SJS: A Type System for JavaScript with Fixed Object Layout

Wontae Choi[1(✉)], Satish Chandra[2], George Necula[1], and Koushik Sen[1]

[1] University of California, Berkeley, USA
{wtchoi,necula,ksen}@cs.berkeley.edu
[2] Samsung Research America, Mountain View, USA
schandra@acm.org

Abstract. We propose a static type system for a significant subset of JavaScript, dubbed SJS, with the goal of ensuring that objects have a statically known layout at the allocation time, which in turn can enable an ahead-of-time (AOT) compiler to generate efficient code.The main technical challenge we address is to ensure fixed object layout, while supporting popular language features such as objects with prototype inheritance, structural subtyping, and method updates, with the additional constraint that SJS programs can run on any available standard JavaScript engine, with no deviation from JavaScript's standard operational semantics. The core difficulty arises from the way standard JavaScript semantics implements object attribute update with prototype-based inheritance. To our knowledge, combining a fixed object layout property with prototype inheritance and subtyping has not been achieved previously.

1 Introduction

JavaScript is the most popular programming language for writing client-side web applications. Over the last decade it has become the programming language for the web, and it has been used to write large-scale complex web applications including Gmail, Google docs, Facebook.com, Cloud9 IDE. The popularity of JavaScript is due in part to the fact that JavaScript can run on any platform that supports a modern web browser, and that applications written in JavaScript do not require to go through an installation process.

Given the breadth of applications written nowadays in JavaScript, significant effort has been put into improving JavaScript execution performance. Modern JavaScript engines implement just-in-time (JIT) compilation techniques combined with inline caching, which rely, among other things, on the fact that the layouts of most JavaScript objects do not change often. These optimization heuristics are ineffective when new fields and method are added to an object [16].

A promising alternative to JIT optimization is to use an ahead-of-time (AOT) compiler backed by a static type system. `asm.js` [2] pioneered this direction in the domain of JavaScript. `asm.js` is a statically-typed albeit low-level subset of JavaScript designed to be used as a compiler target, not by a human programmer. One of the lessons learned from `asm.js` is that a promising strategy for

S. Blazy and T. Jensen (Eds.): SAS 2015, LNCS 9291, pp. 181–198, 2015.
DOI: 10.1007/978-3-662-48288-9_11

improving JavaScript is to design a *subset* of JavaScript that has strong type-safety guarantees, so that it can be compiled into efficient code if a compiler is available, and yet, in the absence of a compiler, can also be run *with the same semantics* on any standard JavaScript engine.

Recently, we started to design a new subset of JavaScript [12], dubbed SJS, that can be compiled efficiently by AOT compilers. Unlike `asm.js`, our design includes popular high-level features of JavaScript, such as objects with prototype-based inheritance, structural subtyping, closures, and functions as first-class objects. Like `asm.js`, an important goal is to enable an AOT compiler to translate attribute accesses into direct memory accesses, which requires that objects have statically known layouts.

The first major technical challenge that we face is how to ensure fixed object layout, in the presence of a rich set of high-level language features, while also retaining the operational semantics as given by standard JavaScript engines. The challenge is due in large part to the way standard JavaScript semantics implements object attribute update. JavaScript allows writing to attributes that are unknown at object creation; a new attribute can be inserted into an object simply by writing to it, thereby altering the object's layout. Even if we addressed this issue, e.g. by having a type system disallow writes to unknown attributes, the problem does not go away, due to JavaScript's treatment of prototype inheritance. For read operations, an attribute that cannot be found in the object itself is looked-up recursively in the object's prototype chain. However, when updating an attribute, a new attribute is created in the inheritor object itself, even if the attribute is present in the prototype chain. Essentially, attribute updates do not follow the prototype chain. *This can lead to objects changing their layout even for programs that update attributes that seemingly are already available for reading.* We elaborate in Sect. 2 how this particular semantics interacts with high-level features such as structural subtyping and method updates.

Contributions. In this paper, we propose the underlying type system of SJS, with the following main contributions:

- The type system of SJS supports many attractive and convenient high-level features, such as prototype-based inheritance, closures, structural subtyping, and functions as first-class objects, and ensures that all objects have a statically known attribute layout once initialized. This makes SJS a good candidate for AOT compilation and optimization. As far as we know, this is the first type system ensuring fixed object layout for JavaScript programs with this combination of features.
- The type system of SJS is described as a composition of a standard base type system for records, along with *qualifiers* on object types designed to ensure the fixed object layout. This presentation of the type system highlights the design of the type qualifiers for fixed object layout, which is a novel contribution of this type system.

In this paper we focus on the design of the type system and the type checking algorithm. The paper also includes a brief summary of implementation and

evaluation results. We refer to the companion technical report [12] for the other interesting aspects of the SJS language, such as type inference, typing declarations, type-directed compilation. The full details of the preliminary performance evaluation results and how the top-level language (SJS) integrates the proposed type system into JavaScript are also available in the technical report.

Comparison with Related Designs. A number of efforts are underway to design statically-typed languages for the web where programs could be type-checked statically and maintained easily. TypeScript [4,21] is a typed *super*set of JavaScript designed to simplify development and maintenance. Unlike SJS's type system, TypeScript's type system does not guarantee the fixed object layout property. Therefore, TypeScript programs cannot be compiled into efficient code ahead of time in the way SJS programs can.

As mentioned earlier, `asm.js` [2] is a statically-typed subset of JavaScript aimed at AOT compilation. If a program is written in `asm.js`, it can run efficiently in the Firefox browser with performance comparable with equivalent C programs. A key advantage of `asm.js`, is that being a strict subset of JavaScript, it can run on *any* JavaScript engine, even if the engine is not tuned for `asm.js`, albeit at a regular JavaScript speed. However, since `asm.js` only supports primitive types and operations, the language is not suitable for regular object-oriented programming. SJS intends to offer the same kind of performance advantage, while mostly retaining the expressivity of JavaScript.

RPython [6] is a typed subset of Python designed for AOT compilation to efficient low-level code. Like SJS, RPython fixes object layouts statically in order to enable optimization. However, RPython's type system does not face the same challenges that we address in SJS, because Python does not use prototype-based inheritance. For a language not using a delegation-based prototype inheritance, a traditional notion of object type is sufficient to ensure the fixed object layout property.

2 Design Rationale for the SJS Type System

To illustrate the issues with dynamic object layout in JavaScript as well as our proposed type system, we consider the example program shown in Fig. 1.

```
1: var o1 = { a : 1, f : function (x) { this.a = 2 } }
2: var o2 = { b : 1, __proto__ : o1 }
3: o1.a = 3       //OK
4: o2.a = 2       //BAD
5: o2.f()         //BAD
```

Fig. 1. Example JavaScript program to demonstrate dynamic object layout.

In this example, in line 1 we create an object `o1` with a field `a` and a method `f`. In line 2 we create another object with a field `b` and with the prototype `o1`[1].

[1] Good programming practices of JavaScript discourage the use of non-standard `__proto__` field; however, we use this field to keep our examples concise.

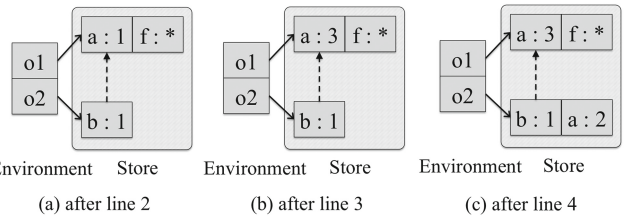

Fig. 2. Program state diagrams for Fig. 1. The dotted line is the prototype reference. The asterisk (*) is a function value

According to JavaScript semantics, the object `o2` will include a reference to the prototype object `o1`, as shown in Fig. 2(a). The value of `o2.a` in this state would be 1, which is found by searching for the nearest definition of the field `a` in the prototype chain for `o2`. Furthermore, since the value of the field `a` is aliased between `o1` and `o2`, the update to `o1.a` from line 3 results in the state shown in Fig. 2(b), and is immediately visible to `o2.a`.

The interesting behavior in this program is in line 4. According to JavaScript semantics, when an inherited field is updated in an object, the field is added to the object itself, and the update happens in the newly added field, resulting in the state shown in Fig. 2(c).

Note that the same effect of object changing its layout would happen at line 5 with the method call `o2.f()`. This method call would first resolve the method `o2.f` to the method `f` inherited from the prototype `o1`, and would then invoke the method with the implicit parameter `this` set to `o2`. We say that `o2` is the *receiver* object for this method invocation.

This example illustrates that in general we cannot assign fixed offsets relative to the location of the object in memory where to find attributes (e.g. `o2.a` refers to different locations at different times.) This poses challenges to efficient execution of JavaScript. A naive implementation would use potentially multiple memory accesses to retrieve the intended attribute value. Modern JavaScript JIT-compilers attempt to optimize attribute lookup computation by caching lookup computation for frequently appearing object layouts at each object operation.[2] Without statically known offset, an AOT compiler would have to either generate inefficient code for attribute lookup, or encode a JIT-compiler-like strategy at runtime.

2.1 Type System for Enforcing Static Object Layout

We propose a type system for a subset of JavaScript to ensure that well-typed programs have the following properties (hereon, we use the term *attribute* to refer to either a field or a method. In standard JavaScript, the term *property* is used instead of the term attribute.):

[2] This representation is called hidden class representation and the caching technique is called inline caching [11]. As noted before, this optimization can fail to apply under certain conditions [16].

- **Prop. 1.** All accesses must be to attributes that have been previously defined (in self or in a prototype.)
- **Prop. 2.** The layout of objects does not change after allocation, both in terms of the set of attributes, and in terms of their types.
- **Prop. 3.** Allow prototype inheritance as a language feature, as implemented in standard JavaScript runtime systems.
- **Prop. 4.** Allow subtyping in assignments, so a subtype instance can be used in contexts in which a base type instance can be used.

In addition, primitive operations do not result in runtime type errors. We believe that these properties are important for program maintainability, as well as for performance on modern JavaScript runtimes. At the same time we believe that it is important to enforce these properties without changes to JavaScript interpreters and just-in-time compilers, so we designed SJS as a subset of JavaScript that preserves standard behavior.

The safety of accessing an attribute (**Prop. 1**) can be enforced with standard static typing techniques that assign fixed static types to variables and attributes. The type of an object must mention the attributes inherited from the prototype chain to allow access to them. However, such a type system would be too forgiving: it would accept the program shown in Fig. 1, violating the fixed layout requirement (**Prop. 2**).

To support fixed layout (**Prop. 2**) and prototype inheritance (**Prop. 3**), while using the standard JavaScript execution model, we need to ensure that: *for any field update statement,* `e1.a = ...`, *the object denoted by* `e1` *must define the field* `a`. We say that an object *owns* the attributes that are defined in the object itself, as opposed to those that are inherited from a prototype. To enforce this property, the types of objects will include the list of attributes guaranteed to be owned by the object, in addition to the list of all attributes guaranteed to be accessible in the object.

Returning to the example from Fig. 1, the type of `o1` will mention that the field `a` and `f` are owned, while the type of `o2` will mention only `b` as owned. Based on these types, the assignment `o2.a = 2` from line 4 will be ill-typed, as we intended.

However, this is not enough to ensure static object layout. Consider replacing line 4 with the method invocation `o2.f()`. This would also attempt to set the field `a` for object `o2`, and should be disallowed. The problem is, however, that the body of the method `f` is type checked in the context of the receiver object `o1`, where it is defined, and in that context the assignment `this.a` is allowed.

There are several options here. One is to require that an object must own all attributes owned by its prototype, such that a function inherited from the prototype can assume that all attributes it may want to update are owned. In the context of our example, this would force us to redefine the fields `a` and `f` in `o2`. This is not a good option because it essentially disables completely the prototype inheritance mechanism and the flexibility it gives.

We therefore decided to allow the set of owned attributes to be different for an object and its prototype. The option that we propose is based on the observation that only a subset of the owned attributes are updated in methods using the receiver syntax, i.e., `this.a`. These are the only attributes that must

be owned by all inheriting objects. We therefore propose to maintain a second set of attribute names for an object type: the subset of the owned attributes that must be owned also by its inheritors. We call these attributes *inheritor-owned* attributes. For the example in Fig. 1, the attribute `a` of `o1` is updated using receiver syntax, i.e., `this.a`, which means that `a` should be an inheritor-owned attribute of `o1`. This means that `a` should be an owned attribute for inheritors, e.g., `o2`. This, in turn, means that we should disallow the definition of `o2` in line 2.

We can summarize the requirements of our type system as follows. Object types are annotated with a set of owned attributes and a set of inheritor-owned attributes, with the following rules:

- **Rule 1**: Owned attributes are defined directly in an object.
- **Rule 2**: Only owned attributes of an object can be updated.
- **Rule 3**: Methods can only update inheritor-owned attributes of their receiver object (using `this.a` notation).
- **Rule 4**: Inheritor-owned attributes are among the owned attributes.
- **Rule 5**: The inheritor-owned attributes of an object include all the inheritor-owned attributes of the prototype object.

Applying these ideas to our example program, we assign the following type to variable `o1`:

$$\texttt{o1} : \{\texttt{a} : \mathit{Int}, \texttt{f} : \mathit{Int} \Rightarrow \mathit{Int}\}^{\mathbf{P}(\{\texttt{a},\texttt{f}\},\{\texttt{a}\})}$$

This type is composed of the base record type and the object-type qualifier written as superscript. The base record type says that the attributes `a` and `f` are all the accessible attributes. The double arrow in the type $\mathit{Int} \Rightarrow \mathit{Int}$ marks that this is the type of a method (i.e., a function that takes an implicit receiver object parameter), and distinguishes the type from $\mathit{Int} \rightarrow \mathit{Int}$, which we reserve for function values; we do not make the receiver type a part of the method type.[3] The object-type qualifier part of `o1` says that the object is precisely typed (marked as **P**, explained later), is guaranteed to own the attributes `a` and `f`, and all of its inheritors must own at least attribute `a`.

In our type system line 2 is ill-typed because it constructs an object that owns only the attribute `b`, yet it inherits from object `o1` that has an inheritor-owned attribute `a` (**Rule 5**). This is reasonable, because if we allow the definition of `o2`, say with type $\{a : \mathit{Int}, b : \mathit{Int}, f : \mathit{Int} \Rightarrow \mathit{Int}\}^{\mathbf{P}(\{b\},\{\})}$, then it would be legal to invoke `o2.f()`, which we know should be illegal because it causes the layout of `o2` to change. To fix this type error we need to ensure that `o2` also owns `a`. Note that the assignment in line 3 (`o1.a = 3`) is well-typed, as it should, because `a` is among the owned fields mentioned in the static type of `o1`.

2.2 Subtyping

Consider again the example in Fig. 1 with the object layouts as shown in Fig. 2(a). The assignment `o1.a = 3` from line 3 is valid, but the assignment `o2.a = 2` from line 4 is not, even though `o2` inherits from its prototype `o1`. This shows

[3] This is to allow comparison of method attribute types in subtyping.

```
6 : var o3 = { a : 11, c : 12,  f : function (x) { this.c = 13 } }
7 : o1 = o3;    //BAD

8 : var o4 = { a : 14, __proto__ : o1 }
9 : o4.f ();    //BAD

10: var o5 = { a : 1, b : 2, f : function (x) { this.a = 2 } }
11: var o6 = { a : 1, b : 3, f : function (x) { this.b = 3 } }
12: o6.f = function (x) { this.b = 4 }  // OK
13: var o7 = if ... then o5 else o6
14: o7.f = function (x) { this.b = 4 }  // BAD
15: console.log(o7.a);                  // OK

16: var o8 = if ... then o1 else o3  // OK
17: o8.f(3);                         // OK
18: o8.c = 2;                        // OK
19: var o9 = { a: 14, __proto__: o8} // BAD
```

Fig. 3. Example JavaScript program (continued from Fig. 1).

that inheritance does not automatically create a subtype relationship when fixed object layout is a concern.

In the spirit of a dynamic language like JavaScript, we propose to use a structural subtyping relationship between types, generated by the structure of the types and not by their prototype relationships.

Consider, for example, a new object `o3` such that the assignment `o1 = o3` is safe. The object `o3` would have to contain the attributes `a` and `f`. Furthermore, `o3` must own all the attributes owned by `o1`, so that it can be used in all the attribute-update operations where `o1` can be used. An example is available in line 6–7 of Fig. 3. The type of `o3` is

$$\texttt{o3} : \{\texttt{a} : Int, \texttt{c} : Int, \texttt{f} : Int \Rightarrow Int\}^{\mathbf{P}(\{\texttt{a},\texttt{c},\texttt{f}\},\{\texttt{c}\})}$$

To support subtyping (**Prop. 4**), the general rule is that an object type A is a subtype of B, if and only if (a) A contains all the attributes of B with the same type (as in the usual width subtyping), and (b) the owned attributes of A include all the owned attributes of B. However, this is still not enough to support fixed layout (**Prop. 2**), in presence of prototype inheritance as implemented in JavaScript (**Prop. 3**), and subtyping (**Prop. 4**).

Challenge: Subtyping and Prototype Inheritance. In our example, after the assignment `o1 = o3` the static type of `o1` suggests that the set of inheritor-owned attributes is {`a`}, while the true inheritor-owned attributes of the runtime object are {`c`}. This suggests that it would be unsafe to use the object `o1` as a prototype in a new object creation, as in the continuation of our example in line 8–9 of Fig. 3. If the object creation in line 8 is well typed, with the type:

$$\texttt{o4} : \{\texttt{a} : Int, \texttt{f} : Int \Rightarrow Int\}^{\mathbf{P}(\{\texttt{a}\},\{\texttt{a}\})}$$

then, when executing line 9 the field `c` would be added to the receiver object `o4`.

One way to get out of this impasse is to restrict the subtype relationship to pay attention also to the inheritor-modified attributes. In particular, to allow the assignment `o1 = o3` followed by a use of `o1` as a prototype, we must ensure

that the static type of o1 includes all the inheritor-owned attributes from the type of o3. This would mean that the inheritor-owned attributes in a supertype must be a superset of the inheritor-owned attributes in the subtype.

However, we show next that this is not enough if we want to allow method updates.

Challenge: Subtyping and Method Update. It is common in JavaScript to change the implementation of a method, especially on prototype objects, e.g., in order to change the behavior of a library. This technique is sometimes called monkey patching.Consider the code fragment in line 10–15 of Fig. 3. In our type system, the types of o5 and o6 can be:

$$\texttt{o5} : \{\texttt{a} : \mathit{Int}, \texttt{b} : \mathit{Int}, \texttt{f} : \mathit{Int} \Rightarrow \mathit{Int}\}^{\mathbf{P}(\{\texttt{a},\texttt{b},\texttt{f}\},\{\texttt{a}\})}$$
$$\texttt{o6} : \{\texttt{a} : \mathit{Int}, \texttt{b} : \mathit{Int}, \texttt{f} : \mathit{Int} \Rightarrow \mathit{Int}\}^{\mathbf{P}(\{\texttt{a},\texttt{b},\texttt{f}\},\{\texttt{b}\})}$$

The method update in line 12 is safe because it updates the method f of o6, with a method that modifies the same set of receiver fields, which are owned by o6 and all objects that may be inheriting from it. This can be verified statically by comparing the receiver attributes that may be changed by the new method (b) with the list of inheritor-owned fields listed in the type of o6.

In this example, subtyping arises in line 13. Notice that the type of o7 must be a supertype of the type of both o5 and o6. The access in line 15 is safe. However, the assignment in line 14 is unsafe, because it may associate with object o5 a method that changes the field b of the receiver object. This is unsafe since b is not listed as inheritor-owned, so the updated method is not safe for inheritance.

This example suggests that one way to ensure soundness of the assignment of o5 to o7 is to ensure that the inheritor-owned attributes in a supertype (e.g., type of o7, which is used for checking statically the safety of method update) must be a subset of the inheritor-owned attributes in the subtype, e.g., type of o5. In this particular case, the inheritor-owned attributes of the static type of o7 must be empty, i.e. a strict subset of that of the static types of o5 and o6. This is exactly opposite of the inclusion direction between the inheritor-owned attributes in a subtype relation proposed in the previous section to handle subtyping and prototype inheritance.

Solution: Subtyping with Approximate Types. We saw that a type system that supports fixed layout (**Prop. 2**) and prototype inheritance (**Prop. 3**) must reject the use of subtyping in line 13. We feel that this would be extremely restrictive, and not fulfill subtyping (**Prop. 4**). Moreover, prototype inheritance, method update, and the inheritor-owned fields, are about inheriting and sharing implementations, while subtyping is about interface compatibility. There are many more occurrences in practice of subtyping in assignments and method calls than there are prototype assignments and method updates.

Therefore, we propose to relax the subtyping relation to make it more flexible and more generally usable, but restrict the contexts where it can be used. In particular, for prototype definition or method update, we only allow the use of objects for which we know statically the dynamic type.

To implement this strategy, we use two kinds of object types. The *precise* object type that we used so far (marked as **P**), which includes a set of all attributes and their types, along with a set of owned attributes, and a set of inheritor-owned attributes. A precise object type means that the static type of the object is the same as the dynamic type, i.e., no subtyping has been used since the object construction. Expressions of precise type can appear in any context where an object is expected.

We also introduce an *approximate* object type, written as $\{Attr\}^{\mathbf{A}(\{Own\})}$, also including a set of attributes and their types, and a set of owned attribute names, but no inheritor-owned attributes. Approximate types allow subtyping, and are only an approximate description of the actual dynamic type of the object. These objects can be used for read/write attribute access and for method invocation, but cannot be used as prototypes or for method updates. Therefore, we do not need to track the inheritor-owned attributes for approximate types.

We can summarize the additional rules in our type system for dealing with subtyping

- **Rule 6**: There is no subtyping relation on precise object types.
- **Rule 7**: An approximate object type is a supertype of the precise object type with the same attributes and the same owned attributes.
- **Rule 8**: An approximate object type A is a subtype of another approximate object type B as long as the subtype A has a superset of the attributes and a superset of the owned attributes of the supertype B (as in standard width subtyping).
- **Rule 9:** Only objects with precise type can be used as prototypes.
- **Rule 10:** Method update can only be performed on objects of precise type, and only when the method writes only inheritor-owned attributes of the object (extension of **Rule 3**).

Returning to our motivating example, both `o1` and `o3` have precise distinct types, which do not allow subtyping, so the assignment `o1 = o3` from line 6 is ill-typed. However, the assignment at line 16 of Fig. 3 will be legal if the static type of `o8` is the following approximate type:

$$\texttt{o8} : \{\texttt{a} : Int, \texttt{c} : Int, \texttt{f} : Int \Rightarrow Int\}^{\mathbf{A}(\{\texttt{a,c,f}\})}$$

Moreover, we can perform attribute lookup and method invocation via `o8` as shown in line 17–18 of Fig. 3, because these operations are allowed on approximate types. However, it would be illegal to use `o8` as prototype, as in line 19 of Fig. 3. This is because an object with approximate type cannot be used as a prototype.

With approximate types, the subtyping assignment at line 13 can be well-typed: by giving the static type of `o7` the approximate type

$$\texttt{o7} : \{\texttt{a} : Int, \texttt{b} : Int, \texttt{f} : Int \Rightarrow Int\}^{\mathbf{A}(\{\texttt{a,b,f}\})}$$

The method update from line 14 will still be ill-typed because method update cannot be applied to an object with approximate type. This shows how the introduction of approximate types supports subtyping in certain contexts, while avoiding the unsoundness that can arise due to interaction of subtyping and prototype inheritance.

We have shown informally a type system that fulfills all of access safety (**Prop. 1**), fixed layout (**Prop. 2**), prototype inheritance (**Prop. 3**), and subtyping (**Prop. 4**), while placing few restrictions. We discuss this type system formally in Sect. 3.

3 A Formal Account of the Type System

This section provides a formal definition of the type system of SJS and a proof of the fixed object layout property. Throughout this section, we use a simplified core language that is designed to capture the essence of the prototype-based object-oriented programming in JavaScript. The language supports mutable objects, prototype inheritance, dynamic method updates, higher-order functions, and local variable bindings. To simplify the presentation, we do not include in the language: functions as objects, constructor functions, accessing undefined variables, and lookup of fields by dynamic names (e.g., `obj[''key'']`). Furthermore, we postpone the introduction of a number of other features to the companion technical report [12]: first-class method functions, recursive data types, and accessing `this` in a non-method function.

3.1 Expression

The syntax definition of the core language expressions is shown in Fig. 4. We are going to use the metavariables e for an expression, n for an integer number, x for a variable identifier, and a for an attribute identifier. A few expression types have type annotations in order to simplify type checking. The expression $\{a_1 : e_1, \ \ldots, \ a_n : e_n\}_T$ defines a new object with attributes $a_1, \ldots, a_n$ initialized with expressions $e_1, \ldots, e_n$, respectively. T is the type of the resulting object. The expression $e_1.a{=}e_2$ updates attribute a of the object e_1 with the value of e_2. The expression $e_1.a(e_2)$ invokes method a of object e_1 with argument e_2. The expression `this` accesses the receiver object. The expression $\{a_1 : e_1, \ \ldots\}_T$ `prototype` e_p creates a new object with prototype e_p. T is the expected type of the resulting object.[4]

3.2 Types and Qualifiers

Figure 4 also defines the types. The novel elements in this type system are the object-type qualifiers (q). If we erase the object-type qualifiers we are left with a standard object type system [5] with few modifications. Object-type qualifiers track the layout information required to constrain object operations in order to guarantee the fixed layout property in the presence of the JavaScript operational semantics.

[4] Please note that deviating from JavaScript (`prototype` expression) is for the clear presentation. The SJS language itself supports the usual prototyping mechanism of JavaScript, which is based on a prototype attribute of constructors. We refer to the companion technical report [12] for more details.

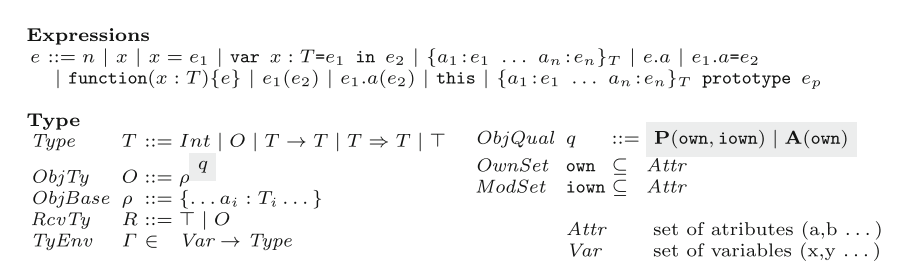

Fig. 4. Syntax: expressions and types. The highlighted items are specific to our object-type qualifiers.

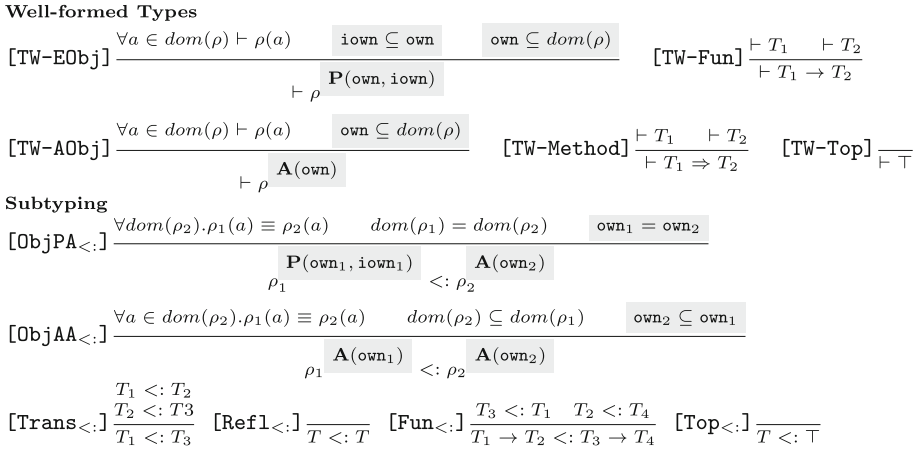

Fig. 5. Well-formed types and subtyping. The highlighted items are specific to our object-type qualifiers.

Types (T) include the integer type (Int), object types (O), function types ($T \to T$), method types ($T \Rightarrow T$), and the top type ($\top$). A receiver type (R) is either the top type, when typing a non-method function, or an object type, when typing a method function. A type environment (Γ) is a map from variables to types. Object types are composed of a base object type (ρ) and an object-type qualifier (q). Object types can have either a precise qualifier ($\mathbf{P}(\texttt{own}, \texttt{iown})$) or an approximate qualifier ($\mathbf{A}(\texttt{own})$). Owned attribute sets (`own`), and inheritor-owned attribute sets (`iown`) are subsets of corresponding objects' attributes.

Operations on Object Types. $dom(\rho)$ denotes all attributes of the base object type ρ. We write $\texttt{own}(q)$ to denote the owned attribute set of the qualifier q We similarly define $\texttt{iown}(q)$ to denote the inheritor-owned attribute set of the qualifier q when q is precise. We are also going to use $\rho(a)$ to denote the type of attribute a in ρ.

Well-Formed Types. Figure 5 defines the rules to check well-formedness of a type, especially for object types. An object type with a precise qualifier is well-formed if all the inheritor-owned attributes are among the owned attributes, all owned attributes are among the attributes, and all attributes have well-formed types. The well-formedness check for an object type with an approximate qualifier is similarly defined without the check for inheritor-owned attributes.

Subtyping and Type Equality. Figure 5 also defines the subtyping relation. There is no subtyping between precise objects. However, precise objects can be relaxed to an approximate object having the same base object type and owned set ($\texttt{[ObjPA}_{<:}\texttt{]}$). This ensures that any read and write operation that is allowed by a precise type is still available after relaxed to an approximate type. Subtyping between approximate objects ($\texttt{[ObjAA}_{<:}\texttt{]}$) is defined as a traditional width-subtyping extended with an additional inclusion check between `own` sets: a subtype should own strictly more than a supertype. This ensures that any read and write operation allowed by a supertype can be safely performed on an object with a subtype.[5] We also have transitivity ($\texttt{[Trans}_{<:}\texttt{]}$), function ($\texttt{[Fun}_{<:}\texttt{]}$). We do not need subtyping among method types because that method types only appears as an attribute type (we will see this in the type system section), and only the equivalence of attributes are checked. Type equivalence ($\equiv$) is a syntactic equivalence check.

3.3 Typing Rules

The static typing rules are defined in Fig. 6. The type system is composed of two kinds of rules: *expression typing judgment* and *attribute-update typing judgment*.

Expression Typing. The expression typing judgment $R, \Gamma \vdash e : T$ means that expression e under receiver type R and type environment Γ has type T.

Variables and Functions. Rules `[T-Var]`, `[T-VarUpd]`, and `[T-LetVar]` handle variable lookup, variable update, and local binding. `[T-This]` applies to the `this` expression when the current receiver type is an object type. `this` cannot be used when the current receiver type is $\top$.

Functions. `[T-Fun]` extends the traditional typed lambda calculus with a receiver type in the context. Since functions, unlike methods, are invoked without a receiver object, the function body is type checked with the receiver type set to the top type ($\top$). As a consequence, accessing the `this` variable within a function is not allowed.

Objects. `[T-Obj]` types an object literal without inheritance. The created object has a well-formed type ρ^q as annotated in the expression. Each attribute of ρ^q should be an owned attribute and should appear in the object literal expression. The safety of initialization expressions and initialization operations are delegated

[5] Allowing depth-subtyping between mutable objects will make the type system unsound. We refer to Abadi and Cardell's work [5] for more details.

Expression Typing

$$\texttt{[T-Var]}\ \frac{\Gamma(x) = T}{R, \Gamma \vdash x : T} \qquad \texttt{[T-VarUpd]}\ \frac{\Gamma(x) = T_1 \qquad R, \Gamma \vdash e : T_2 \qquad T_2 <: T_1}{R, \Gamma \vdash x = e : T_1}$$

$$\texttt{[T-LetVar]}\ \frac{R, \Gamma \vdash e_1 : T_1 \qquad \vdash T \qquad T_1 <: T \qquad R, \Gamma[x \mapsto T] \vdash e_2 : T_2}{R, \Gamma \vdash \texttt{var}\ x : T = e_1\ \texttt{in}\ e_2 : T_2} \qquad \texttt{[T-FCall]}\ \frac{R, \Gamma \vdash e_1 : T_1 \to T_2 \qquad R, \Gamma \vdash e_2 : T_3 \qquad T_3 <: T_1}{R, \Gamma, \vdash e_1(e_2) : T_2}$$

$$\texttt{[T-Fun]}\ \frac{\top, \Gamma[x \mapsto T_1] \vdash e : T_2 \qquad \vdash T_1}{R, \Gamma \vdash \texttt{function}(x : T_1)\{e\} : T_1 \to T_2} \qquad \texttt{[T-This]}\ \frac{}{\rho^q, \Gamma \vdash \texttt{this} : \rho^q}$$

$$\texttt{[T-Attr]}\ \frac{\rho = \{\ldots a : T \ldots\} \qquad R, \Gamma \vdash e : \rho^q \qquad T \neq T_1 \Rightarrow T_2}{R, \Gamma \vdash e.a : T} \qquad \texttt{[T-AttrUpd]}\ \frac{R, \Gamma \vdash_{AU} \rho^q.a{=}e_2 \qquad a \in \texttt{own}(q) \qquad R, \Gamma \vdash e_1 : \rho^q}{R, \Gamma \vdash e_1.a{=}e_2 : \top}$$

$$\texttt{[T-MCall]}\ \frac{R, \Gamma \vdash e_1 : \rho^q \qquad \rho = \{\ldots a : T_1 \Rightarrow T_2 \ldots\} \qquad R, \Gamma \vdash e_2 : T_3 \qquad T_3 <: T_1}{R, \Gamma \vdash e_1.a(e_2) : T_2}$$

$$\texttt{[T-Obj]}\ \frac{\vdash \rho^q \quad dom(\rho) = \{a_1 \ldots a_n\} \quad \forall i \in [1, n].R, \Gamma \vdash_{AU} \rho^q.a_i{=}e_i \quad q = \mathbf{P}(\texttt{own}, \texttt{iown})}{R, \Gamma \vdash \{a_1 : e_1 \ldots a_n : e_n\}_{\rho^q} : \rho^q}$$

$$\texttt{[T-Proto]}\ \frac{\begin{array}{c}\vdash \rho^q \qquad R, \Gamma \vdash e_p : {\rho_p}^{q_p} \qquad dom(\rho) = dom(\rho_p) \cup \{a_1, \ldots, a_n\} \\ \forall i \in [1, n].R, \Gamma \vdash_{AU} \rho^q.a_i{=}e_i \quad \forall a \in dom(\rho_p).\rho(a) \equiv \rho_p(a) \quad \texttt{iown}_p \subseteq \texttt{iown} \\ q = \mathbf{P}(\texttt{own}, \texttt{iown}) \qquad q_p = \mathbf{P}(\texttt{own}_p, \texttt{iown}_p) \qquad \texttt{own} = \{a_1, \ldots, a_n\}\end{array}}{R, \Gamma \vdash \{a_1 : e_1 \ldots a_n : e_n\}_{\rho^q}\ \texttt{prototype}\ e_p : \rho^q}$$

Attribute-Update Typing

$$\texttt{[T-AttrUpdV]}\ \frac{\rho = \{\ldots a : T \ldots\} \qquad T \neq T_1 \Rightarrow T_2 \qquad R, \Gamma \vdash e : T' \qquad T' <: T}{R, \Gamma \vdash_{AU} \rho^q.a{=}e}$$

$$\texttt{[T-AttrUpdM]}\ \frac{\begin{array}{c}O = \rho^q \qquad \rho = \{\ldots a : T_1 \Rightarrow T_2 \ldots\} \qquad {\rho^{q'}}, \Gamma[x \mapsto T_1] \vdash e : T_2 \\ q = \mathbf{P}(\texttt{own}, \texttt{iown}) \qquad q' = \mathbf{A}(\texttt{own}') \qquad \texttt{own}' = \texttt{iown}\end{array}}{R, \Gamma \vdash_{AU} O.a{=}\texttt{function}(x : T_1)\{e\}}$$

Fig. 6. Type system. The highlighted items are specific to object-type qualifiers.

to the *attribute-update typing judgments*, `[T-AttrUpdV]` and `[T-AttrUpdM]` described in the next section. `[T-Attr]` types an attribute read access. The rule restricts the reading of a method attribute. It is well-known that subtyping along with escaping methods can break the soundness of a type system [5]. `[T-MCall]` handles method calls. The rule checks only the parameter type and the return type since the safety of passing the receiver object is already discharged when the method is attached. `[T-AttrUpd]` types an attribute update. The rule requires the target attribute to be owned by the base object type. The determination of the type and type safety of the attribute-update operation is delegated to the attribute-update typing judgments. Note that the attribute-update typing judgment does not provide a type for the assignment result to prevent methods from escaping an object.

Inheritance. `[T-Proto]` types an object literal with inheritance. The rule is basically an extension of `[T-Obj]`, with the following new checks: (1) attributes should be either owned fields of ρ^q or fields inherited from $\rho_p^{q_p}$, (2) the type of an attribute defined in prototype should remain the same in the newly defined object, and (3) inheritor-owned attributes of the newly defined object should

include all the inheritor-owned attributes of the prototype object. The rule also requires $\rho_p^{q_p}$ to be a precise object type. Like in `[T-Obj]`, the type safety of initialization expressions and initialization operations are delegated to the attribute-update typing rules.

Attribute-Update Typing. Attribute updates are handled by a different set of judgment rules. The attribute-update typing judgment $R, \Gamma \vdash_{AU} O.a{=}e$ means that "expression e is well typed under receiver type R (for the current method or function body) and type environment Γ, and the value of e can be safely assigned to attribute a of an object of type O. The judgment has two rules.

Field Update. If a non-method attribute is updated (`[T-AttrUpdV]`), the rule just typechecks the expression e.

Method Update. The method-update rule (`[T-AttrUpdM]`) requires the right-hand side expression to be a function literal[6] and the base object type to be a precise object type (we can only perform method update on objects whose type is known precisely, and in particular whose inheritor-owned set is known). This rule addresses the situations when the method is inherited and the receiver object is some subtype of the receiver type O. The method body is checked with an approximate version of the receiver type O whose owned attributes set is restricted to the inheritor-owned attributes of O. This ensures that the function body can only update the `iown` attributes of the receiver object.

3.4 Properties of the Type System

Theorem *(Fixed Object Layout). A well-typed program never modifies object layouts after object construction.*

Proof. (Sketch) To show this property, we first define an operational semantics of the core language such that any attempt to modify an object layout will result in the execution getting stuck. Then we show the usual type soundness property, i.e., a well-typed program never gets stuck. The fixed object layout property is a corollary of the soundness theorem. The full version of the proof and necessary definitions, such as operational semantics and value typing, are available in the companion technical report [12] (Section B).

4 Summary of Implementation and Evaluation

We have implemented a proof-of-concept type checker and compiler for SJS to evaluate the language. The SJS prototype supports the core type system described in this paper, along with typed arrays, hash tables, integer and floating point numbers, first-class methods, and recursively-defined object types.

[6] This syntactic restriction is posed to keep the presentation simple. The companion technical report [12] (Section A.2) extends the type system to remove this restriction.

We evaluate the usability of the language and the feasibility of type-based compilation. This section provides a short summary of the evaluation. The full details are in the companion technical report [12]. The programs used in this section can be found at http://goo.gl/nBtgXj.

Usability. We considered two programs from the *octane* benchmark suite [3] and two webapps from *01.org* [1] to evaluate the usability of the type system. Programs are moderate-sized (about 500 to 2000 lines of code) and use objects extensively. We managed to typecheck all four programs, after commenting out small portions of code handling Ajax communication, because we do not have enough contextual information to decide the types for this part.

SJS requires programmers to provide type annotations to infer the base type (type qualifiers are inferred without any user interaction). For the benchmarks, one type annotation is required per 8.34 lines of code. The majority of the annotations (86.5 %) are for function parameters, since SJS requires every function parameter to be annotated. The rest of the annotations are for local variables, **this** variables, attributes, returns, and some assignments when there is an ambiguity that the type inference engine cannot handle. Overall, we found that only 2.8 % of expressions and local variables need annotations.

Performance. We wrote a prototype ahead-of-time compiler to translate SJS to C. The compiler uses a flat object representation, which ensures at most two indirections when accessing an object attribute. Then it invokes an off-the-shelf C compiler to produce an executable binary. Besides the flat object representation, and the standard optimizations performed by the C compiler, the SJS compiler does not perform any high-level optimizations.

In our experiment, we used eight programs to evaluate the potential performance benefits of statically-known object layout. We compared the execution time of the output of our compiler with the execution time when using the just-in-time compiler from `node.js` version 0.10.29. On programs using prototype-based inheritance and subtpying, the executables produced by the SJS compiler showed notably better performance (1.5–2.5x). For programs using objects without inheritance, the binaries generated by the SJS compiler showed some improvement (1.02–1.25x). Finally, SJS showed poorer performance (0.65–0.87x) than `node.js` on programs with mostly numeric and array operations. We refer to the companion technical report [12] for more details on the evaluation. Considering the fact that the prototype SJS compiler does not perform any high-level optimizations, we believe that the results show that knowing statically the layout of objects can allow an ahead-of-time compiler to generate faster code for programs that use objects extensively.

5 Related Work

Inheritance Mechanism and Object Layout. There is a strong connection between the inheritance mechanism a language uses and the way a language

ensures a fixed object layout property, which enables static compilation. Common inheritance mechanisms include class-based inheritance (e.g., SmallTalk, C++, Java, and Python), cloning-based prototype inheritance (Cecil [10])[7], and delegation-based prototype inheritance (e.g., Self [11], JavaScript, and Cecil).

Plain object types can be used to ensure fixed object layout property for a language using either class-based inheritance or cloning/sharing-based prototype inheritance. In both cases, it is impossible to change the offset of an attribute of an object once it is computed. Therefore, the type system only needs to ensure the following two requirements: (i) all objects generated using the same constructor should have the same layout, and (ii) an attribute cannot be added or removed once an object is created. Indeed, statically-typed languages in this category exactly implements these restrictions through their type system. Even static type systems proposed to enable static compilation of dynamic languages, such as StrongTalk [9] and RPython [6], impose these requirements.

However, these requirements are not enough for a language using a delegation-based inheritance mechanism, as we discussed in Sect. 2. Cecil solves this problem by making delegation explicit. When inheritance happens, attributes to be delegated to the prototype are marked with the keyword `share`. Then, updating a delegated attribute of an inheritor object changes the original owner of the attribute, rather than adding the attribute to the inheritor object.

Object Calculus. Our base type system borrows several ideas from the *typed imperative object calculus* of Abadi and Cardelli [5], especially subtyping of object types and how to handle method detachment in the existence of subtyping. Unfortunately, we could not use the type system as is because it uses cloning-based inheritance rather than prototype-based inheritance. Our notion of method type is also different from theirs in that ours exclude a receiver type from attached method types to have a simple formalism at the cost of not supporting recursive data types. We refer to the companion technical report [12] (Section A.1) for an extension of SJS to support recursive data types.

The type system proposed by Bono and Fisher [8], based on Fisher et al.'s earlier work [14], separates objects into *prototype objects* and *proper objects* similar to precise objects and approximate objects in SJS. Prototype/proper objects are similar to precise/approximate objects except in the context of subtyping. Despite the similarity, the two systems achieve opposite goals: Bono and Fisher's calculus is designed to support extensible (i.e., flexible) objects, while our type system tries to ensure that objects have a fixed layout. Moreover, their notion of prototyping is not based on delegation. Thus, the calculus is not suitable for JavaScript programs.

[7] A cloning-based inheritance approach populates inherited attributes to an inheritor object when extending the inheritor object with a prototype. After that, all read and write operations are performed local to the inheritor object, without consulting the prototype object. This approach has an effect of fixing object layout at the object creation time.

Type Systems for Dynamically Typed Language. Several static type systems for dynamically typed languages have been proposed [6,9,15,24,25] as well as for JavaScript [2,4,7,13,17–23]. However, only `asm.js` [2] and RPython [6], which we already discussed in Sect. 1, have the same goals as SJS: to define a typed subset of the base language, which can be compiled efficiently. Other type systems are designed to provide type safety and often to retrofit an existing code base. Therefore, it is difficult to compare them directly with SJS type system.

Acknowledgments. The work of the first author is supported in part by a research internship at Samsung Research America. The work of the last author is supported in part by Samsung Research America. This research is partially supported by NSF grants CCF-1018730, CCF-1017810, CCF-1409872, and CCF-1423645. The authors thank Colin S. Gordon, Frank Tip, Manu Sridharan, and the anonymous reviewers for their comments and suggestions.

References

1. 01.org. https://01.org/html5webapps/webapps/
2. asm.js. http://asmjs.org/
3. Octane Benchmarks. https://developers.google.com/octane/
4. TypeScript. http://www.typescriptlang.org
5. Abadi, M., Cardelli, L.: A Theory of Objects, 1st edn. Springer, New York (1996)
6. Ancona, D., Ancona, M., Cuni, A., Matsakis, N.D.: RPython: a step towards reconciling dynamically and statically typed oo languages. In: DSL 2007 (2007)
7. Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for javascript. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 428–452. Springer, Heidelberg (2005)
8. Bono, V., Fisher, K.: An imperative, first-order calculus with object extension. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 462–497. Springer, Heidelberg (1998)
9. Bracha, G., Griswold, D.: Strongtalk: typechecking smalltalk in a production environment. In: OOPSLA 1993 (1993)
10. Chambers, C., Group, T.C.: The Cecil language - specification and rationale (2004)
11. Chambers, C., Ungar, D.: Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In: PLDI 1989 (1989)
12. Choi, P.W., Chandra, S., Necula, G., Sen, K.: SJS: a typed subset of JavaScript with fixed object layout. Technical report UCB/EECS-2015-13, EECS Department, University of California, Berkeley, April 2015
13. Chugh, R., Herman, D., Jhala, R.: Dependent types for JavaScript. In: OOPSLA 2012 (2012)
14. Fisher, K., Honsell, F., Mitchell, J.C.: A lambda calculus of objects and method specialization. Nord. J. Comput. **1**(1), 3–37 (1994)
15. Furr, M., An, J.H.D., Foster, J.S., Hicks, M.: Static type inference for ruby. In: SAC 2009 (2009)
16. Gong, L., Pradel, M., Sen, K.: JITProf: pinpointing JIT-unfriendly JavaScript code. In: ESEC/FSE 2015 (2015)

17. Heidegger, P., Thiemann, P.: Recency types for analyzing scripting languages. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 200–224. Springer, Heidelberg (2010)
18. Lerner, B.S., Politz, J.G., Guha, A., Krishnamurthi, S.: TeJaS: retrofitting type systems for JavaScript. In: DLS 2013 (2013)
19. Politz, J.G., Guha, A., Krishnamurthi, S.: Semantics and types for objects with first-class member names. In: FOOL 2012 (2012)
20. Rastogi, A., Chaudhuri, A., Hosmer, B.: The ins and outs of gradual type inference. In: POPL 2012 (2012)
21. Rastogi, A., Swamy, N., Fournet, C., Bierman, G., Vekris, P.: Safe and efficient gradual typing for TypeScript. Technical report MSR-TR-2014-99, July 2014
22. Swamy, N., Fournet, C., Rastogi, A., Bhargavan, K., Chen, J., Strub, P.Y., Bierman, G.: Gradual typing embedded securely in JavaScript. In: POPL 2014 (2014)
23. Thiemann, P.: Towards a type system for analyzing JavaScript programs. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 408–422. Springer, Heidelberg (2005)
24. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed Scheme. In: POPL 2008 (2008)
25. Tobin-Hochstadt, S., Felleisen, M.: Logical types for untyped languages. In: ICFP 2010 (2010)