Access Permission Contracts

Phillip Heidegger, Annette Bieniusa, and Peter Thiemann

University of Freiburg, Germany
http://proglang.informatik.uni-freiburg.de/jscontest/
{heidegger,bieniusa,thiemann}@informatik.uni-freiburg.de

Abstract. The ideal software contract is a full specification of the behavior of an operation. Often, in particular in the context of scripting languages, a full specification can be cumbersome to state and may not even be desired. In such cases, a partial specification, which describes select aspects of the behavior, may be used to raise the confidence in an implementation of the operation to a reasonable level.
We propose a novel style of contract for object-based languages that permits the partial specification of side effects. Specifically, our contract language attaches access permissions to functions and methods. An access permission describes the side effects of a method using sets of access paths that express read and write permissions for the properties of the objects accessible from the method. We specify a monitoring semantics for access permissions and implement this semantics as an extension of an existing contract system for JavaScript. We find that adding access permissions to contracts increases the effectiveness of error detection by contract monitoring by 6-13%.

1 Introduction

Design by contract is a methodology for software development based on specifications (contracts) of operations [31, 32]. The correctness of an implementation with respect to a contract may be statically guaranteed by program verification or it may be dynamically checked with contract monitoring. As the latter variant permits more expressive specifications and puts less demands on the skills of the programmer, it is widely used in practice as evidenced by implementations of contract checking in various forms and for many languages [1,16–18,24,26,27,37].

Originally, contracts were meant to provide full specifications. However, contracts for partial specifications, which only specify certain aspects of an operation, also have their uses. For example, in a dynamically-typed language, a contract could have the form of a type signature and impose restrictions like a type system [2, 36]. Contract monitoring for such a system essentially detects type errors at method boundaries.

In previous work [24], we proposed a contract system for JavaScript which is based on type signatures. This contract system is value-oriented in the sense that a contract specifies restrictions on the values that are passed to a method and returned from it. However, a value-oriented contract misses an important

facet of the semantics of a method because a type signature does not specify its side effects. In this work, we fix this omission by extending the contract language and contract monitoring with access permissions.

An access permission explicitly states the set of paths (sequences of property accesses) that a method may access from the objects in scope. Being able to state such permissions is important in a language like JavaScript, where a side effect is the raison d'être of many operations. A value-oriented contract does not suffice in many cases. To support this claim, consider the following code:

```
function redirectTo (url) {
  window.location = url;
}
```

A suitable type-signature contract for redirect would be (string) → **undefined** stating that the argument must be a string and that the undefined result value should be returned.[1] However, the interesting information about the function is that it changes the location property of the window object, which has the further effect of redirecting the web browser to a new page. To specify this effect, our extended contract language enables us to extend the above contract with an *access permission*:

```
... with [window.location]
```

This access permission lets the function access and modify the location property of window but denies access to any other object. Contract monitoring for the thus extended contract enforces the permission at run time. For example, if the function's implementation above were replaced by

```
function redirectTo (url) {
  window.location = url;
  myhistory.push (url);
}
```

while keeping the same type signature and access permission, then monitoring would report a contract violation as soon as the function accesses myhistory.

We envision that access permissions are useful in a wide range of applications:

Security: Web browsers maintain a number of "magic properties" where an assignment causes a significant change of the browser's state (for example, window.location). Wrapping a monitored contract around a suspicious piece of code can easily reveal this kind of side effect.

Modularity: JavaScript programs typically rely on a number of libraries and freely include third-party code (mash-ups) that may change arbitrarily between different program runs. Programmers do not want this code to corrupt their global variables and to inflict arbitrary changes to their object structures. Wrapping a monitored contract around the third-party code again confines these effects and guarantees the integrity of the program's state.

[1] `undefined` is a special value in JavaScript. Methods without an explicit return statement return `undefined`.

Test-driven development: Specifying contracts with access permissions simplifies testing because it is always clear which functions are independent from one another and thus need not be tested together. Breaches are, again, detected by monitoring.

Contributions

1. Design of a contract framework with access permissions for object networks.
2. Specification of a formal semantics of access permissions and their dynamic enforcement (monitoring).
3. Implementation of access permissions as an extension of an existing contract and testing framework for JavaScript [24]. The implementation is based on program rewriting and performs contract monitoring at run time. That is, before performing a property access, the transformed program first asks for permission and signals a contract violation unless permission is granted.
4. Assessment of the effectiveness of access permission contracts by observing the impact of random code modifications on hand-annotated case studies.
5. Benchmark data on the performance of the transformed programs in different browsers.

Overview

In Section 2, we motivate access permissions with examples. Section 3 contains the formalization along with a soundness proof of monitoring and a discussion of alternative designs. Section 4 describes the implementation. Section 5 reports the two case studies that we performed and presents a performance evaluation. Finally, Section 6 discusses related work and Section 7 concludes.

2 Motivation

2.1 Modular Layout Computation

Suppose you are a JavaScript developer who has just been assigned a maintenance task on a large AJAX application. In particular, you need to work on the code that performs a layout computation for a bunch of view objects. To start with, it would be advantageous to know which properties are modified by the code. Using our framework, a developer can gradually specify access permissions for the code until it runs without contract violation on a sufficiently large number of test cases. For example, the final specification may be as follows:

```
/*c {}.(int, int) → boolean with [this.x, this.y, this.w, this.h] */
Frame.prototype.layout = function (width, height) { ... }
```

The special comment /*c ... */ specifies a contract for a method. The part before **with** defines the type signature. In the subsequent access permission, this refers to the receiver object of the method call. The access paths specify that only properties named x,y, w, or h of the receiver object may be written.

An access path starts with any variable name in scope followed by a sequence of property names. It permits reading any property reachable by dereferencing some prefix of the access path and writing the properties reachable by dereferencing the entire access path. The special variable names this, $1, $2, ... refer to the receiver object of a method call and to the first, second, and so on parameter.

2.2 Read-only Objects

Many libraries rely on a programming pattern to define JavaScript functions with keyword parameters. The idea is to define a function with one parameter which is always an object. The properties of this object play the role of keyword parameters as in this example:

```
c = createCanvas({width: 100, height: 200, background: "green"});
```

As it is generally considered bad programming style to assign to parameters, this parameter object should not be changed, either. Such changes could be forbidden with a contract:

```
/*c ({}) → undefined with [$1.*.@] */
```

This specification uses two new features in the access permission: as in shell file name patterns, the * stands for any sequence of property names. The final @ stands for the empty set of property names. Thus, the first parameter must be read-only. Read permission is granted for all properties reachable from $1, but write permission is granted only for those access paths that end in a property name that is contained in the empty set, that is, for *no* access path.

2.3 Observer

In an implementation of the observer pattern, the programmer would like to make sure that an observer only reads and writes properties below the state component of the subject. This restriction may be expressed with the contract

```
/*c ({}) → any with [$1.state.*.?] */
Observer.prototype.update = function (subject) {
  ... subject.state.value = ...
}
```

With this access permission, any property below state is readable and writable but state itself is read-only. The final ? stands for any property name.

2.4 Regular Expression Permissions

Let's return to the example from the introduction, where we wanted to ensure that a method only accesses and modifies the window.location property. In the context of enforcement of security properties, it is more likely that we want to forbid access to a few chosen properties, whereas we do not care about accesses to the majority of properties. In such a situation, we might write an access permission like the following:

variable	$x \in$	Var
property name	$p \in$	$Prop$
access path	$\pi \in$	$Path = Prop^*$
path language	$L \in$	$PLang = \wp(Path)$
expression	$e ::=$	$x \mid \lambda x.e \mid e(e) \mid \texttt{new} \mid e.p \mid e.p := e \mid \texttt{permit}\, x : L_r, L_w\, \texttt{in}\, e$

Fig. 1. Syntax.

... **with** [window./^([^s].*|.[^t].*|..[^a].*|...[^t].*|.{4}[^u].*|.{5}[^s].*|.{7,})/]

It specifies an access path that accepts read and write accesses only to properties of the window object that match the regular expression enclosed in slashes. The particular regular expression in the example matches all property names different from status.[2]

3 Designing the Framework

Before delving into the implementation of monitoring for access permissions, we formally define their meaning using a suitable, minimalist core calculus. This approach enables us to discuss potential pitfalls and alternative design choices.

Let's fix some notation before we start. Let A and B be sets. We write $\wp(A)$ for the power set of A, $A + B$ for the disjoint union of A and B, and $A \times B$ for their Cartesian product. $A \rightharpoonup B$ denotes the set of finite (partial) functions from A to B with $\emptyset$ standing for the empty mapping and if $f \in A \rightharpoonup B$, then $f \downarrow_{A'}$ denotes the restriction of f to domain $A' \subseteq A$ and $dom(f) \subseteq A$ denotes the domain of f. The updated function $f' = f[a \mapsto b]$ is defined by $f'(a) = b$ and $f'(a') = f(a')$, for all $a' \neq a$. We also write $[a \mapsto b] = \emptyset[a \mapsto b]$ for the singleton map with domain $\{a\}$. If we write $f(a)$ as part of a premise, this use implies the additional premise $a \in dom(f)$.

3.1 Syntax and Semantics

Figure 1 specifies the syntax of an imperative object calculus extended with permissions. The calculus extends a call-by-value lambda calculus with object construction (`new` creates a fresh object devoid of properties), reading of an object's property, and writing/defining an object's property. The syntax is mostly standard and close to that of existing JavaScript core languages [23, 25].

The novel construct of the calculus is the expression $\texttt{permit}\, x : L_r, L_w\, \texttt{in}\, e$ that specifies an access permission. It enforces a restriction on accesses to variable x during evaluation of e governed by the two languages L_r and L_w. Both languages specify a set of access paths (sequences of properties) starting from

[2] We plan a shorthand syntax for such cases, but we prefer to stick with the implemented features in the examples.

the object bound to x (which must be in scope). Read accesses to descendants of x are limited to paths in L_r whereas write accesses are limited to paths in L_w. Evaluation of e stops if it tries to perform any access that is not permitted.

L_r and L_w should not be arbitrary. The read language L_r should be prefix closed, because it does not make sense to permit reading of $x.a.b$ without permitting to read $x.a$, too. Similarly, writing of $x.a.b$ is not possible without reading $x.a$, first. So, each path in the write language L_w should extend a path in the read language by one property, that is, $L_w \subseteq \{\pi.p \mid \pi \in L_r, p \in \mathit{Prop}\}$.

The only hard requirement on L_r and L_w is that the membership test (in rule CHECK PERMISSION) is decidable. A practical implementation would require these languages to be regular to effectively perform the above sanity checks.

Figure 2 defines the validity of a big-step evaluation judgment of the form

$$\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow H'; u'; v$$

This judgment declares that given a variable environment ρ and indexed collections $\mathcal{R}$ and $\mathcal{W}$ of read and write permissions, the expression e transforms the initial heap H to the final heap H' and returns value v. Furthermore, it threads a serial number $u \in \mathit{Uid}$ for the heap that is incremented at each property write operation and at each permit expression. The permissions $\mathcal{R}$ and $\mathcal{W}$ are indexed by the serial number of the heap for which the permissions were granted. The indexing of the permissions uniquely identifies different executions of permit expressions and determine their relative order with respect to heap modifications.

A value $v \in \mathit{Val}$ is either an integer, a reference, or a closure consisting of an environment and a (lambda-) expression. The representation of a reference is a pair of a heap address ℓ and a collection $\mathcal{M}$ of access paths, indexed by serial numbers. The collection $\mathcal{M}$ records all permitted access paths that have been traversed during evaluation so far to obtain this reference value. The indexing plays the same role as before.

A heap maps a location to an object and an object maps a property name to a pair of a serial number and a value. The serial number indicates the time of the write operation that last assigned the property.

The evaluation rules VAR, LAM, and APP for variables, lambda abstraction, and function application are standard. They do not depend on the permissions $\mathcal{R}$ and $\mathcal{W}$, but pass them unchanged on to their sub-evaluations. The serial number is just threaded through.

The evaluation rule NEW creates a new object in the heap. This object has no properties and its collection of access paths is empty. The latter indicates that the newly created object is not accessible from any variable or already existing object through a sequence of property accesses. For that reason, the new object is completely unrestricted. Any of its properties may be written and read. The serial number does not change because no object in the heap is modified.

The rule GET defines the read operation of object properties. It relies on some auxiliary operations defined in Figure 4. After computing the location ℓ and the collection $\mathcal{M}$ of already traversed access paths of the object, the premise

$$
\begin{array}{rrll}
 & \mathit{Int} & & \text{set of integers} \\
\ell \in & \mathit{Loc} & & \text{set of locations / addresses} \\
u \in & \mathit{Uid} & = \mathit{Int} & \text{set of unique ids} \\
H \in & \mathit{Heap} & = \mathit{Loc} \rightharpoonup \mathit{Obj} & \\
 & \mathit{Obj} & = \mathit{Prop} \rightharpoonup \mathit{Uid} \times \mathit{Val} & \\
\mathcal{P}, \mathcal{R}, \mathcal{W} \in & & \mathit{Uid} \rightharpoonup \mathit{PLang} & \\
\mathcal{M}, \mathcal{N} \in & \mathit{PMap} & = \mathit{Uid} \rightharpoonup \mathit{Path} & \\
(\ell, m) \in & \mathit{Ref} & = \mathit{Loc} \times \mathit{PMap} & \\
v \in & \mathit{Val} & = \mathit{Int} + \mathit{Ref} + \mathit{Env} \times \mathit{Expr} & \\
\rho \in & \mathit{Env} & = \mathit{Var} \rightharpoonup \mathit{Val} &
\end{array}
$$

VAR
$$\rho, \mathcal{R}, \mathcal{W} \vdash H; u; x \hookrightarrow H; u; \rho(x)$$

LAM
$$\rho, \mathcal{R}, \mathcal{W} \vdash H; u; \lambda x.e \hookrightarrow H; u; (\rho \downarrow_{FV(\lambda x.e)}, \lambda x.e)$$

APP
$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0 \hookrightarrow H'; u'; (\rho', \lambda x.e) \quad \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_1 \hookrightarrow H''; u''; v_1 \quad \rho'[x \mapsto v_1], \mathcal{R}, \mathcal{W} \vdash H''; u''; e \hookrightarrow H'''; u'''; v}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_0(e_1) \hookrightarrow H'''; u'''; v}$$

NEW
$$\frac{\ell \notin \mathit{dom}(H)}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; \texttt{new} \hookrightarrow H[\ell \mapsto \emptyset]; u; (\ell, \emptyset)}$$

GET
$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow H'; u'; (\ell, \mathcal{M}) \quad \mathcal{R} \vdash_M \mathcal{M}.p}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e.p \hookrightarrow H'; u'; H'(\ell)(p) \oplus \mathcal{M}.p}$$

PUT
$$\frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1 \hookrightarrow H'; u'; (\ell, \mathcal{M}) \quad \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_2 \hookrightarrow H''; u''; v \quad \mathcal{W} \vdash_M \mathcal{M}.p}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1.p := e_2 \hookrightarrow H''[\ell \mapsto H''(\ell)[p \mapsto (u'', v)]]; u'' + 1; v}$$

PERMIT
$$\frac{\rho[x \mapsto \rho(x) \oplus [u \mapsto \varepsilon]], \mathcal{R}[u \mapsto L_r], \mathcal{W}[u \mapsto L_w] \vdash H; u + 1; e \hookrightarrow H'; u'; v}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; \texttt{permit}\, x : L_r, L_w\, \texttt{in}\, e \hookrightarrow H'; u'; v}$$

Fig. 2. Semantics.

CHECK PERMISSION
$$\frac{\forall u \in \mathrm{dom}(\mathcal{P}) \cap \mathrm{dom}(\mathcal{M}) : \mathcal{M}(u) \in \mathcal{P}(u)}{\mathcal{P} \vdash_M \mathcal{M}}$$

Fig. 3. Checking permissions.

$$(u, v) \oplus \mathcal{M} := \begin{cases} (\ell, \mathcal{M} \ominus_u \mathcal{N}) & \text{if } v = (\ell, \mathcal{N}) \\ v & \text{if } v \notin \mathit{Ref} \end{cases}$$

$$(\mathcal{M} \ominus_u \mathcal{N})(u') := \begin{cases} \mathcal{N}(u') & \text{if } u' \in \mathrm{dom}(\mathcal{N}) \\ \mathcal{M}(u') & \text{if } u' \in \mathrm{dom}(\mathcal{M}) \backslash \mathrm{dom}(\mathcal{N}) \wedge u < u' \\ \text{undefined} & \text{if } u' \in \mathrm{dom}(\mathcal{M}) \backslash \mathrm{dom}(\mathcal{N}) \wedge u \geq u' \\ \text{undefined} & \text{if } u' \notin \mathrm{dom}(\mathcal{M}) \cup \mathrm{dom}(\mathcal{N}) \end{cases}$$

$$(\mathcal{M}.p)(u) := \begin{cases} \mathcal{M}(u).p & \text{if } u \in \mathit{dom}(\mathcal{M}) \\ \text{undefined} & \text{if } u \notin \mathit{dom}(\mathcal{M}) \end{cases}$$

Fig. 4. Auxiliary definitions.

$\mathcal{R} \vdash_M \mathcal{M}.p$ checks the read permission for these paths extended with property p. This check is specified by rule CHECK PERMISSION (Fig. 3) which requires that, for each currently active index u, the access path for u is contained in the set of permitted access paths for u. As the returned value is located at the end of all access paths in $\mathcal{M}.p$, it is augmented with this information using the operator $\oplus$. It attaches the new access path to the location using the operator $\ominus_u$ if the returned value is a reference. Otherwise, the value is passed through as is.

In an application $\mathcal{M} \ominus_u \mathcal{N}$, the first argument $\mathcal{M}$ contains the newly discovered access paths, the second argument $\mathcal{N}$ contains the access paths as they are stored in the heap, and the subscript u is the serial number of the last write to the property. The definition in Figure 4 distinguishes three cases depending on when the property was last written and where the written value came from. Let u' be the serial number of an execution of a permit expression.

1. The object's property value already has an access path for index u' (in $\mathcal{N}$). In that case, the property has been overwritten since the introduction of u' and the existing access path is kept as it reflects an access path at the time when the permission u' was created.
2. The object's property value has no access path for index u' in $\mathcal{N}$ and it has been written before the permission with index u' was installed as can be seen from $u < u'$. In this case, we attach the new u'-path to the value.
3. There is no access path for index u' and the property has been written after the permission with index u' was installed (viz. $u \geq u'$). In this case, no u'-path is attached because this property was not linked to the data structure when u' was created.

The examples in Section 3.2 illustrate these three cases.

The rule PUT specifies the operation that writes and (if necessary) defines a property. It first computes the location ℓ and the collection $\mathcal{M}$ of access paths of the object and then checks the write permission to the object with the premise $\mathcal{W} \vdash_M \mathcal{M}.p$. It overwrites the object's property with the new value and assigns it a new, incremented serial number.

```
1 let x = new in
2   x.a = new;
3   x.b = new;
4   permit x :
5     {a,b,b.a},{a,b.a} in
6     x.a = x.b;
7     x.a.a = 42
```

(a) Valid access

```
1 let x = new in
2   x.a = new;
3   x.b = new;
4   x.a = x.b;
5   permit x :
6     {a,b,b.a},{a,b.a} in
7     x.a.a = 42
```

(b) Invalid access

```
1 let x = new in
2 let y = new in
3   x.a = new;
4   permit y : {a}, {a} in
5   permit x : {a}, {a} in
6     x.a = y;
7     x.a.a = 42
```

(c) Nested permissions

Fig. 5. Exercising the definition of $\circledcirc_u$.

The rule PERMIT specifies the access permission operation. Each such permission is bound to the serial number u of the heap in which the permission is installed. It increments the serial number to avoid clashes with the next permission. Then, evaluation proceeds with the body of the permit-expression, but with an updated variable binding for x, which records the serial number u for the heap reachable from the object bound to x (if any) by attaching $[u \mapsto \varepsilon]$ to it, and updated read and write permissions, which record the stated permission set L_r and L_w for the object network reachable from x.

An access permission is dynamically scoped because the access permissions are propagated with the flow of execution and the rule CHECK PERMISSION only considers the entry points in the domain of the current access permission $\mathcal{P}$. In particular, access permissions are not captured by closures created while they are in force: Closure creation (rule LAM) ignores the access permissions and function application (rule APP) continues to use the current permissions with the body of the invoked function. Hence, after evaluation of the body of an access permission is complete, its entry point u could be garbage collected both from the value and from the heap.

3.2 Examples

The code fragments in Figure 5 serve to illustrate the different cases of the $\circledcirc_u$ operator. For conciseness, we take the liberty of extending the language with a let expression and sequential execution in the usual way.

The code fragments (a) and (b) differ only in the placement of the permit expression. The code fragment (a) installs the permission *before* the assignment x.a = x.b whereas version (b) installs the permission afterwards. In both cases, let the permit expression be associated with serial number u' and let x.b contain a location ℓ_b paired with an empty map (according to rule NEW).

In version (a), the expression x.b returns the location ℓ_b paired with the map $[u' \mapsto b]$ (according to case 2 of $\circledcirc_u$: $u < u'$ because it was generated by the preceding assignment x.b = **new**). This value is written to x.a. The following access to x.a returns $(\ell_b, [u' \mapsto b])$ according to case 1 of $\circledcirc_u$ which governs that the paths stored in the object take precedence. For the final write access, the

extended access map $[u' \mapsto b.a]$ is checked against the set of write permissions and succeeds.

In version (b), x.a = x.b is executed before the permit expression. Hence, x.a contains $(\ell_b, \emptyset)$ and the GET rule makes it return $(\ell_b, [u' \mapsto a])$ according to case 1 of $\circledcirc_u$. For the write operation, the extended access map $[u' \mapsto a.a]$ is checked against the set of write permissions and fails.

The code in Figure 5(c) is supposed to exercise case 3 of the definition of $\circledcirc_u$. After establishing the two permissions, the environment ρ is: $[x \mapsto (\ell_x, [u_3 \mapsto \varepsilon]), y \mapsto (\ell_y, [u_2 \mapsto \varepsilon])]$ where the u_i are sorted according to their indexes. After the assignment x.a = y (with serial number u_4) the object in location ℓ_x is: $\{a : (u_4, (\ell_y, [u_2 \mapsto \varepsilon]))\}$. In line 7, x.a evaluates to

$$(u_4, (\ell_y, [u_2 \mapsto \varepsilon])) \oplus [u_3 \mapsto a] = (\ell_y, [u_3 \mapsto a] \circledcirc_{u_4} [u_2 \mapsto \varepsilon]) = (\ell_y, [u_2 \mapsto \varepsilon])$$

Observe that case 3 of $\circledcirc_u$ applies because $u_4 \geq u_3$. In consequence, u_3 vanishes from the domain of the map because the object that was reachable via x.a before line 6 has become garbage. With this reasoning the update of x.a.a is permitted because it is equivalent to y.a and realizable in the heap after line 5.

3.3 Properties

It is important to see that the semantics of access permissions has some subtle implications. For instance, if x is bound to some object ℓ, then the access permission $\texttt{permit}\, x : \emptyset, \emptyset\, \texttt{in}\, e$ does *not* ensure that evaluation of e does not access any property of ℓ. To see this, consider the following example:

$$\texttt{let}\, y = \texttt{new}\, \texttt{in}\, \texttt{let}\, x = y\, \texttt{in}\, \texttt{permit}\, x : \emptyset, \emptyset\, \texttt{in}\, y.a := 42 \tag{1}$$

This code evaluates successfully to 42 because the access permission only applies to property accesses that happen via x, whereas the binding of y is not aware of the permission.

So what do access permissions actually enforce? The problem with example (1) is that x and y are aliases of one another. A meaningful characterization of the guarantees of an access permission must consider aliasing. To formulate a precise statement, we extend the evaluation judgment to also trace all read and write accesses in sets $T^r, T^w \subseteq \mathit{Loc} \times \mathit{Prop}$:

$$\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow' H'; u'; v\ [T^r, T^w]$$

Figure 6 shows the modified rules for property read and write; the remaining rules just union the trace sets from the subcomputations as in the PUT' rule.

We further need to refer to all heap locations reachable from a given object location. This notion is formalized with a mapping $\mathit{reach} : \mathit{Heap} \times \mathit{Val} \rightharpoonup \wp(\mathit{Loc})$, which returns the set of locations that are reachable from an input value v by dereferencing along any path $\pi \in \mathit{Path}$, using the auxiliary function *deref* (see Figure 7). The *access* function yields tuples of locations and property names for all accessible properties along a path $\pi \in \Pi$.

$$\textsc{get'}\quad \frac{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e \hookrightarrow' H'; u'; (\ell, \mathcal{M})\ [T^r, T^w] \qquad \mathcal{R} \vdash_M \mathcal{M}.p}{\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e.p \hookrightarrow' H'; u'; (H'(\ell)(p) \oplus \mathcal{M}.p)\ [T^r \cup \{(\ell, p)\}, T^w]}$$

$$\textsc{put'}\quad \frac{\begin{array}{c}\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1 \hookrightarrow' H'; u'; (\ell, \mathcal{M})\ [T_1^r, T_1^w] \\ \rho, \mathcal{R}, \mathcal{W} \vdash H'; u'; e_2 \hookrightarrow' H''; u''; v\ [T_2^r, T_2^w] \qquad \mathcal{W} \vdash_M \mathcal{M}.p\end{array}}{\begin{array}{c}\rho, \mathcal{R}, \mathcal{W} \vdash H; u; e_1.p := e_2 \\ \hookrightarrow' H''[\ell \mapsto H''(\ell)[p \mapsto (u'', v)]]; u'' + 1; v\ [T_1^r \cup T_2^r, T_1^w \cup T_2^w \cup \{(\ell, p)\}]\end{array}}$$

Fig. 6. Tracing property read and write.

$$\begin{array}{lll}
\mathit{reach}(H, \{v_1, \ldots, v_n\}) & = \bigcup_i \mathit{reach}(H, v_i) & \\
\mathit{reach}(H, v) & = \mathit{deref}(H, v, \mathit{Path}) & \\
\mathit{deref}(H, v, \Pi) & = \bigcup\{\mathit{deref}(H, v, \pi) \mid \pi \in \Pi\} & \\
\mathit{deref}(H, (u, v), \pi) & = \mathit{deref}(H, v, \pi) & \\
\mathit{deref}(H, v, \pi) & = \begin{cases} \mathit{deref}'(H, \ell, \pi) & v = (\ell, m) \\ \emptyset & v \notin \mathit{Ref} \end{cases} & \\
\mathit{deref}'(H, \ell, \varepsilon) & = \{\ell\} & \\
\mathit{deref}'(H, \ell, p.\pi) & = \begin{cases} \mathit{deref}(H, H(\ell)(p), \pi) & p \in \mathit{dom}(H(\ell)) \\ \emptyset & \text{otherwise} \end{cases} & \\
\mathit{access}(H, v, \Pi) & = \bigcup\{\mathit{access}(H, v, \pi) \mid \pi \in \Pi\} & \\
\mathit{access}(H, (\ell, \mathcal{M}), \pi.p) & = \{(\ell', p) \mid \ell' \in \mathit{deref}'(H, \ell, \pi)\} & \\
\mathit{access}(H, v, \pi) & = \emptyset \qquad \text{if } v \notin \mathit{Ref} &
\end{array}$$

Fig. 7. Heap traversal.

Theorem 1. *Suppose that* $\rho, \mathcal{R}, \mathcal{W} \vdash H_0; \texttt{permit}\, x : L_r, L_w\, \texttt{in}\, e \hookrightarrow' H_1; v\ [T^r, T^w]$ *and that* $\mathit{reach}(H_0, \rho(FV(e) \setminus \{x\})) \cap X = \emptyset$ *where* $X = \mathit{reach}(H_0, \rho(x))$.

Then $T^r \cap (X \times \mathit{Prop}) \subseteq \mathit{access}(H_0, \rho(x), L_r)$ *and* $T^w \cap (X \times \mathit{Prop}) \subseteq \mathit{access}(H_0, \rho(x), L_w)$.

The second assumption just says that x does not share with the remaining variables. The conclusion of the theorem says that for every access pair $(\ell, p) \in T_r$ where ℓ happens to be reachable from $\rho(x)$ this access must be sanctioned by the language L_r of read permissions. The latter is formalized via the *access* function: It splits every access path in L_r in a prefix π and last property p, computes the dereferenced locations from $\rho(x)$ along path π and pairs the results (at most one) with p.

To prove this theorem, we establish an invariant, which we formulate for the judgment without the traces because they are not needed to prove it. The assumption $\rho, \mathcal{R}, \mathcal{W} \vdash H_0; u_x; \texttt{permit}\, x : L_r, L_w\, \texttt{in}\, e \hookrightarrow H_1; u_1; v$ in the theorem

can only hold (by inversion) if its premise also holds:

$$\rho[x \mapsto \rho(x) \oplus [u_x \mapsto \varepsilon]], \mathcal{R}[u_x \mapsto L_r], \mathcal{W}[u_x \mapsto L_w] \vdash H_0; u_x + 1; e \hookrightarrow H_1; u_1; v \tag{2}$$

Let's further assume that $\rho(x) = (\ell_x, m_x) \in \mathit{Ref}$—otherwise, the theorem is trivially true because $v \notin \mathit{Ref} \Rightarrow \mathit{deref}(H_0, v, \pi) = \emptyset$, for all π, so that $X = \emptyset$.

Definition 1. *A value v is* primarily reachable *(short: p.r.) from ℓ_x with index u_x in H_0 if either*

- *$v = (\ell, m)$ with $u_x \in \mathit{dom}(m)$ implies that $\ell \in \mathit{deref}'(H_0, \ell_x, m(u_x))$,*
- *$v = (\rho, \lambda y.e')$ with ρ primarily reachable, or*
- *$v \in \mathit{Int}$.*

An environment ρ is p.r. if $(\forall y \in \mathit{dom}(\rho))\ \rho(y)$ is p.r. A heap H is p.r. if $\forall \ell \in \mathit{dom}(H)$ and $\forall p \in \mathit{dom}(H(\ell))\ H(\ell)(p)$ p.r. (All with respect to the same fixed ℓ_x, u_x, and H_0.)

Lemma 1. *For each judgment $\rho', \mathcal{R}', \mathcal{W}' \vdash H'; u'; e' \hookrightarrow H''; u''; v''$ occurring in the derivation of (2) it holds that: if ρ' and H' are p.r. from ℓ_x with index u_x in H_0, then so are H'' and v''.*

Proof. By induction on the derivation. Each case refers to the variables used in the respective rule in Figure 2.

Case VAR: obviously true.

Case LAM: obviously true.

Case APP: By the assumption on ρ and H, induction on e_0 yields H' and ρ' p.r. As now ρ and H' are p.r., induction yields that H'' and v_1 p.r. As $\rho'[x \mapsto v_1]$ and H'' are p.r., induction yields H''' and v p.r., which proves the result.

Case NEW: The heap $H[\ell \mapsto \emptyset]$ and the value $(\ell, \emptyset)$ are both p.r.

Case GET: By induction, H' and $(\ell, \mathcal{M})$ are p.r. But that means, if $u_x \in \mathit{dom}(\mathcal{M})$ then $\ell \in \mathit{deref}'(H_0, \ell_x, \mathcal{M}(u_x))$. It remains to show that $H'(\ell)(p) \oplus \mathcal{M}.p$ is p.r. The only interesting case occurs if $H'(\ell)(p) = (u, (\ell', \mathcal{N}))$, in which case the returned value is $(u, (\ell', \mathcal{N})) \oplus \mathcal{M}.p = (\ell', \mathcal{M}.p \circledcirc_u \mathcal{N})$.

If $u_x \in \mathit{dom}(\mathcal{N})$, then the heap location has changed its content since the access permission associated with u_x and it has been overwritten with a value reachable in H_0 from ℓ_x on path $\mathcal{N}(u_x)$. This path annotation has to stay in force to ensure p.r. of the result: $(\mathcal{M}.p \circledcirc_u \mathcal{N})(u_x) = (\mathcal{N})(u_x)$, for which p.r. holds by the inductive assumption.

If $u_x \notin \mathit{dom}(\mathcal{N})$, then the contents of the heap location has not yet been reached from ℓ_x. There are two cases, which can be distinguished by comparing u and u_x. If $u \leq u_x$, then the heap location has not changed since H_0 and the result can be marked as visited. This is expressed by $(\mathcal{M}.p \circledcirc_u \mathcal{N})(u_x) = (\mathcal{M}.p)(u_x) = \mathcal{M}(u_x).p$. By the property read that happens in this rule, it is clear that $\ell' \in \mathit{deref}'(H_0, \ell_x, \mathcal{M}(u_x).p)$.

If, however, $u > u_x$, then the heap location has changed since H_0, but the new value has not been reachable from ℓ_x in H_0. For that reason, the value must not receive a u_x annotation. This is expressed by $(\mathcal{M}.p \circledcirc_u \mathcal{N})(u_x) =$undefined.

```
1 /*c ({x:int},{y:int}) → int with [$1.x, $2.y] */
2 function f(p,q) {
3   q.y = 24;
4   q.x = 8;
5 };
6 var o = {x: 42, y: 3};
7 f(o,o);
```

Fig. 8. Calling a function with aliased parameters.

Case PUT: by induction H' and $(\ell, \mathcal{M})$ are p.r. Hence, H'' and v are also p.r. by induction. So is the final heap as the rule overwrites a value with a p.r. value.

Case PERMIT: immediate by induction.

Towards the proof of Theorem 1, which is by induction on the evaluation judgment with traces, we observe that the top-level judgment "seeds" the lemma in a non-trivial way. The environment $\rho[x \mapsto \rho(x) \oplus [u_x \mapsto \varepsilon]]$ is p.r. with respect to u_x, ℓ_x, and H_0 (from (2)) because $\ell_x \in \mathit{deref}'(H_0, \ell_x, \varepsilon)$ and no other environment entry refers to u_x. Similarly, the heap H_0 is p.r. because does not contain any reference to u_x. Thus, the lemma tells us that H_1 and v are also p.r.

3.4 Discussion

Our calculus is more general than the actually implemented contract language. The main differences are that the implemented language only deals with access languages defined by path strings and that access permissions can only be placed on function definitions, which is equivalent to applying a permit expression to the function body.

Our semantics of access permissions is not the only possible choice. To discuss other choices, we consider the code in Figure 8. It illustrates that aliasing is the main complication as already indicated in Section 3.3.

The annotation of function f specifies that the property x of the first parameter and the property y of the second parameter may be read and modified. Our semantics guarantees that the access to q.x is rejected no matter whether p and q are aliased.

The main alternative to our intensional definition would be an extensional one, where permissions are not attached to specific values but to the underlying object network. In the example, the permission $1.x would grant read/write access to the x property of the object bound to p at the time of the call. Similarly, $2.y grants access to the y property of the object bound to q.

However, this alternative semantics would be brittle in the presence of aliasing. If p and q were not aliased, then the alternative semantics would prohibit the update q.x inside of f. If p and q were aliased as in the call f (o,o) in Figure 8, then the alternative semantics would permit the update q.x as it in fact updates

p.x, which is permitted by the access paths. We believe that this behavior is counter-intuitive for the person that specifies the access permission.

To restore some order, the alternative semantics could be refined, for example, by taking the union of the read permissions and the intersection of the write permissions (or some other combination). However, this choice would also behave in a counter-intuitive way: Using the intersection of the write permissions, already the assignment to q.y would provoke an effect violation.

Another reason to dismiss this choice is that it does not admit a compositional definition. For a counterexample, consider the access permission [x.a, x.b, y.c]. Clearly, this permission should grant write permission for x.a, x.b, and y.c, so if x and y were not aliased, then the definition would have to admit the union of the effects. However, if x and y were aliased, then the definition would have to union the effects of x.a and x.b and then take the intersection with the effects of y.c, so that no writes were permitted.

Yet another choice would be to leave the semantics of an access permission unspecified in the case that the read permissions for two different parameters overlap. Again, we dismissed this choice because it has no compositional specification and is obviously unsatisfactory.

4 Implementation

The implementation of the framework for monitoring access permissions consists of two parts. The first part is an off-line JavaScript compiler which is written in OCaml. The second part is a JavaScript library that handles the dynamic aspects of enforcing access permissions. It is available as part of JSConTest[3], a JavaScript framework for contract-based testing [24].

The implementation supports the full JavaScript language according to the standard [13]. In contrast to the formalization in Section 3.1, we do not extend the syntax of JavaScript with an additional permit expression. Instead, contracts can be specified as a special kind of comment /∗c ... ∗/ for each function and method. This approach simplifies the annotation process because it does not require any restructuring or rewriting of the user code. Also, the annotations do not change the semantics of the original code.

4.1 Transformation

The annotated user code is then compiled to a version which monitors access permissions at run time. Figure 9 shows a sample of the transformation steps that are performed by the off-line compiler. All operations that involve heap accesses, like reading and writing of properties, are redirected to JSConTest library calls that handle the access management dynamically.

As an example of the transformation of a function definition, consider the code in Figure 10. The function declaration is replaced by a variable assignment

[3] http://proglang.informatik.uni-freiburg.de/jscontest/

```
⟦ e1[e2] ⟧                 = pRead(⟦e1⟧,⟦e2⟧)
⟦ e1[e2] = e3 ⟧            = pAssign(⟦e1⟧,⟦e2⟧ ,⟦e3⟧)
⟦ f(x1,...,xn) ⟧           = fCall(f,[⟦x1⟧,...,⟦xn⟧])
⟦ o.m(x1,...,xn) ⟧         = mCall(o,m,[⟦x1⟧,..., ⟦xn⟧])
⟦ new O(x1,...,xn) ⟧       = cCall(O,[⟦x1⟧,..., ⟦xn⟧])
⟦ for (var i in o) { e } ⟧ = for (var i in o) { if (mCall(o,"hP",[i])) { ⟦ e ⟧} }
⟦ function f() { s } ⟧     = var f = (function () {
                               function f'() { ⟦ s ⟧ }
                               return JsConTest.effects.enableWrapper(f');
                             }());
```

Fig. 9. Some select rewrite rules for code under test.

```
/** (any) → any                          TESTS.c1 = ... // contract
    with [ $1.a.f ] */                   var f = (function () {
function f(x) {                            function f'(x) {
  var z = x.a;                        ⇝      var z = JSConTest.effects.pRead(x,"a");
  return z.f;                              return JSConTest.effects.pRead(z,"f");
};                                         }
                                           return JSConTest.effects.enableWrapper(f');
                                         }());
```

Fig. 10. Example of a transformation.

which uses an anonymous function that is invoked directly as the right hand side. This standard JavaScript pattern creates a new private scope and avoids name space pollution. Because function declarations are evaluated when loading the source code, the compiler also has to reorder the source elements to ensure semantically equivalent initialization. The wrapper of the function f marks the parameters with the corresponding access path information at run time such that during the execution pRead can check if it has permission to read $1.a. The library call to pRead also creates a wrapper with the access path $1.a for z. Reading the property f of z uses the access path stored in the wrapper of z, extends it to $1.a.f, and checks if reading this path is permitted. The permission is granted because the effect of f is $1.a.f. No access violation is reported. The library function enableWrapper generates a fresh unique identifier each time a function is called (cf. serial numbers in the formalization and Section 4.2).

Similar to the formal system, each reference stores additional information about the access path that has been used to reach the corresponding object. To this end, each object is replaced by a wrapper that contains a reference to the actual object and a map with access path information.

Calls to native or non-transformed code would fail if wrapped objects were passed. Because it is not possible to decide statically which function is applied at a call side, the framework strips parameter objects of the access meta data before passing them to the function. However, it conserves the meta data by storing the wrappers on a global stack that is used to re-wrap the objects if the

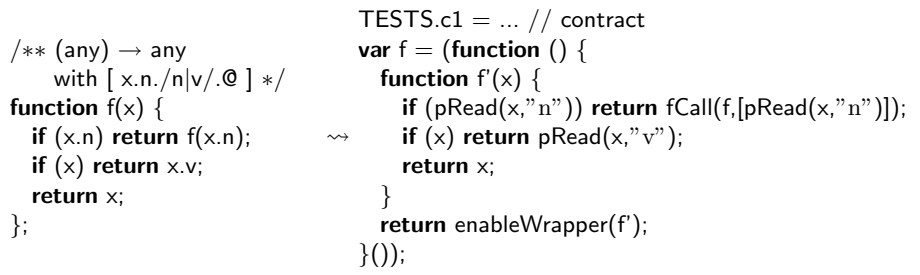

Fig. 11. Example of a transformation with recursive function call.

callee itself is a transformed function. This approach is compatible with uses of eval, although monitoring does not extend to eval-generated code.

For inter-operability with non-transformed code, it is also necessary to remove wrappers when storing object properties. To this end, an additional map (__infos__) is attached to each object. This map stores the wrappers for each of the properties. The function pRead uses this map to lookup the wrappers that belong to an object and return the wrapped object.

As the library stores the access path information in the __infos__ property of the objects, this property must not become accessible to user code. Therefore, we provide a substitute for hasOwnProperty (hoP) that masks out the __infos__ property. We also transform the statement **for** (**var** i **in** o) { e } to ensure that internal properties used by the implementation do not leak out to the program. Technically, this protection is achieved by changing the body e to **if** (hP(o,i) { e }. The functions pRead and pAssign also safeguard the special property __infos__.

If native code or non-transformed code iterates over all properties of an object, then it is not possible to detect the access and prohibit it by hiding the __infos__ property. We are not aware of any solution short of modifying the underlying JavaScript engine. However, in the case studies that we performed the special properties caused no problem.

4.2 Function calls

This example section considers a code fragment with a recursive function (Figure 11). It demonstrates the interplay of wrapping, unwrapping, and access permissions. Given a linked list, the function returns the value (v) of the last node after recursively following the references to the next pointers (n). For expository purposes, the function's implementation does not abide by its contract.

Suppose that the code under test contains the following call to f:

```
var x = { n: { n: { n: undefined, v: 24 }, v: 11}, v: 5 };
f(x);
```

To enable the framework to locate the permissions that need to be respected by read and write operations in the function body, the access permission for f

is stored under uid 0 in a global effect store when the function is called. At this point, the effect store contains { 0: x.n./n|v/.@ }.

Then, the object wrappers for the parameters are created. For parameter x, the wrapper is given by

{ ref: { n: { n: { n: **undefined**, v: 24 }, v: 11}, v: 5 }, pmap: { 0: x } }

The pmap property stores the information that in the (outermost) function call which is given the uid 0, the access path for the object is x.

Now the execution of the actual function body commences with the expression pRead(x,"n"). Comparing the actual access path x.n with the permission x.n./n|v/.@ yields no violation, so the property access is granted, yielding a reference to {n: { n: **undefined**, v: 24}, v: 11} which is returned in a wrapper as

{ ref: { n: { n: **undefined**, v: 24 }, v: 11}, pmap: { 0: x.n } }

Performing the recursive call with this object as parameter creates a fresh uid 1, and extends the wrapper for the parameter to

{ ref: { n: { n: **undefined**, v: 24 }, v: 11}, pmap: { 0: x.n, 1: x } }

Additionally, the global effect store is extended to { 0,1: x.n./n|v/.@ }.

When reading the property n again, the read access is admitted, and the wrappers are again extended similar to the previous case when calling the recursive function again. The parameter x is now wrapped as

{ ref: {n: **undefined**, v: 24}, pmap: { 0: x.n.n, 1: x.n, 2: x }}

Finally, when f tries to read n this time, the access to the property is not granted because the access path for uid 0 only grants access to {x.n./n|v/.@ }, but the read operation tries to dereference {x.n.n.n }. This leads to an access violation.

5 Evaluation

To evaluate the feasibility of monitoring for access permissions we measure its effectiveness and performance on several libraries and applications. To this end, we hand-annotated the code with function and method contracts and ran it with monitoring enabled. We also applied random code modifications [9] to check to what extent the enforcement of access permissions detects changes in the program's behavior.

5.1 Case Study: Linked Lists

The first case study concerns a small third-party library (200 LOC) which implements a singly-linked list data structure.[4] Its interface comprises one constructor for list nodes and six methods to operate on the list: add, remove, find, indexOf, size, and toString.

For each method we developed contracts with access permissions. For example, the toString method was annotated with the contract:

[4] `https://github.com/nzakas/computer-science-in-javascript`

	int, no effects		int, with effects		top, with effects	
fulfilled contracts	1011	18.0 %	711	12.7 %	1055	18.4 %
rejected contracts	4607	82.0 %	4907	87.3 %	4721	81.6 %
reasons for reject (one mutant may be counted multiple times)						
contract failure	2020	43.9 %	1643	33.5 %	1096	19.0 %
signaled error	2034	44.1 %	2136	43.5 %	2243	38.8 %
browser timeout	553	12.0 %	243	5.0 %	369	6.4 %
read violation	-	0.0 %	1018	20.7 %	1004	17.4 %
write violation	-	0.0 %	1606	32.7 %	1593	27.6 %
read/write violation	-	0.0 %	1842	37.5 %	1823	31.6 %

Table 1. Testing random mutations of the singly-linked list case study.

```
/*c js:ll.() → string with [this.head.*.@] */
List.prototype.stringOf = function () { ... }
```

This contract permits reading of all properties that are reachable from head. js:ll is a custom contract developed for this case study. It defines an "instance-of" test and a random generator for linked lists (16 LOC). Annotating the code and implementing the custom contract took about one hour. The contracts for the six functions and the code defining the custom contract is presented in the appendix A.

From the implementation we derived about 5600 random mutations and tested each mutant against the original contracts. The mutation operations were renaming of variables and properties, replacing an integer constant by another, swapping **null** and **undefined**, removing a return statement, and exchanging a binary operator (e.g., + with −, || with &&). Each of the six functions was tested with 1000 randomly generated test cases.

We ran the code under test in several configurations:

- contracts specifying integer lists without effects: only violations of the type contracts are detected (int, no effects),
- contracts specifying integer lists with effects: type and access path violations are detected (int, with effect), and
- contracts for arbitrary lists with effects: list elements are unrestricted and access path violations are detected (top, with effects).

Table 1 shows the results of the test cases. The rows of the table have the following meaning:

- fulfilled: mutation not detected because the mutant fulfills all six contracts.
- rejected: the mutant fails at least one of the six tests.

We can read off the effectiveness of effect monitoring from these two rows. Adding access permissions to integer contracts improves the detection rate for mutations from 82% to 87.3%, an improvement of 6.4%. Adding access permissions to very liberal type contracts (top with effects) has about the same detection rate as (int, no effects).

The remaining rows break down the reasons for the failure of a mutant.

- contract failure: the test revealed an unexpected return value;
- signaled error: the test was stopped because an error (e.g. dereferencing of undefined, calling a method on null) was signaled;
- browser timeout: the test server timed out due to non-terminating code;[5]
- read violations: the code violates the read access permissions;
- write violations: the code violates the write access permissions;
- read/write violations: the code violates a read or write access permission.

Why is it possible that a mutant counts in more than one row? To see this, let's consider a mutant with a test outcome of s,s,r,w,c,r. This outcome states that the first and second function passed 1000 tests successfully, the third and sixth function were stopped because of a read access violation (r), the forth was stopped due to a write violation (w), and the fifth was stopped because the function returned a value that violated the return contract of the function (c).

A mutant only counts as "fulfilled" if all six functions pass all tests (they all report (s)). Otherwise, the mutant counts as "rejected". Because a rejection may have more than one reason, the example outcome s,s,r,w,c,r counts as a contract failure, a read violation, and a write violation. For that reason, the counts in these rows are not disjoint so that the percentages do not add to 100%.

Of special interest are the cases where the contract system did not detect the mutation of the code as these cases indicate the effectiveness of the annotations. A manual inspection of these mutants revealed that in many cases the mutated code is semantically equivalent to the original version, e.g. `x.p` was changed to `x.q`, where both properties `p` and `q` were always undefined. In other cases, the contract is fulfilled by a mutant because the modification did not change any property access or return value from a type perspective, e.g. **return** `true` is changed to **return** `false`. While these mutants may violate the intended semantics, we cannot expect more because we started from a partial specification, not from the full specification of a linked-list data structure.

We also manually inspected ten randomly selected mutants that timed out. All of these mutants timed out because of an infinite loop. Hence, we have reason to believe that our choice of timeout is sensible.

5.2 Case Study: Richards Benchmark

A second case study was performed on the Richards benchmark code, which is part of the Google V8 benchmark suite.[6] It simulates the task dispatcher of an operating system. The code implements 29 functions in 650 LOC. A person without prior knowledge of the code under test provided the contracts and implemented custom generators in about four hours. It took about another two hours to develop the access permissions.

[5] Non-termination is signaled after a timeout of 15 seconds for one function. On average, the run time for a test case is less than 10ms.

[6] `http://v8.googlecode.com/svn/data/benchmarks/v6/richards.js`

	no effects		with effects	
fulfilled contracts	1148	38.9%	911	30.8%
rejected contracts	1807	61.1%	2044	69.2%
reason for rejection (one mutant may be counted multiple times)				
failed contracts	872	48.3%	866	42.4%
signaled error	1052	58.2%	1037	50.7%
browser timeout	28	1.5%	30	1.5%
read violation	0	0.0%	202	9.9%
write violation	0	0.0%	149	7.4%
read/write violation	0	0.0%	349	17.1%

Table 2. Testing random mutations of the Richards case study.

Table 2 shows the result of testing about 2950 mutated versions. We execute for each mutant 50 Tests per function to test effect and contract violations. We chose to run a smaller number of tests to reduce the overall run time and to check if a large number of test cases is required to obtain a high detection rate of mutants.

For this application, adding access permissions increased the detecting rate from 61.1% to 69.2%, which amounts to a 13% improvement. This increase is quite surprising as the percentages of detected read or write violations are much smaller than in the linked-list case study.

5.3 Performance Evaluation

All case studies and benchmarks were executed on a Lenovo Thinkpad X61s notebook with a Core 2 Duo processor with 1.60 GHz and 2GB Ram running the Google-Chrome browser (7.0.517.44) on top of Linux version 2.6.35-23-generic. In this setting, a test run of a mutant is about four times slower with monitoring enabled than without monitoring. This slowdown is consistent for both case studies.

To give ballpark numbers, running 1000 tests for each of the six functions of the linked-list test suite takes about 6 seconds with monitoring compared to 1.5 seconds without. For Richards, running 50 tests for each of the 29 functions takes 1.85 seconds with monitoring compared to 0.5 seconds without.

For the Richards benchmark we also timed the original code (without mutation) once with monitoring and once without to measure the slowdown for code that never violates the effect annotations. This experiment masks out the effects of contract violations, which cause the program to stop earlier on faulty mutants than on correct code. However, the slowdown is similar: running 1000 test cases for each of the 29 functions took 32.4 seconds with monitoring enabled versus 7.4 seconds without, a slowdown factor of 4.4.

6 Related Work

Effect systems are closely related to access permissions. Effect systems have been conceived for functional languages [20] to describe and infer the scope of side effects, with the goal of detecting parallelizable code fragments and improving memory management.

There are too many papers on effect systems to do them all justice here. However, Greenhouse and Boyland [22] introduce effect annotations for Java which closely resemble our contracts. In contrast to our system, effects are collected for regions which comprise a set of objects. Their approach aims for tracking data dependencies of software components. The main differences to our work are that most effect systems are integrated in type systems and thus geared towards static analysis (whereas ours is geared towards dynamic analysis) and that our prime motivation lies in the detection of software defects.

Similarly related is work on ownership and aliasing control. Again, with the exception of the dynamic ownership system of Boyland and coworkers [8], most ownership systems statically impose tree-like ownership structures on object graphs [3, 12, 34, 38]. The main difference to ownership types is that our system is entirely access path-based whereas ownership types are context-based. Furthermore, some ownership systems forbid the mere existence of references, whereas access permissions forbid the traversal of certain paths.

Bierhoff and Aldrich [7] define a static checker for access permission in Java. It combines typestate and object aliasing information to design and verify protocols for safe object access. They also focus on the correct usage of single resources. Their access permissions are statically verified.

Deutsch's [10] analysis for sharing and aliasing is also entirely based on access paths. It is a static analysis phrased as an abstract interpretation of a storeless semantics.

Run-time monitoring is a well-known approach to providing safety and security guarantees. Erlingsson [15] provides an overview of such applications. As a notable difference, security monitoring is mostly geared towards eliminating (sequences of) uses of undesired operations and can often be implemented by finite automata, whereas access path monitoring rules out undesired accesses and requires a more specific implementation techniques (e.g., for dealing with aliasing).

Similar monitoring ideas using program transformation have been explored to provide safety and security in scripting languages, in particular for JavaScript.

BrowserShield [35] provides run-time monitoring of JavaScript. BrowserShield also rewrites code by redirecting critical operations according to user-specified policies. Likewise, the Google Caja project [21] employs an online compilation process of JavaScript code to a safe subset named Cajita.

Maffeis and co-workers [30] combine several isolation techniques for restricting heap accesses of third-party code. They disallow `eval`, `Function`, and `constructor` within untrusted code and also rewrite property accesses with wrappers to enable run-time checks.

These systems operate within the browser during interactive user sessions and provide complete interposition. In contrast, our tool is focused on development and testing of applications.

Finifter and co-workers [19] design a JavaScript heap analysis framework to detect information leaks. To prevent exploits, third-party code is restricted to a name space by prefixing properties with a unique identifier.

ConScript [33] allows fine-grained application specific security policies that are enforced at run time by a modified JavaScript execution engine. Compared to our approach, they have different goals and less overhead, but are tied to a particular, obsolete browser implementation.

Program specification frameworks like Spec# [5], JML [29], or Eiffel [14] permit the formulation of access permissions as FOL-formulas in Hoare-style pre- and postconditions. Because specialized syntax is missing, the annotation process is rather heavy-weight. Besides, these frameworks are geared towards full specifications, whereas we are only interested in partial specifications.

The situation gets even more complicated in a setting with higher-order functions as indicated by Berger and co-workers [6]. They present an extension of a call-by-value imperative higher-order functions calculus with aliasing. This allows for static reasoning about assertion in the form of Hoare-style triples. In the very same way, Banerjee et al. [4] derive a logic for reasoning about mutation and separation of the heap locally. With the emphasis on the theory of aliasing, these systems are orthogonal to the work presented here.

7 Conclusion

We proposed a novel extension of software contracts with access permissions that specify the side effects of an operation in terms of access paths. We implemented monitoring for access permissions by a program transformation and demonstrated that this implementation has a high, but acceptable overhead (slowdown by a factor of four). As a basis of the implementation, we developed a formalization that enabled us to cleanly specify the interaction of monitoring and aliasing.

Two case studies showed that the specification of contracts with access permissions takes about 30-40 minutes per 100 LOC (for a person not familiar with the source code) and that in return the number of bugs detected by contract monitoring increases between 6% and 13%. We find this improvement remarkable because the baseline, monitoring of type signatures, already captures a large amount of bugs (61% and 82%). Hence, access permissions seem to catch a significant number of extra errors and could be a worthwhile extension of testing frameworks.

We believe that our approach is more widely applicable. Other scripting languages have features similar to JavaScript, so the framework should be easily adaptable to them. More generally, path-based access permissions would make sense for any object-based language.

In future work, we want to progress in various directions. An obvious extension would be a special treatment for effects on the DOM [28]. Because DOM structures are guaranteed to be trees (no aliasing!), many of the complications of general object graphs do not arise in the case of DOM.

We further wish to automatically generate random test data from access permissions to perform tests that exercise the side effects of the operations. As similar notions have been proposed in work on security for mashups, it would be interesting to explore applications to security, too.

There are further options in the design space of permissions like generalizing the path specifications to arbitrary regular expressions on the path level that are worth investigating.

Finally, a long term goal would be the integration of such a monitoring facility into a JavaScript engine to improve the efficiency of testing. For now, we opted for the transformational approach because it is easier to implement, it is more portable and it is more durable under changes to the JavaScript engine.

References

1. P. Abercrombie and M. Karaorman. jContractor: Design by contract for Java. `http://jcontractor.sourceforge.net/`, 2003.
2. A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Proc. 38th ACM symp. popl. In *Proc. 38th ACM Symp. POPL*, Austin, USA, Jan. 2011. ACM Press.
3. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In M. Odersky, editor, *18th ECOOP*, volume 3086 of *LNCS*, pages 1–25, Oslo, Norway, June 2004. Springer.
4. A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In J. Vitek, editor, *22nd ECOOP*, volume 5142 of *LNCS*, pages 387–411, Paphos, Cyprus, 2008. Springer.
5. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *CASSIS 2004: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, pages 49–69. Springer, 2004.
6. M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing in imperative higher-order functions. In B. C. Pierce, editor, *Proc. ICFP 2005*, pages 280–293, Tallinn, Estonia, Sept. 2005. ACM Press, New York.
7. K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *Proc. 22nd ACM Conf. OOPSLA*, pages 301–320, Montreal, QC, CA, 2007. ACM Press, New York.
8. J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP '01: Proc. 15th European Conference on Object-Oriented Programming*, pages 2–27, London, UK, 2001. Springer-Verlag.
9. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11:34–41, April 1978.
10. A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proc. IEEE International Conference on Computer Languages 1992*, pages 2–13, Oakland, CA, Apr. 1992. IEEE.
11. T. D'Hondt, editor. *24th ECOOP*, volume 6183 of *LNCS*, Maribor, Slovenia, 2010. SP.

12. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, Oct. 2005.
13. ECMAScript Language Specification, Dec. 2009. ECMA International, ECMA-262, 5th edition.
14. Eiffel: Analysis, design and programming language, June 2006. ECMA International, ECMA-367, 2nd edition.
15. Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, Caledon Hills, Ontario, Canada, Sept. 1999.
16. R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proc. 16th ACM Conf. OOPSLA*, pages 1–15, Tampa Bay, FL, USA, 2001. ACM Press, New York.
17. R. B. Findler and M. Felleisen. Contracts for higher-order functions. In S. Peyton-Jones, editor, *Proc. ICFP 2002*, pages 48–59, Pittsburgh, PA, USA, Oct. 2002. ACM Press, New York.
18. R. B. Findler, S. Guo, and A. Rogers. Lazy contract checking for immutable data structures. In O. Chitil, Z. Horváth, and V. Zsók, editors, *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007*, number 5083 in Lecture Notes in Computer Science, pages 111–128. Springer, 2008.
19. M. Finifter, J. Weinberger, and A. Barth. Preventing Capability Leaks in Secure JavaScript Subsets. In *Proceedings of Network and Distributed System Security Symposium*, pages 375–388. Internet Society, 2010.
20. D. Gifford and J. Lucassen. Integrating functional and imperative programming. In *Proc. 1986 ACM Conf. on Lisp and Functional Programming*, pages 28–38, 1986.
21. Google-caja: A source-to-source translator for securing javascript-based web. `http://code.google.com/p/google-caja/`.
22. A. Greenhouse and J. Boyland. An object-oriented effects system. In R. Guerraoui, editor, *13th ECOOP*, volume 1628 of *LNCS*, pages 205–229, Lisbon, Portugal, June 1999. Springer.
23. A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In D'Hondt [11].
24. P. Heidegger and P. Thiemann. Contract-driven testing of JavaScript code. In *TOOLS*, Malaga, Spain, June 2010. Springer.
25. P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In D'Hondt [11].
26. R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In P. Wadler and M. Hagiya, editors, *Proceedings of the 8th International Symposium on Functional and Logic Programming FLOPS 2006*, pages 208–225, Fuji Susono, Japan, Apr. 2006. Springer.
27. R. Kramer. iContract — the Java design by contract tool. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 295–307, Santa Barbara, CA, USA, 1998.
28. P. Le Hégaret, R. Whitmer, and L. Wood. W3C document object model. `http://www.w3.org/DOM/`, Aug. 2003.
29. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A Notation for Detailed Design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.
30. S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *ESORICS'09: Proceedings of the 14th European Conference on Research in Computer Security*, pages 505–522, Berlin, Heidelberg, 2009. Springer-Verlag.

31. B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, Oct. 1992.
32. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.
33. L. A. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *IEEE Symposium on Security and Privacy*, May 2010.
34. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP*, volume 1445 of *LNCS*, pages 158–185, Brussels, Belgium, July 1998. Springer.
35. C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web*, 1(3):11, 2007.
36. P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proc. 18th ESOP*, volume 5502 of *LNCS*, pages 1–16, York, UK, Mar. 2009. Springer-Verlag.
37. D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In B. Pierce, editor, *Proc. 36th ACM Symp. POPL*, pages 41–52, Savannah, GA, USA, Jan. 2009. ACM Press.
38. T. Zhao, J. Palsberg, and J. Vitek. Lightweight confinement for Featherweight Java. In *Proc. 18th ACM Conf. OOPSLA*, pages 135–148, Anaheim, CA, USA, 2003. ACM Press, New York.

A Linked-List

A.1 LinkedList with Contracts (exact contracts)

```
1  function LinkedList() { ... }
2  /*c js:ll.(int) → undefined with [this._head.*, this._length] */
3  function add(data) { ... }
4  /*c js:ll.(int) → (int or null) with [this._length.@, this._head.*.@] */
5  function item(index) { ... }
6  /*c js:ll.(int) → (int or null) with [this._head.*, this._length] */
7  function remove(index) { ... }
8  /*c js:ll.() → int with [this._length.@] */
9  function size() { ... }
10 /*c js:ll.() → [int] with [this._head.*.@ ] */
11 function toArray() { ... }
12 /*c js:ll.() → string with [this._head.*.@] */
13 function toString() { ... }
14 LinkedList.prototype = { constructor: LinkedList, add: add, ... };
```

A.2 LinkedList with Contracts with simple contracts

```
1  /*c js:ll.(top) → undefined with [this._head.*, this._length] */
2  function add(data) { ... }
3  /*c js:ll.(top) → top with [this._length.@, this._head.*.@] */
4  function item(index) { ... }
5  /*c js:ll.(top) → top with [this._head.*, this._length] */
6  function remove(index) { ... }
7  /*c js:ll.() → [top] with [this._head.*.@ ] */
8  function toArray() { ... }
```