Towards a Type System for Analyzing JavaScript Programs

Peter Thiemann

Universität Freiburg
http://www.informatik.uni-freiburg.de/~thiemann

Abstract. JavaScript is a popular language for client-side web scripting. It has a dubious reputation among programmers for two reasons. First, many JavaScript programs are written against a rapidly evolving API whose implementations are sometimes contradictory and idiosyncratic. Second, the language is only weakly typed and comes virtually without development tools.

The present work is a first attempt to address the second point. It does so by defining a type system that tracks the possible traits of an object and flags suspicious type conversions. Because JavaScript is a classless, object-based language with first-class functions, the type system must include singleton types, subtyping, and first class record labels. The type system covers a representative subset of the language and there is a type soundness proof with respect to an operational semantics.

Keywords: dynamic type systems, program analysis, objects, functions.

1 Introduction

JavaScript has originally been developed by Brendan Eich at Netscape Corp as a language for client-side web scripting. Judging from the material available, the language has grown evolutionary by demand of its users. As the language gained widespread use and with the involvement of several industrial players, a language definition was published as a standard under the name ECMAScript[4].

JavaScript is a weakly typed object-based language in the sense that it has objects but no classes. It relies on prototyping instead of inheritance to share and extend functionality in the tradition of the Self language [14]. In addition to the usual support for imperative programming, JavaScript has lexically scoped first-class functions so it may be called a functional programming language, too.

To date, JavaScript has numerous users because it is the primary scripting language for dynamic HTML and it is supported by virtually all web browsers. However, programmers do not really appreciate the language. Their main problem is that programs that work with one brand of browser do not work with another brand. In fact, numerous (cook-) books and lots of code have been written to detect and address these problems. A closer look reveals that this point is not criticizing the language but rather the differences in the object hierarchies

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 408–422, 2005.

provided by different browsers. The language itself is stable since 1999 so that only obsolete browsers implement the language in a significantly different way.

Another point deplored by JavaScript programmers is the lack of development tools. Despite the fact that significant libraries and applications have been developed, it has taken until very recently that a debugger is available. Taken together with the differences in the object hierarchies, it can become very hard to debug and maintain JavaScript programs.

To a large extent, the maintenance problem is caused by the weak type system of the language. Although every value of the language has a fixed type at runtime, values are converted automatically from one type to another when the context of use requires it. While many scripting programmers advocate such weak type systems because of the convenience they offer, it is easy to find situations where the automatic conversion leads to surprising results (see Section 2).

We believe that a type-based program analyzer for JavaScript programs would be of interest to developers and maintainers alike. It would make maintenance easier by automatically inferring a type signature as a minimal interface for each function. It would make development easier by rejecting programs with type mismatches outright and by flagging suspicious conversions. It would also make multi-browser development easier because the differences between browsers can be modeled by providing different types for the built-in object hierarchies. Last but not least, it would provide a basis for higher program structuring methods like module systems.

The present work presents a first step towards the construction of such a type system. After discussing some motivating examples in Section 2, we define the syntax of a small, but paradigmatic subset of JavaScript in Section 3. Next, in Section 4, we define the syntax of a suitable type language and typing rules along with an informal description of the respective language construct. Section 5 specifies a small-step operational semantics for the language (extracted from the 160+ page verbal specification) and Section 6 presents a type soundness result. Finally, we discuss related work (Section 7) and conclude.

2 Motivation

The object is the main data structure in JavaScript. Unlike structs in C or objects in Java, a JavaScript object is not statically fixed to a certain number of properties (fields) or even to certain names of properties. An object is rather a finite function from strings to JavaScript values that can be modified in any conceivable way: properties may be dynamically added, removed, or changed. Working with objects has a few pitfalls illustrated with the following transcript (running the JavaScript interpreter Rhino [11]).

```
js> var obj = { x: 1}
js> obj.x
1
js> obj.y
js> print(obj.y)
undefined
```

The first line creates an object with property `x` and value `1`. This property can be accessed with the usual dot notation. Next, we try to access a non-existent property. Instead of failing, the result is the value `undefined` which can even be converted to a string if required by the context (last line). **Our type system can identify the places where such conversions happen.**

Since objects are really functions from strings to values, there is an alternative notation for object access which looks like array access. That is, if there is an object access in a program, the actual property name may be statically unknown.

```
js> var x = "x"
js> obj[x]
1
js> obj["undefined"] = "gotcha"
js> obj[obj.y]
```

(We leave it to the reader to figure out the answer of the interpreter to the last input.) **Our type system uses singleton types to track the values of base type variables to increase the precision of property accesses and updates.**

To evaluate the expression `a.x = 51` in a sensible way, the value of `a` must be an object. This requirement is enforced by JavaScript, but in a potentially surprising way:

```
js> var a = "black hole"
js> a.x = 51
51
js> a.x
js>
```

What is going on? The value in variable `a` has type string and string happens to be a base type. Since base type values do not possess properties, the assignment to property `x` might just fail. Instead, JavaScript creates a wrapper object for the string and creates a property `x` for it. Since `a` refers directly to the string, the wrapper object becomes garbage and the evaluation of `a.x` creates a new wrapper object which does not know anything about property `x`. **Our type system flags such silent wrapper conversions.**

If we were to start the above script with `var a = new String("black hole")` everything would work as expected and `a` could be used in the same way as a base type string value.

```
js> var a = new String("black hole")
js> a.x = 51
51
js> a.x
51
```

Auxiliary		
str	$\in$ *String Constants*	
Expressions		
e	::= `this`	self reference in method calls
	$\mid x$	variable
	$\mid c$	constant (number, string, boolean)
	$\mid$ `{`$str : e, \ldots$`}`	object literal
	$\mid$ `function` $x(x, \ldots)$`{var` $x, \ldots; s$`}`	function expression
	$\mid e[e]$	property reference
	$\mid$ `new` $e(e, \ldots)$	object creation
	$\mid e(e, \ldots)$	function call
	$\mid e$ `=` e	assignment
	$\mid p(e, \ldots)$	primitive operators (addition, etc.)
Statements		
s	::= `skip`	no operation
	$\mid e$	expression statement
	$\mid s$`;`s	sequence
	$\mid$ `if (`e`) then {`s`} else {`s`}`	conditional
	$\mid$ `while (`e`) {`s`}`	iteration
	$\mid$ `return` e	function return

Fig. 1. Syntax of Core JavaScript

3 Core JavaScript

The starting point of our work is a restricted version of the JavaScript language. It is still sufficiently rich to expose the important points in the design of a meaningful type system for the language.

JavaScript is a weakly and dynamically typed, object-based language. The layout of an object is not fixed, rather an object is a dynamic table that maps property names (strings) to property values. Although each value possesses a type, there are exhaustive automatic conversions defined between each pair of types. As there are no classes, there is instead a prototyping mechanism that allows to inherit properties from prototype objects. The language includes first-class functions which also serve as pre-methods. When a function is assigned to a property of an object, the function becomes a method and each reference to `this` in its body resolves to a reference to the object at method invocation time.

Figure 1 summarizes the syntax of Core JavaScript. There are expressions e and statements s. Expressions comprise the following alternatives

- `this` is the reference to the object receiving a method call. It only makes sense in functions used as methods or constructors.
- x variables in the usual way.
- c literals. Each literal has a type given by a function *TypeOf*.
- An object literal creates a new object with properties given by literal strings *str* as property names and their values given by expressions.[1]

[1] JavaScript also allows number literals and identifiers as property names.

- A function expression `function` $f(x_1, \ldots)$`{var` $y_1, \ldots; s$`}` defines an anonymous function with arguments $x_1, \ldots$, local variables $y_1, \ldots$, and body s. The resulting function may be recursive if the function body refers to the functions identifier f, which is *not* visible outside the function body.
- $e_1[e_2]$ is a property reference. e_1 should evaluate to an object and e_2 to a value for naming the property. The name of the property is found by converting the value of e_2 to a string. The full language allows the expression $e\,.\,x$ where the identifier x is the property name. However, this expression is equivalent to $e[\text{“}x\text{”}]$.
- `new` $e_0(e_1, \ldots)$ creates a new object and applies the function value of e_0 with parameters $e_1, \ldots$ as a method to the object. In general, it returns the newly constructed object.
- $e_0(e_1, \ldots)$ applies the function value of e_0 to the values of $e_1, \ldots$. If e_0 is a property reference, *i.e.*, $e_0 = e_{01}[e_{02}]$, then the function is called as a method.
- e_0 `=` e_1 evaluates e_0 as a reference and assigns the value of e_1 to it.
- $p(e_1, \ldots)$ stands for a primitive function call, *e.g.*, an arithmetic operation, comparison, and so on.

Functions are used in two non-standard ways in JavaScript. First, they play the role of pre-methods. A function that is assigned to a property becomes a method. Calling a function via a property reference amounts to a method invocation that additionally binds `this` to the receiver object. Second, they become constructor functions when invoked through the `new` operator. In a constructor function, `this` is bound to the new object and `new` returns whatever the function returns (or `this` if it returns `undefined`).

The full language has further kinds of expressions. There are array literals which behave similarly to object literals, assigning primitives like pre- and post-increment, property deletion, and conditional expressions. We have also “cleaned up” the syntax of function bodies: all local variable declarations are gathered at the beginning, whereas the full language allows `var` declarations everywhere in the body.

The language of statements comprises the traditional constructs. The full language includes three further variations of loops, labeled `continue` and `break` statements, a `switch` statement and a `with` statement for opening an object as the innermost scope. They are left out because they are easy to simulate with the remaining constructs. Exceptions in the style of Java are present in the full language, but are left out of our consideration because their type-based analysis adds considerable complication but not much novelty [12, 16]. Finally, JavaScript has a `var` statement to declare a variable as local in the current function. This statement may occur anywhere in the body of a function and affects all occurrences of the declared variable even in front of the declaration. Our syntax assumes that a semantic analysis already collected the local variables at the beginning of each function body.

Types
$\tau ::= \sum_{i \in T, T \subseteq \{\bot,u,b,s,n,o\}} \varphi_i$
Rows
$\varrho ::= str : \tau, \varrho$
$\quad | \quad \delta\tau$
Type environments
$\Gamma ::= \Gamma(x : \tau)$
$\quad | \quad \emptyset$

Type summands and indices
$\varphi_\bot ::= \texttt{Undef}$
$\varphi_u ::= \texttt{Null}$
$\varphi_b ::= \texttt{Bool}(\xi_b)$
$\xi_b ::= \texttt{false} \mid \texttt{true} \mid \top$
$\varphi_s ::= \texttt{String}(\xi_s)$
$\xi_s ::= str \mid \top$
$\varphi_n ::= \texttt{Number}(\xi_n)$
$\xi_n ::= num \mid \top$
$\varphi_f ::= \texttt{Function}\ (\texttt{this} : \tau; \varrho \to \tau)$
$\varphi_o ::= \texttt{Obj}(\sum_{i \in T, T \subseteq \{b,s,n,f,\bot\}} \varphi_i)(\varrho)$

Fig. 2. Syntax of types

4 Types for Core JavaScript

What is the purpose of a type system for a language like JavaScript? The usual goal of a type system is to avoid runtime errors due to type mismatches. However, such runtime errors are fairly rare in JavaScript since most of the time there is a suitable conversion function to reconcile a type mismatch. But three causes for runtime errors remain and they must be rejected by our type system.

1. Function calls and the `new` expression both expect their first operand to evaluate to a function. If that is not the case, they raise an exception.
2. Applying an arithmetic operator to an object raises an exception unless the object is a wrapper object for a number.
3. Accessing a property of the `null` object (available as a literal constant) or of the value `undefined` raises an exception.
4. Accessing a non-existent variable causes an exception.

Section 2 has also demonstrated type conversions which are convenient in some circumstances but may cause unexpected results.

4.1 Syntax of Types

Figure 2 defines the type language. The language definition[4] prescribes a value structure consisting of the base types undefined, null, boolean, number, and string as well as different kinds of objects (plain objects, wrapper objects for boolean, number, and string, function objects, array objects, and regular expression objects). Hence, the type system is based on discriminative sum types which are used at two levels. The outer level distinguishes between the different base types and objects, whereas the inner level distinguishes different features of objects. The feature component indicates the type of the implicit *VALUE* property of an object. Besides this feature component, the object type reflects the properties of an object by a row type as its second component. Row types ϱ are slightly unusual because their labels are string constants. However, this

choice is dictated by the language definition which states that property names are strings and that every value that is used for addressing a property is first converted to a string. The $\delta\tau$ at the end of a row provides a default type for all properties not mentioned explicitly. Rows are considered modulo the equations

$$str_1 : \tau_1, \ldots, str_n : \tau_n, \delta\tau = str_1 : \tau_1, \ldots, str_n : \tau_n, str : \tau, \delta\tau$$
$$\ldots, str_i : \tau_i, \ldots, str_j : \tau_j, \ldots = \ldots, str_j : \tau_j, \ldots, str_i : \tau_i, \ldots$$

where $n \in \mathbf{N}$, $str \notin \{str_1, \ldots, str_n\}$, and all str_i are different.

The type syntax clearly shows that there are wrapped and unwrapped versions of booleans, strings, and numbers. The types for `null` and objects are intrinsically wrapped (or unwrapped), and functions are always wrapped into an object. We write $\bot$ for the empty sum. Base types are further refined by *type indices*. Our definition allows as index either a constant of the appropriate type or $\top$. A constant refines the type to a singleton type whereas $\top$ does not impose a restriction. Singleton string types serve to address the properties of an object.

Each function may take `this` as an implicit parameter if the function is used as a method or a constructor. Furthermore, the number of formal parameters in a function definition need not match the number of actual parameters. If there are too few parameters, the remaining are taken to be `undefined`. If there are too many, the excess ones are ignored. Alternatively, the function body can access the actual parameter list via an array, the object `arguments`. Hence, the function type has a special place for `this` and requires an row type in place of the argument types.

4.2 Subtyping

It is rather delicate to choose a subtyping relation for Core JavaScript. Our design keeps subtyping separate from type conversion, which is modeled by a matching relation. While subtyping is vital to model the dynamically typed nature of the language, applying a conversion too early may result in rejecting a perfectly legal program.

The subtyping relation $<:$ consists of the usual subtyping for variants in the form of type summands, function arguments are dealt with in the usual contravariant manner, and rows are only subject to depth subtyping.

All elimination rules rely on the matching relation $\trianglerighteq$. This relation makes sure that a type can be converted to the form desired by the elimination. Matching is explained with the typing rule for $e_0[e_1]$ in the next subsection.

4.3 Typing Rules

The type system defines a number of judgments.

- $\Gamma \vdash e : \tau$ types an expression e in a context where its value is required.
- $\Gamma \vdash_{ref} e : \tau/\tau'$ types an expression e in a context where a reference may be required. In such a context, τ' is the type of the base object of the reference. For example, if $e = e_1[e_2]$ then e_1 is the base object. The type of the base object is required to provide the type of `this` for a method call.

- $\Gamma \vdash_{lhs} e : \tau$ types a left-hand side use in an assignment.
- $\Gamma \vdash_{stm} s \rhd \tau$ types a statement that may return a value of type τ.
- $\vdash_{acc} \varrho \mathbin{@} \tau \mapsto \tau'$ computes the type for an object access.
- $\vdash_{upd} \varrho \mathbin{@} \tau \hookleftarrow \tau'$ computes the type for an object update.

There are only two typing rules for expressions in a value context, a subtyping rule and a rule that delegates the actual work to the $\vdash_{ref}$ judgment by ignoring the base type.

$$\frac{\Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash e : \tau'} \qquad \frac{\Gamma \vdash_{ref} e : \tau/\tau'}{\Gamma \vdash e : \tau}$$

The next set of rules considers expressions (potentially) in a reference context. The rules for variables and constants are standard. Of course, neither of them provides a base object so the base type is $\bot$ in both cases.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash_{ref} x : \tau/\bot} \qquad \Gamma \vdash_{ref} c : \mathit{TypeOf}(c)/\bot$$

The typing for object literals is similar to the corresponding rule in Abadi and Cardelli's object calculus [1]. The main difference is in the choice of literal strings instead of labels. The object constructed by the object literal has no special features as indicated by the feature `Undef`.

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash_{ref} \{str_1 : e_1, \ldots, str_n : e_n\} : \mathtt{Obj}(\mathtt{Undef})(str_1 : \tau_1, \ldots, str_n : \tau_n)/\bot}$$

Function expressions are not as straightforward as usual.

$$\frac{\begin{array}{c}\Gamma(\mathtt{this} : \tau_0)(f : \tau)(x_1 : \tau_1) \ldots (x_n : \tau_n)(\mathtt{arguments} : \mathtt{Obj}(\bot)(\varrho)) \\ (y_1 : \tau_1') \ldots (y_n : \tau_n') \vdash_{stm} s \rhd \tau' \\ \tau_0 = \mathtt{Obj}(\varphi')(\varrho') \qquad \tau = \mathtt{Obj}(\mathtt{Function}\ (\mathtt{this} : \tau_0; \varrho \to \tau'))(\delta\bot) \\ \varrho = \text{"}\mathtt{length}\text{"} : \mathtt{Number}(n), \lceil 0 \rceil : \tau_1, \ldots, \lceil n-1 \rceil : \tau_n, \delta\bot\end{array}}{\Gamma \vdash_{ref} \mathtt{function}\ f(x_1, \ldots, x_n)\{\mathtt{var}\ y_1, \ldots, y_m; s\} : \tau/\bot}$$

Besides providing the type assumptions for the arguments and the function, it also establishes the type assumption for `this` and `arguments`. The latter is set up as an array, that is, an object with a `length` property and numeric properties where $\lceil n \rceil$ stands for the string containing the decimal representation of n.

The next rule concerns property access. It is the only rule that creates a meaningful reference in the judgment for reference contexts.

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 \trianglerighteq \mathtt{Obj}(\varphi_1)(\varrho_1) \quad \Gamma \vdash e_2 : \tau_2 \quad \vdash_{acc} \varrho_1 \mathbin{@} \tau_2 \mapsto \tau'}{\Gamma \vdash_{ref} e_1[e_2] : \tau'/\tau_1}$$

As e_1 denotes the base object, τ_1 is the required type of the enclosing object. The auxiliary judgment $\vdash_{acc} \varrho \mathbin{@} \tau \mapsto \tau'$ computes the type of a property stored in an object of type ϱ and accessed with a property name of type τ. Figure 3 contains its definition as well as the definition of its cousin $\vdash_{upd} \varrho \mathbin{@} \tau \hookleftarrow \tau'$ which governs the typing of update operations. Both traverse the row ϱ in one of two

$$\frac{\vdash_{acc} \varrho \mathbin{@} \tau' \mapsto \tau \quad \neg(str \in \tau') \quad \neg(str \,\mathtt{is}\, \tau')}{\vdash_{acc} str : \tau_2, \varrho \mathbin{@} \tau' \mapsto \tau}$$

$$\frac{\vdash_{acc} \varrho \mathbin{@} \tau' \mapsto \tau_1 \quad str \in \tau' \quad \neg(str \,\mathtt{is}\, \tau') \quad \tau_1 <: \tau \quad \tau_2 <: \tau}{\vdash_{acc} str : \tau_2, \varrho \mathbin{@} \tau' \mapsto \tau}$$

$$\frac{str \,\mathtt{is}\, \tau'}{\vdash_{acc} str : \tau, \varrho \mathbin{@} \tau' \mapsto \tau}$$

$$\vdash_{acc} \delta\tau \mathbin{@} \tau' \mapsto \tau$$

$$\frac{\vdash_{upd} \varrho \mathbin{@} \tau' \leftrightarrow \tau \quad \neg(str \in \tau') \quad \neg(str \,\mathtt{is}\, \tau')}{\vdash_{upd} str : \tau_2, \varrho \mathbin{@} \tau' \leftrightarrow \tau}$$

$$\frac{\vdash_{upd} \varrho \mathbin{@} \tau' \leftrightarrow \tau \quad str \in \tau' \quad \neg(str \,\mathtt{is}\, \tau') \quad \tau <: \tau_2}{\vdash_{upd} str : \tau_2, \varrho \mathbin{@} \tau' \leftrightarrow \tau}$$

$$\frac{str \,\mathtt{is}\, \tau'}{\vdash_{upd} str : \tau, \varrho \mathbin{@} \tau' \leftrightarrow \tau}$$

$$\frac{\tau <: \tau_2}{\vdash_{upd} \delta\tau_2 \mathbin{@} \tau' \leftrightarrow \tau}$$

$str \,\mathtt{is}\, \mathtt{String}(str)$	$str \in \mathtt{String}(\xi_s) + \ldots \quad \textit{if } str \le \xi_s$
$str \,\mathtt{is}\, \mathtt{Obj}(\mathtt{String}(str))(\varrho)$	$str \in \mathtt{Obj}(\mathtt{String}(\xi_s) + \ldots)(\varrho) + \ldots \quad \textit{if } str \le \xi_s$
$\lceil n \rceil \,\mathtt{is}\, \mathtt{Number}(n)$	$\lceil n \rceil \in \mathtt{Number}(\xi_n) + \ldots \quad \textit{if } n \le \xi_n$
$\lceil n \rceil \,\mathtt{is}\, \mathtt{Obj}(\mathtt{Number}(n))(\varrho)$	$\lceil n \rceil \in \mathtt{Obj}(\mathtt{Number}(\xi_n) + \ldots)(\varrho) + \ldots \quad \textit{if } n \le \xi_n$
$\lceil \mathtt{false} \rceil \,\mathtt{is}\, \mathtt{Bool}(\mathtt{false})$	$\lceil \mathtt{false} \rceil \in \mathtt{Bool}(\xi_b) + \ldots \quad \textit{if } \mathtt{false} \le \xi_b$
$\lceil \mathtt{true} \rceil \,\mathtt{is}\, \mathtt{Bool}(\mathtt{true})$	$\lceil \mathtt{true} \rceil \in \mathtt{Bool}(\xi_b) + \ldots \quad \textit{if } \mathtt{true} \le \xi_b$
$\lceil \mathtt{false} \rceil \,\mathtt{is}\, \mathtt{Obj}(\mathtt{Bool}(\mathtt{false}))(\varrho)$	where $x \le x$ and $x \le \top$, for $x \in \xi$.
$\lceil \mathtt{true} \rceil \,\mathtt{is}\, \mathtt{Obj}(\mathtt{Bool}(\mathtt{true}))(\varrho)$	

Fig. 3. Property access

modes: If the type τ contains definite information about the property name, then τ' becomes the type of that property (or the type associated to undefined properties). If τ does not contain definite information, then the judgment returns a supertype (subtype for $\vdash_{upd}$) of all applicable property types.

The match relation $\tau \trianglerighteq \varphi$ performs a one-way matching of an arbitrary type τ to a type summand φ. Matching succeeds if τ as a whole can be converted to a type with single summand φ. It acts like a constraint in the sense that it propagates information from τ to φ but not the other way round. Its conversions correspond to the definition in Chapter 9 of the ECMAScript standard [4].

The next rule deals with function call and method invocation.

$$\frac{\tau_0 \trianglerighteq \mathtt{Obj}(\mathtt{Function}\ (\mathtt{this} : \tau'; \lceil 0 \rceil : \tau_1, \ldots, \lceil n-1 \rceil : \tau_n, \varrho \to \tau))(\varrho') \quad \Gamma \vdash_{ref} e_0 : \tau_0/\tau' \quad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash_{ref} e_0(e_1, \ldots, e_n) : \tau/\bot}$$

The rule requires that the function's argument types coincide with the types of the actual function arguments. In addition, the $\vdash_{ref}$ judgment retrieves the type τ' of a suitable base object ($\tau' = \bot$ if no such object exists). For a method invocation, τ' is the type of the receiver object, `this`.

The rule for `new` covers the use of a function as a constructor.

$$\frac{\tau_0 \trianglerighteq \mathtt{Obj}(\mathtt{Function}\ (\mathtt{this} : \tau''; \lceil 0 \rceil : \tau_1, \ldots, \lceil n-1 \rceil : \tau_n, \varrho \to \tau))(\varrho') \quad \Gamma \vdash e_0 : \tau_0 \quad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n \quad \tau'' <: \tau' \quad \tau <: \tau'}{\Gamma \vdash_{ref} \mathtt{new}\ e_0(e_1, \ldots, e_n) : \tau'/\bot}$$

$$\Gamma \vdash_{stm} \texttt{skip} \rhd \tau' \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash_{stm} e \rhd \tau'} \qquad \frac{\Gamma \vdash_{stm} s_1 \rhd \tau \quad \Gamma \vdash_{stm} s_2 \rhd \tau}{\Gamma \vdash_{stm} s_1 \texttt{;} s_2 \rhd \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash_{stm} \texttt{return } e \rhd \tau}$$

$$\frac{\Gamma \vdash e : \tau' \quad \Gamma \vdash_{stm} s_1 \rhd \tau \quad \Gamma \vdash_{stm} s_2 \rhd \tau}{\Gamma \vdash_{stm} \texttt{if (}e\texttt{) then \{}s_1\texttt{\} else \{}s_2\texttt{\}} \rhd \tau} \qquad \frac{\Gamma \vdash e : \tau' \quad \Gamma \vdash_{stm} s : \tau}{\Gamma \vdash_{stm} \texttt{while (}e\texttt{) \{}s\texttt{\}} \rhd \tau}$$

Fig. 4. Typing rules for statements

A call to a constructor function binds `this` to the new object. `this` of type τ'' is also the default return value if the function returns `undefined`. Otherwise, `new` returns whatever the constructor returns (τ). The subtyping built into the rule allows for both of these possibilities.

The rule for assignment infers the type of e_0 as a left-hand side.

$$\frac{\Gamma \vdash_{lhs} e_0 : \tau \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash_{ref} e_0 \texttt{ = } e_1 : \tau}$$

The typing judgment for left-hand sides has two rules, for variables and for property references. Property references are dealt with analogously to the rule for $\vdash_{ref}$, however, using the $\vdash_{upd}$ judgment in place of $\vdash_{acc}$. Since assignment *writes* to the reference, $\vdash_{upd}$ forces the type of the written value to be a lower bound for the type of the stored value. The judgment for $\vdash_{acc}$ behaves the other way round.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash_{lhs} x : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 \trianglerighteq \texttt{Obj}(\varphi_1)(\varrho_1) \quad \Gamma \vdash e_2 : \tau_2 \quad \vdash_{upd} \varrho_1 \mathbin{@} \tau_2 \hookleftarrow \tau'}{\Gamma \vdash_{lhs} e_1[e_2] : \tau'}$$

The typing rules for statements in Figure 4 are entirely standard. There is no subtyping rule for the return type of a statement because subtyping can be applied at the expression level before applying the rule for `return` .

5 Semantics

We specify a small-step operational semantics for Core JavaScript. To that end, we extend the syntax of expressions by store locations l drawn from a set *Loc*, variable references $var(l, str)$, property references $prop(l, str)$, and define one-step reduction relations for statements $\Sigma, l_0, s \rightarrow_s \Sigma', s'$ and expressions $\Sigma, l_0, e \rightarrow_e \Sigma', e'$, where Σ and $\Sigma' : Loc \rightarrow Storable$ are stores, $l_0 \in Loc$ is the address of an activation record, e, e' are expressions, and s, s' are statements. These relations are refined by address computation $\Sigma, l_0, e \rightarrow_a \Sigma', e$ and method access $\Sigma, l_0, e \rightarrow_m \Sigma', e$ which rely on further relations $\rightarrow_l$ (left-hand side of assignment) and $\rightarrow_v$ (variable access). Storables are defined by

$$\begin{aligned} Storable &= (String\ Constants + \{VALUE, \texttt{up}, \texttt{this}\}) \rightarrow Value \\ Value &= \{\texttt{undefined}, \texttt{null}\} + Boolean + Number + String + FValue + Loc \\ FValue &= \texttt{function}\ f@l(x_1, \ldots, x_n)\texttt{\{var}\ y_1, \ldots, y_m\texttt{;}\ s\texttt{\}} \end{aligned}$$

Expressions (see the appendix for the standard contextual rules)

$$\Sigma, l_0, \texttt{this} \to_e \Sigma, \Sigma(l_0)(\texttt{this})$$

$$\frac{\Sigma, l_0, x \to_v \Sigma', var(l, str)}{\Sigma, l_0, x \to_e \Sigma', \Sigma'(l)(str)}$$

$$\Sigma, l_0, var(l, str) = v \to_e \Sigma[l \mapsto \Sigma(l)[str \mapsto v]], v$$

$$\Sigma, l_0, prop(l, str) = v \to_e \Sigma[l \mapsto \Sigma(l)[str \mapsto v]], v$$

$$\Sigma, l_0, \{str_1 : v_1, \ldots, str_n : v_n\} \to_e \Sigma[l \mapsto [str_1 \mapsto v_1, \ldots, str_n \mapsto v_n]], l \quad \text{if } l \notin dom(\Sigma)$$

$$\Sigma, l_0, \texttt{function } f(x_1, \ldots, x_n)\{\texttt{var } y_1, \ldots, y_m; s\} \to_e \Sigma[l \mapsto [VALUE \mapsto \texttt{function } f@l_0(x_1, \ldots, x_n)\{\texttt{var } y_1, \ldots, y_m; s\}]], l \quad \text{if } l \notin dom(\Sigma)$$

$$\Sigma, l_0, l[str] \to_e \Sigma, \Sigma(l)(str)$$

$$\Sigma, l_0, l(v_1, \ldots, v_n) \to_e \Sigma[l' \mapsto [\texttt{this} \mapsto \texttt{null}, \texttt{up} \mapsto l_1, f \mapsto l, x_1 \mapsto v_1, \ldots, y_1 \mapsto \texttt{undefined}, \ldots]], \texttt{AR}(l', s)$$
$$\text{if } l' \notin dom(\Sigma), \Sigma(l)(VALUE) = \texttt{function } f@l_1(x_1, \ldots, x_n)\{\texttt{var } y_1, \ldots, y_m; s\}$$

$$\Sigma, l_0, prop(l'', str)(v_1, \ldots, v_n) \to_e \Sigma[l' \mapsto [\texttt{this} \mapsto l'', \texttt{up} \mapsto l_1, f \mapsto l, x_1 \mapsto v_1, \ldots, y_1 \mapsto \texttt{undefined}, \ldots]], \texttt{AR}(l', s)$$
$$\text{if } l' \notin dom(\Sigma), \Sigma(\Sigma(l'')(str))(VALUE) = \texttt{function } f@l_1(x_1, \ldots, x_n)\{\texttt{var } y_1, \ldots, y_m; s\}$$

$$\Sigma, l_0, \texttt{new } l(v_1, \ldots, v_n) \to_e \Sigma[l' \mapsto [\texttt{this} \mapsto l'', \texttt{up} \mapsto l_1, f \mapsto l, x_1 \mapsto v_1, \ldots, y_1 \mapsto \texttt{undefined}, \ldots], l'' \mapsto [\,]], \texttt{AR}(l', s)$$
$$\text{if } l', l'' \notin dom(\Sigma), \Sigma(l)(VALUE) = \texttt{function } f@l_1(x_1, \ldots, x_n)\{\texttt{var } y_1, \ldots, y_m; s\}$$

$$\Sigma, l_0, \texttt{AR}(l, \texttt{return } v) \to_e \Sigma, v$$

$$\Sigma, l_0, \texttt{AR}(l, \texttt{skip}) \to_e \Sigma, \texttt{undefined}$$

$$\frac{\Sigma, l, s \to_s \Sigma', s'}{\Sigma, l_0, \texttt{AR}(l, s) \to_e \Sigma', \texttt{AR}(l, s')}$$

$$\frac{\Sigma, l_0, e_0 \to_m \Sigma', e_0'}{\Sigma, l_0, e_0(e_1, \ldots) \to_e \Sigma', e_0'(e_1, \ldots)}$$

$$\frac{\Sigma, l_0, e_0 \to_l \Sigma', e_0'}{\Sigma, l_0, e_0 = e_1 \to_e \Sigma', e_0' = e_1}$$

Addresses (where $\to_l = \to_v \cup \to_a$ and $\to_m \supseteq \to_a$ where all cases omitted in $\to_a$ are identical to expression cases with $\to_e$ replaced by $\to_a$ in the conclusion)

$$\Sigma, l_0, x \to_v \Sigma, var(l_0, x) \quad \text{if } x \in dom(\Sigma(l_0))$$

$$\frac{\Sigma, \Sigma(l_0)(\texttt{up}), x \to_v \Sigma', v}{\Sigma, l_0, x \to_v \Sigma', v} \quad \text{if } x \notin dom(\Sigma(l_0)), \texttt{up} \in dom(\Sigma(l_0))$$

$$\Sigma, l_0, x \to_v \Sigma, var(l_0, x) \quad \text{if } x \notin dom(\Sigma(l_0)), \texttt{up} \notin dom(\Sigma(l_0))$$

$$\Sigma, l_0, l[str] \to_a \Sigma, prop(l, str) \qquad \frac{\Sigma, l_0, e_0 \to_e \Sigma', e_0'}{\Sigma, l_0, e_0[e_1] \to_a \Sigma', e_0'[e_1]} \qquad \frac{\Sigma, l_0, e_1 \to_e \Sigma', e_1'}{\Sigma, l_0, v_0[e_1] \to_a \Sigma', v_0[e_1']}$$

Fig. 5. Operational semantics

The elements of *Storable* are objects. They map property names to values. There are three special properties, *VALUE*, `up`, and `this`. The *VALUE* property is used by wrapper objects for primitive types and to store function closures. The `up` property is only used in environment objects that implement lexical scoping. It always points to the next lexically enclosing environment. The `this` property is reserved for the self reference in method calls and constructor functions. We leave the sets *Boolean*, *Number*, and *String* unspecified. Each element of the set *FValue* is a *function closure*. It registers the function name f for recursive use, the location l pointing to the lexically enclosing environment of the function's definition, the names $x_1, \ldots$ of the formal parameters, the names $y_1, \ldots$ of the local variables, and the statement s implementing the function's body.

Each of the reduction relations takes an initial store Σ, a reference to the current environment object l, and a syntactic object and rewrites it into the next state and a transformed object. To express the transformed syntactic objects requires to extend the syntax of expressions by values, which are the results of computations. They are generated by the grammar

$$
\begin{array}{rll}
v ::= & \texttt{undefined} & \\
| & \texttt{null} & \text{null reference} \\
| & c & \text{booleans, numbers, strings} \\
| & l & \text{location} \\
| & \mathit{var}(l, \mathit{str}) & \text{variable reference} \\
| & \mathit{prop}(l, \mathit{str}) & \text{property reference}
\end{array}
$$

The last two cases deserve some further explanation. A variable reference $\mathit{var}(l, \mathit{str})$ consists of the location l of an environment object and a property name (variable name) in that environment. Its main use is to serve as an address in the evaluation of an assignment expression. A property reference $\mathit{prop}(l, \mathit{str})$ consists of the location l of a program object and a property name. It serves as an address for evaluating assignments, but it also plays a role in detecting a method call.

A further new kind of expression is the activation object $\texttt{AR}(l, s)$. The location l is the address of an activation record and s is the statement to execute in this context. Each function application creates a new activation object initialized with the names and values of the parameters, the `this` pointer, the `up` pointer, and the local variables. On entry to an activation $\texttt{AR}(l, s)$, evaluation replaces the currently active context with the activation object l and proceeds with s.

Instead of explaining all the evaluation rules in detail, we concentrate on a few important details. Lookup for `this` only takes place in the toplevel activation object, every other variable is accessed through the scope chain (see definition of $\rightarrow_v$). As yet unknown variables are allocated in the toplevel activation object, which is identified by the absence of an `up` property.

Functions are objects that have a closure stored in the special *VALUE* property. The content of a closure is completely standard.

A function invocation can take three different forms: a function call, a method invocation, and a constructor call. These three forms differ solely in the way that `this` is determined, the rest of the activation object is constructed in the same way every time. In a function call, `this` is set to `null`[2]. For a method call, `this` is the receiving object and in a function called through a `new` expression, `this` is the newly constructed object. The last case is easily identified because its syntax is different. Distinguishing a method call from an ordinary function call requires to evaluate the function part to an address if possible (using the relation $\rightarrow_m$). If the result is a property reference, then we have a method call. Otherwise, it is an ordinary function call.[3]

[2] The standard prescribes that `this` should point to the toplevel activation object.
[3] We omit the `arguments` property to keep the presentation manageable.

6 Type Soundness

The connection between the semantics and the type system is made in the usual way by defining a notion of typed configurations and proving type preservation and progress for that notion. A configuration of Core JavaScript is a triple Σ, l_0, s of a store, a reference to an activation object, and a statement. There is also the auxiliary notion of an expression configuration Σ, l_0, e. The rewrite relations $\rightarrow_s$ and $\rightarrow_e$ of the operational semantics induce corresponding relations on configurations in the obvious way.

In addition to the type environment Γ for variables, there is a heap environment Δ for typing references in the store. It maps store locations to object types of the form $\mathtt{Obj}(\varphi)(\varrho)$. This type assignment must be compatible to the actual store: $\Delta \vdash_h \Sigma$. Compatibility means that $dom(\Delta) = dom(\Sigma)$ and that, for each $l \in dom(\Sigma)$ and $str \in dom(\Sigma(l))$, if $\Delta(l) = \mathtt{Obj}(\varphi)(str : \tau, \varrho)$ then the value $\Sigma(l)(str)$ has type τ. If any of the fields in φ is defined, then its contents describe $\Sigma(l)(\mathit{VALUE})$.

The typing rules for configurations define two judgments $\Delta, \Gamma \vdash_{sc} \Sigma, l_0, s \rhd \tau$ and $\Delta, \Gamma \vdash_{ec} \Sigma, l_0, e : \tau$, for statements and expressions. These judgments are defined analogously to $\vdash_{stm}$ and $\vdash$ but have additional rules for the new expressions introduced by the rewrite steps. Due to space reasons, we only give a few example rules.

$$\frac{\Delta(l) = \mathtt{Obj}(\varphi)(str : \tau, \varrho)}{\Delta, \Gamma \vdash_{ec} \Sigma, l_0, \mathit{var}(l, str) : \tau} \qquad \frac{\Delta, \mathit{TE}(\Delta, l) \vdash_{sc} \Sigma, l, s \rhd \tau}{\Delta, \Gamma \vdash_{ec} \Sigma, l_0, \mathtt{AR}(l, s) : \tau}$$

The second rule for an activation record is particularly interesting because it reconstructs a typing environment from the address l of the activation object of the function and from the heap type Δ. It relies on a function $\mathit{TE}(\Delta, l)$ that traverses the chain of activation objects beginning with l and constructs an environment from the properties (and their types) of the activation objects. We omit its straightforward specification.

Lemma 1 (Type preservation). *Suppose that $\Delta \vdash_h \Sigma$, $\Delta, \Gamma \vdash_{sc} \Sigma, l_0, s \rhd \tau$, and $\Sigma, l_0, s \rightarrow_s \Sigma', s'$.*

Then exists Δ' extending Δ such that $\Delta' \vdash_h \Sigma'$ and $\Delta', \Gamma \vdash_{sc} \Sigma', l_0, s' \rhd \tau$.

Lemma 2 (Progress). *Suppose that $\Delta \vdash_h \Sigma$, $\Delta, \Gamma \vdash_{sc} \Sigma, l_0, s \rhd \tau$.*

Then either $s =$ `skip`, $s =$ `return` v, or there exist Σ' and s' such that $\Sigma, l_0, s \rightarrow_s \Sigma', s'$.

7 Related Work

The main influence to this work is the work on soft typing and dynamic typing [3, 15, 6, 7]. These works define static type systems for dynamically typed languages, Scheme in these cases. The goal of these type systems is to enable compiler optimizations for Scheme programs. Dynamic typing has a twofold impact on the performance of a program. First, there is a memory overhead because each value

must carry with it a representation of its type. In the simplest implementation each value is boxed, that is, it is represented by a pointer to a heap-allocated cell. Clearly, there is also a time penalty for manipulating boxed values. Second, each operation must check that its arguments have the expected type before it can proceed to do the actual work. A soft typing system is able to identify the places where the dynamic type checks may be safely omitted and which values need not carry type information with them. The work by Henglein and Rehof [7] even specifies a translation into ML, a statically typed language that requires no type information at runtime.

Another group of works which is closely related to ours is the construction of type systems for the programming language Erlang [2]. Erlang poses similar problems as JavaScript, but is simpler in some respects. For example, functions have fixed arity and there are no structures comparable with JavaScript's objects. Two type systems have been constructed for Erlang, one based on standard type theory [8] and another one which appears more ad-hoc [10]. Both system work from programmer specified type signatures, whereas our system is targeted towards performing automatic program analysis.

The rows in our object types are clearly inspired by type systems for records [13]. The main difference to a traditional record system is the use of strings as labels, which requires a first-class treatment of labels [5, 9] but in the guise of singleton types.

8 Conclusion

We have presented a first attempt at defining a type system for analyzing a weakly typed scripting language, JavaScript. The system is guided by a matching relation which specifies type convertibility. The matching relation determines how conservative the system is and which conversions should only be flagged instead of being rejected (*e.g.*, converting `null` to an object).

A number of extensions might be considered: the object prototyping mechanism, more general type indices, and polymorphism. However, practical experience is necessary to select the most urgently needed one. An implementation is under way to evaluate the type system with typical JavaScript programs.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, NY, 1993.
3. Robert Cartwright and Mike Fagan. Soft typing. In *Proc. Conference on Programming Language Design and Implementation '91*, pages 278–292, Toronto, Canada, June 1991. ACM.
4. ECMAScript Language Specification. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, December 1999. ECMA International, ECMA-262, 3rd edition.

5. Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Dept. of Computer Science, University of Nottingham, November 1996.
6. Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22:197–230, 1994.
7. Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In Simon Peyton Jones, editor, *Proc. Functional Programming Languages and Computer Architecture 1995*, La Jolla, CA, June 1995. ACM Press, New York.
8. Simon Marlow and Philip Wadler. A practical subtyping system for erlang. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 136–149. ACM Press, 1997.
9. Susumu Nishimura. Static typing for dynamic messages. In Luca Cardelli, editor, *Proc. 25th Annual ACM Symposium on Principles of Programming Languages*, pages 266–278, San Diego, CA, USA, January 1998. ACM Press.
10. Sven-Olof Nyström. A soft-typing system for erlang. In *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pages 56–71. ACM Press, 2003.
11. The Mozilla Organization. Rhino: JavaScript for Java. `http://www.mozilla.org/rhino/`, September 2004.
12. François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 276–290. ACM Press, 1999.
13. Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. The MIT Press, 1994.
14. David Ungar and Randall B. Smith. SELF: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–206, July 1991.
15. Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 19(1):87–152, January 1997.
16. Kwangkeun Yi. An abstract interpretation for estimating uncaught exceptions in standard ML programs. *Science of Computer Programming*, 31(1):147–173, 1998.