SJS: a Typed Subset of JavaScript with Fixed Object Layout

Philip Wontae Choi
Satish Chandra
George Necula
Koushik Sen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-10
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-10.html

March 6, 2015

SJS: a Typed Subset of JavaScript with Fixed Object Layout

Technical Report

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

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

Abstract. We present a proposal for 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 enables 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. We describe the design of SJS, both at the type-system level and source level, along with a local type inference algorithm. We measured experimentally the effort required in adding the necessary typing annotations, and the effectiveness of a simple AOT compiler that exploits the fixed object layout property of SJS.

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. A JavaScript web application can readily be executed by pointing the browser to the application website.

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 can be foiled when new fields and method are added to an object [19]. Also, JIT optimization might be an unsatisfactory solution for a resource constrained environment such as a mobile platform.

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 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.

In this paper we describe another design point for a subset of JavaScript 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`, we want to enable an AOT compiler to translate attribute accesses into direct memory accesses, which requires that objects have statically known layouts.

The main technical challenge that we address 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 operation, 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 Section 2 how this particular semantics interacts with high-level features such as structural subtyping and method updates.

Contributions. In this paper, we propose SJS, a statically-typed subset of JavaScript, with the following main contributions:

– SJS includes many attractive and convenient high-level features, such as prototype-based inheritance, closures, a 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.

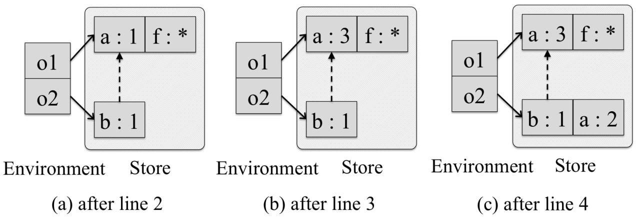
- SJS can support AOT compilation, and can run on standard JavaScript engines with the standard JavaScript semantics.
- The type system of SJS is described as a composition of a standard base type system for records, along with *qualifiers* on object types designed to ensure the fixed object layout. This presentation of the type system highlights the design of the qualifiers, which is a novel contribution of this type system.
- SJS includes inference of types and object-type qualifiers. SJS can use any standard type inference for the base record types, and adds an automatic qualifier inference step. We describe in this paper a design based on a local type-inference algorithm, for source programs with type annotations on function parameters. The types of local variables, object attributes, and function return values are automatically inferred most of the time, with a few exception cases that can be easily annotated manually. The qualifiers are inferred without any user interaction.
- Type annotations are encoded using plain JavaScript expressions, and they do not affect the semantics of programs. This is to allow SJS to be a strict subset of JavaScript, and a SJS program to have the same behavior regardless of whether it is compiled into low-level code, or it runs on an existing JavaScript engine.
- We report a preliminary evaluation of SJS. We ported a number of JavaScript programs to fit in the SJS type system. We measured the number of type annotations needed, and assessed the amount of workaround needed to accommodate the restrictions posed by the type system. We also implemented a proof-of-concept AOT compiler to check the feasibility of statically compiling SJS programs. Our experiments show that even a fairly simple AOT compiler can take advantage of the fixed layout property of SJS to achieve performance that is competitive with that of state-of-the-art JIT optimizers.

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 [5] is a typed *super*set of JavaScript designed to simplify development and maintenance. There are two significant differences between SJS and TypeScript: (i) 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. (ii) TypeScript is a superset of JavaScript—it extends the syntax of JavaScript to include type annotations. Therefore TypeScript programs cannot be run on a JavaScript engine directly.

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`,

```
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.



(a) after line 2 (b) after line 3 (c) after line 4

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

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 [9] 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. Another big difference is that RPython uses a whole-program analysis instead of a local type inference. These two approaches have different advantages: whole program analysis requires a fewer type annotations, while local type inference provides better type errors and requires less analysis time.

Organization Section 2 gives overview and rationale for SJS types and qualifiers with examples. Section 3 provides a formal treatment of the SJS type system with object qualifiers. Section 4 explains an object qualifier inference algorithm. Section 5 discusses SJS, the top level language that relies this type system. Section 6 gives an evaluation of SJS, both in terms of its closeness to JavaScript and feasibility of AOT compilation. Section 7 discusses related work.

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 Figure 1.

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`[3]. According to JavaScript semantics, the object `o2` will include a reference to the prototype object `o1`, as shown in Figure 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 Figure 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 Figure 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.[4] 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.):

- **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.

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

[4] This representation is called hidden class representation and the caching technique is called inline caching [15]. As noted before, this optimization can fail to apply under certain conditions [19].

- **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 Figure 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 Figure 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 Figure 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 distingusihes 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.[5] 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`. Finally, note that line 4 (`o2.a = 2`) is ill-typed because `a` is not among the owned fields mentioned in the static type of `o2`.

2.2 Subtyping

Consider again the example in Figure 1 with the object layouts as shown in Figure 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

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

shows 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 of such an object would be:

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

The type of `o3` is

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

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 $\{\texttt{a}\}$, while the true inheritor-owned attributes of the runtime object are $\{\texttt{c}\}$. This suggests that it would be unsafe to use the object `o1` as a prototype in a new object creation, as in the following continuation of our example:

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

If the object creation in line 8 is well typed, with the type:

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

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 following code fragment:

```
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
```

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 following assignment will be legal

```
16: var o8 = o3 // OK
```

if the static type of `o8` is the following approximate type:

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

Moreover, we can perform attribute lookup and method invocation via `o8` as shown below, because these operations are allowed on approximate types:

```
17: o8.f(3);  // OK
18: o8.c = 2; // OK
```

$$
\begin{array}{l}
e ::= n \mid x \mid x = e_1 \mid \texttt{var}\ x : T\texttt{=}e_1\ \texttt{in}\ e_2 \mid \{a_1 \!:\! e_1\ \ldots\ a_n \!:\! e_n\}_T \mid e.a \mid e_1.a\texttt{=}e_2 \\
\quad \mid \texttt{function}(x : T)\{e\} \mid e_1(e_2) \mid e_1.a(e_2) \mid \texttt{this} \mid \{a_1 \!:\! e_1\ \ldots\ a_n \!:\! e_n\}_T\ \texttt{prototype}\ e_p
\end{array}
$$

Fig. 3. Syntax

However, it would be illegal to use `o8` as prototype, as follows:

```
19: var o9 = { a: 14, __proto__: o8} // BAD
```

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

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

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 Section 3.

3 A Formal Account of Types and Object-Type Qualifiers

This section provides a formal definition of the type system of SJS. 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 until Appendix A: first-class method functions, recursive data types, and accessing `this` in a non-method function.

3.1 Syntax

The syntax definition of the core language expressions is shown in Figure 3. 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 $a_1, \ldots, a_n$

$$\begin{array}{lll}
\textit{Type} & T & ::= \mathit{Int} \mid O \mid T \to T \mid T \Rightarrow T \mid \top \\
\textit{ObjTy} & O & ::= \rho^{\,q} \\
\textit{ObjBase} & \rho & ::= \{\ldots a_i : T_i \ldots\} \\
\textit{RcvTy} & R & ::= \top \mid O \\
\textit{TyEnv} & \Gamma & \in \quad \mathit{Var} \to \mathit{Type}
\end{array}
\qquad
\begin{array}{llll}
\textit{ObjQual} & q & ::= & \mathbf{P}(\texttt{own}, \texttt{iown}) \mid \mathbf{A}(\texttt{own}) \\
\textit{OwnSet} & \texttt{own} & \subseteq & \mathit{Attr} \\
\textit{ModSet} & \texttt{iown} & \subseteq & \mathit{Attr} \\
 & & & \\
 & \mathit{Attr} & & \text{set of atributes (a,b \ldots)} \\
 & \mathit{Var} & & \text{set of variables (x,y \ldots)}
\end{array}$$

Fig. 4. Types

initialized with expressions e_1, …, 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.

3.2 Types and Qualifiers

Figure 4 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 [6] 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.

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. $\mathit{dom}(\rho)$ 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.

3.3 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.

Well-formed Types

$$\texttt{[TW-EObj]}\,\frac{\forall a \in \mathit{dom}(\rho) \vdash \rho(a) \qquad \texttt{iown} \subseteq \texttt{own} \qquad \texttt{own} \subseteq \mathit{dom}(\rho)}{\vdash \rho^{\mathbf{P}(\texttt{own}, \texttt{iown})}} \qquad \texttt{[TW-AObj]}\,\frac{\forall a \in \mathit{dom}(\rho) \vdash \rho(a) \qquad \texttt{own} \subseteq \mathit{dom}(\rho)}{\vdash \rho^{\mathbf{A}(\texttt{own})}}$$

$$\texttt{[TW-Fun]}\,\frac{\vdash T_1 \qquad \vdash T_2}{\vdash T_1 \rightarrow T_2} \qquad \texttt{[TW-Method]}\,\frac{\vdash T_1 \qquad \vdash T_2}{\vdash T_1 \Rightarrow T_2} \qquad \texttt{[TW-Top]} \vdash \top$$

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

Subtyping

$$\texttt{[ObjPA}_{<:}\texttt{]}\,\frac{\forall \mathit{dom}(\rho_2).\rho_1(a) \equiv \rho_2(a) \qquad \mathit{dom}(\rho_1) = \mathit{dom}(\rho_2) \qquad \texttt{own}_1 = \texttt{own}_2}{\rho_1^{\mathbf{P}(\texttt{own}_1, \texttt{iown}_1)} <: \rho_2^{\mathbf{A}(\texttt{own}_2)}}$$

$$\texttt{[ObjAA}_{<:}\texttt{]}\,\frac{\forall a \in \mathit{dom}(\rho_2).\rho_1(a) \equiv \rho_2(a) \qquad \mathit{dom}(\rho_2) \subseteq \mathit{dom}(\rho_1) \qquad \texttt{own}_2 \subseteq \texttt{own}_1}{\rho_1^{\mathbf{A}(\texttt{own}_1)} <: \rho_2^{\mathbf{A}(\texttt{own}_2)}}$$

$$\texttt{[Trans}_{<:}\texttt{]}\,\frac{\begin{array}{c}T_1 <: T_2 \\ T_2 <: T3\end{array}}{T_1 <: T_3} \qquad \texttt{[Refl}_{<:}\texttt{]}\,\frac{}{T <: T} \qquad \texttt{[Fun}_{<:}\texttt{]}\,\frac{T_3 <: T_1 \qquad T_2 <: T_4}{T_1 \rightarrow T_2 <: T_3 \rightarrow T_4} \qquad \texttt{[Top}_{<:}\texttt{]}\,\frac{}{T <: \top}$$

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

3.4 Subtyping and Type Equality

Figure 6 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. 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.

3.5 Typing Rules

The static typing rules are defined in Figure 7. The type system is composed of two kinds of rules: *expression typing judgment* and *attribute-update typing judgment*.

Expression Typing

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

$$\texttt{[T-LetVar]}\,\frac{\begin{array}{c} R, \mathit{\Gamma} \vdash e_1{:}T_1 \qquad \vdash T \qquad T_1 <: T \\ R, \mathit{\Gamma}[x \mapsto T] \vdash e_2 : T_2 \end{array}}{R, \mathit{\Gamma} \vdash \texttt{var}\ x{:}T\texttt{=}e_1\ \texttt{in}\ e_2 : T_2} \qquad \texttt{[T-This]}\,\frac{}{\rho^q, \mathit{\Gamma} \vdash \texttt{this} : \rho^q}$$

$$\texttt{[T-Fun]}\,\frac{\top, \mathit{\Gamma}[x \mapsto T_1] \vdash e : T_2 \qquad \vdash T_1}{R, \mathit{\Gamma} \vdash \texttt{function}(x : T_1)\{e\} : T_1 \to T_2} \qquad \texttt{[T-FCall]}\,\frac{\begin{array}{c} R, \mathit{\Gamma} \vdash e_1 : T_1 \to T_2 \\ R, \mathit{\Gamma} \vdash e_2 : T_3 \qquad T_3 <: T_1 \end{array}}{R, \mathit{\Gamma}, \vdash e_1(e_2) : T_2}$$

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

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

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

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

Attribute-Update Typing

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

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

Fig. 7. Type system (syntax-directed definition). The highlighted items are specific to object-type qualifiers.

Expression Typing. The expression typing judgment $R, \mathit{\Gamma} \vdash e : T$ means that expression e under receiver type R and type environment $\mathit{\Gamma}$ 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 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 [6]. `[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 $R, \Gamma \vdash_{AU} O.a\texttt{=}e$ 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 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.

4 Qualifier Inference

The type inference problem for a SJS program can be solved by first inferring base types (i.e, types without qualifiers), followed by inferring the object-type qualifiers. The reader can inspect the typing rules to see that if we eliminate the highlighted elements, what is left is a standard type system for record types, for which the base type inference can be achieved by adopting existing constraint based type inference algorithms [7, 23]. Hence, in this paper, we focus on the problem of the object-type qualifier inference.

The qualifier inference problem again can be staged into preciseness inference and owned-set inference. The preciseness inference determines whether a qualifier is precise (**P**) or approximate (**A**). The owned-set inference determines the contents of `own` and `iown` of each qualifier. Next, we are going to demonstrate how the inference algorithm works using the following example program:

```
var x = {a : 1, f : function}(z){ this.a = z } }
var y = x
var z = { a : 2, __proto__: y}
```

Setup. Given the input program, the inference algorithm first performs a base type inference, annotates the input program with the inferred base types, and then populates the type annotations with fresh qualifier variables (κ) in the object-type qualifier positions. After applying these steps, the input program becomes:

```
var x : $\rho^{\kappa_1}$ = { a : 1, f : function(z:$Int$){ this.a = z } }$_{\rho^{\kappa_2}}$
var y : $\rho^{\kappa_3}$ = x
var z : $\rho^{\kappa_4}$ = { a : 2, __proto__: y }$_{\rho^{\kappa_5}}$
```

where $\rho = \{\mathtt{a} : Int, \mathtt{f} : Int \Rightarrow \top\}$
and `this` : ρ^{κ_6}

Preciseness inference. The preciseness of object types can be obtained by backward program analysis. According to the type system definition, any object value used as a prototype (`[T-Prototype]`) or being a target of a method update operation (`[T-AttrUpdMethod]`) should have a precise object type. Therefore, starting from these sink points, by backward propagating the preciseness requirement toward sources, we can tell exactly which object-type qualifiers have to be precise. All the other qualifiers become approximate by default.

In the above example, the preciseness analysis starts by setting the object-type qualifiers κ_2, κ_5 to be precise, because they occur in object literals. Next, the type of variable y should be precise because it is used as a prototype. Hence,

κ_3 is precise. Then, the preciseness of κ_3 is backward propagated to the type of variable x. Therefore, qualifier κ_1 become precise. The preciseness inference computes the following result:

$$\begin{array}{ll} \textit{Precise Qualifiers} & \{\kappa_1, \kappa_2, \kappa_3, \kappa_5\} \\ \textit{Approximate Qualifiers} & \{\kappa_4, \kappa_6\} \end{array}$$

Owned and inheritor-owned set inference. Once we know the preciseness of the qualifier variables, we can traverse the typing derivation tree of the program to collect constraints on the owned and inheritor-owned sets. Note that the preciseness information is important in order to know which of the rules `[ObjAA`$_{<:}$`]` and `[ObjPA`$_{<:}$`]` to use for subtyping object types. The collected constraints consist of set equality and set inclusion constraints. Constraints of this form are then solved using existing techniques [8].

In the above example, we generate the following set constraints on object-type qualifiers and compute the following minimal solution:

$$\textit{Constraint} = \left\{\begin{array}{lll} a \in \mathtt{own}(\kappa_2), & f \in \mathtt{own}(\kappa_2), & a \in \mathtt{iown}(\kappa_6), \\ \mathtt{iown}(\kappa_2) = \mathtt{iown}(\kappa_1), & \mathtt{own}(\kappa_2) = \mathtt{own}(\kappa_1), & \mathtt{own}(\kappa_6) \subseteq \mathtt{iown}(\kappa_2), \\ \mathtt{iown}(\kappa_3) = \mathtt{iown}(\kappa_2), & \mathtt{own}(\kappa_3) = \mathtt{own}(\kappa_2), & \mathtt{iown}(\kappa_5) \subseteq \mathtt{own}(\kappa_5), \\ \mathtt{iown}(\kappa_3) \subseteq \mathtt{iown}(\kappa_5), & \mathtt{own}(\kappa_5) = \mathtt{own}(\kappa_4) & \end{array}\right\}$$

$$\textit{Solution} = \left(\begin{array}{l} \mathtt{own}(\kappa_1), \mathtt{own}(\kappa_2), \mathtt{own}(\kappa_1) = \{a, f\} \\ \mathtt{own}(\kappa_4), \mathtt{own}(\kappa_6), \mathtt{iown}(\kappa_1), \mathtt{iown}(\kappa_2), \mathtt{iown}(\kappa_3), \mathtt{iown}(\kappa_5) = \{a\} \end{array}\right)$$

Combining the solution with the preciseness analysis result, we get the following qualifiers:

$$\textit{Qualifiers} = \left(\begin{array}{l} \kappa_1, \kappa_2, \kappa_3 = \mathbf{P}(\{a, f\}, \{a\}) \\ \kappa_4, \kappa_6 = \mathbf{A}(\{a\}) \\ \kappa_5 = \mathbf{P}(\{a\}, \{a\}) \end{array}\right)$$

5 Top-Level Language SJS

In SJS, qualifiers are inferred automatically; however, inference of base types require a small amount of user-provided type hints. Specifically, the base type inference algorithm requires type hints for every function parameters including the implicit receiver parameter (`this`). Types of local variables, object attributes, and function return values are inferred automatically most of the time. In order to avoid any special extension to JavaScript, SJS uses JavaScript expressions to provide type hints. These hint expressions do not perform any computation. Therefore, they will be optimized away by a modern JIT compiling JavaScript engine. Once base types are inferred using the type hints, object-type qualifiers can be inferred automatically using the algorithm described in Section 4.

Type Hints in SJS. We next explain the SJS type-hinting mechanism using the following example:

```
1:  function Point(x,y){
2:   Point.instance = this;
3:   Hint.sameType(x,0);
4:   Hint.sameType(y,0);
5:   this.x = x;
6:   this.y = y;
7:  }

8:  function dist(p){
9:   Hint.sameType(p, Point.instance);
10:  var v = p.x*p.x + p.y*p.y;
11:  return Math.sqrt(v);
12: }

13: var o1 = {x:1, a:"surprise", y:1};
14: dist(o1);
15: var o2 = {x:1, y:1};
16: o2 = Hint.dynCast(o2, o1);
17: var o3 = null;
18: Hint.sameType(o1, Point.instance);
```

Equating types. `Hint.sameType` is a special no-op function that is used to communicate to the type system that the two argument expressions are of the same type. For example, `Hint.sameType(x,0)` specifies that `x` is of type integer (which is the type of the literal 0).

Constructors and type names. In SJS, we assume that every constructor function has a special attribute called `instance` and the constructor assigns `this` to this attribute at the beginning of the body of the constructor. This special assignment directs the type system to treat the function as a constructor. The special assignment also ensures that the special `instance` attribute has the same type as any object created by the constructor. Therefore, the `instance` attribute of a constructor can be used as a name of the type of an object created by the constructor (see line 9).

Object types. The type system requires that all attributes of an object get defined within its constructor. Once an object escapes its constructor, new attributes cannot be added to an object. Therefore, a syntactic scan of the body of a constructor enables us to infer the type of objects constructed by the constructor. Similarly, for object literals, the type system enforces that all attributes are defined in the literal itself.

Function parameters. The type system requires hints for the types of function parameters (line 3, 4, and 9). This requirement enables function-local type inference, which makes the inference result easier to understand. If a function is a method, the type hint for the receiver parameter (`this`) should be provided explicitly. Receiver type hints can be omitted when a method is directly assigned to an object attribute (e.g, $x.a = function()\{...\}$).

```
var get = function(){ this.a };
function F(x){this.a = x;}
F.prototype = {a:0, get:get};
var o = new F(2);
```

(a) JavaScript version

```
var get = function(){ this.a };
var p = {a:0, get:get};
var F = function(x,p){ {a:x} prototype p }
var Fcon = {f:F, p:p};
var o = Fcon.f(2, Fcon.p);
```

(b) the core language version

Fig. 8. Emulating JavaScript constructors in SJS

Local variables, function returns, and attributes. Types of local variables, function returns, and object attributes are often inferred automatically using available information (line 5, 6, 10, 11, and 12). However, a user needs to provide type hints in cases where exact type information is not available. For example, the type of a variable initialized with a (`null`) value cannot be determined because `null` can have any object type (line 17 and 18). Similarly if an array is initialized with an empty array ([·]), its element type cannot be determined.

Type coercion. In SJS, a type coercion can be performed explicitly using the expression `Hint.dynCast(e1, e2)` (line 16), where the type of `e1` is a subtype of the type of `e2`. The function returns the value of `e1`, but the type system considers the return type to be equal to the type of `e2`. Note that SJS only allows explicit up-casting coercions because it is difficult, if not impossible, to check the safety of down-casting at runtime given the limited amount of runtime type information we have during an execution. The only exception is for DOM objects because subtyping between DOM objects can be checked dynamically in most JavaScript engines. Implicit type coercion is only allowed when an expression is passed as an argument to a function or a method (line 14). Note that implicit type coercion during argument passing does not pose any problem in type inference as types of function parameters are inferred within the body of a function.

Constructors. The biggest difference between the formalism introduced in the previous section and JavaScript lies in the handling of constructor functions. In JavaScript, prototype inheritance is implemented through constructor functions while the core language has a dedicated prototype expression. Most JavaScript constructor functions can be desugared into the prototype expressions, as long as the following two restrictions are met. First, all attributes must be defined at the beginning of the constructor. Second, the prototype attribute of a constructor must be initialized immediately after the constructor definition. This is to prevent a situation where a constructor is invoked before a prototype is fully initialized. Figure 8 shows an example of desugaring JavaScript constructors using a prototype expression.

Name	Kind	#Line	#Param	#Var	#Attribute	#Return	#Dyncast	#Total
richards	simulation	539	88	0 (31)	6 (56)	0 (41)	0 (7)	94 (135)
raytracer	rendering	789	124	0 (74)	5 (32)	0 (50)	5 (3)	134 (160)
annex	game	1319	145	1 (198)	8 (153)	0 (96)	21 (97)	175 (544)
hangonman	game	1868	111	7 (253)	7 (48)	0 (121)	13 (86)	138 (508)

Table 1. Number of lines of type hints added

6 Evaluation

We have implemented a proof-of-concept type checker and compiler for SJS to evaluate the language. We evaluate the usability of the language and the feasibility of type-based compilation. The programs used in this section can be found at `http://goo.gl/nBtgXj`.

6.1 Usability of SJS

We ported a number of JavaScript programs to SJS to evaluate the usability of the language. The goal of this effort is to answer the following two questions. (1) *How many type hints do we need to port a JavaScript program to SJS?*, and (2) *How restrictive is the type system?* To answer the latter question we discuss the cases where we needed to modify a program to work around the type checker.

We considered two programs from the *octane* benchmark suite [3] (`richards` and `raytracer`) and two webapps from *01.org* [1] (`annex` and `hangonman`). We selected JavaScript programs that are moderate-sized (about 500 to 2000 lines of code) and that use objects extensively. We managed to fully typecheck `richards` and `raytracer` after adding suitable type hints. In order to typecheck `annex` and `hangonman`, we had to comment out small portions of code handling Ajax communication, because we do not have enough contextual information to decide the types for this part.

Number of type hints. Table 1 summarizes the number of type hints we added for each benchmark program. Columns `Kind` and `#Line` describe the kind and the number of lines of code of each benchmark program, respectively.

SJS requires a programmer to provide type hints for all function parameters. `#Param` column shows the number of type hints added for function parameters. We believe that adding these kind of type hints will not pose a significant cognitive burden to programmers once they become familiar with our language.

SJS also requires a programmer to provide type hints for a small fraction of local variables, object attributes, function return values, and type casts. `#Var` column shows the number of type hints used for local variables, with the number of local variables without type hints in the parenthesis. `#Attribute` shows the number of type hints used for object attributes, with the number of object attributes without type annotations in the parenthesis. `#Return` shows the number of type hints used for function return values, with the number of functions without return type hints in the parenthesis. From the table, we can see that type hints of these kinds can be omitted most of the time because they can be inferred from the context. The exceptional cases also can be easily recognized and handled as described in Section 5.

`#Dyncast` column shows the number of explicit upcasting (`dynCast`), with the number of implicit upcasting listed in the parenthesis. These kind of hints are usually not required by a statically-typed object-oriented programming language. The numbers in this column show that significant number of upcasts do not require type hints, because SJS allows implicit upcasting at function call boundaries.

Restrictiveness of SJS. Next, we discuss the challenges we faced while fitting the benchmark programs to SJS.

Fixing constructors. The first important step necessary to port any program is to identify the set of attributes that belongs to each object in the program. After this step, the constructors of the objects are modified to initialize the corresponding attributes at the beginning of the constructor body with suitable initial values. If an attribute is initialized with `null`, we provide a type hint for the attribute.

Temporary attributes. In some JavaScript programs, we observed a common idiom where a new attribute is temporarily attached to an object to keep track of the course of computation. This cannot be supported in SJS because temporary attributes change the layout of an object. Moreover, we cannot simply promote a temporary attribute to a permanent one, because the program execution could check the existence of the attribute and make a decision based on the outcome. In these situations we rewrote parts of the program depending on temporary attributes. We observed the usage of this idiom in `annex`.

Evidence values. One limitation of the type naming idiom (e.g. `MyConstructor.instance = this`) used in SJS is that it cannot be used for objects not created by a constructor, such as arrays and objects created by literal expressions. To handle these cases, we found that it is convenient to create a dummy *evidence object*, to assign the object to a *type hint variable*, and to use the variable as a representative of all other objects having the same type. The following is an example where we use the variable `hintIntArrArr` to specify that the parameter of the function has type array of arrays of integers:

```
var hintIntArrArr = [[0]];
function sumAA(arg){
  Hint.sameType(arg, hintIntArrArr);
  //do something
}
```

name	#line	#fun	node.js (sec.)	SJS (sec.)	ratio
`matmul`	76	3	362.30	415.90	0.87
`spectral`	98	6	107.21	164.64	0.65
`huffman`	295	17	0.59	0.48	1.25
`bh`	332	15	109.60	83.47	1.31
`simplex`	373	21	21.67	21.39	1.02
`tsp-ga`	442	18	2157.37	1745.45	1.23
`bdd`	479	40	12.99	5.19	2.56
`raytracer`	789	50	10.99	6.37	1.58

Table 2. Experiment summary

Other issues. Among the remaining cases unrelated to objects, we want to emphasize the following two cases. First, some functions can take arguments that have multiple types and take different branches inside the function body based on runtime type analysis (using `typeof` operator) of the arguments. We split such a function into multiple monomorphic functions. Second, SJS does not support downcasting: sound type checking at runtime without instrumentation or change to a JavaScript engine would require us to figure out the owned, inheritor-owned, and the attribute sets of an object inside the downcast function, and this is not possible to do since JavaScript has non-enumerable hidden properties. Therefore, we rewrote any code fragment involving a downcast.

6.2 Feasibility of Ahead of Time Compilation

We have implemented a proof-of-concept compiler for SJS programs. The compiler first translates from SJS to C without any high-level optimizations. The compiler uses a flat object representation, which ensures at most 2 indirections when accessing an object attribute. Then it invokes an off-the-shelf C compiler (LLVM 5.1 with -O3 option) to emit a binary. We use Boehm's conservative garbage collector [11] to perform garbage collection during the execution of a compiled program. We remark that our goal is not to outperform modern JIT compilers. Rather, it is to show that this statically typed subset enables AOT compilers to offer performance that is roughly comparable to JIT compilers.

Benchmarks. We used eight programs to evaluate the feasibility of ahead-of-time compilation. We obtained three of these programs from existing benchmarks [3,4], and the rest were created by implementing textbook algorithms. We could not use the benchmarks for the usability evaluation (except `raytracer`), because most of them use string operations, which is currently not supported by the prototype compiler. Table 2 lists the benchmark programs. Columns #line and #fun report the number of lines of code and the number of functions present in each of these programs, respectively.

Evaluation. We performed our experiments on a dual core *Intel Core I5* 1.6Ghz (2467M) OSX machine with 4GB RAM. We report the execution time of these compiled programs in Table 2. We also ran the original programs in `node.js` (0.10.28), a Google's V8 based JavaScript engine, and report their execution times in the same table.[6] We report the ratio of the times taken by `node.js` relative to SJS in the last column of the table.

To keep the implementation effort low, we ended up using a garbage collector different from that used in `node.js`. To factor out this difference, and to focus on the computation time, the running times reported in the table exclude the time spent on garbage collection.[7]

In our experiment, the binaries generated by the SJS compiler showed notably better performance (1.5-2.5x) on programs (e.g. `bdd` and `raytracer`) using prototype-based inheritance and subtyping. Because of subtyping, several object attribute accesses in these programs required polymorphic inline caches [21] in the `node.js` engine. For programs using objects without prototype-based inheritance and subtyping (e.g. `huffman`, `bh`, `simplex`, `tsp-ga`), the binaries generated by the SJS compiler showed some improvement (1.02-1.25x). Our conjecture is that in these programs most of object attribute accesses can be resolved using a monomorphic inline cache.

Finally, `node.js` showed better performance on small programs with mostly numeric and array operations (e.g. `matmul` and `spectral`). This slowdown is attributed to the fact that our prototype compiler does not perform any meaningful optimizations for these operations compared to `node.js`.

Considering the fact that the prototype SJS compiler does not perform any high-level optimization, we believe that the results show the feasibility of the ahead-of-time compilation of JavaScript.

7 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 [14])[8], and delegation-based prototype inheritance (e.g., Self [15], 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

[6] The latest V8 compiler runs slightly faster than the version in node.js, but we could not find a way to break out GC time and computation time with V8.

[7] Since Boehm's garbage collector is conservative and type agnostic, it runs significantly slower than V8's generational garbage collector on our programs.

[8] 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.

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 [13] and RPython [9], impose these requirements.

However, these requirements are not enough to ensure fixed object layout for a language using a delegation-based inheritance mechanism, as we discussed in Section 2. Cecil solved this problem by letting programmers select what to delegate. When inheritance is used, the inheritor object can select which attributes to delegate to the prototype by using the keyword `share`. Then, updating an delegated attribute of an inheritor object changes the original owner of the attribute, rather than adding the attribute to the inheritor object. This make sense because delegation in Cecil is explicit and intentional, unlike JavaScript and Self.

Object Calculus. Our base type system borrows several ideas from the *typed imperative object calculus* of Abadi and Cardelli [6], 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 appendix for an extension of SJS to support recursive data types.

The type system proposed by Bono and Fisher [12], based on Fisher et al.'s earlier work [17], 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.

Type Systems for Dynamically Typed Language. Several static type systems for dynamically typed languages have been proposed [9, 13, 18, 26, 27] as well as for JavaScript [2, 5, 10, 16, 20, 22, 24, 25]. However, only `asm.js` [2] and RPython [9], which we already discussed in Section 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.

Some proposed type systems support desirable features that are currently lacking in SJS. For example, type systems for JavaScript proposed by Thiemann [25] and Politz et al. [24], support accessing an object using a dynamically computed string as a key. Currently, SJS allows an object access only using a statically fixed string key. Another example is occurrence typing [27], which allows to locally refine type information at a branch using an associated branch condition as a hint. We conjecture that these two features and object-type qualifiers are orthogonal, so they can be added to SJS. However, not every type system can be combined with SJS, especially the ones requiring flow-sensitive reasoning [10, 16, 20], because those type systems could conflict with the fixed layout property requirement.

8 Acknowledgement

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 supported in part by NSF Grants CCF-0747390, CCF-1018729, CCF-1423645, and CCF-1018730.

References

1. 01.org. `https://01.org/html5webapps/webapps/`
2. asm.js. `http://asmjs.org/`
3. Octane Benchmarks. `https://developers.google.com/octane/`
4. The Benchmarks Game. `http://benchmarksgame.alioth.debian.org/`
5. TypeScript. `http://www.typescriptlang.org`
6. Abadi, M., Cardelli, L.: A Theory of Objects. Springer-Verlag New York, Inc.
7. Agesen, O., Palsberg, J., Schwartzbach, M.I.: Type inference of self. In: ECOOP 1993
8. Aiken, A.: Introduction to set constraint-based program analysis. Sci. Comput. Program. (Nov 1999)
9. Ancona, D., Ancona, M., Cuni, A., Matsakis, N.D.: Rpython: A step towards reconciling dynamically and statically typed oo languages. In: DSL 2007
10. Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for javascript. In: ECOOP 2005
11. Boehm, H., Demers, A., Weiser, M.: A garbage collector for c and c++ (2002)
12. Bono, V., Fisher, K.: An imperative, first-order calculus with object extension. In: ECCOP 1998
13. Bracha, G., Griswold, D.: Strongtalk: Typechecking smalltalk in a production environment. In: OOPSLA 1993
14. Chambers, C., Group, T.C.: The cecil language – specification and rationale (2004)
15. Chambers, C., Ungar, D.: Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In: PLDI 1989
16. Chugh, R., Herman, D., Jhala, R.: Dependent types for javascript. In: OOPSLA 2012
17. Fisher, K., Honsell, F., Mitchell, J.C.: A lambda calculus of objects and method specialization. Nordic J. of Computing 1(1), 3–37 (Mar 1994)
18. Furr, M., An, J.h.D., Foster, J.S., Hicks, M.: Static type inference for ruby. In: SAC 2009

19. Gong, L., Pradel, M., Sen, K.: Jitprof: Pinpointing jit-unfriendly javascript code. Tech. Rep. UCB/EECS-2014-144, EECS Department, University of California, Berkeley (Aug 2014), `http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-144.html`
20. Heidegger, P., Thiemann, P.: Recency types for analyzing scripting languages. In: ECOOP 2010
21. Hoelzle, U., Chambers, C., Ungar, D.: Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In: ECOOP 1991
22. Lerner, B.S., Politz, J.G., Guha, A., Krishnamurthi, S.: TeJaS: Retrofitting type systems for JavaScript. In: DLS 2013
23. Oxhøj, N., Palsberg, J., Schwartzbach, M.I.: Making type inference practical. In: ECOOP 1992
24. Politz, J.G., Guha, A., Krishnamurthi, S.: Semantics and types for objects with first-class member names. In: FOOL 2012
25. Thiemann, P.: Towards a type system for analyzing javascript programs. In: ESOP 2005
26. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. In: POPL 2008
27. Tobin-Hochstadt, S., Felleisen, M.: Logical types for untyped languages. In: ICFP 2010

A Type System Extensions

We introduce a number of extensions to SJS that we found useful to type real-world programs. The extensions include recursive object types, first class methods, global objects, DOM objects, and pure prototype objects. All these extensions are orthogonal to the qualifier part of the type system. Therefore, the constraint based qualifier inference algorithm can be reused unchanged. The only exception is the pure prototype object extension, for which we describe a modified inference algorithm. We describe the extensions informally by modifying the original definitions. The modified parts are highlighted in gray.

A.1 Recursive Object Types

To support recursive data types, the SJS type system can be modified to access the information about an object via an object name (sl) and an object definition table (Δ) as in Java and C++:

$$\begin{array}{lll} \mathit{ObjTy} & O & ::= \ sl^{\,q} \\ \mathit{ObjRef} & \Delta & ::= \ \{sl : \rho \ldots\} \ (\in \mathit{SLoc} \rightarrow \mathit{ObjTy}) \\ \mathit{SLoc} & sl & \end{array}$$

We need three modifications to the SJS type system to accomodate object names and the object definition table. First, typing rules and companion rules (subtyping, etc.) need to pass and to consult the object definition table Δ. Second, every usage of a base object type needs to be replaced by a corresponding object type definition lookup ($\Delta(sl)$). Third, type equality rules need to track a list of equalities inferred so far in order to avoid infinite recursion:

$$\texttt{[Obj}_{\equiv}\texttt{-1]}\ \frac{sl_1 \equiv sl_2 \in \Phi \qquad q_1 = q_2}{\Phi \vdash_\Delta sl_1^{q_1} \equiv sl_2^{q_2}}$$

$$\texttt{[Obj}_{\equiv}\texttt{-2]}\ \frac{\begin{array}{cccc} sl_1 \equiv sl_2 \notin \Phi & \Phi' = \Phi \cup \{sl_1 \equiv sl_2\} & \rho_1 = \Delta(sl_1) & \rho_2 = \Delta(sl_2) \\ \multicolumn{2}{c}{\forall a \in \mathit{dom}(\rho_2).\Phi' \vdash_\Delta \rho_1(a) \equiv \rho_2(a)} & \mathit{dom}(\rho_1) = \mathit{dom}(\rho_2) & q_1 = q_2 \end{array}}{\Phi \vdash_\Delta sl_1^{q_1} \equiv sl_2^{q_2}}$$

Φ is the set of equalities inferred so far. Note that the equality check works differently depending on whether the equality is already present in Φ or not. Subtyping rule are also modified similarly.

A.2 First Class Methods

In order allow methods to behave as first class values, we introduce free method types.

$$\begin{array}{llll} \mathit{Type} & T & ::= & \ldots \mid [T]T \Rightarrow T \\ \mathit{ObjTy} & \rho & ::= & \{a : A \ldots\} \\ \mathit{AttrTy} & A & ::= & O \mid T \rightarrow T \mid T \Rightarrow T \end{array}$$

A free method type is the type of a method yet to be attached to an object. Once attached, the corresponding attribute in the object will have a usual method type, and an attached method is still not allowed to escape its object. Note that an object attribute is not allowed to have a free method type. We modify the syntax of a function to have a receiver type hint.

$$e ::= \ldots \mid \texttt{function}\ [T]\ (x : T)\{e\}$$

We need the following modifications to the type system to accomodate free method types. First, we need to add the method expression typing rule. Second, the attribute-update typing judgment needs to expose the type of the argument expression ($R, \mathit{\Gamma} \vdash O.a = e :\ T$). Moreover, the method update rule need not require to have a function expression on the righthand side. The righthand side can be any free method type expression satisfying the contravariant subtyping requirements (including the receiver type).

$$\texttt{[T-Method]}\ \frac{T, \mathit{\Gamma}[x \mapsto T_1] \vdash e : T_2 \qquad \vdash T_1}{R, \mathit{\Gamma} \vdash \texttt{function}[T](x : T_1)\{e\} : [T]T_1 \Rightarrow T_2}$$

$$\texttt{[T-AttrUpdM]}\ \frac{\begin{array}{cccc} O = \rho^q & \rho = \{\ldots a : T_1 \Rightarrow T_2 \ldots\} & & R, \mathit{\Gamma} \vdash e : [O']T_1' \Rightarrow T_2' \\ O' = \rho'^{q'} & q = \mathbf{P}(\texttt{own}, \texttt{iown}) & q' = \mathbf{A}(\texttt{own}') & \texttt{own}' \subseteq \texttt{iown} \\ & \rho <: \rho' & T_1 <: T_1' & T_2' <: T_2 \end{array}}{R, \mathit{\Gamma} \vdash O.a\texttt{=}e : [O']T_1' \Rightarrow T_2'}$$

The attribute-update rule (`[T-AttrUpd]`) needs to be modified to take the result type of the attribute-update typing as the return type of the expression instead of $\top$. Finally, the type equivalence and subtyping definitions need to be extended with the case for free method types. For the subyping rule, the receiver type need to be considered as a contravariant argument.

A.3 The Global Object

In the non-strict mode of JavaScript, the `this` variable returns the global object (e.g., the window object) instead of the undefined value inside a non-method function . This can be supported by passing the global object type in type checking and by using it as a receiver object when a function is invoked.

A.4 Pure Prototype Objects

A common pattern in JavaScript is to create an object which is used only as a prototype. Such objects, which we call as *pure prototype objects*, can have methods that modify attributes not owned by the object. This is legal as long as all the inheriting objects own the attributes being modified and the methods are not called on the prototype object itself. For example, consider the following program:

```
var Animal = { mute:function(){ this.sound = ""; } };
function Cat(){
 this.sound = "Meow";
}
Cat.prototype = Animal;
var cat = new Cat();
cat.mute();
```

In this example, `Animal` is a pure prototype object.

The SJS type system can support pure prototype objects by introducing corresponding pure prototype object-type qualifiers: $\mathbf{PP}(\mathtt{own}, \mathtt{iown})$. Pure prototype objects need to be restricted in two ways. First, since pure prototype objects are not required to own all inheritor owned attributes, invoking a method on them could be unsafe. Therefore, the type system needs to prevent this. Second, pure prototype objects cannot be relaxed through subtyping, because a method can be invoked on the relaxed value, which is still illegal. These restrictions can be imposed by not defining a subtyping rule involving a pure prototype object except for a identity subtyping rule. Apart from these two restrictions, pure prototype objects are first-class values and their methods can be updated because they are not going to be used with subtyping. In the above example, objects `Animal` and `cat` have following types:

$$\begin{array}{ll}\mathtt{Animal} & : \{\mathtt{mute} : \top \Rightarrow \mathit{String}\}^{\mathbf{PP}(\{\mathtt{mute}\},\{\mathtt{sound}\})} \\ \mathtt{cat} & : \{\mathtt{mute} : \top \Rightarrow \mathit{String},\ \ \mathtt{sound} : \mathit{String}\}^{\mathbf{P}(\{\mathtt{sound}\},\{\mathtt{sound}\})}\end{array}$$

Note that the `Animal` object has a pure prototype qualifier, and its inheritor owned attribute set is not a subset of its own attribute set. This is safe because methods of a pure prototype object cannot be invoked. On the contrary, the object `cat`, inheriting `Animal`, meets the usual requirements (i.e, $\mathtt{iown}(\mathtt{cat}) \subseteq \mathtt{own}(\mathtt{cat})$) of a precise object. Therefore, methods of this object can be invoked.

With a simple modification to the preciseness analysis algorithm, the qualifier inference algorithm can support pure prototype qualifiers. The preciseness analysis now need to track preciseness and *impurity* of object-type qualifiers. In this setting, a qualifier is precise if it can be used as a prototype object and if its methods can be updated, and it is impure if it can appear on either sides of a proper subtyping relation or if its method can be invoked. After obtaining a solution from the analysis, we can determine the kind of qualifier for each object as follows: precise and impure objects get precise qualifiers, remaining impure objects get approximate qualifiers, and remaining precise objects get pure prototype qualifiers.

B Dynamic Semantics and Type Safety

This section provides a dynamic semantics of the core language introduced in Section 3, then sketches soundness of the type system with respect to the dynamic semantics.

Operational Semantics: Expression

$$\texttt{[E-Var]}\ \frac{E(x) = l \qquad S(l) = v}{r, E, S \vdash x \Downarrow v, S} \qquad \texttt{[E-VarUpd]}\ \frac{\begin{array}{c} r, E, S \vdash e \Downarrow v, S_1 \\ E(x) = l \qquad S_2 = S_1[l \mapsto v] \end{array}}{r, E, S \vdash x\texttt{=}e \Downarrow v, S_2}$$

$$\texttt{[E-LetVar]}\ \frac{r, E, S \vdash e_1 \Downarrow v_1, S_1 \qquad r, E[x \mapsto l], S_1[l \mapsto v_1] \vdash e_2 \Downarrow v_2, S_2}{r, E, S \vdash \texttt{var}\ x : T\texttt{=}e_1\ \texttt{in}\ e_2 \Downarrow v_2, S_2}\ l \in \mathit{SLoc} \backslash \mathit{dom}(S_1)$$

$$\texttt{[E-This]}\ \frac{}{r, E, S \vdash \texttt{this} \Downarrow r, S} \qquad \texttt{[E-Fun]}\ \frac{S' = S[l \mapsto \texttt{fun}(x, e, E)]}{r, E, S \vdash \texttt{function}(x : T)\{e\} \Downarrow l, S'}\ l \in \mathit{HLoc} \backslash \mathit{dom}(S)$$

$$\texttt{[E-FCall]}\ \frac{\begin{array}{c} r, E, S \vdash e_1 \Downarrow l_1, S_1 \\ r, E, S_1 \vdash e_2 \Downarrow v, S_2 \qquad S_2 \vdash \texttt{app}(\texttt{und}, l_1, v) \Downarrow v', S' \end{array}}{r, E, S \vdash e_1(e_2) : v', S'}$$

$$\texttt{[E-Obj]}\ \frac{\begin{array}{c} r, E, S \vdash e_1 \dots e_n \Downarrow v_1 \dots v_n, S' \\ o \overset{\texttt{let}}{=} \texttt{obj}([a_1 \mapsto v_1,\ \dots\ a_n \mapsto v_n], \texttt{und}) \end{array}}{r, E, S \vdash \{a_1 : e_1 \dots a_n : e_n\}_O \Downarrow l, S'[l \mapsto o]}\ l \in \mathit{HLoc} \backslash \mathit{dom}(S')$$

$$\texttt{[E-Attr]}\ \frac{r, E, S \vdash e \Downarrow l, S' \qquad \mathit{lookup}(S', S'(l), a) = v}{r, E, S \vdash e.a \Downarrow v, S'}$$

$$\texttt{[E-AttrUpd]}\ \frac{\begin{array}{c} r, E, S \vdash e_1 \Downarrow l, S_1 \qquad S_2(l) = o \\ a \in \texttt{own}(o) \qquad r, E, S_1 \vdash e_2 \Downarrow v, S_2 \end{array}}{r, E, S \vdash e_1.a\texttt{=}e_2 \Downarrow v, S_2[l \mapsto o[a \mapsto v]]}$$

$$\texttt{[E-MCall]}\ \frac{\begin{array}{c} r, E, S \vdash e_1 \Downarrow l_1, S_1 \qquad r, E, S_1 \vdash e_2 \Downarrow v, S_2 \\ \mathit{lookup}(S_2, S_2(l_1), a) = l_2 \qquad S_2 \vdash \texttt{app}(l_1, l_2, v) \Downarrow v_2, S_3 \end{array}}{r, E, S \vdash e_1.a(e_2) \Downarrow v_2, S_3}$$

$$\texttt{[E-Proto]}\ \frac{\begin{array}{c} r, E, S \vdash e_1 \dots e_n \Downarrow v_1 \dots v_n, S' \\ r, E, S' \vdash e \Downarrow r', S'' \qquad o \overset{\texttt{let}}{=} \texttt{obj}([a_1 \mapsto v_1,\ \dots\ a_n \mapsto v_n], r') \end{array}}{r, E, S \vdash \{a_1 : e_1 \dots a_n : e_n\}_O\ \texttt{prototype}\ e \Downarrow l, S''[l \mapsto o]}\ l \in \mathit{HLoc} \backslash \mathit{dom}(S'')$$

Operational Semantics: Application

$$\texttt{[E-App]}\ \frac{S(l) = \texttt{fun}(x, e, E) \qquad r, E[x \mapsto l'], S[l' \mapsto v] \vdash e : v', S'}{S \vdash \texttt{app}(r, l, v) \Downarrow v', S'}\ l' \in \mathit{SLoc} \backslash \mathit{dom}(S)$$

Object Lookup

$$\frac{o = \texttt{obj}(am, r) a \in \mathit{dom}(am)}{\mathit{lookup}(S, o, a) = am(a)} \qquad \frac{\begin{array}{c} \qquad\qquad\qquad o = \texttt{obj}(am, l) \\ a \notin \mathit{dom}(am) \qquad S(l) = o' \qquad \mathit{lookup}(S, o', a) = v \end{array}}{\mathit{lookup}(S, o, a) = v}$$

Fig. 9. Operational semantics

B.1 Basic Definitions

$$
\begin{array}{lll}
\mathit{Val} & v & ::= n \mid l \mid \texttt{und} \mid \texttt{err} \\
\mathit{Loc} & l & \in \quad \mathit{SLoc} \cup \mathit{HLoc} \\
\\
\mathit{Env} & E & \in \quad \mathit{Var} \rightarrow \mathit{Loc} \\
\mathit{Store} & S & \in \quad \mathit{Loc} \rightarrow \mathit{StoreVal} \\
\mathit{StoreVal} & sv & \in \quad \mathit{Val} \cup \mathit{Obj} \cup \mathit{Fun} \\
\\
\mathit{Obj} & o & ::= \texttt{obj}(am, r) \\
\mathit{Fun} & f & ::= \texttt{fun}(x, e, E) \\
\\
\mathit{Ref} & r & ::= l \mid \texttt{und} \\
\mathit{AttrMap} & am & \in \quad \mathit{Attr} \rightarrow \mathit{Val}
\end{array}
$$

Fig. 10. Runtime components

Runtime components. Figure 10 provides the definitions of the runtime components. Values consist of: integer values (n), locations (l), the undefined value (`und`), and the error value (`err`). An environment (E) maps variables to locations. A store (S) maps locations to store values. A store value (sv) can be either an object (o) or a function closure (f). A store models both the call stack and heap, and therefore locations are partitioned into stack locations ($\in \mathit{SLoc}$) and heap locations ($\in \mathit{HLoc}$). The distinction is required by the soundness proof. An object (o) is a pair of an attribute map (am) and the location of the prototype object. An attribute map (am) of an object maps the object's own attributes to values. A function closure (f) consists of a parameter variable (x), a function body (e), and an environment (E). The undefined value is a primitive value. Finally, the error value denotes that the program execution is in an undefined state.

Operation on maps. Given a map E, we use $E(x)$ to denote the value of x in E. $E[x \mapsto l]$ creates a new map E', where $E'(y) = E(y)$ for all $y \neq x$ in the domain of E and $E'(x) = l$. We use $E[x_1 \mapsto l_1, \ldots, x_n \mapsto l_n]$ to denote $E[x_1 \mapsto l_1] \ldots [x_n \mapsto l_n]$. We use $\mathit{dom}(E)$ to denote the domain of the map E.

Operation on objects. We also define the following operations on object values:

$$\texttt{obj}(am, r)[a \mapsto v] \equiv \texttt{obj}(am[a \mapsto v], r)$$

$$
\begin{array}{ll}
\mathit{dom}(\texttt{obj}(am, \texttt{und})) & = \mathit{dom}(am) \\
\mathit{dom}(\texttt{obj}(am, o)) & = \mathit{dom}(am) \cup \mathit{dom}(o)
\end{array}
$$

B.2 Operational Semantics.

The big-step operational semantics of the simplified core language is defined in Figure 9. If an undefined case occurs, the program terminates with `err` value indicating that the program is stuck. The operational semantics is defined using two kinds of evaluation judgments. The expression evaluation judgment

$$r, E, S \vdash e \Downarrow v, S'$$

means that expression e evaluates to value v and store S' under reference r to the receiver object, environment E, and store S. For simplicity of exposition, we use a group evaluation judgment $r, E, S \vdash e_1 \ldots e_n \vdash v_1 \ldots v_n, S'$, which is identical to evaluating the expressions in sequence and collecting their results. The application evaluation judgment

$$S \vdash \texttt{app}(r, v, l) \Downarrow v', S'$$

means that the application of the function closure pointed by location l to receiver reference r and argument value v under store S results in the value v' and store S'.

Variables. In SJS, all variables are mutable. This is realized by introducing a level of indirection in environments. When a variable is used in `[E-Var]` (`[E-VarUpd]`), the corresponding location is retrieved from the environment, and the value is retrieved from (or stored to) the store using the location as a key. When a new scope is created in `[E-LetVar]`, a new location l' is created, the environment is extended with a map from the new variable x to l', and the store is extended with a map from l' to value v.

Functions. `[E-Fun]` creates a new function closure and returns the reference to the closure. `[E-FCall]` invokes a closure as a function. `[E-App]` implements the actual function call. The rule receives a reference to a receiver, a pointer to a closure to call, and an argument for the closure. Note that `[E-FCall]` passes `und` as a reference to the receiver object. Therefore, during the execution of the callee, any attempt to access an attribute of the `this` variable will result in a runtime error.

Objects. `[E-Obj]` creates a new object without a prototype. `[E-Attr]` reads an object attribute. *lookup* function implements the actual attribute resolution. *lookup* first searches the current object. If the search fails, *lookup* moves to the prototype object. A *lookup* failure is an undefined behavior and the evaluation gets stuck. `[E-AttrUpd]` updates an object. The rule checks whether the target attribute is owned by the base object. The check makes sure that an object layout is not changed and allows one to prove that a well-typed program has objects with fixed layouts. `[E-MCall]` invokes a closure as a method. The reference to the receiver object is handed to `[E-App]`. Finally, `[E-Proto]` creates a new object with a prototype object.

Deviations from JavaScript operational semantics. The operational semantics of SJS is stricter than the JavaScript semantics. The strictness is introduced to help the type soundness proof, without loss of generality. Three modifications to the operational semantics will bring back the flexibility of the JavaScript: (1) a failed variable lookup should first consult the global object, then return `und`, (2) a failed attribute lookup should return `und`, and (3) an attribute-update operation should not check whether the attribute is owned by the base object. Note that well typed programs will behave equivalently on the strict setting and the relaxed setting.

B.3 Typing Runtime Components

As a second step to formalize the type soundness result, we define typing of runtime components: values, attributes, environments, and stores (Figure 11). Trivial cases, such as typing an integer value, are excluded from the figure.

Note that most judgments are augmented with a store and a store typing (Σ : *Loc* $\rightarrow$ *Type*). The main purpose of store types is to provide precise type information of runtime components to typing judgment.

We expect well-typed runtime components to behave as expected from their types. Among component typing rules, the most important two are `[VT-Obj]` and `[AVT-Me]`.

Safety of updating attributes. `[VT-Obj]` checks that the owned attributes of an object type are actually owned by the object value. This implies that statically checked attribute updates are safe when the static type and the runtime type of an object are equal. The safety of updating objects of an approximate object type also follows because approximate object types always have stricter update permission (`[TS-PA]`, `[TS-AA]`).

Safety of attached methods. `[AVT-Me]` checks whether a method is *safely attached* to the base object (i.e, the method body can be type checked with the type of the current base object), and the check is used by `[VT-Obj]` to check the safety of all attached methods. Type safety of methods, conveyed in the well-typedness of an object value, can then be used to prove the type soundness of a method call.

B.4 Type Augmented Operational Semantics

As a final step to formalize the type soundness result, we augment the operational semantics to maintain a store type through the program execution. We are going to explain this step informally since the type augmented operational semantics is a straight forward modification of the original operational semantics. The augmented operational semantics judgment has following form:

$$\Sigma, r, E, S \vdash e \Downarrow v, S', \Sigma'$$

Value Typing

$$\texttt{[VT-SLoc]}\ \dfrac{\vdash_{S:\Sigma} S(l) : T' \qquad T' <: T \qquad \Sigma(l) = T' \qquad l \in \mathit{SLoc} \qquad S(l) \in \mathit{Val}}{\vdash_{S:\Sigma} l : T}$$

$$\texttt{[VT-HLoc]}\ \dfrac{S(l) \in \mathit{Fun} \cup \mathit{Obj} \quad \Sigma(l) = T \quad l \in \mathit{HLoc} \quad \vdash_{S:\Sigma} S(l) : T}{\vdash_{S:\Sigma} l : T}$$

$$\texttt{[VT-Fun]}\ \dfrac{\vdash_{S:\Sigma} E : \Gamma \qquad \vdash_{S:\Sigma}: \Gamma \qquad \top, \Gamma[x \mapsto T_1] \vdash e : T_2}{\vdash_{S:\Sigma} \texttt{fun}(x, e, E) : T_1 \to T_2}$$

$$\texttt{[VT-Obj]}\ \dfrac{\begin{array}{c} O = \rho^{\mathbf{P}(\texttt{own},\texttt{iown})} \qquad \forall a \in \mathit{dom}(\rho).O \vdash_{S:\Sigma} o.a : \rho(a) \\ o = \texttt{obj}(\mathit{am}, l) \qquad \mathit{dom}(o) = \mathit{dom}(\rho) \qquad \mathit{dom}(\rho) = \texttt{own} \end{array}}{\vdash_{S:\Sigma} o : O}$$

Attribute Value Typing

$$\texttt{[AVT-Fe]}\ \dfrac{T \neq T_1 \Rightarrow T_2 \qquad \mathit{lookup}(S, o, a) = v \qquad T' <:\ \mathit{ty} \qquad \vdash_{S:\Sigma} v : T'}{O \vdash_{S:\Sigma} o.a : T}$$

$$\texttt{[AVT-Me]}\ \dfrac{\begin{array}{c} \mathit{lookup}(S, o, a) = \texttt{fun}(x, e, E) \\ O = \rho^{\mathbf{P}(\texttt{own},\texttt{iown})} \qquad \vdash_{S:\Sigma} E : \Gamma \qquad \rho^{\mathbf{A}(\texttt{iown})}, \Gamma[x \mapsto T_1] \vdash e : T_2 \end{array}}{O \vdash_{S:\Sigma} o.a : T_1 \Rightarrow T_2}$$

Environment Typing

$$\dfrac{\forall x \in \mathit{dom}(E). \vdash_{S:\Sigma} E(x) : \Gamma(x)}{\vdash_{S:\Sigma} E : \Gamma}$$

Store Typing

$$\dfrac{\forall l \in \mathit{dom}(S). \vdash_{S:\Sigma} l : \Sigma(l) \qquad \forall T \in \mathit{range}(\Sigma). \vdash T}{\vdash S : \Sigma}$$

Fig. 11. Runtime-component typing

The new judgment takes store type Σ as an input and generates updated store type Σ'. A store type provides a type level interpretation of a corresponding store (S). Maintaining the soundness of a store type with respect to a corresponding store (i.e, $\vdash S : \Sigma$) is a key to the type soundness proof. During the program execution, a store type is updated when a new location is added to a store. Note that when a new location is introduced, a corresponding type annotation exists most of the time (`[E-LetVar]`, `[E-Fun]`, `[E-Obj]`, and `[E-Proto]`). For example, the `[E-LetVar]` rule and `[E-Obj]` can be extended as follows. The new parts are highlighted:

$$\texttt{[E-LetVar}_{\Sigma}\texttt{]}\ \frac{\begin{array}{c}\Sigma, r, E, S \vdash e_1 \Downarrow v_1, S_1, \Sigma_1 \\ \Sigma_1, r, E[x \mapsto l], S_1[l \mapsto v_1] \vdash e_2 \Downarrow v_2, S_2, \Sigma_2\end{array}}{\Sigma, r, E, S \vdash \texttt{var}\ x : T\texttt{=}e_1\ \texttt{in}\ e_2 \Downarrow v_2, S_2, \Sigma_2[l \mapsto T]}\ l \in SLoc \backslash dom(S_1)$$

$$\texttt{[E-Obj}_{\Sigma}\texttt{]}\ \frac{\begin{array}{c}\Sigma, r, E, S \vdash e_1 \dots e_n \Downarrow v_1 \dots v_n, S', \Sigma' \\ o \overset{\texttt{let}}{=} \texttt{obj}([a_1 \mapsto v_1,\ \dots\ a_n \mapsto v_n], \texttt{und})\end{array}}{\Sigma, r, E, S \vdash \{a_1 : e_1 \dots a_n : e_n\}_O \Downarrow l, S'[l \mapsto o], \Sigma'[l \rightarrow O]}\ l \in HLoc \backslash dom(S')$$

The application evaluation judgment is also modified accordingly ($\Sigma, S \vdash \texttt{app}(r, l, v) \Downarrow v, S', \Sigma'$). In case of the application rule (`[E-App]`), the parameter type is extracted from the function type retrieved from the store type. The receiver object type does not need to be updated because it is already added to the store type, when the object is first created.

$$\texttt{[E-App}_{\Sigma}\texttt{]}\ \frac{\begin{array}{c}S(l) = \texttt{fun}(x, e, E) \qquad \Sigma(l) = T_1 \rightarrow T_2 \\ \Sigma[l' \rightarrow T_1], r, E[x \mapsto l'], S[l' \mapsto v] \vdash e : v', S', \Sigma'\end{array}}{\Sigma, S \vdash \texttt{app}(r, l, v) \Downarrow v', S', \Sigma'}\ l' \in SLoc \backslash dom(S)$$

B.5 Type Soundness

Lemma 1. *(Receiver Subtyping) Assume method body e type checks with object type O as a receiver. Then, the method body can also be type checked with a subtype of O as a receiver, i.e., if* $O, \Gamma \vdash e : T$ *and* $O' <: O$*, then* $O', \Gamma \vdash e : T$

Proof. The proof can be done by induction on the structure of the typing of the expression. □

Lemma 2. *(Preservation) Assume* $\vdash S : \Sigma$, $\vdash_{S:\Sigma} E : \Gamma$, $\vdash_{S:\Sigma} r : R$, *and* $\vdash T$. *If* $R, \Gamma \vdash e : T$ *and* $\Sigma, r, E, S \vdash e \Downarrow v, S', \Sigma'$*, then we have:*

$$\Sigma \leq \Sigma',\ \vdash S' : \Sigma',\ \textit{and}\ \vdash_{S'\Sigma'} v : T'\ \textit{for some}\ T' <: T.$$

The order on type stores ($\leq$*) is defined as:* $\forall l \in dom(\Sigma).\Sigma(l) \equiv \Sigma'(l) \Leftrightarrow \Sigma \leq \Sigma'$.

Proof. The proof can be done by induction on the structure of the evaluation derivation tree. `[E-Num]`, `[E-Var]`, and `[E-This]` are the base cases. Most of the cases follow from the well-typedness of the runtime components and the induction hypothesis. The method call case and the prototype inheritance case are worth mentioning:

- `[E-MCall]`: From the induction hypothesis, the typing of the receiver expression ($e1$), and the evaluation derivation of the receiver expression, we have the well-typedness of the evaluation result of the receiver expression. The well-typedness then provides the typing of the body of the method

([AVT-Me]). The typing of the method body, the induction hypothesis, and the evaluation derivation of the method body, yield the well-typedness of the method call result.

- [E-Proto]: The crux of the proof of the case is to show the type safety of inherited methods. From the method's point of view, the method inheritance is to be bound to a new receiver object whose type is a subtype of the previous receiver object. The safety of rebinding follows from the *Receiver Subtyping* lemma.

□

Theorem 1. *(Type Safety) A well-typed program never gets stuck.*

Proof. The proof follows from the *Preservation* lemma and the fact that the error value (which happens when the evaluation gets stuck) doesn't have a type. □

Theorem 2. *(Fixed Object Layout) A well-typed program never modifies object layouts.*

Proof. The proof follows from the *Type Safety* theorem and the fact that a program modifying an object layout gets stuck under the operational semantics of the core calculus. □

C Type Directed Translation

In this work, we implemented a proof-of-concept compiler from SJS to a low-level language that can bypass expensive hash table based attribute lookups by taking advantage of static object layout information. In this section we discuss the most interesting aspects of the compiler, and the formal definition of the type directed translation from the core language to a lower level language.

C.1 Object Representation

First we note that in the presence of structural subtyping, the offset of a given attribute may not be the same for all subtypes of a given object type. To illustrate this, consider the following example.

Example 1.

```
1: var o1 = { a:"A", b:"B" };
2: var o2 = { b:"B": c:"C" };
3: var o3 = if(*) o1 else o2;
4: console.log(o3.b);
```

The type of `o1`, `o2`, and `o3` are:

$\texttt{o1} : \{\mathsf{a} : \mathit{Str}, \mathsf{b} : \mathit{Str}\}^{\mathbf{P}(\{\mathsf{a},\mathsf{b}\},\{\})}$
$\texttt{o2} : \{\mathsf{b} : \mathit{Str}, \mathsf{c} : \mathit{Str}\}^{\mathbf{P}(\{\mathsf{b},\mathsf{c}\},\{\})}$
$\texttt{o3} : \{\mathsf{b} : \mathit{Str}\}^{\mathbf{A}(\{\mathsf{b}\})}$

Let us assume that we use an array to store the values of the attributes of an object. The representation of object `o1` will then use an array where the first and second cells will store the values of the attributes `a` and `b`, respectively. Similarly, object `o2` will use its first cell to store the value of its attribute `b`. Therefore, attribute `b` will have different offsets in the two objects. At line 4, `o3` can refer to `o1` or `o2`. Therefore, we cannot use a fixed offset to resolve `o3.b` at line 4.

Solution using indirection table. Our solution to resolve this situation is to have objects carry an indirection table that contains the actual attribute offsets. Attribute resolution is done in several steps. At compile time, the attribute name is mapped to an offset into the indirection table. At run time, the indirection table is used to retrieve the offset into the object table, which in turn is used to access the attribute.

- In the above example, objects `o1` and `o2` can be represented as follows:

  ```
  o1 = { tbl:[0,1], obj:["A","B"] }
  o2 = { tbl:[0,1], obj:["B","C"] }
  ```

  The `tbl` attribute stores the indirection table. The `obj` attribute contains the object array. With this representation, the accesses `o1.b` and `o2.b` compile respectively to

  ```
  o1.obj[o1.tbl[1]]
  o2.obj[o2.tbl[0]]
  ```

- Objects get rebundled when coerced to a supertype, reusing the object table with a new indirection table. At line 3, if `o1` gets assigned to `o3`, `o3` will get the following value:

  ```
  o3 = { tbl:[1], obj:["A", "B"] }
  ```

  Note `o3` is a fresh value composed of `o1.obj` and the newly computed indirection table. The original object value `o1` is not affected by the coercion process. If `o2` gets assigned to `o3`, then the value of `o3` will be:

  ```
  o3 = { tbl:[0], obj:["B", "C"] }
  ```

  In either case, `o3.b` at line 4 can be compiled to:

  ```
  o3.obj[o3.tbl[0]]
  ```

C.2 Handling Prototype-Based Inheritance

The above compilation scheme will not work if an object accesses an attribute defined in its prototype. The value of an attribute defined in a prototype is shared by all of its inheritors (for simplicity let us assume that the attribute is not overridden by an inheritor). A change in the attribute value should be visible to all inheritor objects.

To support this sharing mechanism via prototypes, we add another level of indirection. We modify object arrays to store references to values, instead of the values themselves. With this modification, prototype-based inheritance can be implemented by simply copying the contents of the prototype's object array to its inheritors' object arrays. Consider the following example:

```
1: var o4 = { a:10 };
2: var o5 = { b:20 } prototype o4;
3: o5.a;
```

In this program, `o5.a` accesses `o4`'s `a` attribute. Any modification to `o4.a` should be visible from `o5`. Using the modified representation, this program compiles to:

```
let o4 = { tbl:[0], obj:[ref 10] } in
let o5 = { tbl:[0,1], obj:[ o4.obj[o4.tbl[0]], ref 20 ] } in
! (o5.obj[o5.tbl[0]]);
```

where the operator `ref` allocates a new reference cells initialized to the given value, and the operator "`!`" reads the contents of a reference cell. Note that this representation is possible only because the core language programs have fixed object layouts. If, for example, we would allow an update of `o5.a` then the source JavaScript program and the compiled version would have different semantics.

C.3 Type Directed Translation

Target Language

$$\underline{e} ::= n \mid \lambda x.\underline{e} \mid \underline{ee} \mid x \mid \texttt{let}\ x = \underline{e}\ \texttt{in}\ \underline{e} \mid \texttt{ref}\ \underline{e} \mid \texttt{!}\underline{e} \mid \underline{e} := \underline{e} \mid [\underline{e}, \ldots, \underline{e}] \mid \underline{e}[\underline{e}] \mid e; e$$

Target language is a call-by-value lambda calculus with integers, let-bindings, references, and immutable tuples. We use underline ($\underline{e}$) to distinguish target language expressions from source language expressions. $\texttt{ref}\underline{e}$ allocates a new memory cell, initializes the cell with the evaluation result of $\underline{e}$, and returns the address of the cell. $\texttt{!}\underline{e}$ is a dereferencing. $\underline{e} := \underline{e}$ is a reference update. $[e, \ldots, e]$ defines a new tuple and $e[e]$ accesses an element of the tuple by an index. $e; e$ is a sequence expression, which executes subexpressions in a sequence and returns the result of the last subexpression. Essentially, the target language is a subset of the ML language, for which a compilation strategy to efficient machine code is well known. In this paper, we omit the operational semantics and the type system of the target language.

Auxiliary Definitions

Type tagging. We are going to assume the existence of a mapping from types to distinctive integer tags, and refer it by $tag(T)$.

Syntactic Sugars. For the clarity of the description, we define two kind of record expressions ($\{\texttt{obj} : e, \texttt{tbl} : e, \texttt{ty} : e\}$ and $\{\texttt{fun} : e, \texttt{rTy} : e\}$ respectively) and corresponding record field access expression which are a syntactic sugar defined in terms of tuple expressions. A first kind of tuple is used to represent an object value as explained before, with the additional field `ty` (the runtime type tag of the object). A second kind of tuple represents a method value, of which the second field `rTy` indicates the static receiver type of the method.

Ordering. The order function ($Order(O, a) = n$) determines the offset of attributes of a given object type. We assume that (1) the function is injective and (2)given an object, the range of the function and the set of attributes of the object have the same cardinality. Any function satisfying two properties can be used in theory. In this section, we are going to abuse the order function to also accept a type tag, instead of a type, as the first argument.

Computing indirection table. *GetTbl* function computes an indirection table receiving the runtime type of an object and the static type of the object. Like the order function, we are going to abuse *GetTbl* function to accept type tags as arguments, instead of types. Following is the specification of *GetTbl* function.

$$\frac{\begin{array}{c} O_r <: O_s \\ |dom(O_s)| = k \qquad \forall i \in [1,k].\exists a.Order(O_s, a) = i \wedge Order(O_r, a) = n_i \end{array}}{GetTable(O_r, O_s) = [n_1, ..., n_k]}$$

Compilation Figure 12 defines type directed translation. Type directed expression compilation judgment

$$R, \Gamma \vdash e : T \leadsto \underline{e}$$

means that expression e of type T translates to expression $\underline{e}$. We assumes that program is well typed and annotated with type information. We also use simpler form $e \leadsto \underline{e}$ when type information is not necessary. Subjudgment

$$R, \Gamma \vdash O.a = e \leadsto \underline{e}$$

translates the argument expression of an attribute update expression. Another judgment, subtyping compilation judgment

$$T_1 <: T_2 \leadsto \underline{e}$$

creates a coercion function from the current runtime type to the target type. The subtype part (T_1) is not used during the compilation. Since the type directed translation rules are based on the typing rules, Figure 12 only shows parts updated from Figure 7.

Variables are compiled to a variable containing a reference (`[C-Fun]`, `[C-AttrUpdM]`). Variable lookups and updates become a derefencing and a reference update (`[C-Upd]`, `[C-Var]`). Functions are compiled into a lambda function. Methods

Expression Compilation

$$\texttt{[C-Num]}\ \frac{}{R,\mathit{\Gamma} \vdash n : \mathit{Int} \rightsquigarrow n} \qquad \texttt{[C-Var]}\ \frac{\ldots}{R,\mathit{\Gamma} \vdash x : T \rightsquigarrow !x}$$

$$\texttt{[C-Let]}\ \frac{R,\mathit{\Gamma} \vdash e_1 : T_1 \rightsquigarrow \underline{e_1} \qquad T_1 <: T \rightsquigarrow \underline{e_s} \qquad R,\mathit{\Gamma}[x:T] \vdash e_2 : T_2 \rightsquigarrow \underline{e_2}}{R,\mathit{\Gamma} \vdash \texttt{let}\ \ x : T = e_1\ \ \texttt{in}\ \ e_2 : T_2 \rightsquigarrow \texttt{let}\ \ x = \texttt{ref}\ \ \underline{\underline{e_s}(\underline{e_1})}\ \ \texttt{in}\ \ \underline{e_2}}$$

$$\texttt{[C-Upd]}\ \frac{\ldots \qquad R,\mathit{\Gamma} \vdash e : T_2 \rightsquigarrow \underline{e} \qquad T_2 <: T_1 \rightsquigarrow \underline{e_s}}{R,\mathit{\Gamma} \vdash x\texttt{=}e : T_1 \rightsquigarrow \texttt{let}\ \ t = \underline{e}\ \ \texttt{in}\ \ (x := t\ ;\ \underline{e_s}(t))}\ \text{fresh}t \qquad \texttt{[C-This]}\ \frac{}{O,\mathit{\Gamma} \vdash \texttt{this} : O \rightsquigarrow !\texttt{this}}$$

$$\texttt{[C-Fun]}\ \frac{\ldots \qquad R,\mathit{\Gamma}[x \mapsto T_1] \vdash e : T_2 \rightsquigarrow \underline{e}}{R,\mathit{\Gamma} \vdash \texttt{function}(x:T_1)\{e\} : T_1 \to T_2 \rightsquigarrow \lambda x'.\texttt{let}\ \ x = \texttt{ref}\ \ x'\ \ \texttt{in}\ \ \underline{e}}\ \text{fresh } x'$$

$$\texttt{[C-App]}\ \frac{R,\mathit{\Gamma} \vdash e_1 : T_1 \to T_2 \rightsquigarrow \underline{e_1} \qquad R,\mathit{\Gamma} \vdash e_2 : T_3 \rightsquigarrow \underline{e_2} \qquad T_3 <: T_1 \rightsquigarrow \underline{e_s}}{R,\mathit{\Gamma} \vdash e_1(e_2) : T_2 \rightsquigarrow \underline{e_1}(\underline{e_s}(\underline{e_2}))}$$

$$\texttt{[C-Obj]}\ \frac{\ldots \qquad \forall i \in [1,n].R,\mathit{\Gamma} \vdash_{\!AU} O.a_i = e_i \rightsquigarrow \underline{e_i} \wedge \mathit{Order}(O,a_i) = k_i \wedge \underline{e'_{k_i}} = \texttt{ref}\ \underline{e_i}}{R,\mathit{\Gamma} \vdash \{a_1 : e_1, \ldots, a_n : e_n\}_O : O \rightsquigarrow \{\texttt{obj} : [\underline{e'_1},\ \ldots \underline{e'_n}], \texttt{tbl} : [0,\ldots,n-1], \texttt{ty} : \mathit{Tag}(O)\}}$$

$$\texttt{[C-AttrUpd]}\ \frac{\ldots \qquad R,\mathit{\Gamma} \vdash e_1 : O \rightsquigarrow \underline{e_1} \qquad R,\mathit{\Gamma} \vdash_{\!AU} O.a = e_2 \rightsquigarrow \underline{e_2} \qquad \mathit{Order}(O,a) = n}{R,\mathit{\Gamma} \vdash e_1.a := e_2 : \top \rightsquigarrow \texttt{let}\ \ x = \underline{e_1}\ \ \texttt{in}\ \ x.\texttt{obj}[x.\texttt{tbl}[n]] := \underline{e_2}}\ \text{fresh } x$$

$$\texttt{[C-Attr]}\ \frac{\ldots \qquad R,\mathit{\Gamma} \vdash e : O \rightsquigarrow \underline{e} \qquad \mathit{Order}(O,a) = n}{R,\mathit{\Gamma} \vdash e.a : T \rightsquigarrow \texttt{let}\ \ x = \underline{e}\ \ \texttt{in}\ \ !(x.\texttt{obj}[x.\texttt{tbl}[n]])}\ \text{fresh } x$$

$$\texttt{[C-MCall]}\ \frac{\begin{array}{c} \ldots \qquad O(a) = T_1 \Rightarrow T_2 \\ R,\mathit{\Gamma} \vdash e_1 : O \rightsquigarrow \underline{e_1} \qquad R,\mathit{\Gamma} \vdash e_2 : T_3 \rightsquigarrow \underline{e_2} \qquad T_3 <: T_1 \rightsquigarrow \underline{e_s} \qquad \mathit{Order}(O,a) = n \\ \underline{e_{obj}} = \{\texttt{obj} : x.\texttt{obj},\ \texttt{tbl} : \mathit{GetTbl}(x.\texttt{ty}, y.\texttt{rTy}),\ x.\texttt{ty}\} \end{array}}{R,\mathit{\Gamma} \vdash e_1.a(e_2) : T_2 \rightsquigarrow \texttt{let}\ \ x = \underline{e_1},\ y = !(x.\texttt{obj}[x.\texttt{tbl}[n]]),\ z = \underline{e_{obj}}\ \ \texttt{in}\ \ y.\texttt{fun}(z)(\underline{e_2})}\ \text{fresh } x,y,z$$

$$\texttt{[C-Proto]}\ \frac{\begin{array}{c} \ldots \qquad R,\mathit{\Gamma} \vdash e_p : O_p \rightsquigarrow \underline{e_p} \qquad |\mathit{dom}(O)| = m \\ \forall i \in [1,n].R,\mathit{\Gamma} \vdash_{\!AU} O.a_i = e_i \rightsquigarrow \underline{e_i} \wedge \mathit{Order}(O,a_i) = j_i \wedge \underline{e'_{j_i}} = \texttt{ref}\ \underline{e_i} \\ \forall a \in (\mathit{dom}(O) \setminus \texttt{own}(O)).\mathit{Order}(O,a) = k \wedge \mathit{Order}(O_p,a) = l \wedge \underline{e'_k} = \texttt{ref}\ x.\texttt{obj}[x.\texttt{tbl}[l]] \\ \underline{e_{obj}} = \{\texttt{obj} : [\underline{e'_1}, \ldots \underline{e'_m}], \texttt{tbl} : [0,\ldots,n-1], \texttt{ty} : \mathit{Tag}(O)\} \end{array}}{R,\mathit{\Gamma} \vdash \{a_1 : e_1 \ldots a_n : e_n\}_O\ \ \texttt{prototype}\ \ e_p : O \rightsquigarrow \texttt{let}\ \ x = \underline{e}\ \ \texttt{in}\ \ \underline{e_{obj}}}\ \text{fresh } x$$

Attribute Update Compilation

$$\texttt{[C-AttrUpdV]}\ \frac{O(a) = T \qquad T \neq T_1 \Rightarrow T_2 \qquad R,\mathit{\Gamma} \vdash e : T' \rightsquigarrow \underline{e} \qquad T' <: T \rightsquigarrow \underline{e_s}}{R,\mathit{\Gamma} \vdash_{\!AU} O.a = e \rightsquigarrow \underline{e_s}(\underline{e})}$$

$$\texttt{[C-AttrUpdM]}\ \frac{\begin{array}{c} O = \rho^{\mathbf{P}(\texttt{own},\texttt{iown})} \qquad O' = \rho^{\mathbf{A}(\texttt{iown}')} \qquad O(a) = T_1 \Rightarrow T_2 \qquad O',\mathit{\Gamma}[x \mapsto T_1] \vdash e : T_2 \rightsquigarrow \underline{e} \\ \underline{e_m} = \lambda\texttt{this}'.\lambda x'.\texttt{let this}' = \texttt{ref this},\ x = \texttt{ref}\ \ x'\ \ \texttt{in}\ \ \underline{e} \end{array}}{R,\mathit{\Gamma} \vdash_{\!AU} O.a = \texttt{function}(x)e \rightsquigarrow \{\texttt{fun} : \underline{e_m},\ \texttt{rTy} : \mathit{Tag}(O')\}}\ \text{fresh } x'$$

Subtyping Compilation

$$\texttt{[C-ObjPA}_{<:}\texttt{]},\ \texttt{[C-ObjAA}_{<:}\texttt{]}\ \frac{\ldots}{O_1 <: O_2 \rightsquigarrow \lambda x.\{\texttt{obj} : x.\texttt{obj},\ \texttt{tbl} : \mathit{GetTable}(x.\texttt{ty}, O_2),\ \texttt{ty} : x.\texttt{ty}\}}\ \text{fresh } x$$

$$\texttt{[C-Trans}_{<:}\texttt{]}\ \frac{T_1 <: T_2 \rightsquigarrow \underline{e_{12}} \qquad T_2 <: T_3 \rightsquigarrow \underline{e_{23}}}{T_1 <: T_3 \rightsquigarrow \underline{e_{23}} \circ \underline{e_{12}}} \qquad \texttt{[C-Refl}_{<:}\texttt{]}\ \frac{}{T <: T \rightsquigarrow \lambda x.x}\ \text{fresh } x$$

$$\texttt{[C-Fun}_{<:}\texttt{]}\ \frac{T_3 <: T_1 \rightsquigarrow \underline{e_a} \qquad T_2 <: T_4 \rightsquigarrow \underline{e_r}}{T_1 \to T_2 <: T_3 \to T_4 \rightsquigarrow \lambda x.\underline{e_r} \circ x \circ \underline{e_a}}\ \text{fresh } x \qquad \texttt{[C-Top}_{<:}\texttt{]}\ \frac{}{T <: \top \rightsquigarrow \lambda x.x}\ \text{fresh } x$$

Fig. 12. Type Directed Compilation

are represented as a tuple composed of a function value with two explicit parameters (the receiver and the actual parameter) and a receiver type (`[C-AttrUpdM]`). The receiver type information is used to coerce an actual receiver object argument passed to the method to the expected receiver type. Coercion is necessary because that a receiver object may have the runtime type different from the expected type (`[C-MCall]`). x.`fun` and x.`rty` each indicates the function value component and the receiver type of the method value.

Object representations are handled as described in previous subsections. (`[C-Obj]`, `[C-Attr]`, `[C-AttrUpd]`, `[C-Proto]`). A runtime type tag is attached to perform object type coercion ($\texttt{[C-ObjPA}_{<:}\texttt{]}$, $\texttt{[C-ObjAA}_{<:}\texttt{]}$). Note object type coercion functions generated by the object subtyping rules invoke *GetTbl* function when executed, which might require hashtable lookup. However, these are the only points where a hashtable is involved.