Towards an amortized type system for JavaScript

Daniel Franzen and David Aspinall

University of Edinburgh, Edinburgh, UK
{D.Franzen,David.Aspinall}@ed.ac.uk

Abstract

JavaScript programs have access to a wide range of resources and many of those have security implications. Tight bounds on the consumption of those resources can give indication of the functionality provided by the program and minimize the security risks of mobile applications. Resource consumption is typically dependent on the input of the user.

In this paper we introduce an *amortized* type system for a core of JavaScript. The resulting types certify bounds for the resource usage dependent on the input parameters. We define the amortized types and the corresponding typing rules. Furthermore we discuss how to fully automatically infer those resource bounds for arbitrary applications. In addition to the usual example of amortized resource, heap-space, our type system can be applied to many phone specific resources, which we demonstrate using the example of the GPS sensor and others.

The main result of this paper is the soundness of the core type system, proving that a valid type for a program corresponds to a bound on the units used of the specified resource.

1 Introduction

JavaScript plays an important role in our digital lives. Almost all web sites use JavaScript, often as third party code over which the main web site developer has no control. JavaScript code has access to a range of desktop resources, which are limited by the browser's sandbox and origin policy controls. And JavaScript is advancing beyond the browser, for example, together with HTML5, it is the native way of writing mobile applications in the newest generation of mobile phone operating systems, such as Tizen [4] and Firefox OS [1]. Several popular in cross-platform cross-platform programming frameworks, such as PhoneGap [3], are also built around HTML5 and JavaScript. These new environments are of concern because JavaScript code can be granted access to a much wider range of resources than on the desktop, for example, including APIs to access the plethora of phone sensors that may cost the user money, battery life, or privacy.

To access sensors, an application needs a privilege to do so at a certain moment. Take for example a function `logbook`, which tracks the users location. As a parameter it is provided with a list of timestamps. The application saves the current GPS location and the location at each of the timestamps. This application needs access to the GPS sensor. Most operating systems guard the GPS sensor by an access control mechanism to grant GPS access privilege, most often using a *permission* style privilege which works in advance. This means that the user grants the app access to the GPS sensor freely; after that, it can be difficult to tell where and how often the location is being accessed.

Providing extra information about resource usage is complicated, since the typical JavaScript application is a mash-up of different libraries. We propose *automatic inference of resource bounds* to provide a basis on which users can judge the intentions and detailed attack potential of a program. In this paper, we present a type system to statically analyse a fragment of

JavaScript code with a focus on resource consumption. The type system incorporates amortized annotations in all datastructures and functions of the application. The analysis generates constraints between those annotations to describe all execution traces and by solving those constraints we get an upper bound on the resource usage. The amortized annotations inside the datastructures allow us to track reserved resource units dependent on the size of related data structures without dependent type systems. This enables us to prove powerful bounds on the resource usage of mobile applications. Type systems for amortized data structures were introduced by Hofmann et al. in [23] and were extended to object oriented languages in [24]. We apply the idea to JavaScript and consider further resources like phone sensors.

A possible amortized type for the function `logbook` would be:

$$O, \mathit{List}((\mathit{Int}, 1)), 5 \rightarrow \mathit{List}((\mathit{Int}, 0)), 4$$

: the basic type is a function that takes a receiver object O and an integer list as arguments and returns another integer list. The added numeric annotations show that the function needs 5 free unit of the GPS resource to be executed and that 4 of the provided resources are available after the execution. This fifth unit has been used to "pay" for the query for the start location. The annotation in the parameter type $\mathit{List}((\mathit{Int}, 1))$ described that each element in the list contains an integer value and one free resource unit associated with it. This one unit is used to get the user location at the time represented in the integer value. The return value is of type $\mathit{List}((\mathit{Int}, 0))$, which shows that the returned list does not contain any free resource units. If we call the function `logbook(x)` with the variable x of type $\mathit{List}(\mathit{Int}, 4)$, then the data structure x is intuitively split into one list of type $\mathit{List}(\mathit{Int}, 3)$ which stays stored in the variable x and one list of type $(\mathit{Int}, 1)$ which gets consumed by the function.

The application of amortized types to JavaScript exposes new challenges: in class-based languages the only resource consuming operation is the instantiation of a class. In JavaScript the structure of objects can change dynamically, simply by assignment to previously absent fields. This object extension might consume further resources and therefore each assignment to a field has the potential to consume resources. A type system has an advantage here, since it already infers the shape of the values on which the consumption might depend. Furthermore JavaScript allows the programmer to store references to the API functions in custom variables and object fields. This feature is often used in portable code to minimize the size of the source code. Therefore the analysis needs to handle functions as normal values.

Rather than specialising on one resource, we formulated a general resource model: the typing and semantic rules are parametrised by the resource consumption of the different constructs of the language and APIs. To instantiate the model one has to define the consumption of the language operators and the API functions.

As an example for resources we show the definition for heap space consumption, as well as the phone oriented and API based GPS sensor resource. Heap space is a resource that needs to be analysed on the level of the language operations, since each construct might allocate new heap space. Furthermore heap cells can be reserved in chunks (i.e. more than one unit at once) and freed after use. The GPS sensor is only accessed through API functions. Additionally it can only be accessed once at a time and a call to the API function can not be reverted.

The execution of a mobile application often heavily depends on user interaction, i.e. on the input the user provides during runtime. This might be keyed-in numbers and text as well as mouse movements and button presses. The amortized resource model is especially advantageous for those dependencies, since the resource units are associated directly with the data structures and we can represent limits such as $(\text{size(input)} \cdot 5)$ units.

Contributions. We have developed a sound amortized type system for a core of JavaScript:

- We present the types and typing rules depending on the definition of the resource.
- We discuss a type inference algorithm and its main steps.
- We present the corresponding resource annotated semantics.
- We formulate the soundness property that whenever a program is typed, it can be executed with the resources asserted in the type.

This paper is structured as follows: in Section 1 we discussed the importance of resource analysis for JavaScript and general idea of amortized types; in Section 2 we formally introduce our resource model and discuss different kinds of resources for mobile application; Section 3 presents the type system. That includes the definition of the types itself, the typing rules and the outline for the type inference algorithm. Section 4 then defines the corresponding semantic rules and Section 5 shows the soundness result. Finally we discuss related work in Section 6 and conclude in Section 7.

2 Resource modelling

2.1 Amortized annotations

Our system analyses resources in units: examples for a unit are a memory cell, a bigger memory regions or one access to a sensor. Throughout the execution of the program we consider the number n of resource units available to the program and call this collection *freelist*. For memory cells, n is decreased whenever memory is allocated. The resource for sensor accesses is usually only consumed, but sometimes the user might grant more accesses dynamically or accesses might get earned by some protocol. One example of the later would be: the application has to beep every time before the GPS sensor is used.

To be able to perform actions on a data structure of unknown size, we use an amortized approach presented in [23]. Consider for example the `logbook`-function from above. The GPS usage clearly depends on the size of the list. For this purpose, our analysis can store parts of the freelist in data structures. We represent this internal freelists as integer included in the type. For example a value could be typed $(Int, 1)$ to show that the value owns a freelist of size 1. These units can then be used to perform resource consuming operations with this value.

In more complex data structures those annotations let us describe resource usages linearly dependent on the size of the structure: For example a list could be implemented as a recursive object, containing one element as field `head` and another object with a shorter list in the field `tail`. We type this object as $T = \mu\alpha.[(head : (Int, 1)), (tail : (\mu, 0))]$. It asserts, that the `head` of each element in the list contains one reserved resource unit, which can be used to access the GPS sensor once for each element in the list.

Our analysis aims to find minimal values for the different freelists, such that the program can just perform its operations. That includes values for the global freelist and the amortized freelists. A bound on the resource units can then be computed as the sum of all amortized freelists in the initial parameters and the initial global freelist.

2.2 Resource kinds

Our analysis can cover a wide range of resources. Bounds on the resources can defend against various attacks, as discussed in [11]. As examples we consider the PhoneGap framework, which

Parameter	Operation
C_{varR} C_{varW}	read/write variable
C_{memR} $C_{\text{memW}}(t, f)$	read/write object field (member)
C_{funX}	execute function
C_{seq}	concatenation of statements
C_{new}	executing constructor

(a) Resource parameters

$$C_{\text{new}} = 1$$
$$C_{\text{memW}}(\psi) = \begin{cases} 1 & \text{if } \psi = \circ \\ 0 & \text{if } \psi = \bullet \end{cases}$$
$$C_{\text{x}} = 0 \text{ for any other x}$$

(b) Heap resource model

Figure 1: Language level resources

executes JavaScript application on phones and provides an API to access phone functionalities.

- heap space: our system infers bounds on the used heap space. On mobile devices with smaller memory this is particularly important and hackers could attack all phone features.
- GPS sensor: For the location services on mobile devices it is hard to differentiate between the intended use case (for example in a routing application) and a malicious use (tracking the users position).
- notifications (pop-ups and in the top bar) are, always used to confirm a critical operation with the user. Social engineering attacks display multiple notifications to decrease the attention users pay on the important notifications (s. [12]).
- text messages: a common attack on mobile phones is to send multiple premium SMS, charging the user to the advantage of the attacker.
- pictures taken without the knowledge of the user pose a thread to the privacy of the user.

For each of the resource kinds comparing the inferred resource bound with the resources needed for the expected functionality helps the user to decide whether the application can be trusted.

2.3 Resource Model

Some resources types, e.g. heap space, are consumed and returned directly by the language operators. Every assignment might allocate further heap resources. To represent those resource types we use the resource consumption parameters C_{x}. Figure 1(a) shows the parameters and which operation they describe the consumption of and Figure 1(b) shows the values to analyse the heap space.

Assignments to a member field in JavaScript can be either overwriting an existing field or extend the object with a new field. We split C_{memW} into $C_{\text{memW}}(\bullet)$ for overwriting assignment and $C_{\text{memW}}(\circ)$ for extending assignment. Later we are going to use *true* as an alias of $\bullet$ and $false$ as $\circ$ for C_{memW}.

The second part of the resource model *API* is the resource type of the API functions: We notate the resource need of an API function as $[n, n']$, where n describes the units needed to call the API function and n' describes how many of those are usable again after the execution of the function. Examples for different consumption values are shown in Figure 2.

The API `geolocation.watchPosition` accesses the GPS sensor periodically. In order to still infer bounds on this we type it as $[1, 0]$, but additionally require that the callback function, given as parameter, is of type $[1, 0]$. This way we can still infer a resource bound of (1 per time event). The resource consumption can also be weighted as we have done in the notification

Resource	PhoneGap APIs, f	$API(f)$
GPS	geolocation.getCurrentPosition	[1,0]
	geolocation.watchPosition	[1,0] *
SMS	plugins.sms.send	[1,0]
	plugins.socialsharing.shareViaSMS	[1,0]
camera	camera.getPicture	[1,0]
notifications	notification.alert	[3,0]
	notification.confirm	[3,0]
	notification.beep	[1,0]
	notification.vibrate	[1,0]

Figure 2: Resource APIs in PhoneGap

example. The annotation n' can, for example, be used to restrict the number of files opened at the same time. The API function to open a file would have consumption $[1, 0]$, whereas the function to close a file would get the values $[0, 1]$.

In the remainder of this paper we will always assume we are given a resource model (API, C).

3 Types

Figure 3(a) shows the types of JS_0^{T+}. A type in JS_0^T (see [7]) is either Integer (Int), Object (O), Function (G) or a type variable (α). Objects are represented as a list of fields and their types. Each field can be either definite ($\bullet$), which indicates that this field has been allocated already, or potential ($\circ$) which indicates this field might be allocated in the future. Function types consist of the receiver, which will be available in the body of the function as `this`, the type of the parameter and the return type. Note that all function use the variable `x` as parameter, which simplifies the environment.

The amortized annotations appear in two places: First of all, each full type has an amortized annotation n. A value with type $(Int, 5)$ describes a value with the basic type Int and 5 resource units reserved for it. This means the program could access a sensor 5 times, resulting in a new type $(Int, 0)$ for the given value.

Additionally we annotate the function types: we add one annotation n to the parameter side and n' to the return side of the type. This describes that the function needs n units of the resource as input to be executed and n' of those units will be freed after the execution has finished. Thus a function that consumes 2 resource unit and needs an additional unit temporarily has the type $O, T, 3 \to T', 1$.

To extend results about JS_0^T to JS_0^{T+}we define the *erasure* $[\![\cdot]\!]^-$ (Figure 3(b)) of an annotated type. The erased type is a type with the same structure, but no annotations. This intuition can be proven by induction on the definition of the erasure operation.

3.1 Subtypes

For an annotated type to be a subtype of another $t \leq t'$ we require the erased types to be in the same relation: $[\![t]\!]^- \leq [\![t']\!]^-$. Furthermore we have two cases: for assignments `x=e` we have the subtyping $\leq^-$: the type of the variable x has already resource units reserved for its annotations. Thus we get no constraints on the annotations of the value e. In function calls `f(e)` the function assumes that the argument e has reserved resource. In this case all annotations in the actual

$$
\begin{array}{rcll}
t & ::= & O \mid G \mid Int \mid \alpha & \text{(basic type)} \\
t^+ & ::= & (t, n) & \text{(full type)} \\
O & ::= & \mu\alpha.M \mid M & \text{(object)} \\
M & ::= & [(m : tm)^*] & \text{(memberlist)} \\
tm & ::= & (t^+, \psi) & \text{(membertype)} \\
G & ::= & \mu\alpha.(O \times t, n \to t, n) & \text{(function)} \\
\psi & ::= & \bullet \mid \circ & \text{(field state)}
\end{array}
$$

where $n \in \mathbb{N}$ and m ranges over strings

(a) Types in JS_0^{T+}

$$
\begin{array}{rcll}
[\![Int]\!]^- & = & Int & \text{(integer)} \\
[\![(t, n)]\!]^- & = & [\![t]\!]^- & \text{(annotated)} \\
[\![\mu\alpha.M]\!]^- & = & \mu\alpha.[\![M]\!]^- & \text{(obj)} \\
[\![[(m : tm)^*]]\!]^- & = & [(m : [\![tm]\!]^-)^*] & \text{(memberlist)} \\
[\![(t^+, \psi)]\!]^- & = & ([\![t^+]\!]^-, \psi) & \text{(member)} \\
[\![\mu\alpha.(O \times t, n \to t', n')]\!]^- & = & & \text{(function)} \\
 & & \mu\alpha.([\![O]\!]^- \times [\![t]\!]^- \to [\![t']\!]^-) & \\
[\![\alpha]\!]^- & = & \alpha & \text{(recursion)}
\end{array}
$$

(b) Erasure

Figure 3

$$\bullet \leq^\delta \circ \quad \text{(ST-POT)}$$

$$\frac{t \equiv t' \qquad \psi \leq^\delta \psi'}{(t, \psi) \leq^\delta (t', \psi')} \quad \text{(ST-MEM)}$$

$$\frac{t \equiv t'}{t \leq^\delta t'} \quad \text{(ST-EQUIV)}$$

$$\frac{n \leq N \qquad n - n' \leq N - N'}{(O \times t, n \to t', n') \leq^\delta (O \times t, N \to t', N')} \quad \text{(ST-FUN)}$$

$$\frac{\forall m : O'(m) = (t', \psi') \Rightarrow (O(m) = (t, \psi) \wedge (t, \psi) \leq^\delta (t'\psi'))}{O \leq^\delta O'} \quad \text{(ST-OBJ)}$$

$$\frac{n \geq n'}{(t, n) \leq^+ (t', n')} \qquad \frac{}{(t, n) \leq^- (t', n')} \quad \text{(ST-AMORT)}$$

(a) Subtypes

$$\psi \sqcup \psi' = \begin{cases} \bullet & \text{if } \psi = \bullet \text{ and } \psi' = \bullet \\ \circ & \text{otherwise} \end{cases} \quad \text{(M-POT)}$$

$$n \sqcup n' = \min(n, n') \quad \text{(M-AMORT)}$$

$$(t', \psi') \sqcup (t'', \psi'') = (t' \sqcup t'', \psi' \sqcup \psi'') \quad \text{(M-MEM)}$$

$$t \sqcup t = t \quad \text{(M-EQUAL)}$$

$$\frac{\begin{array}{c} Dom(O_1) = Dom(O_2) = Dom(O_3) \\ \bigwedge_{i=1,2,3} O_i(m) = (t_i, n_i, \psi_i) \\ \Rightarrow (t = t' \sqcup t'' \wedge n = n' \sqcup n'' \wedge \psi = \psi' \sqcup \psi'') \end{array}}{O_2 \sqcup O_3 = O_1} \quad \text{(M-OBJ)}$$

(b) Minimum

$$\frac{\begin{array}{c} M \equiv M' \qquad t_1 \equiv t_1' \qquad t_2 \equiv t_2' \\ f = \mu\alpha.(M \times t_1, n \to t_2, n') \\ f' = \mu\alpha.(M' \times t_1', n \to t_2', n') \end{array}}{f \equiv f'} \quad \text{(EQ-FUNC)}$$

$$\frac{t \equiv^+ t'}{(t, n) \equiv^+ (t', n)} \qquad \frac{t \equiv^- t'}{(t, n) \equiv^- (t', n')} \quad \text{(EQ-ANNOT)}$$

(c) Equality

Figure 4: Type Relations

parameter need to be as least as great as in the type of the formal parameter x. This subtyping relation is notated $\leq^+$. The notation $\leq^\delta$ is used for either relation. The rules for subtyping are in Figure 4(a).

The relation $\equiv$ is defined for JS_0^T in [8] to reflect equality on types modulo unfolding of recursive types. We split the relation into $\equiv^+$ and $\equiv^-$ and add equality for the annotations in $\equiv^+$. The changed rules EQ-FUNC and EQ-ANNOT can be found in Figure 4(c).

3.2 Type checking

The type judgement we check has the form:

$$P, \Gamma, n \vdash e : t || \Gamma', n'$$

The environments Γ and Γ' are mappings from variable names $\{\texttt{this}, x\}$ to annotated types t^+ and assert the current type of the variables. The numbers n and n' are the resource annotations for this evaluation of the expression e and t is its type. The program P is a mapping from function names to typed function definitions, so

$$P(f) = \texttt{function}\ \ f(x) : G\{e\}$$

where f is the function name, G is the function type and e is the body of the function. P needs to respect the function $API(f)$. That means for $API(f) = [n, n']$ we require

$$P(f) = \texttt{function}\ \ f(x) : (O \times t, n \to t', n')\{\}$$

Intuitively the type judgement asserts that given P and Γ the expression e evaluates to a value of type t and after the evaluation Γ has been updated to Γ'. The evaluation needs no more then n resource units and n' of those will be reusable after the evaluation.

The typing rules in Figure 5(a) describe the type checking procedure for the annotated types. The relation $\sqcup$ (shown in Figure 4(b)) expresses the minimum of two types or environments. From the construction we get that $(t \sqcup t') \leq^+ t$ and $(t \sqcup t') \leq^+ t'$.

As an example let us consider the rule T-MEMR. It describes a read operation of an object member: The first precondition uses the type judgement recursively to assign the type of the object e and the second precondition reads the type of the member m from this type. The remaining resource units are n' left after the evaluation of e together with the units that were stored in the member m. From that we have to subtract the C_{memR} resources for the read operation.

Typing rule T-VARR uses a splitting constraint $t_1 \oplus t_2 = t_3$. Intuitively this constraint asserts that the three types t_1, t_2 and t_3 describe objects with the same structure, but t_1 and t_2 share the freelists previously hold by t_3. The definition is presented in Figure 6.

Proposition 1. *The splitting relation preserves the structure of the type:* $t_1 \oplus t_2 = t_3 \Rightarrow [\![t_1]\!]^- = [\![t_2]\!]^- = [\![t_3]\!]^-$

Proof. by structural induction on the derivation of the splitting relation. □

In order to transfer properties of the original type system to our annotated system, we need to prove the correspondence between the type and its erasure. For that we extend the erasure to the environment Γ and the program P:

$$\begin{aligned} [\![\Gamma]\!]^- = &\quad \{\texttt{this} : [\![\Gamma(\texttt{this})]\!]^-, x : [\![\Gamma(x)]\!]^-\} \\ [\![P]\!]^- : &\quad m \mapsto \texttt{function}\ \ f : [\![G]\!]^-\{...\}, \\ &\quad \texttt{for}\ \ P(m) = \texttt{function}\ \ f : G[n, n']\{...\} \end{aligned}$$

Proposition 2. *(Annotation erasure) If given* Γ, P *an expression* E *has type* t*, then* E *has type* $[\![t]\!]^-$ *in* $[\![\Gamma]\!]^-$ *and* $[\![P]\!]^-$

$$P, \Gamma, n \vdash E : t || \Gamma', n' \qquad \Rightarrow \qquad [\![P]\!]^-, [\![\Gamma]\!]^- \vdash E : [\![t]\!]^- || [\![\Gamma']\!]^-$$

Proof. by structural induction on the derivation of the type in JS_0^{T+}. □

$$\frac{}{P,\Gamma,n \vdash \texttt{null} : O||\Gamma,n} \quad \text{(T-NULL)}$$

$$\frac{}{P,\Gamma,n \vdash i : Int||\Gamma,n} \quad \text{(T-INT)}$$

$$\frac{\Gamma(V) = (t,n_1) \qquad t' \oplus t'' \leq t \qquad \Gamma' = \Gamma[V \mapsto (t',n_2)] \qquad V \in \{\texttt{this}, x\}}{P,\Gamma,n + C_{\text{varR}} \vdash V : t''||\Gamma',n + n_1 - n_2} \quad \text{(T-VARR)}$$

$$\frac{P,\Gamma,n \vdash e : t_1||\Gamma',n_1' \qquad \Gamma'(x) = (t_2,n_2) \qquad \Gamma'' = \Gamma'[x \to (t_2,n_3)] \qquad t_1 \leq^- t_2}{P,\Gamma,n + C_{\text{varW}} \vdash x = e : t_1||\Gamma'',n_1' + n_2 - n_3} \quad \text{(T-XW)}$$

$$\frac{P,\Gamma,n \vdash e : O||\Gamma',n_1' \qquad O(m) = ((t,n_2),\bullet)}{P,\Gamma,n + C_{\text{memR}} \vdash e.m : t||\Gamma',n_1' + n_2} \quad \text{(T-MEMR)}$$

$$\frac{P,\Gamma,n \vdash e : t_1||\Gamma',n_1' \qquad \Gamma'(var) = (O,n_2) \qquad O(m) = ((t_3,n_3),\psi) \qquad O' = O[m \mapsto ((t_1,n_5),\bullet)] \qquad \Gamma'' = \Gamma'[var \mapsto (O',n_6)] \qquad n' = n_1' + n_2 + n_3 - n_5 - n_6 - C_{\text{memW}}(\psi)}{P,\Gamma,n \vdash var.m = e : t_1||\Gamma'',n'} \quad \text{(T-MEMW1)}$$

$$\frac{P,\Gamma,n \vdash e_1 : O||\Gamma',n_1' \qquad P,\Gamma',n_1' \vdash e_2 : t_2||\Gamma'',n_2' \qquad O(m) = ((t_3,n_3),\bullet) \qquad t_2 \leq^- t_3 \qquad n' = n_2' + n_3 - C_{\text{memW}}(\bullet)}{P,\Gamma,n \vdash e_1.m = e_2 : t_2||\Gamma'',n'} \quad \text{(T-MEMW2)}$$

$$\frac{P,\Gamma,n \vdash e_1 : O_1||\Gamma',n_1' \qquad O_1(m) = ((G,n_2),\bullet) \qquad P,\Gamma',n_1' + n_2 \vdash e_2 : t_3||\Gamma'',n_3' \qquad G = \mu\alpha.(O_4 \times t_4, n_4 \to t_4', n_4') \qquad O_1 \leq^+ O_4, t_3 \leq^+ t_4 \qquad n_3' \geq n_4 \qquad n' = n_3' - n_4 + n_4' - C_{\text{memR}} - C_{\text{funX}}}{P,\Gamma,n \vdash e_1.m(e_2) : t_4'||\Gamma'',n'} \quad \text{(T-MEMX)}$$

$$\frac{P(f) = \texttt{function } f(x) : G\{...\}}{P,\Gamma,n \vdash f : G||\Gamma,n} \quad \text{(T-FUNR)}$$

$$\frac{P,\Gamma,n \vdash e : t_1||\Gamma',n_1' \qquad P(f) = \texttt{function } f(x) : G\{...\} \qquad G = \mu\alpha.(O \times t_3, n_3 \to t_3', n_3') \qquad \{m \mid O(m) = (t',\bullet)\} = \emptyset \qquad t_1 \leq^+ t_3 \qquad n_1' \geq n_3 \qquad n' = n_1' - n_3 + n_3' - C_{\text{funX}}[-C_{\text{new}}]}{P,\Gamma,n \vdash [\texttt{new }] f(e) : t_3'||\Gamma',n'} \quad \text{(T-FUNX)}$$

$$\frac{P,\Gamma,n \vdash e_1 : t_1||\Gamma',n_1' \qquad P,\Gamma',n_1' \vdash e_2 : t_2||\Gamma'',n_2'}{P,\Gamma,n \vdash e_1;e_2 : t_2||\Gamma'',n_2' - C_{\text{seq}}} \quad \text{(T-SEQ)}$$

$$\frac{\Gamma,n \vdash e_1 : Int||\Gamma',n_1' \qquad \Gamma',n_1' \vdash e_2 : t_2||\Gamma_2,n_2' \qquad \Gamma',n_1' \vdash e_3 : t_3||\Gamma_3,n_3'}{\Gamma,n \vdash e_1\,?\,e_2 : e_3 : t_2 \sqcup t_3||\Gamma_2 \sqcup \Gamma_3, \min(n_2',n_3')} \quad \text{(T-COND)}$$

(a) Typing rules

$$\frac{}{var,H,S \xrightarrow[n]{n+C_{\text{varR}}} S(var),H,S \qquad var \in \{x,\texttt{this}\}} \quad \text{(S-VARR)}$$

$$\frac{e,H,S \xrightarrow[n']{n} v,H',S'' \qquad S' = \{\texttt{this} \mapsto S''(\texttt{this}), x \mapsto v\}}{x = e,H,S \xrightarrow[n']{n+C_{\text{varW}}} v,H',S'} \quad \text{(S-XW)}$$

$$\frac{e,H,S \xrightarrow[n']{n} \iota,H',S'}{e.m,H,S \xrightarrow[n']{n+C_{\text{memR}}} H'(\iota)(m),H',S'} \quad \text{(S-MEMR)}$$

$$\frac{e_1,H,S \xrightarrow[n_1']{n_1} \iota,H_1,S_1 \qquad e_2,H_1,S_1 \xrightarrow[n_2']{n_2} v,H_2,S' \qquad H' = H_2\{\iota.m \lhd v\} \qquad \varphi = (m \in Dom(H_2(\iota))) \qquad (n,n') = con((n_1,n_1'),(C_{\text{memW}}(\varphi),0),(n_2,n_2'))}{e_1.m = e_2,H,S \xrightarrow[n']{n} v,H',S'} \quad \text{(S-MEMW)}$$

$$\frac{e_1,H,S \xrightarrow[n_1']{n_1} \iota,H_1,S_1 \qquad H_2(\iota)(m) = f \qquad f(e_2),H_1,S_1 \xrightarrow[n_2']{n_2} v,H',S' \qquad (n,n') = con((n_1,n_1'),(C_{\text{memR}},0),(n_2,n_2'))}{e_1.m(e_2),H,S \xrightarrow[n']{n} v,H',S'} \quad \text{(S-MEMX)}$$

$$\frac{e,H,S \xrightarrow[n_1']{n_1} v',H_1,S' \qquad [n_f,n'f] = \begin{cases} API(f) & f \in API \\ [0,0] & \text{otherwise} \end{cases} \qquad P(f) = \texttt{function } f(x)\{e'\} \qquad e',H_1,\{\texttt{this} \mapsto \mathcal{U}df, x \mapsto v'\} \xrightarrow[n_2']{n_2} v,H',S'' \qquad (n,n') = con((n_1,n_1'),(C_{\text{funX}},0),(n_f,n_f'),(n_2,n_2'))}{f(e),H,S \xrightarrow[n']{n} v,H',S'} \quad \text{(S-FUNX)}$$

$$\frac{e,H,S \xrightarrow[n_1']{n_1} v',H_1,S' \qquad P(f) = \texttt{function } f(x)\{e'\} \qquad \iota \text{ is new in } H_1 \text{ and } H_2 = \{\iota \mapsto []\} \qquad e',H_2,\{\texttt{this} \mapsto \iota, x \mapsto v'\} \xrightarrow[n_2']{n_2} v,H',S'' \qquad (n,n') = con((n_1,n_1'),(C_{\text{funX}} + C_{\text{new}},0),(n_f,n_f'),(n_2,n_2'))}{new\ f(e),H,S \xrightarrow[n']{n} \iota,H',S'} \quad \text{(S-NEW)}$$

$$\frac{e_1,H,S \xrightarrow[n_1']{n_1} v',H_1,S_1 \qquad e_2,H_1,S_1 \xrightarrow[n_2']{n_2} v,H',S' \qquad (n,n') = con((n_1,n_1'),(C_{\text{seq}},0),(n_2,n_2'))}{e_1;e_2,H,S \xrightarrow[n']{n} v,H',S'} \quad \text{(S-SEQ)}$$

$$\frac{e_1,H,S \xrightarrow[n_1']{n_1} v',H'',S'' \qquad e_2,H'',S'' \xrightarrow[n_2']{n_2} v,H',S' \qquad v' > 0 \qquad (n,n') = con((n_1,n_1'),(n_2,n_2'))}{e_1\,?\,e_2 : e_3,H,S \xrightarrow[n']{n} v,H',S'} \quad \text{(S-TRUE)}$$

(b) Semantics

Figure 5: Inference Rules

$$\frac{t_i = (Int, n_i) \qquad n_1 + n_2 = n_3}{t_1 \oplus t_2 = t_3} \quad \text{(SP-Int)}$$

$$\frac{t_i = (G, n_i) \qquad n_1 + n_2 = n_3}{t_1 \oplus t_2 = t_3} \quad \text{(SP-Fun)}$$

$$\frac{t_i = (t'_i, \psi) \qquad t'_1 \oplus t'_2 = t'_3}{t_1 \oplus t_2 = t_3} \quad \text{(SP-Field)}$$

$$\frac{t_i = (O_i, n_i) \qquad \begin{array}{c} O_1 \oplus O_2 = O_3 \\ n_1 + n_2 = n_3 \end{array}}{t_1 \oplus t_2 = t_3} \quad \text{(SP-Obj)}$$

$$\frac{\begin{array}{c} m \in M_1 \Leftrightarrow m \in M_2 \Leftrightarrow m \in M_3 \\ \forall m \in M_1 : M_1(m) \oplus M_2(m) = M_3(m) \end{array}}{M_1 \oplus M_2 = M_3} \quad \text{(SP-Mem)}$$

Figure 6: Splitting Relation

3.3 Type inference

Proposition 2 shows that a type derivation in JS_0^{T+}can be reduced to a derivation in JS_0^T. The other way around, if we derive the type in JS_0^T, we annotate the derivation tree with fresh variables and obtain a set of linear constraints on those variables. With this idea we get a type inference algorithm with the following steps.

Infer basic types:
The type inference for $JS_0^T$is given in [7].

Annotation Constraints generation:
As the next step we go through the derivation tree and replace each rule in $JS_0^T$with the corresponding rule in JS_0^{T+}. Doing this we just add fresh annotation variables to all types and constraints on those variables to the preconditions. As we know the structure of all types already, we can replace each type variable by the appropriate final type. This way we can convert splitting and subtyping constraints into linear constraints on the annotations, too. All occurring annotation constraints define a minimisation linear programming problem (LPP).

$$\begin{array}{rl} \text{Minimize} & c^T \vec{n} \\ \text{with} & A\vec{n} \leq b \\ \text{and} & \vec{n} \geq 0 \end{array}$$

As all constraints are taken from the typing rules, we see that the constraints have at most 6 variables, so the constraint matrix A will be sparse. All coefficients are 0,1 or -1, whereas the constant term might take arbitrary values depending on the resource model. The variables are all positive, since we do allow negative resource units. As a consequence we can avoid one step in the rewriting process into standard form. In each constraint the number of positive and negative coefficients only differs by at most one. Most typing rule will be translated by the system into at most two constraints. The exception is the the rule T-VarR as it uses the splitting relation $\oplus$. This is translated into as many linear constraints as the given datastructure has nodes, thus it is linear in the size of the datastructure.

Annotation Constraint solving:
The LPP can be solved by standard techniques. With the solution we can replace the annotation variables by values and obtain a full type for the analysed expressions. If the LPP solver does not find a finite solution, the analysis outputs an infinite bound. That might be, because the

application uses unbounded many resources or the worst case estimate used in our analysis was not precise enough.

So far we have always spoken about the annotations n to be integers. As it should not be possible to consume fractions of a resources this is reasonable for usages in the resource model. On the other hand a used resource unit can be composed of different parts tracked through different data structures. For example a function calculating the sum of two lists, element by element, might take the needed heap units for the resulting list half from either of the input lists, such that we get a function type of $O, List((Int, 0.5)) \times List((Int, 0.5)), 0 \to List((Int, 0)), 0$. Therefore we can allow the annotations to have decimal values, too. Due to this modification the final resource consumption might be fractional as well for example 7.43 units. Since all resource consuming operations consume only whole units this means, that at least 0.43 units are always stored in amortized annotations and the program in fact is executable with 7 units. As a side-effect we solve general LPPs in polynomial time instead of integer LPPs.

We have implemented those steps in Haskell using the package hjs[2] to parse JavaScript and the GNU Linear Programming Kit to solve the LPP.

3.4 Inferred bounds

With our analysis we can infer different kinds of bounds. To illustrate them we show small examples with their typing and their meaning. We use the PhoneGap function `getCurrentPosition(callback)` as the resource consuming function. On success the function `callback` is executed with the current location as argument. We use some syntax simplifications to make the examples easier to read including the function `wait_till(timestamp)`, which waits till the given timestamp is in the past.

```
function wait_and_locate(timestamp) {
    wait_till(timestamp);
    getCurrentPosition(callback);
    1;


}
```

```
function logbook(list) {
    wait_and_locate(list.head);
    if (list.tail) {
        logbook(list.tail);
    }
    1;
}
```

The function `wait_and_locate` will simply wait for a specific time and then send the current location to the function callback. We infer the function type $null \times Int, 1 \to Int, 0$. This function type describes a **constant bounds** of 1 resource unit.

Function `logbook` can be typed as $null \times \mu\alpha\{head : (Int, 1), tail : (\alpha, 0)\}, 0 \to Int, 0$. We see that each element of the recursive list carries one resource unit with it. So the overall resource consumption of this function is **input dependent**, namely size($list$).

JavaScript has no direct way for programming with concurrency. It is however often used in an event-driven way. Programs initialise themselves and then wait for events as for example time and user input to execute the functionality associated with this event. For example the function `setInterval(callme,milisec)` executes a given function repeatedly with the given rate. As we know the function type of the event handler, we can describe bounds dependent on the number of occurred events: if we find `setInterval` with the handler function `wait_and_locate` as parameter `callme` we get an **event dependent** bound of 1 unit per time event. This works equally with user interaction event registered using the `addEventListener`.

$$con((n_1, n_1'), (n_2, n_2')) = (n, n')$$

$$\text{with } n = \begin{cases} n_1 & \text{if } n_1' \geq n_2 \\ n_2 + (n_1 - n_1') & \text{otherwise} \end{cases}$$

$$\text{and } n' = n - (n_1 - n_1') - (n_2 - n_2')$$

$$con(p_1, ..., p_n) = con(con(p_1, ..., p_{n-1}), p_n)$$

(a) Concatenation operator

$$\begin{aligned} \Sigma(\Gamma) &= \Sigma(\Gamma(\texttt{x})) + \Sigma(\Gamma(\texttt{this})) \\ \Sigma((t, n)) &= n + \Sigma(t) \\ \Sigma([(m : tm)^*] &= \sum_m (\Sigma(tm)) \\ \Sigma((tp, \psi)) &= \Sigma(tp) \\ \Sigma(G) &= 0 \\ \Sigma((\alpha, n)) &= n \end{aligned}$$

(b) Bound extraction

Figure 7: Notations

4 Semantics

To relate our typing system to real resource usage, we annotate the JS_0^T semantics with resource usage. The annotated evaluation judgement looks like this:

$$e, H, S \xrightarrow[n']{n} e', H', S'$$

It asserts that the expression e in the heap H and the Stack S will evaluate to e' and change the heap to H' and the stack to S'. The heap H is a map from addresses to objects, which are themselves maps from field names to values. The stack S maps the variables to values. Values in this context consist of Integers, Functions, Addresses, `null` and $\mathcal{U}df$. Further details can be found in [9]. The evaluation judgement further states that during the evaluation of e a freelist with n elements is necessary and n' of those resource units will be available afterwards. In the semantics we use the same map P, that maps function names to function definitions and types.

The rules to infer the evaluation of member expressions can be found in Figure 5(b). Some rules combine the resource usage of multiple expressions. For those cases we define the concatenation operator in Figure 7(a).

This operator computes how many units are needed for the concatenated operation, depending on the relationship between the two operations' resource consumption. The extension of *con* to multiple pairs makes sense, since the operation is associative.

The annotations to a given evaluation are not unique:

Definition 3. We call an evaluation $e, H, S \xrightarrow[n']{n} e', H', S'$ of an expression e *minimal* if for every other evaluation can be expressed as $e, H, S \xrightarrow[n'+c]{n+c} e', H', S'$

5 Soundness

We are now going to relate the type system and the semantics. Ultimately we want to prove that the amortized annotations can be used as an upper bound for the resource units the program is going to consume. We extract this upper bound as the sum of all resource annotations in the initial typing context as defined as the Σ operator in Figure 7(b).

Furthermore we need to relate the typing context with in the typing relation with the heap and the stack in the evaluation relation. For that we define, what it means for a value to be compatible with a type. The definitions of compatible do not depend on the annotations and are equivalent to [7].

Definition 4. Given a heap H and a program P, we say that $A \subset (Val \times Type)$ is an agreement relation if the following conditions are satisfied:

- if $(null, t) \in A$, then $t = O$ for some object type O.
- if $(n, t) \in A$, then $t = Int$
- if $(f, t) \in A$, then $P(f) = \texttt{function}\ f(x) : G$ with $G = t$
- if $(\iota, t) \in A$, then $t = O$ for some well-formed O, $H(\iota) = \{m_1 : v_1, ..., m_p : v_p\}$ and
  - $O(m) = ((t', n'), \bullet) \Rightarrow$
    $m \in Dom(H(\iota))$ and $(H(\iota)(m), t') \in A$
  - $O(m) = ((t', n'), \circ)$ and $m \in Dom(H(\iota)) \Rightarrow (H(\iota)(m), t') \in A$

Definition 5. Value v is compatible with type t in H

$$P, H \vdash v \blacktriangleleft t$$

if there exists an agreement relation A on H and P with $(v, t) \in A$.
We furthermore relax the notation to annotated types:

$$P, H \vdash v \blacktriangleleft (t, n) \qquad \Leftrightarrow \qquad P, H \vdash v \blacktriangleleft t$$

Definition 6. A program P, the type environment Γ, the Heap H and the stack S are compatible

$$P, \Gamma \vdash H, S \diamondsuit$$

if $P, H \vdash S(\texttt{this}) \blacktriangleleft \Gamma(\texttt{this})$ and $P, H \vdash S(\texttt{x}) \blacktriangleleft \Gamma(\texttt{x})$

Finally we need to relate the annotations in the function types with the actual behaviour:

Definition 7. We call a program P consistent with a compatible stack S, heap H and typing environment Γ, if for each function f with the definition $P(f) = \texttt{function}\ f(x) : G\{e\}$, $G = (O \times t, n \to t', n')$ and the minimal evaluation $e, H, S \xrightarrow[n'_s]{n_s} v, H, S$ holds that

$$\begin{aligned} \Sigma t + n &\geq n_s \\ \Sigma t + n - (\Sigma t' + n') &\geq n_s + n'_s \end{aligned}$$

Theorem 8. *(Soundness) Let an expressions with the type judgement*

$$P, \Gamma, n \vdash e : t || \Gamma', n'$$

with $N = \Sigma\Gamma + n$ and $N' = \Sigma\Gamma' + n' + \Sigma t$ and a compatible stack and heap

$$P, \Gamma \vdash H, S \diamondsuit$$

and the consistent program P be given.
If there exists an evaluation then for the minimal evaluation

$$e, H, S \xrightarrow[n'_s]{n_s} v, H', S'$$

we have $n_s \leq N$ and $n_s - n'_s \leq N - N'$

Proof. This proof is a structural induction on the evaluation derivation. □

6 Related work

6.1 Resource Analysis

To the best of our knowledge this is the first attempt to fit amortized annotations to JavaScript.

The amortized annotations used here are inspired by the work by Hofmann et al. starting with [22]. They developed similar annotations as we used to type functional programs in [23]. Later in [24] this work was also extended to handle object-oriented languages on the example of Resource Aware Java (RAJA), which is able to automatically infer amortized types for object-oriented programs. To do this the authors introduce *views*, which assign different annotations to the class-types in different contexts. Although Hofmann et al. mention that the analysis is not specialized on heap usage, they do not make any suggenstions how to extend the analysis to other resources as treated in this paper by the API function. In other work [21, 20] the amortized annotations are extended to infer polynomial resource bounds, which could improve our analysis as well.

Another approach to infer similar resource bounds is used by Albert et al. [6, 5]. They present a system to infer and solve cost relations, describing the amount of resources required to run the application. The cost relations are more complicated than the typing and integer constraints we use in this work and the solution algorithm is not complete.

Aspinall et al. develop a logic for resources in [10]. This logic treats the resource kind general as we do in this paper. The logic is intended to reason about resource usage in a fragment of the Java VML and is expressed in Isabelle. Since the challenges in JavaScript are quite different from those in Java, those results are not directly portable.

6.2 JavaScript Formalisations

The type system JS_0^T was introduced by Anderson in [8]. All details about the constraint solving algorithm to infer the types can be found in [7].

Other type systems for JavaScript include [13, 25, 28]. The first presents a dependent type system for JavaScript. The nested refinements within the types make the type checking more complicated and the paper does not aim for inference. The type systems presented in the second and third one cover e.g. arbitrary field extensions of objects rather then the potential/definite notation we used in this paper. We chose JS_0^T, because it is a good compromise between handling the dynamic features, while still providing type inference.

Hedin and Sabelfeld [19] also employed a type system for JavaScript to track information flow by assigning either a high or a low confidentiality type to each value and each output. Type errors then show highly confidential values transmitted to a low confidential output. Their type system is much easier and thus not expressive enough to make assertions about the resources we analyse.

Gardner et al. propose a program logic for JavaScript in [17]. With this logic they are able to assert and reason about complex properties of JavaScript programs potentially including resource consumptions. The expressiveness of the logic comes with the requirement of highly detailed and complicated preconditions. In [16] they present the tool JuS, which can infer properties automatically. However recursive functions are not handled and loops require annotations of invariants. This restriction makes it harder to infer properties dependent on the size of recursive data-structures as we do.

In [18] Guha et al. present a subset λ_{JS} and a method to desugar JavaScript applications into this subset. As an use-case they present a type system to exclude certain API methods. The correctness of the desugaring process is only shown with examples and not formally proven.

Another subset of JavaScript, which can be analysed for resource consumption is introduced in [27] by Miller et al. This framework consists of a static checker, which infuses additional runtime enforcements. They handle the `eval` operation by only allowing an even further restricted subset of JavaScript, which gets parsed and validated before execution. The goal is to provide an object capability language to prevent certain attacks mostly focused on privacy and privilege escalation. In some use-cases JavaScript applications are signed by the developer or a trusted party and runtime enforcements would invalidate those signatures. Thus our systems does not incorporate code manipulation.

In [26] Maffeis et al. show an operational semantics for the whole JavaScript standard. Since JavaScript provides multiple challenges for analysis, we did not work on the whole of JavaScript, but restricted our work to JS_0^T.

The tool ADsafe, using JSLint [14], also checks JavaScript code. The objective here is to restrict the application to a saver subset of JavaScript code. JSLint is the analysis tool, which checks preconditions and ADsafe then adds dynamic checks that ensure the wanted security properties. Eliopoulos et al. have verified ADsafe's properties formally in [15] using a type system. Our analysis aims to infer the bound directly, whereas ADsave uses the analysis to verify the preconditions for the dynamic restrictions.

7 Conclusion and Future Work

We have shown a system to automatically infer bounds on the resource consumption for a core of JavaScript. The system infers the types for the used values first and then solves a set of linear constraints describing the needed resource units. The resulting bounds are input size dependent and interaction dependent. The type system has been proven correct in relation to the annotated operational semantics we provided and is expressive enough to cover the basic constructs of JavaScript. For future work we will focus on closing the gap between $JS_0^{T+}$and JavaScript, which includes handling the JavaScript scope chain and aliasing, loops and strings.

The result of our system we aim to provide confidence about resource consumption of third party JavaScript programs by providing bounds on various resource consumptions for JavaScript application. Since the typing serves a certificate of the bound and can be checked much easier than generated, our analysis is suitable for Proof Carrying Code scenarios, in web pages and mobile phone applications.

References

[1] Firefox OS. `https://www.mozilla.org/en-US/firefox/os/`, Apr. 2014.

[2] Haskell JavaScript Parser and Interpreter. http://www.haskell.org/ haskellwiki/Libraries_and_tools/HJS, June 2014.

[3] PhoneGap. `http://phonegap.com/`, Feb. 2014.

[4] Tizen. `https://www.tizen.org/`, Apr. 2014.

[5] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning*, pages 161–203, 2011.

[6] E. Albert, S. Genaim, and A. N. Masud. More precise yet widely applicable cost analysis. In *Verification, Model Checking, and Abstract Interpretation*, pages 38–53, 2011.

[7] C. Anderson and S. Drossopoulou. *Type inference for JavaScript.* PhD thesis, University of London, 2006.

[8] C. Anderson and P. Giannini. Type checking for JavaScript. In *In WOOD '04, volume WOOD of ENTCS. Elsevier, 2004.*, 2004.

[9] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *ECOOP 2005 - Object-Oriented Programming*, pages 428–452. Springer Berlin Heidelberg.

[10] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resources. *Theoretical Computer Science*, pages 411–445, 2007.

[11] D. Aspinall, P. Maier, and I. Stark. Safety guarantees from explicit resource management. In *Formal Methods for Components and Objects*, pages 52–71. Springer, 2008.

[12] R. Böhme and J. Grossklags. The security cost of cheap user interaction. In *Proceedings of the 2011 Workshop on New Security Paradigms Workshop (NSPW*, pages 67–82. ACM, 2011.

[13] R. Chugh, D. Herman, and R. Jhala. Dependent types for Java Script. In *ACM SIGPLAN Notices*, pages 587–606, 2012.

[14] D. Crockford. Jslint: The javascript code quality tool. *URL http://www. jslint. com*, 2011.

[15] S. A. Eliopoulos, J. G. Politz, S. Krishnamurthi, and A. Guha. Type-based verification of JavaScript sandboxing. *USENIX Security Symposium, 2011*, 2011.

[16] P. Gardner and G. Smith. JuS: Squeezing the sense out of javascript programs. *JSTools@ ECOOP*, 2013.

[17] P. A. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for JavaScript. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 31–44, 2012.

[18] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *Proceedings of the 24th European conference on Object-oriented programming (ECOOP)*, pages 126–150. Springer-Verlag, 2010.

[19] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. *2012 IEEE 25th Computer Security Foundations Symposium*, pages 3–18.

[20] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. In *ACM SIGPLAN Notices*, pages 357–370, 2011.

[21] J. Hoffmann and M. Hofmann. Amortized resource analysis with polymorphic recursion and partial big-step operational semantics. In *Programming Languages and Systems*, pages 172–187. 2010.

[22] M. Hofmann. *A Type System for Bounded Space and Functional In-Place Update–Extended Abstract.* 2000.

[23] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 185–197, 2003.

[24] M. Hofmann and D. Rodriguez. Automatic type inference for amortised heap-space analysis. In *Programming Languages and Systems*, page 593–613. Springer, 2013.

[25] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Static Analysis*, pages 238–255. Springer, 2009.

[26] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems (APLAS)*, pages 307–325. Springer-Verlag, 2008.

[27] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. 2008.

[28] P. Thiemann. Towards a type system for analyzing JavaScript programs. In *Programming Languages and Systems*, pages 408–422. Springer Berlin Heidelberg, 2005.