Type Inference for Static Compilation of JavaScript

Satish Chandra* Colin S. Gordon† Jean-Baptiste Jeannin* Cole Schlesinger*
Manu Sridharan* Frank Tip‡ Youngil Choi§

*Samsung Research America, USA
{schandra,jb.jeannin,cole.s,m.sridharan}@samsung.com

†Drexel University, USA
csgordon@cs.drexel.edu

‡Northeastern University, USA
f.tip@northeastern.edu

§Samsung Electronics, South Korea
duddlf.choi@samsung.com

Abstract

We present a type system and inference algorithm for a rich subset of JavaScript equipped with objects, structural subtyping, prototype inheritance, and first-class methods. The type system supports abstract and recursive objects, and is expressive enough to accommodate several standard benchmarks with only minor workarounds. The invariants enforced by the types enable an ahead-of-time compiler to carry out optimizations typically beyond the reach of static compilers for dynamic languages. Unlike previous inference techniques for prototype inheritance, our algorithm uses a combination of lower and upper bound propagation to infer types and discover type errors in *all* code, including uninvoked functions. The inference is expressed in a simple constraint language, designed to leverage off-the-shelf fixed point solvers. We prove soundness for both the type system and inference algorithm. An experimental evaluation showed that the inference is powerful, handling the aforementioned benchmarks with no manual type annotation, and that the inferred types enable effective static compilation.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors

Keywords object-oriented type systems, type inference, JavaScript

1. Introduction

JavaScript is one of the most popular programming languages currently in use [6]. It has become the de facto standard in web programming, and its growing use in large-scale, real-world applications—ranging from servers to embedded devices—has sparked significant interest in JavaScript-focused program analyses and type systems in both the academic research community and in industry.

In this paper, we report on a type inference algorithm for JavaScript developed as part of a larger, ongoing effort to enable type-based *ahead-of-time* compilation of JavaScript programs. Ahead-of-time compilation has the potential to enable lighter-weight execution, compared to runtimes that rely on just-in-time optimizations [5, 9], without compromising performance. This is particularly relevant for resource-constrained devices such as mobile phones where both performance and memory footprint are important. Types are key to doing effective optimizations in an ahead-of-time compiler.

JavaScript is famously dynamic; for example, it contains `eval` for runtime code generation and supports introspective behavior, features that are at odds with static compilation. Ahead-of-time compilation of unrestricted JavaScript is *not* our goal. Rather, our goal is to compile a subset that is rich enough for idiomatic use by JavaScript developers. Although JavaScript code that uses highly dynamic features does exist [39], data shows that the majority of application code does not require that flexibility. With growing interest in using JavaScript across a range of devices, including resource-constrained devices, it is important to examine the tradeoff between language flexibility and the cost of implementation.

The JavaScript compilation scenario imposes several desiderata for a type system. First, the types must be *sound*, so they can be relied upon for compiler transformations. Second, the types must impose enough restrictions to allow the compiler to generate code with good, predictable performance for core language constructs (Section 2.1 discusses some of these optimizations). At the same time, the system must be expressive enough to type check idiomatic coding patterns and make porting of mostly-type-safe JavaScript code easy. Finally, in keeping with the nature of the language, as well as to ease porting of existing code, we desire powerful type inference. To meet developer expectations, the inference must infer types and discover type errors in *all* code,

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984017

including uninvoked functions from libraries or code under development.

No existing work on JavaScript type systems and inference meets our needs entirely. Among the recently developed type systems, TypeScript [8] and Flow [3] both have rich type systems that focus on programmer productivity at the expense of soundness. TypeScript relies heavily on programmer annotations to be effective, and both treat inherited properties too imprecisely for efficient code generation. Defensive JavaScript [15] has a sound type system and type inference, and Safe TypeScript [36] extends TypeScript with a sound type system, but neither supports prototype inheritance. TAJS [29] is sound and handles prototype inheritance precisely, but it does not compute types for uninvoked functions; the classic work on type inference for SELF [12] has the same drawback. Choi *et al.* [21] present a JavaScript type system targeting ahead-of-time compilation that forms the basis of our type system, but their work does not have inference and instead relies on programmer annotations. See Section 6 for further discussion of related work.

This paper presents a type system and inference algorithm for a rich subset of JavaScript that achieves our goals. Our type system builds upon that of Choi et al. [21], adding support for abstract objects, first-class methods, and recursive objects, each of which we found crucial for handling real-world JavaScript idioms; we prove these extensions sound. The type system supports a number of additional features such as polymorphic arrays, operator overloading, and intersection types in manually-written interface descriptors for library code, which is important for building GUI applications.

Our type inference technique builds on existing literature (e.g., [12, 23, 35]) to handle a complex combination of language features, including structural subtyping, prototype inheritance, first-class methods, and recursive types; we are unaware of any single previous type inference technique that soundly handles these features in combination.

We formulate type inference as a constraint satisfaction problem over a language composed primarily of subtype constraints over standard row variables. Our formulation shows that various aspects of the type system, including source-level subtyping, prototype inheritance, and attaching methods to objects, can all be reduced to these simple subtype constraints. Our constraint solving algorithm first computes lower and upper bounds of type variables through a propagation phase (amenable to the use of efficient, off-the-shelf fixed-point solvers), followed by a straightforward error-checking and ascription phase. Our use of both lower *and* upper bounds enables type inference and error checking for uninvoked functions, unlike previous inference techniques supporting prototype inheritance [12, 29]. As shown in Section 2.3, sound inference for our type system is non-trivial, particularly for uninvoked functions; we prove that our inference algorithm is sound.

Leveraging inferred types, we have built a backend that compiles type-checked JavaScript programs to optimized native binaries for both PCs and mobile devices. We have compiled slightly-modified versions of six of the Octane benchmarks [1], which ranged from 230 to 1300 LOC, using our compiler. The modifications needed for the programs to type check were minor (see Section 5 for details).

Preliminary data suggests that for resource-constrained devices, trading off some language flexibility for static compilation is a compelling proposition. With ahead-of-time compilation (AOTC), the six Octane programs incurred a *significantly* smaller memory footprint compared to running the same JavaScript sources with a just-in-time optimizing engine. The execution performance is not as fast as JIT engines when the programs run for a large number of iterations, but is acceptable otherwise, and vastly better than a non-optimizing interpreter (details in Section 5).

We have also created six GUI-based applications for the Tizen [7] platform, reworking from existing web applications; these programs ranged between 250 to 1000 lines of code. In all cases, all types in the user-written JavaScript code were inferred, and *no* explicit annotations were required. We do require annotated signatures of library functions, and we have created these for many of the JavaScript standard libraries as well as for the Tizen platform API. Experiences with and limitations of our system are discussed in Section 5.

Contributions:

- We present a type system, significantly extending previous work [21], for typing common JavaScript inheritance patterns, and we prove the type system sound. Our system strikes a useful balance between allowing common coding patterns and enabling ahead-of-time compilation.
- We present an inference algorithm for our type system and prove it sound. To our best knowledge, this algorithm is the first to handle a combination of structural subtyping, prototype inheritance, abstract types, and recursive types, while also inferring types for uninvoked functions. This inference algorithm may be of independent interest, for example, as a basis for software productivity tools.
- We discuss our experiences with applying an ahead-of-time compiler based on our type inference to several existing benchmarks. We found that our inference could infer all the necessary types for these benchmarks automatically with only slight modifications. We also found that handling a complex combination of type system features was crucial for these programs. Experimental data points to the promise of ahead-of-time compilation for running JavaScript on resource-constrained devices.

2. Overview

Here we give an overview of our type system and inference. We illustrate some requirements and features of typing and type inference by way of a simple example. We also highlight

```
1  var v1 = { d : 1, // o1
2        m : function (x) { this.a = x + this.d }}
3  var v2 = { a : 2 } proto v1; // o2
4  v2.m(3);
5  v2.m("foo"); // type error in our system
6  var v3 = { b : 4 } proto v2; // o3
7  v3.m(4);      // type error in our system
```

Figure 1: An example program to illustrate our type system.

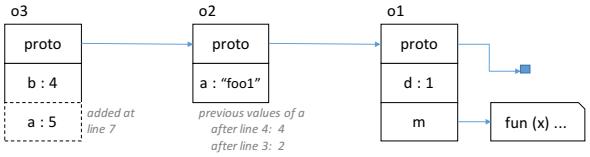


Figure 2: Runtime heap for Figure 1 at line 6.

some challenges in inference, and show in more detail why previous techniques are insufficient for our needs.

2.1 Type System Requirements

Our type system prevents certain dynamic JavaScript behaviors that can compromise performance in an AOTC scenario. In many cases, such behaviors also reflect latent program bugs. Consider the example of Figure 1. We refer to the object literals as `o1`, `o2`, and `o3`. To keep our examples readable, we use a syntactic sugar for prototype inheritance: the expression `{a : 2} proto o1` makes `o1` the prototype parent of the `{a: 2}` object, corresponding to the following JavaScript:

```
function C() { this.a = 2 } // constructor
C.prototype = o1; new C()
```

In JavaScript, the `v2.m("foo")` invocation (line 5) runs without error, setting `v2.a` to `"foo1"`. In SJS, we do not allow this operation, as `v2.a` was initialized to an integer value; such restrictions are standard with static typing.

JavaScript field accesses also present a challenge for AOTC. Figure 2 shows a runtime heap layout of the three objects allocated in Figure 1 (after line 6). In JavaScript, a field read `x.f` first checks `x` for field `f`, and continues up `x`'s prototype chain until `f` is found. If `f` is not found, the read evaluates to `undefined`. Field writes `x.f = y` are peculiar. If `f` exists in `x`, it is updated in place. If not, `f` is created in `x`, *even if* `f` is available up the prototype chain. This peculiarity is often a source of bugs. For our example, the write to `this.a` within the invocation `v3.m(4)` on line 7 creates a new slot in `o3` (dashed box in Figure 2), rather than updating `o2.a`.

Besides being a source of bugs, this field write behavior prevents a compiler from optimizing field lookups. If the set of fields in every object is fixed at the time of allocation—a *fixed object layout* [21]—then the compiler can use a constant indirection table for field offsets.[1] Fixed layout also establishes the availability of fields for reading / writing, obviating the need for runtime checks.[2]

In summary, our type system must enforce the following properties:

- **Type compatibility**, e.g., integer and string values cannot be assigned to the same variable.
- **Access safety** of object fields: fields that are neither available locally nor in the prototype chain cannot be read; and fields that are not locally available cannot be written.

These properties promote good programming practices and make code more amenable to compilation. Note that detection of errors that require flow-sensitive reasoning, like `null` dereferences, is out of scope for our type system; extant systems like TAJS [29] can be applied to find such issues.

2.2 The Type System

Access safety. In our type system, the fields in an object type O are maintained as two rows (maps from field names to types), O^{r} for *readable* fields and O^{w} for *writeable* fields. Readable fields are those present either locally or in the prototype chain, while writeable fields must be present locally (and hence must also be readable). Since `o1` in Figure 1 only has local fields d and m, we have $O_1^{\mathsf{r}} = O_1^{\mathsf{w}} = \langle d, m \rangle$.[3] For `o2`, the readable fields O_2^{r} include local fields $\langle a \rangle$ and fields $\langle d, m \rangle$ inherited from `o1`, so we have $O_2^{\mathsf{r}} = \langle d, m, a \rangle$ and $O_2^{\mathsf{w}} = \langle a \rangle$. Similarly, $O_3^{\mathsf{r}} = \langle d, m, a, b \rangle$ and $O_3^{\mathsf{w}} = \langle b \rangle$. The type system rejects writes to read-only fields; e.g., `v2.d = 2` would be rejected.

Detecting that the call `v3.m(4)` on line 7 violates access safety is less straightforward. To handle this case, the type system tracks two additional rows for certain object types: the fields that attached methods may read (O^{mr}), and those that methods may write (O^{mw}). The typing rules ensure that such *method-accessed fields* for an object type include the fields of the receiver types for all attached methods. Let T_m be the receiver type for method `m` (line 2). Based on the uses of `this` within `m`, we have $T_m^{\mathsf{r}} = \langle d, a \rangle$ and $T_m^{\mathsf{w}} = \langle a \rangle$ (again, writeable fields must be readable). Since `m` is the only method attached to `o1`, we have $O_1^{\mathsf{mr}} = T_m^{\mathsf{r}} = \langle d, a \rangle$ and $O_1^{\mathsf{mw}} = T_m^{\mathsf{w}} = \langle a \rangle$. Since `o2` and `o3` inherit `m` and have no other methods, we also have $O_3^{\mathsf{mr}} = O_2^{\mathsf{mr}} = \langle d, a \rangle$ and $O_3^{\mathsf{mw}} = O_2^{\mathsf{mw}} = \langle a \rangle$.

With these types, we have `a` $\in O_3^{\mathsf{mw}}$ and `a` $\notin O_3^{\mathsf{w}}$: i.e., a method of O_3 can write field field `a`, which is not locally present. Hence, the method call `v3.m(4)` is unsafe.

[1] The compiler may even be able to allocate a field in the same position in all containing objects, eliminating the indirection table.

[2] When dynamic addition and deletion of fields is necessary, a map rather than an object is more suited; see Section 5.1.

[3] For brevity, we elide the field types here, as the discussion focuses on which fields are present.

The type system considers O_3 to be *abstract*, and method invocations on abstract types are rejected. (Types for which method invocations are safe are *concrete*.) Similarly, O_1 is also abstract. Note that rejecting abstract types completely is too restrictive: JavaScript code often has prototype objects that are abstract, with methods referring to fields declared only in inheritors.

The idea of tracking method-accessed fields follows the type system of Choi et al. [20, 21], but they did not distinguish between mr and mw, essentially placing all accessed fields in mw. Their treatment would reject the safe call at line 4, whereas with mr, we are able to type it.[4]

Subtyping. A type system for JavaScript must also support structural subtyping between object types to handle common idioms. But, a conflict arises between structural subtyping and tracking of method-accessed fields. Consider the following code:

```
1 p = cond()
2     ? { m : fun() { this.f = 1 }, f: 2 } // o1
3     : { m : fun() { this.g = 2 }, g: 3 } // o2
4 p.m();
```

Both `o1` and `o2` have concrete types, as they contain all fields accessed by their methods. Since `m` is the only common field between `o1` and `o2`, by structural subtyping, `m` is the only field in the type of `p`. But what should the method-writeable fields of `p` be? A sound approach of taking the union of such fields from `o1` and `o2` yields $\langle f, g \rangle$. But, this makes the type of `p` abstract (neither f nor g is present in `p`), prohibiting the safe call of `p.m()`.

To address this issue, we adopt ideas from previous work [21, 33] and distinguish *prototypal types*, suitable for prototype inheritance, from *non-prototypal types*, suitable for structural subtyping. Non-prototypal types elide method-accessed fields, thereby avoiding bad interactions with structural subtyping. For the example above, we can assign `p` a non-prototypal concrete type, thereby allowing the `p.m()` call. However, an expression `{...} proto p` would be disallowed: without method-accessed field information for `p`, inheritance cannot be soundly handled. For further details, see Section 3.3.

2.3 Inference Challenges

As noted in Section 1, we found that no extant type inference technique was suitable for our needs. The closest techniques are those that reason about prototype inheritance precisely, like type inference for SELF [12] and the TAJS system for JavaScript [29]. Both of these systems work by tracking which values may flow to an operation (a "top-down" approach), and then ensuring the operation is legal for those values. They also gain significant scalability by only analyzing reachable code, as determined by the analysis itself.

But, this approach cannot infer types or find type errors in *unreachable* code, e.g., a function under development that is not yet invoked. Consider this example:

```
1 function f(x) {
2   var y = -x;
3   return x[1];
4 }
5 f(2);
```

Without the final call `f(2)`, the previous techniques would not find the (obvious) type error within `f`. This limitation is unacceptable, as developers expect a compiler to report errors in *all* code.

An alternative inference approach is to compute types based on how variables/expressions are used (a "bottom-up" approach), and then check any incoming values against these types. Such an approach is standard in unification-style inference algorithms, combined with introduction of parametric polymorphism to generalize types as appropriate [26]. Unfortunately, since our type system has subtyping, we cannot apply such unification-based techniques, nor can we easily infer parametric polymorphism.

Instead, our inference takes a hybrid approach, tracking value flow in *lower bounds* of type variables and uses in *upper bounds*. Both bounds are *sets* of types, and the final ascribed type must be a subtype of all upper bound types and a supertype of all lower bound types. Upper bounds enable type inference and error discovery for uninvoked functions, e.g., discovery of the error within `f` above.

If upper bounds alone under-constrain a type, lower bounds provide a further constraint to inform ascription. For example, given the identity function `id(x) { return x; }`, since no operations are performed on `x`, upper bounds give no information on its type. However, if there is an invocation `id("hi")`, inference can use the lower bound information from `"hi"` to ascribe the type string $\rightarrow$ string. Note that as in other systems [3, 8], we could combine our inference with checking of user-written polymorphic types, e.g., if the user provided a type $T \rightarrow T$ (T is a type variable) for `id`.

Once all upper and lower bounds are computed, an assignment needs to be made to each type variable. A natural choice is the greatest lower bound of the upper bounds, with a further check that the result is a supertype of all lower bound types. However, if upper and lower bounds are based solely on value flow and uses, type variables can be *partially* constrained, with $\emptyset$ as the upper bound (if there are no uses) or the lower bound (if no values flow in). In the first case, since our type system does not include a top type, [5] it is not clear what assignment to make. This is usually not a concern in unification-based analyses, which flow information across assignments symmetrically, but it *is* an issue in subtyping-based analyses such as ours.

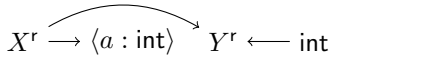
[4] Throughout the paper, we call out extensions we made to enhance the power of Choi et al.'s type system.

[5] We exclude $\top$ from the type system to detect more errors; see discussion in Section 4.2.

Particular care thus needs to be taken to soundly assign type variables whose upper bound is empty. A sound choice would be to simply fail in inference, but this would be too restrictive. We could compute an assignment based on the lower bound types, e.g., their least upper bound. But this scheme is unsound, as shown by the following example:

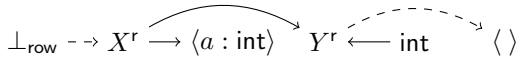
```
function f(x) {
  var y = x; y = 2; return x.a+1;
}
```

Assume `f` is uninvoked. Using a graphical notation (edges reflect subtyping), the relevant constraints for this code are:



`x` has no incoming value flow, but it is used as an object with an integer a field (shown as the $X^{\mathsf{r}} \longrightarrow \langle a : \mathsf{int} \rangle$ edge). For `y`, we see no uses, but the integer `2` flows into it (shown as the $Y^{\mathsf{r}} \longleftarrow \mathsf{int}$ edge). A technique based solely on value flow and uses would compute the upper bound of X^{r} as $\{\langle a : \mathsf{int} \rangle\}$, the lower bound of Y^{r} as $\{\mathsf{int}\}$, and the lower bound of X^{r} and upper bound of Y^{r} as $\emptyset$. But, ascribing types based on these bounds would be *unsound*: they do not capture the fact that if `x` is ascribed an object type, then `y` must also be an object, due to the assignment `y = x`.

Instead, our inference *strengthens* lower bounds based on upper bounds, and vice-versa. For the above case, bound strengthening yields the following constraints (edges due to strengthening are dashed):



Given the type $\langle a : \mathsf{int} \rangle$ in the upper bound of X^{r}, we strengthen X^{r}'s lower bound to $\bot_{\mathsf{row}}$ (a subtype of all rows), as we know that any type-correct value flowing into `x` must be an object. As Y^{r} is now reachable from $\bot_{\mathsf{row}}$, $\bot_{\mathsf{row}}$ is added to Y^{r}'s lower bound. With this bound, the algorithm strengthens Y^{r}'s upper bound to $\langle\,\rangle$, a supertype of all rows. Given these strengthened bounds, inference tries to ascribe an object type to `y`, and detects a type error with int in Y^{r}'s lower bound, as desired. Apart from aiding in correctness, bound strengthening simplifies ascription, as any type variable can be ascribed the greatest-lower bound of its upper bound (details in Section 4.2).

3. Terms, Types, and Constraint Generation

This section details the terms and types for a core calculus based on that of Choi *et al.* [20], modelling a JavaScript fragment equipped with integers, objects, prototype inheritance, and methods. The type system includes structural subtyping, abstract types, and recursive types. As this paper focuses on inference, rather than presenting the typing relation here, we show the constraint generation rules for inference instead, which also capture the requirements for terms to be well-typed. An associated technical report presents the full typing relation [18].

$$
\begin{array}{lll}
\textbf{fields} & a & \in \mathcal{A} \\
\textbf{expressions} & e & ::= n \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid x \mid x := e_1 \\
& & \mid \{\cdot\} \mid \{a_1 : e_1, \ldots, a_n : e_n\}\ \mathsf{proto}\ e_p \mid \mathsf{null} \mid \mathsf{this} \\
& & \mid e.a \mid e_1.a := e_2 \mid \mathsf{function}\,(x)\,\{e_1\} \mid e_1.a\,(e_2)
\end{array}
$$

Figure 3: Syntax of terms.

$$
\begin{array}{lrcl}
\textbf{types} & \tau, \sigma \in \mathcal{T} & ::= & \mathsf{int} \mid \nu \mid \alpha \mid \\
& & \mid & [\nu]\,\tau_1 \Rightarrow \tau_2 \mid [\cdot]\,\tau_1 \Rightarrow \tau_2 \\
\textbf{rows} & r, w, mr, mw & ::= & \langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle \\
\textbf{base types} & \rho & ::= & \{r \mid w\} \\
\textbf{object types} & \nu & ::= & \rho^q \mid \mu\alpha.\nu \\
\textbf{qualifiers} & q & ::= & \mathbf{P}\,(mr, mw) \mid \mathbf{NC} \mid \mathbf{NA}
\end{array}
$$

Figure 4: Syntax of types.

3.1 Terms

Figure 3 presents the syntax of the calculus. The metavariable a ranges over a finite set of fields $\mathcal{A}$, which describe the fields of objects. Expressions e include base terms n (which we take to be integers), and variable declaration ($\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$), use ($x$), and assignment ($x := e$). An object is either the empty object $\{\cdot\}$ or a record of fields $\{a_1 : e_1, \ldots, a_n : e_n\}$ $\mathsf{proto}\ e_p$, where e_p is the object's prototype. We also have the null and the receiver, this.

Field projection $e.a$ and assignment $e_1.a := e_2$ take the expected form. The calculus includes first-class methods (declared with the `function` syntax, as in JavaScript), which must be invoked with a receiver argument. Our implementation also handles first-class functions, but they present no additional complications for inference beyond methods, so we omit them here for simplicity. Additional details can be found in the extended version [18].

3.2 Types

Figure 4 presents the syntax of types. Types τ include a base type (integers), objects (ν), and two method types: unattached methods ($[\tau_r]\,\tau_1 \Rightarrow \tau_2$), which retain the receiver type τ_r, and attached methods ($[\cdot]\,\tau_1 \Rightarrow \tau_2$), wherein the receiver type is elided and assumed to be the type of the object to which the method is attached. (If $e_1.a := e_2$ assigns a new method to $e_1.a$, e_2 is typed as an unattached method. Choi et al [21] restricted e_2 to method literals, whereas our treatment is more general.)

Object types comprise a base type, ρ, and a qualifier, q. The base type is a pair of rows (finite maps from names to types), one for the readable fields r and one for the writeable

fields w.[6] Well-formedness for object types (detailed in Section 3.3) requires that writeable fields are also readable. We choose to repeat the fields of w into r in this way because it enables a simpler mathematical treatment based on row subtyping. Object types also contain recursive object types $\mu\alpha.\nu$, where α is bound in ν and may appear in field types.

Object qualifiers q describe the field accesses performed by the methods in the type, required for reasoning about access safety (see Section 2.2). A *prototypal* qualifier $\mathbf{P}\,(mr, mw)$ maintains the information explicitly with two rows, one for fields readable by methods of the type (mr), and another for method-writeable fields (mw). At a method call, the type system ensures that all method-readable fields are readable on the base object, and similarly for method-writeable fields. The $\mathbf{NC}$ and $\mathbf{NA}$ qualifiers are used to enable structural subtyping on object types, and are discussed further in Section 3.3.

3.3 Subtyping and Type Equivalence

Any realistic type system for JavaScript must support structural subtyping for object types. Figure 5 presents the subtyping rules for our type system. In the premises, we sometimes write $\tau_1 \equiv \tau_2$ as a shorthand for $\tau_1 <: \tau_2 \wedge \tau_2 <: \tau_1$, and similarly $r_1 \equiv r_2$ as a shorthand for $r_1 <: r_2 \wedge r_2 <: r_1$. The S-ROW rule enables width subtyping on rows and row reordering, and the S-NONPROTO rule lifts those properties to nonprototypal objects (ignore the qualifier k for the moment). Note from S-ROW that overlapping field types must be equivalent, disallowing depth subtyping—such subtyping is known to be unsound with mutable fields [11, 27]. Depth subtyping would be sound for read-only fields, but we disallow it to simplify inference.

As discussed in Section 2.2, there is no good way to preserve information about method-readable and method-writeable fields across use of structural subtyping. Hence, other than row reordering enabled by S-ROW and S-PROTO, there is *no* subtyping between distinct prototypal types, which are the ones that carry method-readable and method-writeable information. To employ structural subtyping, a prototypal type must first be converted to a *non-prototypal* $\mathbf{NC}$ or $\mathbf{NA}$ type (distinction to be discussed shortly), using the S-PROTOCONC or S-PROTOABS rules. After this conversion, structural subtyping is possible using S-NONPROTO. Since non-prototypal types have no specific information about which fields are accessed by methods, they cannot be used for prototype inheritance or method updates; see Section 3.5.

The type system also makes a distinction between *concrete* object types, on which method invocations are allowed, and *abstract* types, for which invocations are prohibited. For prototypal types, concreteness can be checked directly, by ensuring that all method-readable fields are readable on the object and similarly for method-writeable fields, i.e., $r <: mr$ and $w <: mw$ (the assumptions of the S-PROTOCONC rule). For non-prototypal types, we employ separate qualifiers $\mathbf{NC}$ and $\mathbf{NA}$ to distinguish concrete from abstract. Rule S-PROTOCONC only allows concrete prototypal types to be converted to an $\mathbf{NC}$ type, whereas rule S-PROTOABS allows any prototypal type to be converted to an $\mathbf{NA}$ type. The type system only allows a method call if the receiver type can be converted to an $\mathbf{NC}$ type (see Section 3.5). The S-CONCABS rule allows any $\mathbf{NC}$ type to be converted to the corresponding $\mathbf{NA}$ type, as this only removes the ability to invoke methods.

[6] Note that row types cannot be ascribed to terms directly; they only appear as part of object types.

$$\text{S-ROW}\ \frac{\forall a \in \mathsf{dom}(r').a \in \mathsf{dom}(r) \wedge r[a] \equiv r'[a]}{r <: r'}$$

$$\text{S-NONPROTO}\ \frac{r_1 <: r_2 \quad w_1 <: w_2 \quad k = \mathbf{NC} \vee k = \mathbf{NA}}{\{r_1 \mid w_1\}^{k} <: \{r_2 \mid w_2\}^{k}}$$

$$\text{S-PROTO}\ \frac{r_1 \equiv r_2 \quad w_1 \equiv w_2 \quad mr_1 \equiv mr_2 \quad mw_1 \equiv mw_2}{\{r_1 \mid w_1\}^{\mathbf{P}(mr_1, mw_1)} <: \{r_2 \mid w_2\}^{\mathbf{P}(mr_2, mw_2)}}$$

$$\text{S-PROTOCONC}\ \frac{r <: mr \quad w <: mw}{\{r \mid w\}^{\mathbf{P}(mr,mw)} <: \{r \mid w\}^{\mathbf{NC}}}$$

$$\text{S-PROTOABS}\ \frac{}{\{r \mid w\}^{\mathbf{P}(mr,mw)} <: \{r \mid w\}^{\mathbf{NA}}}$$

$$\text{S-CONCABS}\ \frac{}{\{r \mid w\}^{\mathbf{NC}} <: \{r \mid w\}^{\mathbf{NA}}}$$

$$\text{S-METHOD}\ \frac{}{[\tau]\,\tau_1 \Rightarrow \tau_2 <: [\cdot]\,\tau_1 \Rightarrow \tau_2}$$

$$\text{S-TRANS}\ \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \qquad \text{S-REFL}\ \frac{}{\tau <: \tau}$$

$$\text{WF-NONOBJECT}\ \frac{\tau \text{ is not an object type or a type variable}}{\Delta \Vdash \tau}$$

$$\text{WF-NC}\ \frac{r <: w \quad \forall a \in \mathsf{dom}(r).\ \Delta \Vdash r[a]}{\Delta \Vdash \{r \mid w\}^{\mathbf{NC}}}$$

$$\text{WF-NA}\ \frac{r <: w \quad \forall a \in \mathsf{dom}(r).\ \Delta \Vdash r[a]}{\Delta \Vdash \{r \mid w\}^{\mathbf{NA}}}$$

$$\text{WF-P}\ \frac{\begin{array}{r} r <: w \quad \forall a \in \mathsf{dom}(r).\ \Delta \Vdash r[a] \\ mr <: mw \quad \forall a \in \mathsf{dom}(mr).\ \Delta \Vdash mr[a] \\ \forall a \in \mathsf{dom}(mr) \cap \mathsf{dom}(r).\ mr[a] \equiv r[a] \end{array}}{\Delta \Vdash \{r \mid w\}^{\mathbf{P}(mr,mw)}}$$

$$\text{WF-REC}\ \frac{\Delta, \alpha \Vdash \nu}{\Delta \Vdash \mu\alpha.\nu} \qquad \text{WF-VAR}\ \frac{}{\Delta, \alpha \Vdash \alpha}$$

Figure 5: Subtyping and object-type well-formedness.

Revisiting the example in Figure 1, here are the types for objects O_1, O_2 and O_3.

$$O_1 : \{\langle d : \mathsf{int}, m : [\cdot]\,\mathsf{int} \Rightarrow \mathsf{void}\rangle \mid \langle d, m\rangle\}^{\mathbf{P}(\langle d,a\rangle,\langle a\rangle)}$$

$$O_2 : \{\langle d : \mathsf{int}, m : [\cdot]\,\mathsf{int} \Rightarrow \mathsf{void}, a : \mathsf{int}\rangle \mid \langle a\rangle\}^{\mathbf{P}(\langle d,a\rangle,\langle a\rangle)}$$

$$O_3 : \{\langle d : \mathsf{int}, m : [\cdot]\,\mathsf{int} \Rightarrow \mathsf{void}, a : \mathsf{int}, b : \mathsf{int}\rangle \mid \langle b\rangle\}^{\mathbf{P}(\langle d,a\rangle,\langle a\rangle)}$$

(We omit writing the types of fields in rows duplicatively.) In view of the subtyping relation presented above, the conversion of the prototypal type of O_2 to a $\mathbf{NC}$ type is allowed

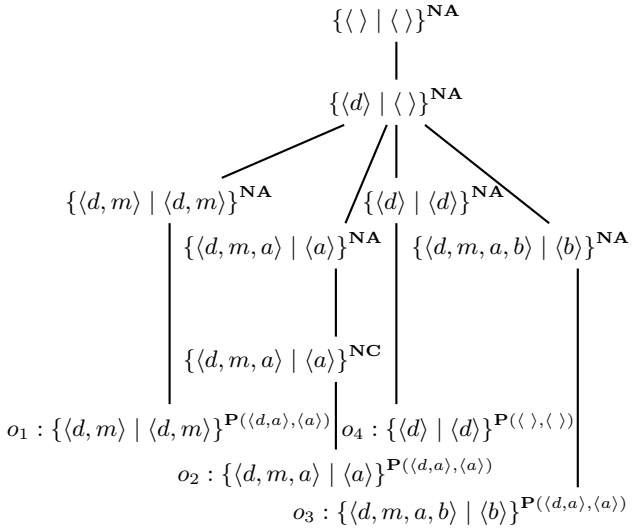


Figure 6: Lattice of object types

(by S-PROTOCONC), so a method call at line line 4 is allowed. By contrast, the conversion of the prototypal type of O_3 to a $\mathbf{NC}$ type is not allowed (because the condition $w <: mw$ is not satisfied in S-PROTOCONC), and so the method call at line 7 is disallowed. Figure 6 gives a graphical view of the associated type lattice, showing the order between prototypal, $\mathbf{NC}$ and $\mathbf{NA}$ types.

The $\mathbf{NA}$ qualifier aids in expressivity. Consider extending Figure 1 as follows:

```
8    var v4 = cond() ?
9                v3 :
10               { d : 2 } // o4
```

The type of O_4 is $\{\langle d : \mathsf{int}\rangle \mid \langle d\rangle\}^{\mathbf{P}(\langle\,\rangle,\langle\,\rangle)}$. To ascribe a type to `v4`, we need to find a common supertype of the types of O_3 and O_4. We cannot simply upcast O_3 to the type of O_4 because there is no subtyping on prototypal types; S-NONPROTO does not apply. We also cannot apply S-PROTOCONC to O_3, as O_3 is not concrete. However, we *can* use S-PROTOABS and S-NONPROTO, in that order, to upcast the type of O_3 to $\{\langle d : \mathsf{int}\rangle \mid \langle\,\rangle\}^{\mathbf{NA}}$ (see Figure 6). This type is also a supertype of the type of O_4, and therefore, a suitable type to be ascribed to `v4`.[7] $\mathbf{NA}$ serves as a top element in the object type lattice, which also simplifies type ascription as we will illustrate in Section 4.2.

Rule S-METHOD introduces a limited form of method subtyping to allow an unattached method to be attached to an object, thereby losing its receiver type. This stripping of receiver types is important for object subtyping. In the subtyping example in Section 2.2, without attached methods, `o1` and `o2` would not have a common supertype with `m` present, as the receiver types for their `m` methods would differ; this would make `p.m()` a type error. We exclude any other form of method subtyping, as we have not encountered a need for it in practice. More general function/method subtyping poses additional challenges for inference, due to contravariance, but extant techniques could be adopted to handle these issues [22, 23]; we plan to do so when a practical need arises.

Subtyping is reflexive and transitive. Recursive types are equi-recursive and admit α-equivalence, that is:

$$\mu\alpha.\nu <: \nu\,[\alpha \mapsto \mu\alpha.\nu]$$
$$\nu\,[\alpha \mapsto \mu\alpha.\nu] <: \mu\alpha.\nu$$
$$\mu\alpha.\nu <: \mu\beta.(\nu[\alpha \mapsto \beta])$$

which directly implies:

$$\mu\alpha.\nu \equiv \nu\,[\alpha \mapsto \mu\alpha.\nu]$$
$$\mu\alpha.\nu \equiv \mu\beta.(\nu[\alpha \mapsto \beta])$$

Note that it is possible to expand a recursive type and then apply rule S-NONPROTO or S-PROTO to achieve a form of width subtyping.

Figure 5 also shows the well-formedness for types, $\Delta \Vdash \tau$, in the context Δ representing a set of bound variables. All non-object non-variable types are well-formed (rule WF-NONOBJECT). For object types, well-formedness requires that any writeable field is also readable, and that all field types are also well-formed (rules WF-NC and WF-NA). In prototypal types, well-formedness further requires that method-writeable fields are also method-readable, and that for any field a that is both readable and method-readable, the mr and r rows agree on a's type (rule WF-P). Finally, rules WF-REC and WF-VAR respectively introduce and eliminate type variables to enable well-formedness of recursive types.

3.4 Constraint Language

Here, we present the constraint language used to express our type inference problem. Constraints primarily operate over *families of row variables*, rather than directly constraining more complex source-level object types. Section 3.5 reduces inference for the source type system to this constraint language, and Section 4 gives an algorithm for solving such constraints.

Figure 7 defines the constraint language syntax. The language distinguishes *type variables*, which represent source-level types, and *row variables*, which represent the various components of a source-level object type. Each type variable X has five corresponding row variables: X^{r}, X^{w}, X^{mr}, X^{mw}, and X^{all}. The first four correspond directly to the r, w, mr, and mw rows from an object type. To enforce the condition $\Vdash \{r \mid w\}^q$ (Figure 5) on all types, we impose the well-formedness conditions in Figure 7 on all X. The last

[7] While the type system of Choi et al. [21] restricts subtyping on prototypal types, their system does not have the notion of $\mathbf{NA}$, and hence cannot type the example above.

type variables	X, Y		*range over source types*
variable sorts	s	::=	$\mathsf{r} \mid \mathsf{w} \mid \mathsf{mr} \mid \mathsf{mw} \mid \mathsf{all}$
row variables	X^s, Y^s		*range over row / non-object types*
literals	L	::=	$\mathsf{int} \mid \bot_{\mathsf{row}} \mid \langle \ldots, a : X, \ldots \rangle$
			$\mid [X_R]\, X_1 \Rightarrow X_2$
constraints	C	::=	$L <: X^s \mid X^s <: L \mid X^s <: Y^s$
			$\mid C \wedge C$
			$\mid X^s <: Y^s \backslash \{a_1, \ldots, a_n\}$
			$\mid \mathsf{proto}(X) \mid \mathsf{concrete}(X)$
			$\mid \mathsf{strip}(X)$
			$\mid \mathsf{attach}(X_b, X_f, X_v)$
acceptance criteria	A	::=	$\mathsf{notmethod}(X) \mid \mathsf{notproto}(X)$

well-formedness $X^{\mathsf{all}} <: X^{\mathsf{r}} <: X^{\mathsf{w}}$
$\wedge\, X^{\mathsf{all}} <: X^{\mathsf{mr}} <: X^{\mathsf{mw}}$

Figure 7: Constraint language. We give the language syntax above the line, and well-formedness constraints below.

variable, X^{all}, is used to ensure that types of fields in both r and mr are equivalent; if ascription fails for X^{all}, there must be some inconsistency between X^{r} and X^{mr}.

Type literals include int, unattached methods, and rows. The $\bot_{\mathsf{row}}$ type ensures a complete row subtyping lattice and is used in type propagation (see Section 4.1). To handle non-object types, row variables are "overloaded" and can be assigned non-object types as well. Our constraints ensure that if any row variable for X is assigned a non-row type τ, then all row variables for X will be assigned τ, and hence X should map to τ in the final ascription.

The first three constraint types introduced in Figure 7 express subtyping over literals and row variables. We write $X^s \equiv L$ as a shorthand for $L <: X^s \wedge X^s <: L$ and $X^s \equiv Y^t$ as a shorthand for $X^s <: Y^t \wedge Y^t <: X^s$. Constraints can be composed together using the $\wedge$ operator. A constraint $X^s <: Y^s \backslash \{a_1, \ldots, a_n\}$ means that X^s must be a subtype of the type obtained by removing the fields $a_1, \ldots, a_n$ from Y^s. Such constraints are needed for handling prototype inheritance, discussed further in Section 3.5.

The $\mathsf{proto}(X)$ and $\mathsf{concrete}(X)$ constraints enable inference of object type qualifiers. Constraint $\mathsf{proto}(X)$ means the ascribed type for X must be prototypal, while $\mathsf{concrete}(X)$ means the type for X must be a subtype of an $\mathbf{NC}$ type. The $\mathsf{strip}(X)$ constraint ensures X is assigned an *attached* method type, with no receiver type. Conversion of unattached method types to attached occurs during ascription (Section 4.2), so the constraint syntax only includes unattached method types.

The constraint $\mathsf{attach}(X_b, X_f, X_v)$ in Figure 7 handles method attachment to objects. For a field assignment $e_1.a := e_2$, X_b, X_f, and X_v respectively represent the type of e_1, the type of a in e_1's (object) type, and the type of e_2. Intuitively, this constraint ensures the following condition:

$$(X_v^{\mathsf{r}} <: [X_R]\_ \Rightarrow \_) \implies (\mathsf{proto}(X_b) \wedge X_b^{\mathsf{mr}} <: X_R^{\mathsf{r}} \wedge X_b^{\mathsf{mw}} <: X_R^{\mathsf{w}} \wedge \mathsf{strip}(X_f))$$

That is, when X_v is an unattached method type with receiver X_R, then X_b is prototypal, its method-readable and method-writeable fields must respectively include the readable and writeable fields of X_R, and X_f is an attached method type. Note that $\mathsf{attach}(X_b, X_f, X_v)$ is *not* a macro for the above condition, as we do not directly support an implication operator in the constraint language. Instead, the condition is enforced directly during constraint propagation (Figure 10, Rule (xii)).

The acceptance criteria in Figure 7 are additional conditions on solutions that need only be checked after the constraints have been solved. The two possible criteria are checking that a variable is not assigned a method type, $\mathsf{notmethod}(X)$, and ensuring a variable is not assigned a prototypal type, $\mathsf{notproto}(X)$.

3.5 Constraint Generation

Constraint generation takes the form of a judgement

$$X_R, \Gamma \vdash e : X \mid C,$$

to be read as: *in a context with receiver type X_R and inference environment Γ, expression e has type X such that constraints in C are satisfied.* Figure 8 presents rules for constraint generation; see Section 4.1 for an example.

Rules C-INT and C-VAR generate straightforward constraints. The constraints for C-OBJEMP ensure the empty object is assigned type $\{\cdot \mid \cdot\}^{\mathbf{P}(\cdot,\cdot)}$. The rule C-THIS is the only rule directly using the carrier's type X_R. The constraint $X^{\mathsf{w}} <: \langle\ \rangle$ in rule C-NULL ensures that X is assigned an object type. (Recall that $X^{\mathsf{r}} <: X^{\mathsf{w}}$.)

The rule for variable declaration C-VARDECL passes on the constraints generated by its subexpressions (C_1, C_2), with additional constraints $Y_1^{\mathsf{r}} <: X_1^{\mathsf{r}} \wedge Y_1^{\mathsf{w}} <: X_1^{\mathsf{w}}$, which are sufficient to ensure that the type Y_1 of the expression e_1 is a subtype of the fresh inference variable X_1 ascribed to x in the environment (no constraint on Y_1^{mr}, X_1^{mr}, Y_1^{mw} or X_1^{mw} is needed). Constraining both the r and w rows is consistent with the S-NONPROTO subtyping rule (Figure 5). We put x in the initialization scope of e_1 in order to allow for the definition of recursive functions.

The C-METHDECL rule constrains the type of the body e using fresh variables Y_1 and Y_R for the parameter and receiver types. Y_R is constrained to be non-prototypal and concrete, as in any legal method invocation, the receiver type must be a subtype of an $\mathbf{NC}$ type. (Recall that prototypal types, if they are concrete, can be safely cast to $\mathbf{NC}$.) The rule for method application C-METHAPP ensures that the type X_1 of e_1 is concrete, and that its field a has a method type

$$\boxed{X_R, \Gamma \vdash e : X \mid C}$$

$$\textsc{C-Int}\ \frac{\mathsf{fresh}\ X}{X_R, \Gamma \vdash n : X \mid X^{\mathsf{r}} \equiv \mathsf{int}} \qquad \textsc{C-Var}\ \frac{\Gamma(x) = X}{X_R, \Gamma \vdash x : X \mid \emptyset} \qquad \textsc{C-This}\ \frac{}{X_R, \Gamma \vdash \mathsf{this} : X_R \mid \emptyset}$$

$$\textsc{C-VarDecl}\ \frac{X_R, \Gamma\,[x \mapsto X_1] \vdash e_1 : Y_1 \mid C_1 \qquad \mathsf{fresh}\ X_1 \qquad X_R, \Gamma\,[x \mapsto X_1] \vdash e_2 : X \mid C_2}{X_R, \Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : X \mid C_1 \wedge C_2 \wedge Y_1^{\mathsf{r}} <: X_1^{\mathsf{r}} \wedge Y_1^{\mathsf{w}} <: X_1^{\mathsf{w}}}$$

$$\textsc{C-VarUpd}\ \frac{x : X_1 \in \Gamma \qquad X_R, \Gamma \vdash e_1 : X \mid C_1}{X_R, \Gamma \vdash x := e_1 : X \mid C_1 \wedge X^{\mathsf{r}} <: X_1^{\mathsf{r}} \wedge X^{\mathsf{w}} <: X_1^{\mathsf{w}}} \qquad \textsc{C-Null}\ \frac{\mathsf{fresh}\ X}{X_R, \Gamma \vdash \mathsf{null} : X \mid X^{\mathsf{w}} <: \langle\,\rangle}$$

$$\textsc{C-MethDecl}\ \frac{\mathsf{fresh}\ Y_R, Y_1, X \quad \mathsf{has\_this}(e) \quad Y_R, \Gamma\,[x \mapsto Y_1] \vdash e : Y_2 \mid C}{X_R, \Gamma \vdash \mathsf{function}\ (x)\ \{e\} : X \mid C \wedge Y_R^{\mathsf{w}} <: \langle\,\rangle \wedge \mathsf{concrete}(Y_R) \wedge \mathsf{notproto}(Y_R) \wedge X^{\mathsf{r}} \equiv ([Y_R]\, Y_1 \Rightarrow Y_2)}$$

$$\textsc{C-MethApp}\ \frac{\mathsf{fresh}\ X_M, Y_R, X_3, X \qquad X_R, \Gamma \vdash e_1 : X_1 \mid C_1 \qquad X_R, \Gamma \vdash e_2 : X_2 \mid C_2}{\begin{array}{l} X_R, \Gamma \vdash e_1.a\,(e_2) : X \mid \quad C_1 \wedge C_2 \wedge X_1^{\mathsf{r}} <: \langle a : X_M \rangle \wedge X_M^{\mathsf{r}} \equiv ([Y_R]\, X_3 \Rightarrow X) \wedge \mathsf{strip}(X_M) \wedge \mathsf{concrete}(X_1) \\ \qquad\qquad \wedge\, \mathsf{concrete}(Y_R) \wedge Y_R^{\mathsf{w}} <: \langle\,\rangle \wedge \mathsf{notproto}(Y_R) \wedge X_2^{\mathsf{r}} <: X_3^{\mathsf{r}} \wedge X_2^{\mathsf{w}} <: X_3^{\mathsf{w}} \end{array}}$$

$$\textsc{C-ObjEmp}\ \frac{}{X_R, \Gamma \vdash \{\cdot\} : X \mid \mathsf{proto}(X) \wedge X^{\mathsf{r}} \equiv \langle\,\rangle \wedge X^{\mathsf{mr}} \equiv \langle\,\rangle} \qquad \textsc{C-Attr}\ \frac{\mathsf{fresh}\ X \quad X_R, \Gamma \vdash e : X_1 \mid C}{X_R, \Gamma \vdash e.a : X \mid C \wedge X_1^{\mathsf{r}} <: \langle a : X \rangle \wedge \mathsf{notmethod}(X)}$$

$$\textsc{C-AttrUpd}\ \frac{\mathsf{fresh}\ X_f \quad X_R, \Gamma \vdash e_1 : X_b \mid C_1 \quad X_R, \Gamma \vdash e : X_v \mid C_2}{X_R, \Gamma \vdash e_1.a := e : X_v \mid C_1 \wedge C_2 \wedge X_b^{\mathsf{w}} <: \langle a : X_f \rangle \wedge X_v^{\mathsf{r}} <: X_f^{\mathsf{r}} \wedge X_v^{\mathsf{w}} <: X_f^{\mathsf{w}} \wedge \mathsf{attach}(X_b, X_f, X_v)}$$

$$\textsc{C-ObjLit}\ \frac{\mathsf{fresh}\ X \qquad \forall i \in 1..n.\ \mathsf{fresh}\ X_i \qquad \forall i \in 1..n.\ X_R, \Gamma \vdash e_i : Y_i \mid C_i \quad X_R, \Gamma \vdash e_p : X_p \mid C_p}{\begin{array}{l} X_R, \Gamma \vdash \{a_1 : e_1, \ldots, a_n : e_n\}\ \mathsf{proto}\ e_p : X \mid \quad C_p \wedge \bigwedge_i (C_i \wedge Y_i^{\mathsf{r}} <: X_i^{\mathsf{r}} \wedge Y_i^{\mathsf{w}} <: X_i^{\mathsf{w}} \wedge \mathsf{attach}(X, X_i, Y_i)) \\ \qquad\qquad \wedge\, X^{\mathsf{w}} \equiv \langle a_1 : X_1, \ldots, a_n : X_n \rangle \wedge X^{\mathsf{r}} <: X_p^{\mathsf{r}} \wedge X_p^{\mathsf{r}} <: X^{\mathsf{r}} \backslash \{a_1, \ldots, a_n\} \\ \qquad\qquad \wedge\, \mathsf{proto}(X) \wedge \mathsf{proto}(X_p) \wedge X^{\mathsf{mr}} <: X_p^{\mathsf{mr}} \wedge X^{\mathsf{mw}} <: X_p^{\mathsf{mw}} \end{array}}$$

Figure 8: Constraint generation.

X_M with appropriate argument type X_3 and return type X. The $\mathsf{strip}(X_M)$ constraint ensures X_M is an *attached* method type. Note that a relation between X_1 and Y_R is ensured by an attach constraint when method a is attached to object e_1, following C-AttrUpd or C-ObjLit.

The last three rules deal more directly with objects. Constraint generation for attribute use C-Attr applies to non-methods (for methods, C-MethApp is used instead); the rule generates constraints requiring that e has an object type X_1 with a readable field a, such that a does not have a method type (preventing detaching of methods). The attribute update rule C-AttrUpd constrains a to be a *writable* field of e_1 ($X_b^{\mathsf{w}} <: \langle a : X_f \rangle$), and ensures that X_f is a supertype of e_2's type X_v. Finally, it uses the attach constraint to handle a possible method update.

Finally, the rule C-ObjLit imposes constraints governing object literals with prototype inheritance. Its constraints dwarf those of other rules, as object literals encompass potential method attachment for each field (captured by $\mathsf{attach}(X_l, X_i, Y_i)$) in addition to prototype inheritance. For the literal type X, the constraints ensure that the writeable fields X^{w} are precisely those declared in the literal. The readable fields must include those inherited from the prototype ($X^{\mathsf{r}} <: X_p^{\mathsf{r}}$); note that $X^{\mathsf{r}} <: X^{\mathsf{w}}$ is imposed by well-formedness. Furthermore, the constraint $X_p^{\mathsf{r}} <: X^{\mathsf{r}} \backslash \{a_1, \ldots, a_n\}$ ensures that additional readable fields do not appear "out of thin air," by requiring that any

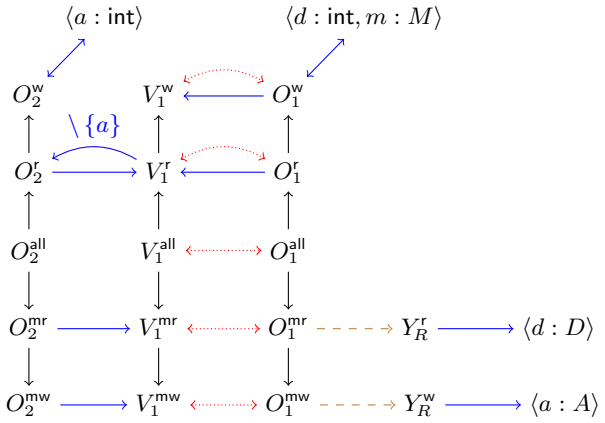


Figure 9: Selected constraints for the example of Figure 1.

fields in X^{r} apart from the locally-present $a_1, \ldots, a_n$ be present in the prototype. Finally, we ensure both X and X_p are prototypal, and that any method-accessed fields from X_p are also present in X.

Example Figure 9 shows a graph representation of some constraints for `o1`, `o2`, and `v1` from lines 1–3 of Figure 1. Nodes represent row variables and type literals, with variable names matching the corresponding program entities (Y_R corresponds to `this` on line 2). Each edge $X^s \to Y^t$ represents a constraint $X^s <: Y^t$. Black solid edges represent well-

formedness constraints (Figure 7), while blue solid edges represent constraints generated from the code (Figure 8). Dashed or dotted edges are added during constraint solving, and will be discussed in Section 4.1.

We first discuss constraints for the body of the method declared on line 2. For the field read `this.d`, the C-ATTR rule generates $Y_R^{\mathsf{r}} \to \langle d : D \rangle$. Similarly, the C-ATTRUPD rule generates $Y_R^{\mathsf{w}} \to \langle a : A \rangle$ for the write to `this.a`.

For the containing object literal `o1`, the C-OBJLIT rule creates row variables for type O_1 and edges $O_1^{\mathsf{w}} \leftrightarrow \langle d : \mathsf{int}, m : M \rangle$ (due to type equality). It also generates a constraint $\mathsf{attach}(O_1, M, F)$ (not shown in Figure 9) to handle method attachment to field m (F is the type of the line 2 function); we shall return to this constraint in Section 4.1. The assignment to `v1` yields the V_1 row variables and the $O_1^{\mathsf{r}} \to V_1^{\mathsf{r}}$ and $O_1^{\mathsf{w}} \to V_1^{\mathsf{w}}$ edges via C-VARDECL.[8]

For `o2` on line 3, C-OBJLIT yields the $O_2^{\mathsf{w}} \leftrightarrow \langle a : \mathsf{int} \rangle$ edges for the declared `a` field. The use of `v1` as a prototype yields the constraints $O_2^{\mathsf{r}} \to V_1^{\mathsf{r}}$, $O_2^{\mathsf{mr}} \to V_1^{\mathsf{mr}}$, and $O_2^{\mathsf{mw}} \to V_1^{\mathsf{mw}}$, capturing inheritance. We also have $V_1^{\mathsf{r}} \xrightarrow{\setminus\{a\}} O_2^{\mathsf{r}}$ to prevent "out of thin air" readable fields on O_2. Finally, we generate $\mathsf{proto}(V_1)$ (not shown) to ensure V_1 gets a prototypal type.

4. Constraint Solving

Constraint solving proceeds in two phases. First, *type propagation* computes lower and upper bounds for every row variable, extending techniques from previous work [22, 35]. Then, *type ascription* checks for type errors, and, if none are found, computes a satisfying assignment for the type variables.

4.1 Type Propagation

Type propagation computes a lower bound $\lfloor X^s \rfloor$ and upper bound $\lceil X^s \rceil$ for each row variable X^s appearing in the constraints, with each bound represented as a *set* of types. Intuitively, X^s must be ascribed a type between its lower and upper bound in the subtype lattice. Figure 10 shows the rules for type propagation. Given initial constraints $\mathcal{C}$, propagation computes the smallest set of constraints $\mathcal{C}'$, and the smallest sets of types $\lceil X^s \rceil$ and $\lfloor X^s \rfloor$ for each variable X^s, verifying the rules of Figure 10. In practice, propagation starts with $\mathcal{C}' = \mathcal{C}$ and $\lceil X^s \rceil = \lfloor X^s \rfloor = \emptyset$ for all X^s. It then iteratively grows $\mathcal{C}'$ and the bounds to satisfy the rules of Figure 10 until all rules are satisfied, yielding a least fixed point.

Rule (ii) adds the standard well-formedness rules for object types. Rules (iii)–(vi) show how to update bounds for the core subtype constraints. Rule (v) states that if we have $X^s <: Y^t$, then any upper bound of Y^t is an upper bound of X^s, and vice-versa for any lower bound of X^s. Rule (vi) propagates upper bounds in a similar way for constraint $X^s <: Y^s \setminus \{a_1, \ldots, a_n\}$, but it removes fields $\{a_1, \ldots, a_n\}$ from each upper bound before propagation. Lower bounds are *not* propagated in Rule (vi), as the right-hand side of the constraint is not a type variable.

Rules (vii) and (viii) perform *bound strengthening*, a crucial step for ensuring soundness (see Section 2.3). The rules leverage predicates $\mathsf{top}(L)$ and $\mathsf{bot}(L)$, defined as follows:

$$\mathsf{top}(L) = \begin{cases} \langle\,\rangle, & \text{if } L \text{ is a row type} \\ L & \text{otherwise} \end{cases}$$

(i) $\mathcal{C} \subseteq \mathcal{C}'$;

Well-formedness

(ii) $X^{\mathsf{all}} <: X^{\mathsf{r}} <: X^{\mathsf{w}} \in \mathcal{C}'$ and $X^{\mathsf{all}} <: X^{\mathsf{mr}} <: X^{\mathsf{mw}} \in \mathcal{C}'$;

Subtyping

(iii) if $X^s <: L \in \mathcal{C}'$, then $L \in \lceil X^s \rceil$;

(iv) if $L <: X^s \in \mathcal{C}'$, then $L \in \lfloor X^s \rfloor$;

(v) if $X^s <: Y^t \in \mathcal{C}'$, then $\lfloor X^s \rfloor \subseteq \lfloor Y^t \rfloor$ and $\lceil Y^t \rceil \subseteq \lceil X^s \rceil$;

(vi) if $X^s <: Y^t \setminus \{a_1, \ldots, a_n\} \in \mathcal{C}'$, then for any $\langle F \rangle \in \lceil Y^t \rceil$, add $\langle F \setminus \{a_1, \ldots, a_n\} \rangle$ to $\lceil X^s \rceil$;

Bound strengthening

(vii) if $L \in \lfloor X^s \rfloor$, then $\mathsf{top}(L) \in \lceil X^s \rceil$;

(viii) if $L \in \lceil X^s \rceil$, then $\mathsf{bot}(L) \in \lfloor X^s \rfloor$;

Prototypalness and concreteness

(ix) if $\mathsf{proto}(Y) \in \mathcal{C}'$, $X^{\mathsf{r}} <: Y^{\mathsf{r}} \in \mathcal{C}'$ and $X^{\mathsf{w}} <: Y^{\mathsf{w}} \in \mathcal{C}'$, then $\mathsf{proto}(X) \in \mathcal{C}'$, $X^{\mathsf{r}} \equiv Y^{\mathsf{r}} \in \mathcal{C}'$, $X^{\mathsf{w}} \equiv Y^{\mathsf{w}} \in \mathcal{C}'$, $X^{\mathsf{mr}} \equiv Y^{\mathsf{mr}} \in \mathcal{C}'$ and $X^{\mathsf{mw}} \equiv Y^{\mathsf{mw}} \in \mathcal{C}'$;

(x) if $\mathsf{concrete}(Y) \in \mathcal{C}'$, $X^{\mathsf{r}} <: Y^{\mathsf{r}} \in \mathcal{C}'$, and $X^{\mathsf{w}} <: Y^{\mathsf{w}} \in \mathcal{C}'$, then $\mathsf{concrete}(X) \in \mathcal{C}'$;

(xi) if $\mathsf{proto}(X) \in \mathcal{C}'$ and $\mathsf{concrete}(X) \in \mathcal{C}'$ then $X^{\mathsf{r}} <: X^{\mathsf{mr}} \in \mathcal{C}'$ and $X^{\mathsf{w}} <: X^{\mathsf{mw}} \in \mathcal{C}'$;

Attaching methods

(xii) if $\mathsf{attach}(X_b, X_f, X_v) \in \mathcal{C}'$ and $[X_R]\, Y_1 \Rightarrow Y_2 \in \lceil X_v^{\mathsf{r}} \rceil$, then $\mathsf{proto}(X_b) \in \mathcal{C}'$, $X_b^{\mathsf{mr}} <: X_R^{\mathsf{r}} \in \mathcal{C}'$, $X_b^{\mathsf{mw}} <: X_R^{\mathsf{w}} \in \mathcal{C}'$, and $\mathsf{strip}(X_f) \in \mathcal{C}'$.

(xiii) if $\mathsf{strip}(X) \in \mathcal{C}'$, $X^{\mathsf{r}} <: Y^{\mathsf{r}} \in \mathcal{C}'$, and $X^{\mathsf{w}} <: Y^{\mathsf{w}} \in \mathcal{C}'$, then $\mathsf{strip}(Y) \in \mathcal{C}'$;

Inferring equalities (not essential for soundness)

(xiv) if $\langle f_1 : F_1, \ldots, f_n : F_n, \ldots \rangle \in \lfloor X^s \rfloor$ and $\langle f_1 : G_1, \ldots, f_n : G_n \rangle \in \lceil X^s \rceil$, then $\forall s. \{F_1^s \equiv G_1^s, \ldots, F_n^s \equiv G_n^s\} \subseteq \mathcal{C}'$;

(xv) if $\langle f_1 : F_1, \ldots, f_n : F_n, \ldots \rangle \in \lceil X^s \rceil$ and $\langle f_1 : G_1, \ldots, f_n : G_n, \ldots \rangle \in \lceil X^s \rceil$, then $\forall s. \{F_1^s \equiv G_1^s, \ldots, F_n^s \equiv G_n^s\} \subseteq \mathcal{C}'$;

(xvi) if $[X_R]\, X_1 \Rightarrow X_2 \in \lceil X^s \rceil$ and $[Y_R]\, Y_1 \Rightarrow Y_2 \in \lceil X^s \rceil$, then $\forall s. \{X_1^s \equiv Y_1^s, X_2^s \equiv Y_2^s\} \subseteq \mathcal{C}'$.

Figure 10: Propagation rules.

[8] The code uses JavaScript `var` syntax rather than let from the calculus.

$$\mathsf{bot}(L) = \begin{cases} \bot_{\mathsf{row}}, & \text{if } L \text{ is a row type} \\ L & \text{otherwise} \end{cases}$$

The rules ensure that any lower bound $\lfloor X^s \rfloor$ includes the best type information that can be inferred from $\lceil X^s \rceil$, and vice-versa.

Rules (ix)–(xi) handle the constraints for prototypalness and concreteness. Recall from Section 3.3 that a prototypal type is only related to itself by subtyping (modulo row reordering). So, if we have $\mathsf{proto}(Y)$ and $X <: Y$, it must be true that $X \equiv Y$ and also $\mathsf{proto}(X)$ (to handle transitive subtyping). Rule (ix) captures this logic at the level of row variables. The subtyping rules (Figure 5) show that for any concrete (**NC**) type Y, if $X <: Y$, then X must also be concrete, either as an **NC** type (S-NonProto) or a concrete prototypal type (S-ProtoConc); Rule (x) captures this logic. Finally, if we have both $\mathsf{proto}(X)$ and $\mathsf{concrete}(X)$, Rule (xi) imposes the assumptions from the S-ProtoConc rule of Figure 5, ensuring any method-accessed field is present in the type.

Rules (xii) and (xiii) handle method attachment. Rule (xii) enforces the meaning of attach as discussed in Section 3.4. To understand Rule (xiii), say that X and Y are both *unattached* method types such that $X <: Y$. If we add $\mathsf{strip}(Y)$ to make Y an attached method, $X <: Y$ still holds, by the S-Method subtyping rule (Figure 5). However, if $\mathsf{strip}(X)$ is introduced, then $\mathsf{strip}(Y)$ must also be added, or else $X <: Y$ will be violated.

Rules (xiv)–(xvi) introduce new type equalities that enable the inference to succeed in more cases (the rules are not needed for soundness). Rule (xiv) equates types of shared fields for any rows $r_1 \in \lfloor X^s \rfloor$ and $r_2 \in \lceil X^s \rceil$; the types must be equal since $r_1 <: r_2$ and the type system has no depth subtyping. Rule (xv) imposes similar equalities for two rows in the same upper bound, and Rule (xvi) does the same for methods.

Example. We describe type propagation for the example of Figure 9. For the graph, type propagation ensures that if there is a path from row variable X^s to type L in the graph, then $L \in \lceil X^s \rceil$. E.g., given the path $O_2^{\mathsf{r}} \to O_2^{\mathsf{w}} \to \langle a : \mathsf{int} \rangle$, propagation ensures that $\{\langle a : \mathsf{int} \rangle\} \subseteq \lceil O_2^{\mathsf{r}} \rceil$. The new subtype / equality constraints added to $\mathcal{C}'$ in the rules in Figure 10 correspond to adding new edges to the graph. For the example, the C-MethDecl rule generates a constraint $F^{\mathsf{r}} \equiv [Y_R]\, Y_1 \Rightarrow Y_2$ (not shown in Figure 9) for the method literal on line 2 of Figure 1. Once propagation adds $[Y_R]\, Y_1 \Rightarrow Y_2$ to $\lceil F^{\mathsf{r}} \rceil$, handling of the $\mathsf{attach}(O_1, M, F)$ constraint (Rule (xii)) constrains the method-accessible fields of O_1 to accommodate receiver Y_R. Specifically, the solver adds the brown dashed edges $O_1^{\mathsf{mr}} \to Y_R^{\mathsf{r}}$ and $O_1^{\mathsf{mw}} \to Y_R^{\mathsf{w}}$.

The $\mathsf{proto}(V_1)$ constraint, combined with $O_1^{\mathsf{r}} <: V_1^{\mathsf{r}}$, leads the solver to equate all corresponding row variables for O_1 and V_1 (Rule (ix)). This leads to the addition of the red dotted edges in Figure 9. These new red edges make all the literals reachable from O_2^{all}; e.g., we have path $O_2^{\mathsf{all}} \to O_2^{\mathsf{r}} \to V_1^{\mathsf{r}} \to O_1^{\mathsf{r}} \to O_1^{\mathsf{w}} \to \langle d : \mathsf{int}, m : M \rangle$. So, propagation yields:

$$\{\langle a : \mathsf{int} \rangle, \langle d : \mathsf{int}, m : M \rangle, \langle d : D \rangle, \langle a : A \rangle\} \subseteq \lceil O_2^{\mathsf{all}} \rceil$$

Via Rule (xv), the types of a and d are equated across the rows, yielding $A \equiv D \equiv \mathsf{int}$. Hence, the inference discovers `this.a` and `this.d` on line 2 both have type int, *without* observing the invocations of `m`.

Implementation. Our implementation computes type propagation using the iterative fixed-point solver available in WALA [10]. WALA's solver accommodates generation of new constraints during the solving process, a requirement for our scenario. WALA's solver includes a variety of optimizations, including sophisticated worklist ordering heuristics and machinery to only revisit constraints when needed. By leveraging this solver, these optimizations came for free and saved significant implementation work. As the sets of types and fields in a program are finite, the fixed-point computation terminates.

4.2 Type Ascription

Algorithm 1 Type ascription.

```
1:  procedure AscribeType(X)
2:      if strip(X) ∈ C' then strip receivers in ⌈X^s⌉, ⌊X^s⌋
3:      for each X^s do
4:          if ⌈X^s⌉ = ∅ then Φ(X^s) ← default
5:          else
6:              Φ(X^s) ← glb(⌈X^s⌉)            ▷ Fails if no glb
7:              for each L ∈ ⌊X^s⌋ do
8:                  if L ≮: Φ(X^s) then fail
9:      if Φ(X^r) = int ∨ Φ(X^r) = default then
10:         Φ(X) ← Φ(X^r)
11:     else if Φ(X^r) is method type then
12:         if notmethod(X) ∈ C' then fail
13:         Φ(X) ← Φ(X^r)
14:     else                                   ▷ Φ(X^r) must be a row
15:         ρ ← {Φ(X^r) | Φ(X^w)}
16:         if proto(X) ∈ C' then
17:             if notproto(X) ∈ C' then fail
18:             Φ(X) ← ρ^P(Φ(X^mr),Φ(X^mw))
19:         else if concrete(X) ∈ C' then Φ(X) ← ρ^NC
20:         else Φ(X) ← ρ^NA
```

Algorithm 1 shows how to ascribe a type to variable X, given bounds for all row variables X^s and the implied constraints $\mathcal{C}'$. Here, we assume each type variable can be ascribed independently, for simplicity; an associated technical report gives a slightly-modified ascription algorithm that handles variable dependencies and recursive types [18].

If required by a $\mathsf{strip}(X)$ constraint, line 2 handles stripping the receiver type in all method literals of $\lceil X^s \rceil$ and

$\lfloor X^s \rfloor$. For each X^s, we check if its upper bound is empty, and if so assign it the default type. For soundness, the same default type must be used everywhere in the final ascription; our implementation uses int. Conceptually, an empty set upper bound corresponds to a $\top$ (top) type. However we do not allow $\top$ in our system, as it would hide problems like objects and ints flowing into the same (unused) location, e.g., `x = { }; x = 3`.

If the upper bound is non-empty, we compute its *greatest lower bound* (glb) (line 6). The glb of a set of row types is a row containing the union of their fields, where each common field must have the same type in all rows. For example:

$$\mathsf{glb}(\{\langle a : \mathsf{int}\rangle, \langle b : \mathsf{string}\rangle\}) = \langle a : \mathsf{int}, b : \mathsf{string}\rangle$$

$$\mathsf{glb}(\{\langle a : \mathsf{int}\rangle, \langle a : \mathsf{string}\rangle\}) \text{ is undefined}$$

If no glb exists for two upper bound types, ascription fails with a type error.[9] Given a glb, the algorithm then checks that every type in the lower bound is a subtype of the glb (line 8). If this does not hold, then some use in the program may be invalid for some incoming value, and ascription fails (examples forthcoming).

Once all glb checks are complete, lines 9–20 compute a type for X based on its row variables. If $\Phi(X^{\mathsf{r}})$ is an integer, method, or default type, then X is assigned $\Phi(X^{\mathsf{r}})$. Otherwise, an object type for X is computed based on its row variables. The appropriate qualifier is determined based on the presence of $\mathsf{proto}(X)$ or $\mathsf{concrete}(X)$ constraints in $\mathcal{C}'$, as seen in lines 16–20. The algorithm also checks the acceptance criteria (Section 3.4), ensuring ascription failure if they apply (they are introduced by the C-METHDECL and C-ATTR rules in Figure 8).

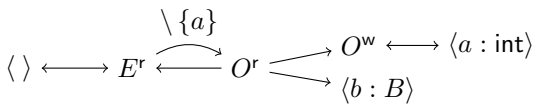
Notice that **NA** is crucial to enable ascription based exclusively on glb of upper bounds. Absent **NA**, if an object of abstract type τ flows from `x` to `y`, the types of `x` and `y` must be *equal*, as τ would have no supertypes in the lattice. Hence, qualifiers would have to be considered when deciding which fields should appear in object types, losing the clean separation in Algorithm 1. Note also, abstractness is not syntactic (in Figure 1, `v3` is only abstract because of inheritance), so even computing abstractness could require another fixed point loop.

Example. Returning to O_2 in the example of Figure 9, $\lceil O_2^{\mathsf{r}} \rceil = \{\langle a : \mathsf{int}\rangle, \langle d : \mathsf{int}, m : M\rangle\}$ after type propagation. Given type $[\cdot]\,\mathsf{int} \Rightarrow \mathsf{void}$ for M, $\mathsf{glb}(\lceil O_2^{\mathsf{r}} \rceil) = \langle a : \mathsf{int}, d : \mathsf{int}, m : [\cdot]\,\mathsf{int} \Rightarrow \mathsf{void}\rangle$. $\Phi(O_2^{\mathsf{w}})$, $\Phi(O_2^{\mathsf{mr}})$, and $\Phi(O_2^{\mathsf{mw}})$ are computed similarly. Since we have $\mathsf{proto}(O_2)$ (by C-OBJLIT, Figure 8), at line 18 ascription assigns O_2 the following type, shown previously in Section 3.3:

$$\{\langle d : \mathsf{int}, m : [\cdot]\,\mathsf{int} \Rightarrow \mathsf{void}, a : \mathsf{int}\rangle \mid \langle a\rangle\}^{\mathbf{P}(\langle d,a\rangle,\langle a\rangle)}$$

Using glb of upper bounds for ascription ensures a type captures what is needed from the term, rather than what is available. In Figure 1, note that `v3` is only used to invoke method `m`. Hence, only `m` will appear in the upper bound of V_3^{r}, and the type of `v3` will only include `m`, despite the other fields available in object `o3`.

Type error examples. We now give two examples to illustrate detection of type errors. The expression `({a: 3} proto {}).b` erroneously reads a non-existent field `b`. For this code, the constraints are:



E is the type of the empty object, and O the type of the parenthesized object literal. The $\langle\rangle \leftrightarrow E^{\mathsf{r}}$ edges are generated by the C-OBJEMP rule. As O inherits from the empty object, we have $O^{\mathsf{r}} \to E^{\mathsf{r}}$, modeling inheritance of readable fields, and also $E^{\mathsf{r}} \xrightarrow{\setminus\{a\}} O^{\mathsf{r}}$, ensuring any readable field of O except a is inherited from E. Since E is the empty object, these constraints ensure a is the only readable field of O.

Propagation and ascription detect the error as follows. $\langle a : \mathsf{int}\rangle$ is *not* added to $\lceil E^{\mathsf{r}} \rceil$, though it is reachable, due to the $\setminus\{a\}$ filter on the edge from E^{r} to O^{r}. Instead, we have $\{\langle b : B\rangle\} \subseteq \lceil E^{\mathsf{r}} \rceil$: intuitively, since b is not present locally in O, it can only come from E. Further, we have $\{\langle\rangle\} \subseteq \lfloor E^{\mathsf{r}} \rfloor$. Since $\langle\rangle \not<: \langle b : B\rangle$, line 8 of Algorithm 1 reports a failure.

As a second example, consider:

```
({m: fun () { this.f = 3; }}).m()
```

The invocation is in error, since the object literal `o` is abstract (it has no `f` field). Our constraints are:

$$\langle m : M\rangle \leftrightarrow O^{\mathsf{w}} \dashrightarrow O^{\mathsf{mw}} \dashrightarrow Y_R^{\mathsf{w}} \longrightarrow \langle f : \mathsf{int}\rangle$$

As in Figure 9, the brown dashed edge stems from method attachment. From the invocation and C-METHAPP, we have $\mathsf{concrete}(O)$. We also have $\mathsf{proto}(O)$ (from C-OBJLIT), leading (via Rule (xi)) to the dotted edge from O^{w} to O^{mw}. Now, we have a path from $\langle m : M\rangle$ to O^{w}, and from O^{w} to $\langle f : \mathsf{int}\rangle$. Since $\langle m : M\rangle \not<: \langle f : \mathsf{int}\rangle$, line 8 will again report an error.

4.3 Soundness of Type Inference

We prove soundness of type inference, including soundness of constraint generation, constraint propagation, and type ascription. We also prove our type system sound. Our typing judgment and proofs can be found in an associated technical report [18].

Our proof of soundness of type inference relies on three lemmas on constraint propagation and ascription, subtyping constraints, and well-formedness of ascripted types.

Definition 1 (Constraint satisfaction)**.** *We say that a typing substitution Φ, which maps fields in $\mathcal{A}$ to types in $\mathcal{T}$, satisfies the constraint C if, after substituting for inference variables in C according to Φ, the resulting constraint holds.*

[9] We compute glb over a semi-lattice excluding $\bot_{\mathsf{row}}$, to get the desired failure with conflicting field types.

benchmark	size	benchmark	size
access-binary-trees	41	splay	230
access-fannkuch	54	crypto	1296
access-nbody	145	richards	290
access-nsieve	33	navier	355
bitops-3bit-bits-in-byte	19	deltablue	466
bitops-bits-in-byte	20	raytrace	672
bitops-bitwise-and	7	cdjs	684
bitops-nsieve-bits	29	calc	979
controlflow-recursive	22	annex	688
math-cordic	59	tetris	826
math-partial-sums	31	2048	507
math-spectral-norm	45	file	278
3d-morph	26	sensor	266
3d-cube	301		

Table 1: Size is non-comment non-blank lines of code. Programs from the Sunspider suite appear on the left, those from Octane and Jetstream on the top right, and the Tizen apps on the bottom right.

Lemma 1 (Soundness of constraint propagation and ascription)**.** *For any set of constraints $\mathcal{C}$ generated by the rules of Figure 8, on variables $X_1, \ldots, X_n$ and their associated row variables, if constraint propagation and ascription succeeds with assignment Φ, then $\forall i, \forall s, \Phi(X_i) \vdash \mathcal{C}$ and $\Phi(X_i^s) \vdash \mathcal{C}$.*

Lemma 2 (Soundness of subtyping constraints)**.** *For a set of constraints $\mathcal{C}$ containing the constraints $X^{\mathsf{r}} <: Y^{\mathsf{r}}$ and $X^{\mathsf{w}} <: Y^{\mathsf{w}}$, if constraint generation and ascription succeeds with assignment Φ, then $\Phi(X) <: \Phi(Y)$.*

Lemma 3 (Well-formedness of ascripted types)**.** *For a set of constraints $\mathcal{C}$ containing constraints on variable X, if constraint generation and ascription succeeds with assignment Φ, then $\Vdash \Phi(X)$*

Theorem 1 (Soundness of type inference)**.** *For all terms e, receiver types X_R, and contexts Γ, if $X_R, \Gamma \vdash e : X \mid C$ and $\Phi \vdash C$, then $\Phi(X_R), \Phi(\Gamma) \vdash e : \Phi(X)$.*

5. Evaluation

We experimented with a number of standard benchmarks (Table 1), among them a selection from the Octane suite [1] (the same ones used in recent papers on TypeScript [36] and ActionScript [35]), several from the SunSpider suite [2], and `cdjs` from Jetstream [4].[10] In all cases, our compiler relied on the inferred types to drive optimizations. A separate developer team also created six apps for the Tizen mobile OS (further details in Section 5.2). In all these programs, inference took between 1 and 10 seconds. We have used type inference on additional programs as well, which are not reported here; our regression suite runs over a hundred programs.

[10] For SunSpider, we chose all benchmarks that did not make use of `Date` and `RegExp` library routines, which we do not support. For Octane, we chose all benchmarks with less than 1000 LOC.

All the features our type inference supports—structural subtyping, prototype inheritance, abstract types, recursive object types, etc.—were necessary in even this small sampling of programs. As one example, the `raytrace` program from Octane stores items of two different types in a single array; when read from the array, only an implicit "supertype" is assumed. Our inference successfully infers the common supertype. We also found the ability to infer types and find type errors in uninvoked functions to be useful in writing new code as well as typing legacy code.

5.1 Practical Considerations

Our implementation goes beyond the core calculus to support a number of features needed to handle real-world JavaScript programs. For user code, the primary additional features are support for constructors and prototype initialization (see discussion in Section 5.2) and support for polymorphic arrays and heterogeneous maps. The implementation also supports manually-written type declarations for external libraries: such declarations are used to give types for JavaScript's built-in operators and standard libraries, and also for native platform bindings. These type declaration files can include more advanced types that are not inferred for user-written functions, specifically types with parametric polymorphism and intersection types. We now give further details regarding these extensions.

Maps and arrays JavaScript supports dictionaries, which are key-value pairs where keys are strings (which can be constructed on the fly)[11] and values are of heterogeneous types. Our implementation supports maps, albeit with a homogeneous polymorphic signature $\texttt{string} \to \tau$, where τ is any type. Our implementation permits array syntax (`a[f]`) for accessing maps, but not for record-style objects. Arrays are supported similarly, with the index type `int` instead of `string`. Note that maps (and arrays) containing different types can exist in the same program; we instantiate the τ at each instance appropriately.

Constructors Even though we present object creation as allocation of object literals, JavaScript programmers often use constructors. A constructor implicitly declares an object's fields via assignments to fields of `this`. We handle constructors by distinguishing them syntactically (as functions with a capitalized name) and using syntactic analysis to discover which fields of `this` they write.

Operator overloading JavaScript operators such as `+` are heavily overloaded. Our implementation includes a separate environment file with all permissible types for such operators; the type checker selects the appropriate one, and the backend emits the required conversion. Many of the standard functions are also overloaded in terms of the number or types of arguments and are handled similarly.

[11] By contrast, object fields are *fixed* strings.

benchmark	workarounds	classes / types in TypeScript
`splay`		2 / 15
`crypto`	**C**,**U**	8(1) / 142
`richards`	**C**	7(1) / 30
`navier`		1(1) / 41
`deltablue`	**I**, **P**	12 / 61
`raytrace`	**I**	14(1) / 48
`cdjs`	**U**, **P**	—

Table 2: Workarounds needed in selected Octane benchmarks and cdjs. Each workaround impacted multiple lines of code. For relevant benchmarks, the last column quotes from Rastogi et al. [36] the number of classes (abstract ones in parentheses) and type annotations added to type check these programs in TypeScript.

Generic and native functions Some runtime functions, such as an allocator for a new array, are generic by nature. Type inference instantiates the generic parameter appropriately and ensures that arrays are used consistently (per instance). As this project arose from pursuing native performance for JavaScript applications on mobile devices, we also support type-safe interfacing with native platform functions via type annotations supplied in a separate environment file.

5.2 Explanation of Workarounds

Our system occasionally requires workarounds for type inference to succeed. The key workarounds needed for the Octane programs and `cdjs` are summarized in Table 2; our modified versions are available in the supplementary materials for this paper. The SunSpider programs did not require any major workaround.[12] After these workarounds, types were inferred fully automatically.

C (Constructors). JavaScript programs often declare a behavioral interface by defining methods on a prototype, as follows:

```
function C() { ... } // constructor
C.prototype.m1 = function () {...}
C.prototype.m2 = function () {...}
...
```

We support this pattern, provided that such field writes (including the write to the prototype field itself) appear immediately and contiguously after the constructor definition. Without this restriction, we cannot ensure in a flow-insensitive type system that the constructor is not invoked before all the prototype properties have been initialized. The code refactoring required to accommodate this restriction is straightforward (see Figure 11 for an example). We did not see any cases in which the prototype was updated more than once.

U (Unions). Lack of flow sensitivity also precludes type checking (and inference) for unions distinguished via a type test. This feature is useful in JavaScript programs, and we encountered it in one of the Octane programs. In the original `crypto`, the `BigInteger` constructor may accept a number, or a string and numeric base (arity overloading as well); we split the string case into a separate function, and updated call sites as appropriate. For `cdjs`, there were two places where the fields present in an object type could differ depending on the value of another field. We changed the code to always have all fields present, to respect fixed object layout.

[12] A trivial workaround had to do with the current implementation requirement that only constructor names to begin with an uppercase letter.

```
// Original
function TaskControlBlock(...) {
  this.link = link;
  this.id = id;
  this.priority = priority;
  this.queue = queue;
  this.task = task;
  ...
}
var STATE_RUNNING = 0;
...
TaskControlBlock.prototype.setRunning =
  function () {
    this.state = STATE_RUNNING;
  };
...
```

```
// Refactored
var STATE_RUNNING = 0;
...
function TaskControlBlock(...) {
  this.link = link;
  this.id = id;
  this.priority = priority;
  this.queue = queue;
  this.task = task;
  ...
}
TaskControlBlock.prototype.setRunning =
  function () {
    this.state = STATE_RUNNING;
  };
...
```

Figure 11: Code fragment from `richards`. In the refactored code (below), we simply moved the constant declarations out of the way (**C**).

P (Polymorphism). Although inferring polymorphic types is well understood in the context of languages like ML, its limits are less well understood in a language with mutable records and subtyping. We do not attempt to infer parametric polymorphism, although this feature is known to be useful in JavaScript programs and did come up in `deltablue` and `cdjs`. We plan to support generic types via manual annotations, as we already do for environment functions. For now, we worked around the issue with code duplication. See Figure 12 for an example.

I (Class-based Inheritance). Finally, JavaScript programs often use an *ad hoc* encoding of class-based inheritance: programmers develop their own shortcuts (or use libraries) that

```
1  Planner.prototype.removePropagateFrom =
2    function (out) {
3      out.determinedBy = null;
4      out.walkStrength = Strength.prototype.WEAKEST;
5      out.stay = true;
6      var unsatisfied = new OrderedCollection();
7      // Original
8      // var todo = new OrderedCollection();
9      var todo = new OrderedCollectionVariable();
10     todo.add(out);
11   };
```

Figure 12: Excerpt from modified `deltablue`. `OrderedCollections` were being populated with different types, which cannot be typed without parametric polymorphism. As a workaround, a duplicate type `OrderedCollectionVariable` was created, and appropriate sites (like line 9 above) were changed to use the new type.

use "monkey patching"[13] and introspection. We cannot type these constructs, but our type system can support class-based inheritance via prototypal inheritance, with some additional verbosity (see Figure 13). The latest JavaScript specification includes class-based inheritance, which obviates the need for encoding classes by other means. We intend to support the new class construct in the future.

Usability by developers. With our inference system, developers remain mostly unaware of the types being inferred, as the inference is automatic and no explicit type ascription is generated. For inference failures, we invested significant effort to provide useful error messages [31] that were understandable without knowledge of the underlying type theory. While some more complex concepts like intersection types are needed to express types for certain library routines, these types can be written by specialists, so developers solely interacting with the inference need not deal with such types directly.

More concretely, the Tizen apps listed in Table 1 were created by a team of developers who were not experts in type theory. The apps required porting of code from existing web applications (e.g., for `tetris` and `2048`) as well as writing new UI code leveraging native Tizen APIs. To learn our subset of JavaScript, the developers primarily used a manual we wrote that described the restrictions of the subset without detailing the type inference system; an associated technical report gives more details on this manual [18].

[13] "Monkey patching" here refers to adding previously non-existent methods to an object (violating fixed layout) or modifying the pre-existing methods of global objects such as `Object.prototype` (making code difficult to read accurately, and thwarting optimization of common operations). Our system permits dynamic update of *existing* methods of developer-created objects, preserving fixed layout.

```
1  // Original
2  Object.defineProperty(Object.prototype,
3    "inheritsFrom", ...)
4  function EqualityConstraint(var1, var2, strength) {
5    EqualityConstraint.superConstructor
6      .call(this, var1, var2, strength);
7  }
8  EqualityConstraint.inheritsFrom(BinaryConstraint);
```

```
1  // Refactored
2  function EqualityConstraintInheritor() {
3    this.execute = null;
4  }
5  EqualityConstraintInheritor.prototype =
6    BinaryConstraint.prototype;
7  function EqualityConstraint(var1, var2, strength) {
8    this.strength = strength;
9    this.v1 = var1;
10   this.v2 = var2;
11   this.direction = Direction.NONE;
12   this.addConstraint();
13 }
14 EqualityConstraint.prototype =
15   new EqualityConstraintInheritor();
```

Figure 13: Excerpt showing a change in `deltablue` to work around *ad hoc* class-based inheritance. The refactored code (bottom) avoids monkey-patching `Object` with a new introspective method `inheritsFrom`.

5.3 More Problematic Constructs

Certain code patterns appearing in common JavaScript frameworks make heavy use of JavaScript's dynamic typing and introspective features; such code is difficult or impossible to port to our typed subset. As an example, consider the `json2.js` program,[14] a variant of which appears in Crockford [25]. A core computation in the program, shown in Figure 14, consists of a loop to traverse a JSON data structure and make in-place substitutions. In JavaScript, arrays are themselves objects, and like objects, their contents can be traversed with a for-in loop. Hence, the single loop at line 4 applies equally well to arrays and objects. Also note that in different invocations of `walk`, the variable `v` may be an array, object, or some value of primitive type.

Our JavaScript subset does not allow such code. We were able to write an equivalent routine in our subset only after significant refactoring to deal with maps and arrays separately, as shown in Figure 15; moreover, we had to "box" values of different types into a common type to enable the recursive calls to type check. Clearly, this version loses the economy of expression of dynamically-typed JavaScript.

JavaScript code in frameworks (even non-web frameworks like `underscore.js`[15]) is often written in a highly introspective style, using constructs not supported in our subset. One common usage is extending an object's properties in-place

[14] `https://github.com/douglascrockford/JSON-js`

[15] `http://underscorejs.org/`

```
  function walk(k, v) {
      var i, n;
      if (v && typeof v === object) {
          for (i in v) {
              n = walk(i, v[i]);
              if (n !== undefined) {
                  v[i] = n;
              }
          }
      }
      return filter(k, v);
  }
```

Figure 14: JSON structure traversal.

```
function JSONVal() {
    this.tag = ...
    this.a = null; // array
    this.m = null; // map
    this.intval = 0; // int value
    this.strval = ""; // string value
}

function walk(k, v) { // v instance of JSONVal
  var i, j, n;
  switch (v.tag) {
    case Constants.INT:
    case Constants.STR:
      break;
    case Constants.MAP:
      for (var i in v.m) {
        n = walk(i, v.m[i]);
        if (n !== undefined) { v.m[i] = n; }
      }
      break;
    case Constants.ARRAY:
      for (j = 0; j < v.a.length; j++) {
        // j+"" converts j to a string
        n = walk(j+"",v.a[j]);
        if (n !== undefined) { v.a[j] = n; }
      }
      break;
  }
  return filter(k,v);
}
```

Figure 15: JSON structure traversal in our subset of JavaScript.

```
Object.prototype.extend = function (dst, src) {
   for (var prop in src) {
      dst[prop] = src[prop];
   }
}
```

Figure 16: `extend` in JavaScript

using the pattern shown in Figure 16. The code treats all objects—including those meant to be used as structs—as maps. Moreover, it also can add properties to `dst` that may not have been present previously, violating fixed-object layout. We do not support such routines in our subset.

As mentioned before, the full JavaScript language includes constructs such as `eval` that are fundamentally incompatible with ahead-of-time compilation. We also do not support adding or modifying behavior (aka "monkey patching") of built-in library objects like `Object.prototype` (as is done in Figure 16). The community considers such usage as bad practices [25].

Even if we take away these highly dynamic features, there is a price to be paid for obtaining type information for JavaScript statically: either a programmer stays within a subset that admits automatic inference, as explored in this paper and requiring the workarounds of the kinds described in Section 5.2; or, the programmer writes strong enough type annotations (the last column of Table 2 shows the effort required in adding such annotations for the same Octane programs in [36]).

Whether this price is worth paying ultimately depends on the value one attaches to the benefits offered by ahead-of-time compilation.

5.4 The Promise of Ahead-of-Time Compilation

As mentioned earlier, we have implemented a compiler that draws upon the information computed by type inference (Section 2.1) and generates optimized code. The details of the compiler are outside the scope of the paper, but we present preliminary data to show that AOTC for JavaScript yields advantages for resource-constrained devices.

We measured the space consumed by the compiled program against the space consumed by the program running on v8, a modern just-in-time compiler for JavaScript. The comparative data is shown in Figure 17. The Octane programs were run with their default parameters.[16] As the figure shows, ahead-of-time compilation yielded significant memory savings vs. just-in-time compilation.

We also timed these benchmarks for runtime performance on AOTC compiled binaries and the v8 engine. Figure 18 shows the results for one of the programs, `deltablue`; the figure also includes running time on *duktape*, a non-optimizing interpreter with a compact memory footprint. We observe that (i) the non-optimizing interpreter is quite a bit slower than the other engines, and (ii) for smaller numbers of iterations, AOTC performs competitively with v8. For larger iteration counts, v8 is significantly faster. Similar behavior was seen for all six Octane programs (see Figure 19). The AOTC slowdown over v8 for the largest number of runs ranged from 1.5X (`navier`) to 9.8X (`raytrace`). We expect significant further speedups from AOTC as we improve our

[16] Except for `splay`, which we ran for 80 as opposed to 8000 elements; memory consumption in `splay` is dominated by program data.

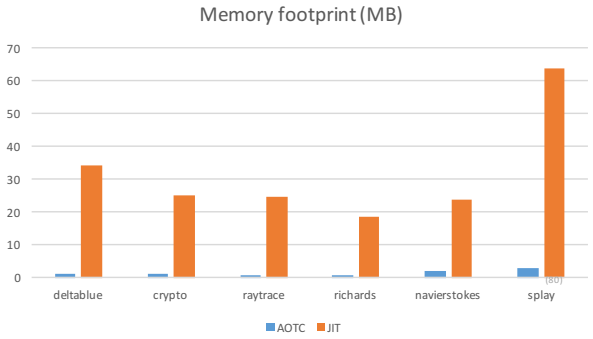


Figure 17: Memory comparison

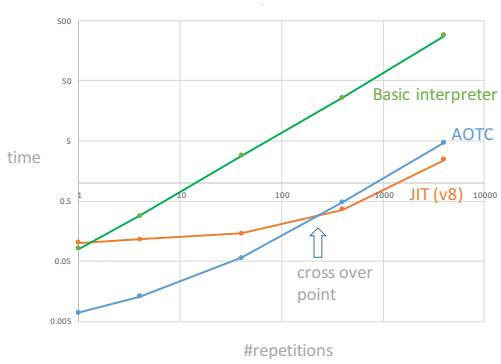


Figure 18: Running times for `deltablue`. Note the log-log scale.

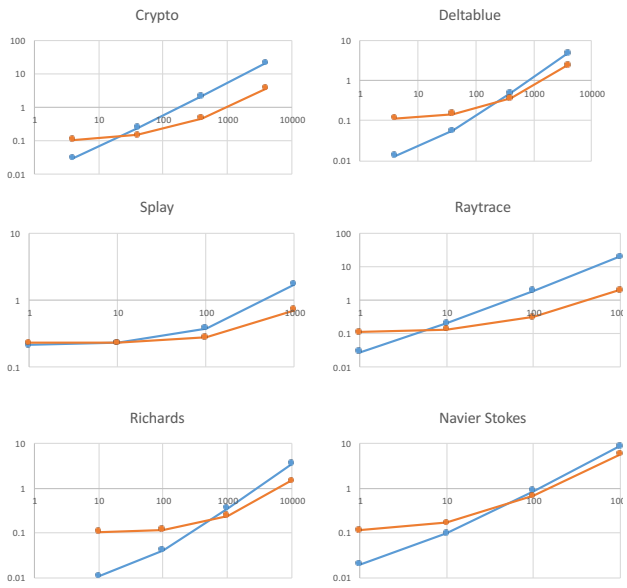


Figure 19: Crossover behavior of our AOTC system vs. the v8 runtime for the six Octane programs.

optimizations and our garbage collector. Full data for the six Octane programs, both for space and time, are presented in an associated technical report [18].

Interoperability. In a number of scenarios, it would be useful for compiled code from our JavaScript subset to interoperate with unrestricted JavaScript code. The most compelling case is to enable use of extant third-party libraries without having to port them, e.g., frameworks like jQuery[17] for the web[18] or the many libraries available for Node.js.[19] Additionally, if a program contains dynamic code like that of Figure 14 or Figure 16, and that code is not performance-critical, it could be placed in an unrestricted JavaScript module rather than porting it.

Interoperability with unrestricted JavaScript entails a number of interesting tradeoffs. The simplest scheme would be to invoke unrestricted JavaScript from our subset (and vice versa) via a foreign function interface, with no shared heap. But, this would impose a high cost on such calls, due to marshalling of values, and could limit expressivity, e.g., passing functions would be difficult. Alternately, our JavaScript subset and unrestricted JavaScript could share the same heap, with additional type checks to ensure that inferred types are not violated by the unrestricted JavaScript. The type checks could "fail fast" at any violation, like in other work on gradual typing [36, 41, 44]. But, this could lead to application behavior differing on our runtime versus a standard JavaScript runtime, as the standard runtime would not perform the additional checks. Without "fail fast," the compiled code may need to be deoptimized at runtime type violations, adding significant complexity and potentially slowing down code with no type errors. At this point, we have a work-in-progress implementation of interoperability with a shared heap and "fail fast" semantics, but a robust implementation and proper evaluation of these tradeoffs remain as future work.

[17] `http://jquery.com`

[18] Note that running our compiled code in a web browser would require an implementation of the DOM APIs, which our current implementation does not support.

[19] `http://nodejs.org`

6. Related Work

Related work spans type systems and inference for JavaScript and dynamic languages in general, as well as the type inference literature more broadly.

Type systems and inference for JavaScript. Choi et al. [20, 21] presented a typed subset of JavaScript for ahead-of-time compilation. Their work served as our starting point, and we built on it in two ways. First, our type system extends theirs with features that we found essential for real code, most crucially abstract types (see discussion throughout the paper). We also present a formalization and prove these extensions sound (see the technical report [18]). Second, whereas they relied on programmer annotations to obtain

types, we developed and implemented an automatic type inference algorithm.

Jensen *et al.* [29] present a type analysis for JavaScript based on abstract interpretation. They handle prototypal inheritance soundly. While their analysis could be adapted for compilation, it does not give a typing discipline. Moreover, their dataflow-based technique cannot handle partial programs, as discussed in Section 2.3.

TypeScript [8] extends JavaScript with type annotations, aiming to expose bugs and improve developer productivity. To minimize adoption costs, its type system is very expressive but deliberately unsound. Further, it requires type annotations at function boundaries, while we do global inference. Flow [3] is another recent type system for JavaScript, with an emphasis on effective flow-sensitive type inference. Although a detailed technical description is unavailable at the time of this writing, it appears that our inference technique has similarity to Flow's in its use of upper- and -lower bound propagation [19]. Flow's type language is similar to that of TypeScript, and it also sacrifices strict soundness in the interest of usability. It would be possible to create a sound gradually-typed version of Flow (i.e., one with dynamic type tests that may fail), but this would not enforce fixed object layout. For TypeScript, a sound gradually-typed variant already exists [36], which we discuss shortly.

Early work on type inference for JavaScript by Thiemann [42] and Anderson *et al.* [14] ignored essential language features such as prototype inheritance, focusing instead on dynamic operations such as property addition. Guha *et al.* [28] present a core calculus λ_{JS} for JavaScript, upon which a number of type systems have been based. TeJaS [30] is a framework for building type checkers over λ_{JS} using bidirectional type checking to provide limited inference. Politz *et al.* [34] provide a type system enforcing access safety for a language with JavaScript-like dynamic property access.

Bhargavan *et al.* [15] develop a sound type system and inference for Defensive JavaScript (DJS), a JavaScript subset aimed at security embedding code in untrusted web pages. Unlike our work, DJS forbids prototype inheritance, and their type inference technique is not described in detail.

Gradual typing for JavaScript. Rastogi *et al.* [35] give a constraint-based formulation of type inference for ActionScript, a gradually-typed class-based dialect of JavaScript. While they use many related techniques—their work and ours are inspired by Pottier [22]—their gradually-typed setting leads to a very different constraint system. Their (sound) inference aims at proving runtime casts safe, so they need not validate upper bound constraints. They do not handle prototype inheritance, relying on ActionScript classes.

Rastogi *et al.* [36] present Safe TypeScript, a *sound, gradual* type system for TypeScript. After running TypeScript's (unsound) type inference, they run their (sound) type checker and insert runtime checks to ensure type safety. Richards *et al.* [40] present StrongScript, another TypeScript extension with sound gradual typing. They allow the programmer to enforce soundness of some (but not all) type annotations using a specific type constructor, thus preserving some flexibility. They also use sound types to improve compilation and performance. Being based on TypeScript, both systems require type annotations, while we do not (except for signatures of external library functions). Moreover, they do not support general prototype inheritance or mutable methods, but rather rely on TypeScript's classes and interfaces.

Type inference for other dynamic languages. Agesen *et al.* [12] present inference for Self, a key inspiration for JavaScript which includes prototype inheritance. Their constraint-based approach is inspired by Palsberg and Schwartzbach [32]. However, their notion of type is a set of values computed by data flow analysis, rather than syntactic typing discipline.

Foundations of type inference and constraint solving. Type inference has a long history, progressing from early work [26] through record calculi and row variables [45, 46] through more modern presentations. Type systems for object calculi with object extension (e.g., prototype-based inheritance) and incomplete (abstract) objects extends back to the late 1990s [16, 17, 27, 37]. To our knowledge, our system is the first to describe inference for a language with both abstract objects and prototype inheritance.

Trifonov and Smith [43] describe constraint generation and solving in a core type system where (possibly recursive) types are generated by base types, $\bot$, $\top$ and $\rightarrow$ only. They introduce techniques for removing redundant constraints and optimizing constraint representation for faster type inference. Building on their work, Pottier [22, 23] crisply describes the essential ideas for subtyping constraint simplification and resolution in a similar core type system. We do not know of any previous generalization of this work that handles prototype inheritance. In both of these systems, lower and upper bounds for each type variable are already defined while resolving and simplifying constraints. Both lines of work support partial programs, producing schemas with arbitrary constraints rather than an established style of polymorphic type.

Pottier and Rémy [24] describe type inference for ML, including records, polymorphism, and references. Rémy and Vouillon [38] describe type inference for class-based objects in Objective ML. These approaches are based on row polymorphism rather than subtyping, and they do not handle prototype inheritance or non-explicit subtyping.

Aiken [13] gives an overview of program analysis in the general framework of set constraints, with applications to dataflow analysis and simple type inference. Most of our constraints would fit in his framework with little adaptation, and his resolution method also uses lower and upper bounds. His work is general and does not look into specific program construct details like objects, or a specific language like JavaScript.

Acknowledgements

We thank the anonymous reviewers for their detailed feedback, which significantly improved the presentation of the paper.

References

[1] Octane Benchmarks. `https://developers.google.com/octane/`.

[2] SunSpider Benchmarks. `https://www.webkit.org/perf/sunspider/sunspider.html`.

[3] Flow. `http://www.flowtype.org`.

[4] JetStream Benchmarks. `http://browserbench.org/JetStream/`.

[5] JavaScriptCore JavaScript engine. `http://trac.webkit.org/wiki/JavaScriptCore`.

[6] The Redmonk Programming Language Rankings: June 2015. `https://redmonk.com/sogrady/2015/07/01/language-rankings-6-15/`.

[7] Tizen Platform. `https://www.tizen.org/`.

[8] TypeScript. `http://www.typescriptlang.org`.

[9] V8 JavaScript Engine. `https://developers.google.com/v8/`.

[10] T.J. Watson Libraries for Analysis (WALA). `http://wala.sf.net`, 2015.

[11] Martín Abadi and Luca Cardelli. A Theory of Primitive Objects: Untyped and First-order Systems. *Information and Computation*, 125(2):78–102, 1996. doi:10.1006/inco.1996.0024.

[12] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type Inference of SELF. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 247–267, London, UK, UK, 1993. Springer-Verlag. ISBN 3-540-57120-5. doi:10.1007/3-540-47910-4_14.

[13] Alexander Aiken. Introduction to Set Constraint-based Program Analysis. *Sci. Comput. Program.*, November 1999. ISSN 0167-6423. doi:10.1016/S0167-6423(99)00007-6.

[14] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards Type Inference for JavaScript. In *ECOOP 2005*. doi:10.1007/11531142_19.

[15] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Language-based defenses against untrusted browser origins. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 653–670, Washington, D.C., 2013. USENIX. ISBN 978-1-931971-03-4. URL `https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/bhargavan`.

[16] Viviana Bono, Michele Bugliesi, and Luigi Liquori. A Lambda Calculus of Incomplete Objects. In *Mathematical Foundations of Computer Science 1996*, pages 218–229. Springer, 1996. doi:10.1007/3-540-61550-4_150.

[17] Viviana Bono, Michele Bugliesi, Mariangiola Dezani-Ciancaglini, and Luigi Liquori. Subtyping Constraints for Incomplete Objects. In *TAPSOFT'97: Theory and Practice of Software Development*, pages 465–477. Springer, 1997. doi:10.1007/BFb0030619.

[18] Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Young-il Choi. Type Inference for Static Compilation of JavaScript. Technical report, Samsung Research America, August 2016. URL `http://arxiv.org/abs/1608.07261`.

[19] Avik Chaudhuri. Personal communication, 2016.

[20] Philip Wontae Choi, Satish Chandra, George Necula, and Koushik Sen. SJS: A Typed Subset of JavaScript with Fixed Object Layout. Technical Report UCB/EECS-2015-13, EECS Department, University of California, Berkeley, Apr 2015. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-13.html`.

[21] Wontae Choi, Satish Chandra, George C. Necula, and Koushik Sen. SJS: A Type System for JavaScript with Fixed Object Layout. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, pages 181–198, 2015. doi:10.1007/978-3-662-48288-9_11.

[22] François Pottier. A Framework for Type Inference with Subtyping. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 228–238, September 1998. doi:10.1145/291251.289448.

[23] François Pottier. Simplifying Subtyping Constraints: A Theory. *Information & Computation*, 170(2):153–183, November 2001. doi:10.1006/inco.2001.2963.

[24] François Pottier and Didier Rémy. The Essence of ML Type Inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[25] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, 2008.

[26] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982. doi:10.1145/582153.582176.

[27] Kathleen Fisher and John C Mitchell. A Delegation-based Object Calculus with Subtyping. In *Fundamentals of Computation Theory*, pages 42–61. Springer, 1995. doi:10.1007/3-540-60249-6_40.

[28] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 126–150. Springer, 2010. doi:10.1007/978-3-642-14107-2_7.

[29] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *SAS*, pages 238–255, 2009. doi:10.1007/978-3-642-03237-0_17.

[30] Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. TeJaS: Retrofitting Type Systems for JavaScript. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 1–16, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2433-5. doi:10.1145/2508168.2508170.

[31] Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. A practical framework for type inference error explanation. In *Proceedings of the 2016 ACM International Conference on Object Oriented Programming Systems Lan-*

guages & Applications, OOPSLA '16, New York, NY, USA, 2016. ACM.

[32] Jens Palsberg and Michael I. Schwartzbach. Object-oriented Type Inference. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 146–161, New York, NY, USA, 1991. ACM. ISBN 0-201-55417-8. doi:10.1145/117954.117965.

[33] Jens Palsberg and Tian Zhao. Type Inference for Record Concatenation and Subtyping. *Inf. Comput.*, 189(1):54–86, 2004. doi:10.1016/j.ic.2003.10.001.

[34] Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Semantics and Types for Objects with First-class Member Names. In *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*, page 37, 2012.

[35] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The Ins and Outs of Gradual Type Inference. In *Proceedings of the 39th ACM Symposium on Principles of Programming Languages (POPL'12)*, pages 481–494. ACM, 2012. doi:10.1145/2103621.2103714.

[36] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe and Efficient Gradual Typing for TypeScript. In *Proceedings of the 39th ACM Symposium on Principles of Programming Languages (POPL'15)*. ACM, 2015. doi:10.1145/2775051.2676971.

[37] Didier Rémy. From classes to objects via subtyping. In *European Symposium on Programming*, pages 200–220. Springer, 1998. doi:10.1007/BFb0053572.

[38] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. doi:10.1002/(SICI)1096-9942(1998)4:1<27::AID-TAPO3>3.0.CO;2-4.

[39] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi:10.1145/1806596.1806598.

[40] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete Types for TypeScript. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 76–100, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-86-6. doi:10.4230/LIPIcs.ECOOP.2015.76.

[41] Jeremy G Siek, Michael M Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic References for Efficient Gradual Typing. In *European Symposium on Programming Languages and Systems*, ESOP, pages 432–456. Springer, 2015. doi:10.1007/978-3-662-46669-8_18.

[42] Peter Thiemann. Towards a Type System for Analyzing Javascript Programs. In *ESOP 2005*. ISBN 3-540-25435-8, 978-3-540-25435-5. doi:10.1007/978-3-540-31987-0_28.

[43] Valery Trifonov and Scott F. Smith. Subtyping Constrained Types. In *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings*, pages 349–365, 1996. doi:10.1007/3-540-61739-6_52.

[44] Michael M Vitousek, Andrew M Kent, Jeremy G Siek, and Jim Baker. Design and Evaluation of Gradual Typing for Python. In *Dynamic Language Symposium (DLS)*. ACM, 2014. doi:10.1145/2775052.2661101.

[45] Mitchell Wand. Complete Type Inference for Simple Objects. In *LICS*, volume 87, pages 37–44, 1987.

[46] Mitchell Wand. Type Inference for Record Concatenation and Multiple Inheritance. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 92–97. IEEE, 1989. doi:10.1016/0890-5401(91)90050-C.