part of the implementation's performance characteristics. Because all types were statically known in our benchmarks, the only difference between versions translated by LikeScript and versions translated by TypeScript are the presence of our intrinsics.

7.1 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[4] and Octane[5] benchmark suites, and translated to LikeScript code. Benchmarks which cannot be rewritten to use classes cannot take advantage of our intrinsics, and so produce identical JavaScript code whether compiled by LikeScript 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`.

7.2 Evaluation technique

For each benchmark, a type-erased and typed form were compiled, called the "TypeScript" and "LikeScript" 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 LikeScript.

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.

7.3 Performance

Benchmark	TypeScript runtime	LikeScript 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%

Of our five benchmarks, three showed marked improvements when using LikeScript, 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.

8. Conclusions

Dynamic languages provide no type guarantees and are difficult to analyze, making the application of IDEs and tools such as automated refactoring difficult. Nonetheless, they are popular in industry because of the rapid deployability of dynamic code. We have described LikeScript, an extension to JavaScript which provides sound types while also allowing all the underlying dynamism of JavaScript. Our language and implementation bring gradual typing to a dynamic language, while maintaining backwards compatibility with the behaviors that dynamic language programmers know and expect. Through the adaption of like types, we introduce a trace-preserving intermediary step in program evolution, thereby allowing developers to annotate their programs and libraries with assurance that they will not negatively influence client code. LikeScript

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

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

provides gradual typing that is truly applicable to dynamic programs, allowing the programmer to choose any point in the spectrum of dynamic-to-static types as is appropriate for their APIs. With fully concrete types, we show through an augmented implementation of JavaScript that performance benefits are achievable, and even without, assurance is improved by the presence of like types with optional blame tracking.

References

[1] Esteban Allende, Oscar Calla, Johan Fabry, ric Tanter, and Marcus Denker. Gradual typing for smalltalk. *Science of Computer Programming*, (0), 2013.

[2] 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.

[3] Gilad Bracha. Pluggable Type Systems. *OOPSLA04, Workshop on Revival of Dynamic Languages*, 2004.

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

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

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

[7] European Association for Standardizing Information and Communication Systems (ECMA). *ECMA-262: ECMAScript Language Specification*. Fifth edition, December 2009.

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

[9] 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.

[10] Simon Jensen, Anders Mller, and Peter Thiemann. Type analysis for JavaScript. In *Symposium on Static Analysis (SAS)*, volume 5673 of *LNCS*, pages 238–255. Springer, 2009.

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

[12] 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.

[13] Gregor Richards, Sylvain Lesbrene, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the ACM Programming Language Design and Implementation Conference (PLDI)*, June 2010.

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

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

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

[17] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Symposium on Dynamic languages (DLS)*, 2006.

[18] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *POPL*, pages 395–406, 2008.

[19] D. Unger and R. B. Smith. Self: The power of simplicity. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 227–242, 1987.

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

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

[22] 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.

[23] 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 *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204. ACM, 2013.