# Efficient Dynamic Access Analysis Using JavaScript Proxies

Technical Report *

Matthias Keil  Peter Thiemann
Institute for Computer Science
University of Freiburg
Freiburg, Germany
{keilr,thiemann}@informatik.uni-freiburg.de

## Abstract

JSConTest introduced the notions of effect monitoring and dynamic effect inference for JavaScript. It enables the description of effects with path specifications resembling regular expressions. It is implemented by an offline source code transformation.

To overcome the limitations of the JSConTest implementation, we redesigned and reimplemented effect monitoring by taking advantange of JavaScript proxies. Our new design avoids all drawbacks of the prior implementation. It guarantees full interposition; it is not restricted to a subset of JavaScript; it is self-maintaining; and its scalability to large programs is significantly better than with JSConTest.

The improved scalability has two sources. First, the reimplementation is significantly faster than the original, transformation-based implementation. Second, the reimplementation relies on the fly-weight pattern and on trace reduction to conserve memory. Only the combination of these techniques enables monitoring and inference for large programs.

***Categories and Subject Descriptors*** D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features—Classes and objects; D.3.3 [*SOFTWARE ENGINEERING*]: Software/Program Verification—Programming by contract,Validation; D.4.6 [*OPERATING SYSTEMS*]: Security and Protection—Access controls

***General Terms*** Design, Languages, Security, Verification

***Keywords*** Access Permission Contracts, JavaScript, Proxies

## 1. Introduction

JSConTest [22] introduced the notions of effect monitoring and dynamic effect inference [23] for JavaScript. It enables the programmer to specify the effect of a function using access permission contracts. These contracts consist of an anchor specifying a start object and a regular expression specifying the admissible access paths that a contract-annotated function may use. Matching paths can be assigned read or write permission.

The inference component of JSConTest may help a software maintainer who wants to investigate the effect of an unfamiliar function by monitoring its execution and then summarizing the observed access traces to access permission contracts.

JSConTest is implemented by an offline source code transformation. This approach enabled a quick development, but it comes with a number of drawbacks. First, it requires a lot of effort to construct an offline transformation that guarantees full interposition and that covers the full JavaScript language: the implemented transformation has known omissions (e.g., no support for `with` and prototypes) and it does not apply to code created at run time using `eval` or other mechanisms. Second, the transformation is subject to bitrotting because it becomes obsolete as the language evolves. Third, the implementation represents access paths with strings and checks them against the specification using the built-in regular expression matching facilities of JavaScript. This approach quickly fills up memory with many large strings and processes the matching of regular expressions in a monolithic way.

In this work, we present JSConTest2, a redesign and reimplementation of JSConTest using JavaScript proxies [14, 43], a JavaScript extension which is scheduled for the upcoming ECMAScript 6 standard. This new implementation addresses all shortcomings of the previous version. First, the proxy-based implementation guarantees full interposition for the full language and for all code regardless of its origin, including dynamically loaded code and code injected via `eval`. Second, maintenance is alleviated because there is no transformation that needs to be adapted to changes in the language syntax. Also, future extensions are catered for as long as the proxy API is supported. By adapting ideas from code contracts [16], we also avoided a custom syntax extension. Third, our new implementation represents access paths in a space efficient way. It also incrementalizes the path matching by encoding its state in an automaton state, which is represented by a regular expression. It applies the fly-weight pattern to reduce memory consumption of the states. Last but not least, the new implementation is significantly faster than the previous one.

JSConTest2 employs Brzozowski's derivatives of regular expressions [10] to perform the path matching incrementally and efficiently. It applies a rewriting system inspired by Antimirov's techniques [2] for deciding subset constraints for regular expressions to simplify regular expressions if more than one contract is applied to an object at the same time.

To evaluate the scalability of JSConTest2, we applied path monitoring to a number of example programs including web page dumps. The main problem we had to deal with was excessive

* This report is a slightly edited versions of the paper appeared in the *Proceedings of the 9th symposium on Dynamic languages*. To avoid confusions we revised the notation of access permission contracts. Further, we split the theorem of *Syntactic derivative of contracts*.

*DLS '13*, October 28, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM 978-1-4503-2433-5/13/10. . . $15.00.
http://dx.doi.org/10.1145/2508168.2508176

memory use. We explain several techniques for reducing memory consumption, including the reduction of regular expression based effect contracts using an adaptation of Antimirov's techniques.

***Contributions***

- Reimplementation of JSConTest using JavaScript proxies
- Formalization of violation logging and contract enforcement
- Reduced memory use by simplification of regular expressions
- Practical evaluation with case studies

***Overview*** Section 2 gives some examples and rationales for JSConTest2. Section 3 gives a high-level overview of the approach taken in this paper. Section 4 recalls proxies and membranes from related work. Section 5 defines the syntax of access paths and access contracts. Section 6 formalizes a core language and defines the semantics for path logging and contract enforcement. Section 7 explains the techniques used to reduce memory consumption. Sections 8 and 9 describe the implementation and some experiences in applying JSConTest2. Section 10 discusses related work. It is followed by a conclusion.

Appendix A and B shows the formal semantics for violations and merged proxies. Section C states some auxiliary functions used. The proofs of semantic containment, syntactic, derivative, syntactic containment, and correctness are shown in the appendix D, E, F, and G.

## 2. Effects for JavaScript

JavaScript is the language of the Web. More than 90% of all Web pages provide functionality using JavaScript. Most of them rely on third-party libraries for calenders, social networks, or feature extensions. Some of these libraries are statically included with the main script, others are loaded dynamically.

Software development and maintenance is tricky in JavaScript because dynamically loaded libraries have arbitrary access to the application state. Some libraries override global objects to add features, others manipulate data stored in the browser's DOM or in cookies, yet others may send data to the net. In addition, there are security concerns if the application has to guarantee confidentiality or integrity of data. As all scripts run with the same authority, the main script has no handle on the use of data by an included script.

As all resources in a JavaScript program are accessible via property read and write operations, controlling those operations is sufficient to control the resources. Thus, effect monitoring and inference have a role to play in the context of test-driven development, in maintenance to analyze a piece of software, or in security to prevent the software from compromising confidentiality or integrity.

JSConTest2 monitors read and write operations on objects through access permission contracts that specify allowed effects as outlined in the introduction. A contract restricts effects by defining a set of permitted access paths starting from some anchor object.

### 2.1 Contracts and the Contract API

This section introduces the contract syntax and the JSConTest2 API. In a first example, a developer may want to ascertain that only some parts of an object are accessed.

```
var protected =
  __APC.permit('(a.?+b*)', {a:{a:3,b:5},b:{a:7,b:11}});
```

Here, _APC is the object that encapsulates the JSConTest2 implementation. Its **permit** method takes a contract and an object as parameters and returns a "contracted" object where only access paths that are explicitly permitted by the contract are admitted. The contract consists of two alternative parts connected by **+**. The first part, **'a.?'**, gives read/write access to all properties of the object in the a property, but a itself is read-only. The second part, **'b*'**, allows read and write access to an arbitrarily long chain of properties named b.

Here is an example with some uses of the contracted object.

```
1 var x = __APC.permit('a.b', {a:{b:3}, b:{b:5}});
2 y = x.a;
3 y.b = 3;
```

The access permission contract **'a.b'** specifies the singleton set $\{a.b\}$ of permitted access paths. The contract allows us to read and write property a.b and to read the prefix a. Properties which are not addressed by a contract are neither readable nor writeable. The read and write operations in lines 2 and 3 abide by the contract, but reading from x.b or writing to x.a would not be permitted and would cause a violation.

Only the last property of a path in the set of permitted access paths is writeable and all prefixes are readable. The special property @ stands for a "blank" property that matches no other property. Using it at the end of a contract specifies a read-only path as shown in the following example.

```
var x = __APC.permit('a.b.@', {a:{b:3}, b:{b:5}});
x.a.b = 3; // violation
```

One could imagine contracts for defining write-only paths, for instance, in a security context. This case is not covered by our implementation, but it would be straightforward to provide an interface that separates read and write permissions.

The next example demonstrates how contracts interact with assignments.

```
var x = __APC.permit('((a+a.b)+b.b.@)', {a:{b:3}, b:{b:5}});
x.a = x.b;
x.a.b=7; // violation
```

The contract **'((a+a.b)+b.b.@)'** allows read access to x.b and x.b.b as well as read and write access to x.a and x.a.b. Reading x.b yields a contracted object {b:5} with contract **'b.@'**, where b is read-only. This object is assigned to x.a so that x.a and x.b are now aliases. The strategy of JSConTest2 is to obey the contracts along all access paths. Reading x.a again yields an object with contract **'(e+b)&b.@'**, where **e** stands for the empty word and **&** is the conjunction operator. Thus, the resulting contract **'(e+b)&b.@'** simplifies to **'b.@'** such that writing to x.a.b causes a violation.

In addition to using full property names in contracts, the syntax admits regular expressions for property names, too. For example, the contract **'(/^get.+/+next)*.length.@'** allows us to read the length property after reading a chain of properties that either start with get or that are equal to next.

### 2.2 A Security Example

As an example from a security context, consider the following scenario, which was used as an exploit to extract the contacts out of a GMail account.[1]

```
<script type="text/javascript"
  src="http://docs.google.com/data/contacts?out=js&
  show=ALL&psort=Affinity&callback=google&max=99999">
</script>
```

This script element is a JSONP request that loads the Google Mail contacts and sends it to the google function, which is given as callback. The following listing shows what the data given to the callback function could look like.

```
var contacts = {
  Success: true,
  Errors: [],
  Body: {
    AuthToken: {
```

[1] This exploit has been fixed in 2006.

```
      Value: '********'
    },
    Contacts: [
    {
      Name: 'Jimmy Example',
      Email: 'email@example.org',
      Addresses: [],
      Phones: [],
      Ims: []
    },
    // More contacts
    ]
  }
};
```

To restrict access to the contacts object, the developer could wrap it into a contract as follows.

```
return __APC.permit(
  '((Success.@+Errors.?*)+Body.Contacts.?.Name)',
  contacts);
```

This contract enables read access to Success (**'Success.@'**), read-/write access to everything below Errors (**'Errors.?*'**). Furthermore, only the Name propertiy can be accessed on each element of the contacts array Body.Contacts. Access to the properties AuthToken as well as to the actual contact data (e.g., Email, Phones, Ims) is not permitted, thus substantially diminishing the value of an exploit.

In this case, an access permission contract should restrict the `google` function from using its argument arbitrarily. To be effective, a HTTP proxy would have to insert the contract in the HTTP request resulting from the script tag.

```
var google = __APC.permitArgs('arguments.0.
  ((Success.@+Errors.?*)+Body.Contacts.?.Name)',
  function(contacts) {
    // do something
});
```

The **permitArgs** method takes an access permission contract and a function and returns a wrapped function, such that each call to the wrapped function enforces the contract. Arguments are addressed by position so that **'arguments.0'** addresses the first argument. The remaining contract specification is as before.

Because of the transparent implementation of contract enforcement, the function that is wrapped is arbitrary: it may be defined in the same source, it may be loaded dynamically, or it may be the result of `eval`. Contract enforcement works in all cases.

## 3. The JSConTest2 Approach

JSConTest2 implements a contracted target object by wrapping it in a proxy object that intercepts all operations on the target and either forwards them to the target or signals a contract violation. The monitoring requires storing a set of access paths and the contract along with the proxy. When reading a property of a contracted object that contains another object, then the read operation must return a contracted object that carries the remaining contract after the read operation (cf. the examples in Section 2.1). This "contract inheritance" is an instance of the membrane pattern that is often used in connection with proxies. Section 4 gives an introduction to proxies and membranes.

As contracts are closely related to regular expressions, the remaining contract after a read operation can be nicely characterized using Brzozowski-derivatives of regular expressions. Section 5 formally defines contracts and their semantics in terms of access paths, it defines the derivative operation on contracts, establishes its basic properties, and finishes by defining readable and writeable paths. This section form the basis for Section 6, which formalizes the semantics of the **permit** operations.

Section 7 addresses some practical problems that arise from the implementation. Under certain circumstances, the same object may be subject to multiple contracts. A naive implementation would create an inefficient chain of proxy objects, which can be avoided by merging the path set and contract information. However, these merge operations themselves lead to memory bloat, which can be addressed by using suitable data structures and aggressive contract simplification.

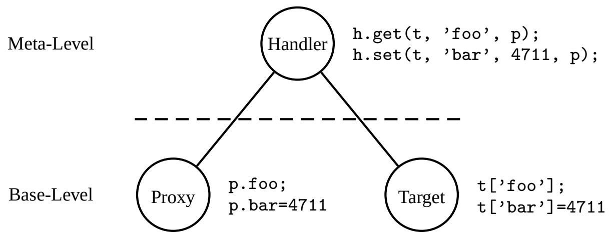

**Figure 1.** Example of proxy operation.

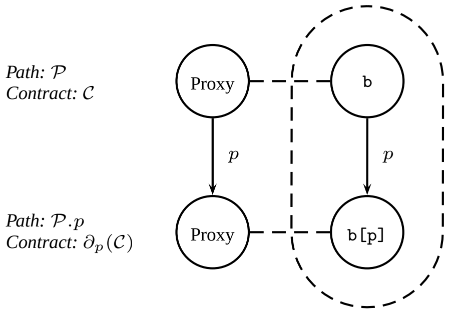

**Figure 2.** Example of property access through membrane.

## 4. Proxies and Membranes

### 4.1 Proxies

A JavaScript proxy [14] is an object whose behavior is controlled by a handler object. A typical use case is to have the handler mediate access to an arbitrary target object, which may be a native or proxy object. The proxy is then intended to be used in place of the target and is not distinguishable from other objects. However, the proxy may modify the original behavior of the target object in many respects.

The handler object defines trap functions that implement the operations on the proxy. Operations like property lookup or property update are forwarded to the corresponding trap. The handler may implement the operation arbitrarily; in the simplest case, it forwards the operation to the target object. The handler may also be a proxy.

Figure 1 contains a simple example, where the handler `h` causes the proxy `p` to behave as a wrapper for a target object `t`. Performing the property access `p.foo` on the proxy object results in a meta-level call to the corresponding trap on the handler object `h`. Here, the handler forwards the property access to the target object. The property write is handled similarly.

### 4.2 Membranes

Our technique to implement objects under a contract is inspired by *Revocable Membranes* [14, 37, 43]. A membrane serves as a regulated communication channel between an object and the rest of the program. It ensures that all parts of the objects behind a membrane also remain behind. For example, each property access on a wrapped object (e.g. `obj.p`) returns another wrapped object. Therefore, after wrapping, no new direct references to target objects behind the membrane become available. One use of this mechanism

$$
\begin{array}{llllll}
\textit{Literal} & \ni \ell & ::= & @ & \textit{(empty literal)} \\
 & & \mid & ? & \textit{(universe)} \\
 & & \mid & r & \textit{(regular expression)} \\
 & & \mid & !r & \textit{(negation)} \\
 & & & & \\
\textit{Contract} & \ni \mathcal{C} & ::= & \emptyset & \textit{(empty set)} \\
 & & \mid & \mathcal{E} & \textit{(empty contract)} \\
 & & \mid & \ell & \textit{(literal)} \\
 & & \mid & \mathcal{C}* & \textit{(Kleene star)} \\
 & & \mid & \mathcal{C}+\mathcal{C} & \textit{(logical or)} \\
 & & \mid & \mathcal{C}\&\mathcal{C} & \textit{(logical and)} \\
 & & \mid & \mathcal{C}.\mathcal{C} & \textit{(concatenation)}
\end{array}
$$

**Figure 3.** Syntax of access permission contracts.

is to revoke all references to an object network or to enforce write protection [14, 37, 43].

In our use of membranes (cf. Figure 2), each handler contains a path $\mathcal{P}$, and a contract $\mathcal{C}$ describing the allowed field accesses. Each property access `obj`$.p$ on a wrapped object returns a wrapped object whose path is $\mathcal{P}.p$. In addition, the handler traps enforce the contract $\mathcal{C}$. If the access on property $p$ is allowed by contract $\mathcal{C}$ the handler forwards the request to the target object and wraps the returned object with the new contract $\partial_p(\mathcal{C})$, which is the derivative of $\mathcal{C}$ with respect to $p$ (explained in Section 5.3). If this access is not allowed, then the handler prevents it in a configurable way.

The Figure 2 shows a membrane arising from an allowed property access. The information on the left is contained in the handler objects and the objects inside the membrane on the right are the target objects of the proxies. Thus, our implementation logs all access paths to wrapped objects in their handlers.

## 5. Access Permission Contracts

This section defines the syntax and semantics of access permission contracts and access paths.

### 5.1 Access Paths

Let $\mathcal{A}$ be a set of property names and $\iota \notin \mathcal{A}$ be a special blank property that does not occur in any JavaScript object. Its sole purpose is to indicate read-only accesses. Let $p \in \mathcal{A} \cup \{\iota\}$ range over all properties. An access path $\mathcal{P} \in (\mathcal{A} \cup \{\iota\})^*$ is a sequence of properties. We write $\epsilon$ for the empty path and $\mathcal{P}.\mathcal{P}$ for the concatenation of two paths (considered as sequences).

### 5.2 Contracts

Figure 3 defines the syntax of contracts. Contract literals $\ell$ are the primitive building blocks of contracts. Each literal defines a property access. A literal $\ell$ is either the empty literal @, the universe literal ?, a regular expression $r$, or a negated regular expression $!r$. The empty literal @ stands for the blank property $\iota$. It should not be confused with the empty set contract $\emptyset$. The universe literal ? represents the set of all JavaScript property names. A regular expression $r$ describes a set of matching property names. We assume that these expressions are JavaScript regular expressions, which we treat as abstract in this work.

Contracts are regular expressions extended with intersection. A contract $\mathcal{C}$ is either an empty set $\emptyset$, an empty contract $\mathcal{E}$, a single literal $\ell$, a Kleene star $\mathcal{C}*$, a disjunction $\mathcal{C}+\mathcal{C}$, a conjunction $\mathcal{C}\&\mathcal{C}$, or a concatenation $\mathcal{C}.\mathcal{C}$. Beware that a literal may contain a regular expression at the character level.

Each contract defines a set of access paths as defined in Figure 4. This definition follows the usual semantics of regular expressions with a few specialities. The empty literal yields the empty property.

$$
\begin{array}{lcl}
\mathcal{L}[\![@]\!] & = & \{\iota\} \\
\mathcal{L}[\![?]\!] & = & \mathcal{A} \\
\mathcal{L}[\![r]\!] & = & \{p \mid r \succ p\} \\
\mathcal{L}[\![!r]\!] & = & \mathcal{A} \backslash \mathcal{L}[\![r]\!] \\
\mathcal{L}[\![\emptyset]\!] & = & \{\} \\
\mathcal{L}[\![\mathcal{E}]\!] & = & \{\epsilon\} \\
\mathcal{L}[\![\mathcal{C}*]\!] & = & \{\epsilon\} \cup \mathcal{L}[\![\mathcal{C}.\mathcal{C}*]\!] \\
\mathcal{L}[\![\mathcal{C}+\mathcal{C}']\!] & = & \mathcal{L}[\![\mathcal{C}]\!] \cup \mathcal{L}[\![\mathcal{C}']\!] \\
\mathcal{L}[\![\mathcal{C}\&\mathcal{C}']\!] & = & \mathcal{L}[\![\mathcal{C}]\!] \cap \mathcal{L}[\![\mathcal{C}']\!] \\
\mathcal{L}[\![\mathcal{C}.\mathcal{C}']\!] & = & \{\mathcal{P}.\mathcal{P}' \mid \mathcal{P} \in \mathcal{L}[\![\mathcal{C}]\!], \mathcal{P}' \in \mathcal{L}[\![\mathcal{C}']\!]\}
\end{array}
$$

**Figure 4.** Language of contracts.

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
 & & & \emptyset+\mathcal{C} & \rightsquigarrow & \mathcal{C} \\
\mathcal{E}* & \rightsquigarrow & \mathcal{E} & @+\mathcal{C} & \rightsquigarrow & \mathcal{C} \\
 & & & \mathcal{C}+\mathcal{C} & \rightsquigarrow & \mathcal{C} \\
\mathcal{E}.\mathcal{C} & \rightsquigarrow & \mathcal{C} & & & \\
@.\mathcal{C} & \rightsquigarrow & @ & \emptyset\&\mathcal{C} & \rightsquigarrow & \emptyset \\
\emptyset.\mathcal{C} & \rightsquigarrow & \emptyset & @\&\mathcal{C} & \rightsquigarrow & @ \\
 & & & \mathcal{C}\&\mathcal{C} & \rightsquigarrow & \mathcal{C}
\end{array}
$$

**Figure 5.** Normalization rules for contracts.

$\mathcal{A}$ is the set of all property names. $r \succ p$ is a predicate that indicates whether property $p$ matches regular expression $r$ (as a standard regular expression on characters).

We say that the contract literal $\ell$ matches property $p$, written as $\ell \succcurlyeq p$, iff $p \in \mathcal{L}[\![\ell]\!]$. We further say that a contract $\mathcal{C}$ matches path $\mathcal{P}$, written $\mathcal{C} \succcurlyeq \mathcal{P}$, iff $\mathcal{P} \in \mathcal{L}[\![\mathcal{C}]\!]$.

The last property $p$ of an access path $\mathcal{P}.p$ is readable and writeable. All properties along the prefix $\mathcal{P}$ are readable. A contract ending with the empty literal $\mathcal{C}.@$ is a read-only contract. It matches access paths of the form $\mathcal{P}.\iota$ that end with the blank property $\iota$, which never occurs in a program.

Figure 5 contains normalization rules for contracts. We say that a contract $\mathcal{C}$ is *normalized* iff it cannot be further reduced by these rules. From now on, we regards all contracts as normalized.

### 5.3 Derivatives of Contracts

In this section we introduce the notion of a derivative for a contract, which is defined analogously to the derivative of a regular expression [10, 39]. Derivatives are best explained in terms of a language quotient, which is the set of suffixes of words in the language after taking away a prescribed prefix.

**Definition 1** (Left quotient)**.** *Let $L \subseteq \mathcal{A}^*$ be a language. The* left quotient *$\mathcal{P}^{-1}L$ of the language $L$ with respect to an access path $\mathcal{P}$ is defined as:*

$$\mathcal{P}^{-1}L \;=\; \{\mathcal{P}' \mid \mathcal{P}.\mathcal{P}' \in L\} \tag{1}$$

Clearly, it holds that $\{\mathcal{P}.\mathcal{P}' \mid \mathcal{P}' \in \mathcal{P}^{-1}L\} \subseteq L$. It is also immediate from the definition that $(p.\mathcal{P})^{-1}L = \mathcal{P}^{-1}(p^{-1}L)$.

To compute the derivative of a contract $\mathcal{C}$ w.r.t. an access path $\mathcal{P}$ we have to introduce an auxiliary function $\nu$ to determine if a contract $\mathcal{C}$ matches the empty path $\epsilon$. Figure 6 contains its definition.

**Definition 2** (Nullable)**.** *A contract $\mathcal{C}$ is nullable iff its language $\mathcal{L}[\![\mathcal{C}]\!]$ contains the empty access path $\mathcal{E}$.*

**Lemma 1** (Nullable)**.**

$$\mathcal{E} \in \mathcal{L}[\![\mathcal{C}]\!] \;\Leftrightarrow\; \nu(\mathcal{C}) = \top \tag{2}$$

If we access a target object by reading property $p$ on an object with contract $\mathcal{C}$, then the access language for the target ob-

$$\begin{array}{lcllcl}
\nu(@) & = & \bot & \nu(\mathcal{E}) & = & \top \\
\nu(?) & = & \bot & \nu(\mathcal{C}*) & = & \top \\
\nu(r) & = & \bot & \nu(\mathcal{C}+\mathcal{C}') & = & \nu(\mathcal{C}) \vee \nu(\mathcal{C}') \\
\nu(!r) & = & \bot & \nu(\mathcal{C}\&\mathcal{C}') & = & \nu(\mathcal{C}) \wedge \nu(\mathcal{C}') \\
\nu(\emptyset) & = & \bot & \nu(\mathcal{C}.\mathcal{C}') & = & \nu(\mathcal{C}) \wedge \nu(\mathcal{C}')
\end{array}$$

**Figure 6.** The predicate "is nullable".

$$\begin{array}{lcl}
\partial_p(@) & = & \emptyset \\
\partial_p(?) & = & \mathcal{E} \\
\partial_p(r) & = & \begin{cases} \mathcal{E}, & r \succ p \\ \emptyset, & \text{otherwise} \end{cases} \\
\partial_p(!r) & = & \begin{cases} \emptyset, & r \succ p \\ \mathcal{E}, & \text{otherwise} \end{cases} \\
\partial_p(\emptyset) & = & \emptyset \\
\partial_p(\mathcal{E}) & = & \emptyset \\
\partial_p(\mathcal{C}*) & = & \partial_p(\mathcal{C}).\mathcal{C}* \\
\partial_p(\mathcal{C}+\mathcal{C}') & = & \partial_p(\mathcal{C})+\partial_p(\mathcal{C}') \\
\partial_p(\mathcal{C}\&\mathcal{C}') & = & \partial_p(\mathcal{C})\&\partial_p(\mathcal{C}') \\
\partial_p(\mathcal{C}.\mathcal{C}') & = & \begin{cases} \partial_p(\mathcal{C}).\mathcal{C}'+\partial_p(\mathcal{C}'), & \nu(\mathcal{C}) \\ \partial_p(\mathcal{C}).\mathcal{C}', & \text{otherwise} \end{cases}
\end{array}$$

**Figure 7.** Derivative of a contract by a property.

ject is $p^{-1}\mathcal{L}[\![\mathcal{C}]\!]$. As for regular expressions, we can compute a derivative contract $\partial_p(\mathcal{C})$ of $\mathcal{C}$ with respect to $p$ symbolically, such that $p^{-1}\mathcal{L}[\![\mathcal{C}]\!] = \mathcal{L}[\![\partial_p(\mathcal{C})]\!]$. Figure 7 contains the definition of the derivative for a single property. We extend this definition to access paths by

$$\begin{array}{lcl}
\partial_{\mathcal{E}}(\mathcal{C}) & = & \mathcal{C} \\
\partial_{p.\mathcal{P}}(\mathcal{C}) & = & \partial_{\mathcal{P}}(\partial_p(\mathcal{C}))
\end{array}$$

**Lemma 2** (Derivatives of Contracts)**.** *For all paths $\mathcal{P}$ it holds that:*

1. $\mathcal{L}[\![\partial_{\mathcal{P}}(\mathcal{C})]\!] = \mathcal{P}^{-1}\mathcal{L}[\![\mathcal{C}]\!]$
2. $\mathcal{L}[\![\mathcal{P}.\partial_{\mathcal{P}}(\mathcal{C})]\!] \subseteq \mathcal{L}[\![\mathcal{C}]\!]$
3. $\mathcal{P} \in \mathcal{L}[\![\mathcal{C}]\!] \Leftrightarrow \nu(\partial_{\mathcal{P}}(\mathcal{C}))$

### 5.4 Matching

By Lemma 2, $\mathcal{C}' = \partial_{\mathcal{P}}(\mathcal{C})$ defines the language containing the remaining paths after reading $\mathcal{P}$. If path $\mathcal{P}$ is not a prefix of a path in $\mathcal{L}[\![\mathcal{C}]\!]$, then $\mathcal{C}'$ must be the empty set $\emptyset$. If $\mathcal{P}$ is an element of $\mathcal{L}[\![\mathcal{C}]\!]$, then the language of $\mathcal{C}'$ contains the empty path $\mathcal{E}$. By definition, each path and each prefix of a path is readable. Thus, readability and writeability can be determined by checking whether the remaining language is the empty set.

**Definition 3** (Readable)**.** *An access path $\mathcal{P}$ is readable with respect to contract $\mathcal{C}$ iff the derivative of contract $\mathcal{C}$ with respect to path $\mathcal{P}$ results in contract $\mathcal{C}'$ with $\mathcal{L}[\![\mathcal{C}']\!] \neq \emptyset$. That is:*

$$\mathcal{C} \vdash_{\mathcal{R}} \mathcal{P} \Leftrightarrow \mathcal{L}[\![\partial_{\mathcal{P}}(\mathcal{C})]\!] \neq \emptyset \quad (3)$$

Every path $\mathcal{P}$ in $\mathcal{L}[\![\mathcal{C}]\!]$ is writeable. By Lemma 2, we know that a path $\mathcal{P}$ is an element of the language defined by $\mathcal{C}$ iff $\epsilon \in \mathcal{L}[\![\partial_{\mathcal{P}}(\mathcal{C})]\!]$.

**Definition 4** (Writeable)**.** *An access path $\mathcal{P}$ is writeable with respect to contract $\mathcal{C}$ iff the derivative of $\mathcal{C}$ with respect to path $\mathcal{P}$ is nullable. That is:*

$$\mathcal{C} \vdash_{\mathcal{W}} \mathcal{P} \Leftrightarrow \nu(\partial_{\mathcal{P}}(\mathcal{C})) \quad (4)$$

$$\begin{array}{llcl}
\textit{Expression} & \ni e & ::= & c \mid x \mid \lambda x.e \mid e(e) \mid \textbf{new}\, e \\
 & & \mid & e[e] \mid e[e] = e \mid \textbf{permit}\, \mathcal{C}\, \textbf{in}\, e \\
\\
\textit{Location} & \ni \xi & & \\
\textit{Value} & \ni v & ::= & c \mid \xi \\
\\
\textit{Monitor} & \ni \mathcal{M} & ::= & \emptyset \mid \mathcal{M} \lhd_{\mathcal{R}} \mathcal{P} \mid \mathcal{M} \lhd_{\mathcal{W}} \mathcal{P} \\
 & & \mid & \mathcal{M}; \mathcal{M}' \\
\\
\textit{Access Handler} & \ni H & ::= & \langle \mathcal{P}, \mathcal{C} \rangle \\
\textit{Proxy} & \ni P & ::= & \langle \xi, H \rangle \\
\\
\textit{Prototype} & \ni \pi & ::= & v \\
\textit{Closure} & \ni f & ::= & \emptyset \mid \langle \rho, \lambda x.e \rangle \\
\textit{Object} & \ni o & ::= & \emptyset \mid o[str \mapsto v] \\
\textit{Storable} & \ni s & ::= & \langle o, f, \pi \rangle \mid P \\
\textit{Environment} & \ni \rho & ::= & \emptyset \mid \rho[x \mapsto v] \\
\textit{Heap} & \ni \mathcal{H} & ::= & \emptyset \mid \mathcal{H}[\xi \mapsto s]
\end{array}$$

**Figure 8.** Syntax and semantic domains of $\lambda_J$.

## 6. Formalization

This section presents the formal semantics of path monitoring and contract enforcement in terms of a JavaScript core calculus $\lambda_J$ extended with access permission contracts and access paths.

### 6.1 Syntax

$\lambda_J$ (Figure 8) is a call-by-value lambda calculus extended with objects and object-proxies. The syntax is close to JavaScript core calculi from the literature [20, 27].

A $\lambda_J$ expression is either a constant $c$ including **undefined** and **null**, a variable $x$, a lambda expression, an application, an object creation, a property reference, a property assignment, or a permit expression. The novel **permit** expression applies the given contract $\mathcal{C}$ to the object arising from expression $e$.

### 6.2 Semantic Domains

Figure 8 defines the semantic domains of $\lambda_J$.

The heap maps a location $\xi$ to a storable $s$, which is either a proxy object $P$ or a triple consisting of an object $o$, a function closure $f$, and a value $\pi$ as prototype. A Proxy is a wrapper for a location $\xi$ augmented with an access handler $H$, which is a tuple consisting of a path $\mathcal{P}$ and a contract $\mathcal{C}$. An object $o$ maps a string to a value. A function closure consists of an expression $e$ and an environment $\rho$, which maps a variable to a value $v$.

Further, a monitor $\mathcal{M}$ is a collection used for effect monitoring. It records all paths that have been accessed during the evaluation. The notation $\mathcal{M} \lhd_{\mathcal{R}} \mathcal{P}$ adds path $\mathcal{P}$ as read effect to the monitor. Synonymously, $\mathcal{M} \lhd_{\mathcal{W}} \mathcal{P}$ adds a write effect. $\mathcal{M}; \mathcal{M}'$ denotes the union of two collections.

Figure 9 introduces some abbreviated notations. A property lookup or a property update on a storable $s = \langle o, f, \pi \rangle$ is relayed to the underlying object. The property access $s(str)$ returns **undefined** by default if the accessed string is not defined in $o$ and the prototype of $s$ is not a location $\xi$. The notation $\mathcal{H}[\xi, str \mapsto v]$ updates a property of storable $\mathcal{H}(\xi)$, $\mathcal{H}[\xi \mapsto \pi]$ initializes an object, and $\mathcal{H}[\xi \mapsto f]$ defines a function. Further, we write $\langle \xi, \mathcal{P}, \mathcal{C} \rangle$ for a protected location.

### 6.3 Evaluation of $\lambda_J$

Program execution is modeled by a big-step evaluation judgment of the form $\mathcal{H}, \rho \vdash e \Downarrow \mathcal{H}' \mid v \mid \mathcal{M}$. The evaluation of expression $e$ with initial heap $\mathcal{H}$ and environment $\rho$ results in final heap $\mathcal{H}'$,

$$
\begin{array}{lcl}
\langle o, f, \pi\rangle(\mathit{str}) & = & \begin{cases} v, & o = o'[\mathit{str} \to v] \\ o'(\mathit{str}), & o = o'[\mathit{str}' \to v] \\ \mathcal{H}(\xi)(\mathit{str}), & o = \emptyset \ \wedge\ \pi = \xi \\ \textbf{undefined}, & o = \emptyset \ \wedge\ \pi = c \end{cases} \\
\langle o, f, \pi\rangle[\mathit{str} \mapsto v] & = & \langle o[\mathit{str} \mapsto v], f, \pi\rangle \\
\mathcal{H}[\xi, \mathit{str} \mapsto v] & = & \mathcal{H}[\xi \mapsto \mathcal{H}(\xi)[\mathit{str} \mapsto v]] \\
\mathcal{H}[\xi \mapsto \pi] & = & \mathcal{H}[\xi \mapsto \langle \emptyset, \emptyset, \pi\rangle] \\
\mathcal{H}[\xi \mapsto f] & = & \mathcal{H}[\xi \mapsto \langle \emptyset, f, \textbf{null}\rangle] \\
\langle \xi, \langle \mathcal{P}, \mathcal{C}\rangle\rangle & = & \langle \xi, \mathcal{P}, \mathcal{C}\rangle
\end{array}
$$

**Figure 9.** Abbreviations.

(CONST)
$$\frac{}{\mathcal{H}, \rho \vdash c \Downarrow \mathcal{H} \mid c \mid \emptyset}$$

(VAR)
$$\frac{}{\mathcal{H}, \rho \vdash x \Downarrow \mathcal{H} \mid \rho(x) \mid \emptyset}$$

(ABS)
$$\frac{\xi \notin \mathit{dom}(\mathcal{H})}{\mathcal{H}, \rho \vdash \lambda x.e \Downarrow \mathcal{H}[\xi \mapsto \langle \rho, \lambda x.e\rangle] \mid \xi \mid \emptyset}$$

(APP)
$$\frac{\begin{array}{c} \mathcal{H}, \rho \vdash e_0 \Downarrow \mathcal{H}' \mid \xi \mid \mathcal{M} \\ \mathcal{H}', \rho \vdash e_1 \Downarrow \mathcal{H}'' \mid v_1 \mid \mathcal{M}' \\ \mathcal{H}'', \xi \vdash_{\mathsf{App}} v_1 \Downarrow \mathcal{H}''' \mid v \mid \mathcal{M}'' \end{array}}{\mathcal{H}, \rho \vdash e_0(e_1) \Downarrow \mathcal{H}''' \mid v \mid \mathcal{M}; \mathcal{M}'; \mathcal{M}''}$$

(NEW)
$$\frac{\mathcal{H}, \rho \vdash e \Downarrow \mathcal{H}' \mid v \mid \mathcal{M} \qquad \xi \notin \mathit{dom}(\mathcal{H}')}{\mathcal{H}, \rho \vdash \textbf{new}\, e \Downarrow \mathcal{H}'[\xi \mapsto v] \mid \xi \mid \mathcal{M}}$$

(GET)
$$\frac{\begin{array}{c} \mathcal{H}, \rho \vdash e_0 \Downarrow \mathcal{H}' \mid \xi \mid \mathcal{M} \\ \mathcal{H}', \rho \vdash e_1 \Downarrow \mathcal{H}'' \mid \mathit{str} \mid \mathcal{M}' \\ \mathcal{H}'', \xi \vdash_{\mathsf{Get}} \mathit{str} \Downarrow \ \mid \mathcal{H}''' \mid v \mid \mathcal{M}'' \end{array}}{\mathcal{H}, \rho \vdash e_0[e_1] \Downarrow \mathcal{H}''' \mid v \mid \mathcal{M}; \mathcal{M}'; \mathcal{M}''}$$

(PUT)
$$\frac{\begin{array}{c} \mathcal{H}, \rho \vdash e_0 \Downarrow \mathcal{H}' \mid \xi \mid \mathcal{M} \\ \mathcal{H}', \rho \vdash e_1 \Downarrow \mathcal{H}'' \mid \mathit{str} \mid \mathcal{M}' \\ \mathcal{H}'', \rho \vdash e_2 \Downarrow \mathcal{H}''' \mid v \mid \mathcal{M}'' \\ \mathcal{H}''', \xi \vdash_{\mathsf{Put}} \mathit{str}, v \Downarrow \mathcal{H}'''' \mid v' \mid \mathcal{M}''' \end{array}}{\mathcal{H}, \rho \vdash e_0[e_1] = e_2 \Downarrow \mathcal{H}'''' \mid v' \mid \mathcal{M}; \mathcal{M}'; \mathcal{M}''; \mathcal{M}'''}$$

(PERMIT)
$$\frac{\begin{array}{c} \mathcal{H}, \rho \vdash e \Downarrow \mathcal{H}' \mid \xi \mid \mathcal{M} \\ \xi' \notin \mathit{dom}(\mathcal{H}') \qquad \mathcal{H}'' = \mathcal{H}'[\xi' \mapsto \langle \xi, \epsilon, \mathcal{C}\rangle] \end{array}}{\mathcal{H}, \rho \vdash \textbf{permit}\, \mathcal{C}\ \textbf{in}\, e \Downarrow \mathcal{H}'' \mid \xi' \mid \mathcal{M}}$$

**Figure 10.** Inference rules for $\lambda_J$.

value $v$, and monitor $\mathcal{M}$. The Figures 10, 11, 12, and 13 contain its inference rules.

The rules (CONST), (VAR), (ABS) are standard. (NEW) allocates a new object based on the evaluated prototype. The rule (PERMIT) creates a new proxy object for the location resulting from the subexpression. The handler of this proxy contains an empty access path $\epsilon$ and the initial contract $\mathcal{C}$.

Function application, property lookup and property assignment distinguish two cases: either the operation applies directly to a target object or it applies to a proxy. If the given reference is a non-proxy object, then the usual rules apply: (APP-NOPROXY) for calling the closure and rules (GET-NOPROXY) and (PUT-NOPROXY) for property read and write. Otherwise, if the given reference is a proxy, then the proxy rules formalize the operation implemented by the trap.

In case of a function proxy, the contract is applied to the argument to protect the function's input. If the argument is a constant (APP-PROXY), then the constant need not be wrapped and the function application is forwarded to the target object. Otherwise, if the argument is a location (APP-MEMBRANE), then the argument is wrapped in a new proxy with the function's contract and then

(APP-NOPROXY)
$$\frac{\begin{array}{c} \langle o, \langle \dot\rho, \lambda x.e\rangle, \pi\rangle = \mathcal{H}(\xi) \\ \mathcal{H}, \dot\rho[x \mapsto v] \vdash e \Downarrow \mathcal{H}' \mid v' \mid \mathcal{M} \end{array}}{\mathcal{H}, \xi \vdash_{\mathsf{App}} v \Downarrow \mathcal{H}' \mid v' \mid \mathcal{M}}$$

(APP-PROXY)
$$\frac{\begin{array}{c} \langle \xi', \mathcal{P}, \mathcal{C}\rangle = \mathcal{H}(\xi) \\ \mathcal{H}, \mathcal{H}(\xi') \vdash_{\mathsf{App}} c \Downarrow \mathcal{H}' \mid v' \mid \mathcal{M} \end{array}}{\mathcal{H}, \xi \vdash_{\mathsf{App}} c \Downarrow \mathcal{H}' \mid v' \mid \mathcal{M}}$$

(APP-MEMBRANE)
$$\frac{\begin{array}{c} \langle \xi'', \mathcal{P}, \mathcal{C}\rangle = \mathcal{H}(\xi) \qquad \xi''' \notin \mathit{dom}(\mathcal{H}') \\ \mathcal{H}[\xi''' \mapsto \langle \xi', \epsilon, \mathcal{C}\rangle], \mathcal{H}(\xi'') \vdash_{\mathsf{App}} \xi''' \Downarrow \mathcal{H}' \mid v' \mid \mathcal{M} \end{array}}{\mathcal{H}, \xi \vdash_{\mathsf{App}} \xi' \Downarrow \mathcal{H}' \mid v' \mid \mathcal{M}}$$

**Figure 11.** Inference rules for function application.

(GET-NOPROXY)
$$\frac{\langle o, f, \pi\rangle = \mathcal{H}(\xi)}{\mathcal{H}, \xi \vdash_{\mathsf{Get}} \mathit{str} \Downarrow \mathcal{H} \mid o(\mathit{str}) \mid \emptyset}$$

(GET-PROXY)
$$\frac{\begin{array}{c} \langle \xi', \mathcal{P}, \mathcal{C}\rangle = \mathcal{H}(\xi) \qquad \mathcal{C} \vdash_{\mathcal{R}} \mathit{str} \\ \mathcal{H}, \xi' \vdash_{\mathsf{Get}} \mathit{str} \Downarrow \mathcal{H}' \mid c \mid \mathcal{M} \end{array}}{\mathcal{H}, \xi \vdash_{\mathsf{Get}} \mathit{str} \Downarrow \mathcal{H}' \mid c \mid \mathcal{M} \lhd_{\mathcal{R}} \mathcal{P}.\mathit{str}}$$

(GET-MEMBRANE)
$$\frac{\begin{array}{c} \langle \xi', \mathcal{P}, \mathcal{C}\rangle = \mathcal{H}(\xi) \qquad \mathcal{C} \vdash_{\mathcal{R}} \mathit{str} \\ \mathcal{H}, \xi' \vdash_{\mathsf{Get}} \mathit{str} \Downarrow \mathcal{H}' \mid \xi'' \mid \mathcal{M} \\ P = \langle \xi'', \mathcal{P}.\mathit{str}, \partial_{\mathit{str}}(\mathcal{C})\rangle \qquad \xi''' \notin \mathit{dom}(\mathcal{H}') \end{array}}{\mathcal{H}, \xi \vdash_{\mathsf{Get}} \mathit{str} \Downarrow \mathcal{H}'[\xi''' \mapsto P] \mid \xi''' \mid \mathcal{M} \lhd_{\mathcal{R}} \mathcal{P}.\mathit{str}}$$

**Figure 12.** Inference rules for property reference.

(PUT-NOPROXY)
$$\frac{\langle o, f, \pi\rangle = \mathcal{H}(\xi)}{\mathcal{H}, \xi \vdash_{\mathsf{Put}} \mathit{str}, v \Downarrow \mathcal{H}[\xi, \mathit{str} \mapsto v] \mid v \mid \emptyset}$$

(PUT-PROXY)
$$\frac{\begin{array}{c} \langle \xi', \mathcal{P}, \mathcal{C}\rangle = \mathcal{H}(\xi) \qquad \mathcal{C} \vdash_{\mathcal{W}} \mathit{str} \\ \mathcal{H}, \xi' \vdash_{\mathsf{Put}} \mathit{str}, v \Downarrow \mathcal{H}' \mid v' \mid \mathcal{M} \end{array}}{\mathcal{H}, \xi \vdash_{\mathsf{Put}} \mathit{str}, v \Downarrow \mathcal{H}' \mid v' \mid \mathcal{M} \lhd_{\mathcal{W}} \mathcal{P}.\mathit{str}}$$

**Figure 13.** Inference rules for property assignment.

passed to the target function. This wrapping may happen multiple times.

To perform a property read on a proxy, the handler first checks whether the access is allowed by the contract. If so and in case the accessed value is a constant, then the value gets returned in rule (GET-PROXY). Otherwise, if the value is a location, the location gets wrapped by the derivative of the original contract with respect to the accessed property $\partial_{str}(\mathcal{C})$ and an extended path $\mathcal{P}.str$ (GET-MEMBRANE). The accessed object is now wrapped with the derivative that describes the remaining permitted paths. Thus, a property access to a wrapped object only returns constants or wrapped objects so that the membrane remains intact. In both cases the monitor registers the accessed path.

In a similar way, the property assignment checks if the property is writeable with respect to the given contract (PUT-PROXY). If so, the value gets assigned and the monitor extended by a write effect.

Both, the get and put rules, signal a violation by being stuck. The behavior of the implementation is configurable: it may raise an exception and stop execution or it may just log the violation and continue. This behavior could be formalized as well by introducing separate crash judgments. We omit the definition of these judgments because their inference rules largely duplicate the rules from Figures 10, 11, 12, and 13.

## 7. Reduction

As the target object of a proxy may be a proxy itself, there may be a chain of proxies to traverse before reaching the actual non-proxy target object. Such chains waste memory, they increase the run time of all operations, and the intermediate proxies may contain redundant information. To avoid the creation of inefficient chains of nested proxies, we create a "proxy of a proxy" as follows: the new proxy directly refers to the target of the existing proxy, its path set is the union of the new path and the already existing pathset, and its contract is merged with the contract of the already existing handler. Fortunately, contracts can be merged easily by using the conjunction operator $\&$, which means that the restrictions enforced along *all* reaching paths are enforced. Extending the formalization, the handler contains a set of paths, instead of a single path.

However, naively following this approach leads to two problems. First, a path update operation has to extend every path in a set and the redundant parts in a set waste a lot of memory. Second, the combination of contracts may result in a contract whose parts cancel or subsume one another. Redundant parts in a combined contract also waste memory and make the computation of the derivative more expensive. As a result, in our initial experiments, test cases with many objects could not be analyzed in a sensible amount of time.

This section reports some optimizations to reduce memory consumption and to improve the run time.

### 7.1 Trie Structure

In a first step, we changed the representation of a set of access paths to a trie structure [17]. Let $\mathit{PathTrie} \ni \mathcal{T} ::= \emptyset \mid \mathcal{T}[p \mapsto \mathcal{T}']$ be a trie structure used as prefix tree to store paths $\mathcal{P}$. At each node $\mathcal{T}$, there is at most one association $[p \mapsto \mathcal{T}']$ for each property $p$. It indicates that there is a path of the form $p.\mathcal{P}$ in the trie, where $\mathcal{P}$ is in $\mathcal{T}'$. We use $[\epsilon \mapsto \emptyset]$ to mark the end of a path.

A path $\mathcal{P}$ in a trie $\mathcal{T}$ is thus represented by the concatenation of the properties $p$ on a path from the root to an edge labeled with the empty path $\epsilon$.

Using trie structures enables us to share prefixes in path sets, which reduces the memory usage significantly. In particular, linked data structures like lists and trees give rise to many shared prefixes which are represented efficiently by the trie structure.

### 7.2 Contract Rewriting

Besides the normalization of contracts (Section 5.2) the containment reduction of contracts is important to reduce memory consumption. Containment reduction means that a contract $\mathcal{C}+\mathcal{C}'$ can be reduced to $\mathcal{C}$ if $\mathcal{L}[\![\mathcal{C}']\!] \subseteq \mathcal{L}[\![\mathcal{C}]\!]$. Similarly, $\mathcal{C}\&\mathcal{C}'$ can be reduced to $\mathcal{C}'$ if $\mathcal{L}[\![\mathcal{C}']\!] \subseteq \mathcal{L}[\![\mathcal{C}]\!]$. The implementation also accounts for commutativity.

First, we define a semantic containment relation on contracts.

**Definition 5** (Containment)**.** *A contract $\mathcal{C}$ is contained in another contract $\mathcal{C}'$, written as $\mathcal{C} \sqsubseteq \mathcal{C}'$, iff $\mathcal{L}[\![\mathcal{C}]\!] \subseteq \mathcal{L}[\![\mathcal{C}']\!]$.*

The containment relation on contracts is reflexive and transitive and thus forms a preorder. As it is defined semantically, we need a syntactic decision procedure for containment of contracts to put it to use. This procedure is inspired by Antimirov's calculus [2], which provides a non-deterministic decision procedure to solve the containment problem for ordinary regular expressions.

From the definition of containment (Definition 5), we obtain that $\mathcal{C} \sqsubseteq \mathcal{C}'$ iff for all paths $\mathcal{P} \in \mathcal{L}[\![\mathcal{C}]\!]$ the derivative of contract $\mathcal{C}'$ with respect to path $\mathcal{P}$ is nullable $\nu(\partial_{\mathcal{P}}(\mathcal{C}'))$.

**Lemma 3** (Containment)**.**

$$\mathcal{C} \sqsubseteq \mathcal{C}' \;\Leftrightarrow\; \nu(\partial_{\mathcal{P}}(\mathcal{C}')) \textit{ for all } \mathcal{P} \in \mathcal{L}[\![\mathcal{C}]\!] \tag{5}$$

The proof is by Lemma 1 and 2.

As this lemma does not yield an effective way of deciding containment, we aim to enumerate the access paths of $\mathcal{C}$ by iteratively extracting its possible first properties and forming the derivatives on both sides as in Antimirov's procedure.

**Definition 6** (next)**.** *The function next : $\mathcal{C} \to \mathcal{A}$ returns the first properties $p$ of path elements $\mathcal{P}$ in $\mathcal{L}[\![\mathcal{C}]\!]$.*

$$\mathit{next}(\mathcal{C}) \;=\; \{p \mid p.\mathcal{P} \in \mathcal{L}[\![\mathcal{C}]\!]\} \tag{6}$$

From Antimirov's theorem [2], we obtain that $\mathcal{C} \sqsubseteq \mathcal{C}'$ iff $\nu(\mathcal{C})$ implies $\nu(\mathcal{C}')$ and $\forall p : \partial_p(\mathcal{C}) \sqsubseteq \partial_p(\mathcal{C}')$.

**Lemma 4** (Containment2)**.**

$$\begin{aligned} \mathcal{C} \sqsubseteq \mathcal{C}' \;\Leftrightarrow\; & (\forall p \in \mathit{next}(\mathcal{C}))\; \partial_p(\mathcal{C}) \sqsubseteq \partial_p(\mathcal{C}') \\ & \wedge\; (\nu(\mathcal{C}) \Rightarrow \nu(\mathcal{C}')) \end{aligned} \tag{7}$$

The proof combines the assertion of Lemma 3 with the stepwise derivative of Lemma 2.

Unfortunately, it is in general impossible to construct all derivatives with respect to all first properties. In case of a question mark literal or a negation, there may be infinitely many first literals. Thus, the implementation needs to apply some kind of approximation.

#### 7.2.1 Literal-derivative of Contracts

To abstract from the derivative of a contract with respect to a property, we introduce a literal-based derivative which forms the derivative of a contract with respect to a literal $\ell$. This operation admits derivatives with respect to the contract literals $@$, $?$, $r$, and $!r$, and performs some approximation by returning a contract that is contained in the derivatives resulting from expanding the literals.

As the literals include character-level regular expressions that we treat as abstract, we rely on a relation $\sqsubseteq_r$ that decides containment of literals and on an operation $\sqcap_r$ that builds the intersection of two literals. Both are considered as abstract operators on the language of regular expression literals and we do not specify their implementation (but similar methods as for contracts apply).

$$\ell \sqsubseteq_r r \;\Leftrightarrow\; \mathcal{L}[\![\ell]\!] \subseteq \mathcal{L}[\![r]\!] \tag{8}$$

$$\mathcal{L}[\![\ell \sqcap_r r]\!] \;=\; \mathcal{L}[\![\ell]\!] \cap \mathcal{L}[\![r]\!] \tag{9}$$

Figure 14 contains the definition of the syntactic derivative with respect to a literal. Deriving a literal $\ell$ w.r.t. itself results in the

$$
\begin{array}{lcl}
\nabla_\ell(@) & = & \begin{cases} \mathcal{E}, & \ell = @ \\ \emptyset, & \text{otherwise} \end{cases} \\
\nabla_\ell(?) & = & \mathcal{E} \\
\nabla_\ell(r) & = & \begin{cases} \mathcal{E}, & \ell \sqsubseteq_r r \\ \emptyset, & \text{otherwise} \end{cases} \\
\nabla_\ell(!r) & = & \begin{cases} \mathcal{E}, & \ell \sqcap_r r = \emptyset \\ \emptyset, & \text{otherwise} \end{cases} \\
\nabla_\ell(\emptyset) & = & \emptyset \\
\nabla_\ell(\mathcal{E}) & = & \emptyset \\
\nabla_\ell(\mathcal{C}*) & = & \nabla_\ell(\mathcal{C}).\mathcal{C}* \\
\nabla_\ell(\mathcal{C}+\mathcal{C}') & = & \nabla_\ell(\mathcal{C})+\nabla_\ell(\mathcal{C}') \\
\nabla_\ell(\mathcal{C}\&\mathcal{C}') & = & \nabla_\ell(\mathcal{C})\&\nabla_\ell(\mathcal{C}') \\
\nabla_\ell(\mathcal{C}.\mathcal{C}') & = & \begin{cases} \nabla_\ell(\mathcal{C}).\mathcal{C}'+\nabla_\ell(\mathcal{C}'), & \nu(\mathcal{C}) \\ \nabla_\ell(\mathcal{C}).\mathcal{C}', & \text{otherwise} \end{cases}
\end{array}
$$

**Figure 14.** Derivative of a contract by a contract literal.

$$
\begin{array}{lcl}
\mathsf{first}(@) & = & \{@\} \\
\mathsf{first}(?) & = & \{?\} \\
\mathsf{first}(r) & = & \{r\} \\
\mathsf{first}(!r) & = & \{!r\} \\
\mathsf{first}(\emptyset) & = & \{\} \\
\mathsf{first}(\mathcal{E}) & = & \{\} \\
\mathsf{first}(\mathcal{C}*) & = & \mathsf{first}(\mathcal{C}) \\
\mathsf{first}(\mathcal{C}+\mathcal{C}') & = & \mathsf{first}(\mathcal{C}) \cup \mathsf{first}(\mathcal{C}') \\
\mathsf{first}(\mathcal{C}\&\mathcal{C}') & = & \{\ell \sqcap_r \ell' \mid \ell \in \mathsf{first}(\mathcal{C}), \ell' \in \mathsf{first}(\mathcal{C}')\} \\
\mathsf{first}(\mathcal{C}.\mathcal{C}') & = & \begin{cases} \mathsf{first}(\mathcal{C}) \cup \mathsf{first}(\mathcal{C}'), & \nu(\mathcal{C}) \\ \mathsf{first}(\mathcal{C}), & \text{otherwise} \end{cases}
\end{array}
$$

**Figure 15.** first on contracts.

empty contract $\mathcal{E}$. Applying any derivative to the empty set $\emptyset$ yields the empty set $\emptyset$. The derivative of a regular expression literal $r$ w.r.t. a literal $\ell$ is the empty contract $\mathcal{E}$ if the language of $\ell$ is subsumed by the regular expression $r$, that is, if we can make a step with each character in the literal. Similarly, the derivative of a negated regular expression literal $!r$ w.r.t. a literal $\ell$ is the empty contract $\mathcal{E}$ if no property of the language of $\ell$ can make a step in $r$. Otherwise, the result is $\emptyset$. The remaining cases are exactly as in the derivative with respect to a property.

The following lemma states the connection between the derivative by contract literal and the derivative by a property.

**Lemma 5** (Syntactic derivative of contracts). $\forall \ell$ :

$$\mathcal{L}[\![\nabla_\ell(\mathcal{C})]\!] \subseteq \bigcap_{p \in \mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C})]\!] \tag{10}$$

### 7.2.2 Containment

Using the syntactic derivation, we are able to abstract the derivative of a contract w.r.t. an infinite property set by forming derivatives with respect to literals. Before combining this abstraction with the containment lemma (Lemma 4), we define a function to obtain the first literals of a contract as shown in Figure 15.

The first literal of a literal $\ell$ is $\ell$. $\emptyset$ and $\mathcal{E}$ have no first literals. The first literals of a Kleene star contract $\mathcal{C}*$ are the first literals of its subcontract. The first literals of a disjunction are the union of the first literals of its subcontracts. For a conjunction, the set of first literals is the set of all intersections $\ell \sqcap_r \ell'$ of the first literals of both conjuncts. The first literals of a concatenation are the first literals of its first subcontract if the first subcontract is not nullable. Otherwise, it is the union of the first literals of both subcontracts.

The language of the first literals is defined to be the union of the languages of its literals.

$$\mathcal{L}[\![\mathsf{first}(\mathcal{C})]\!] = \bigcup_{\ell \in \mathsf{first}(\mathcal{C})} \mathcal{L}[\![\ell]\!] \tag{11}$$

**Lemma 6** (first).

$$\mathit{next}(\mathcal{C}) \ = \ \mathcal{L}[\![\mathit{first}(\mathcal{C})]\!] \tag{12}$$

**Lemma 7** (Syntactic derivative of contracts 2). $\forall \ell \in \mathit{first}(\mathcal{C})$ :

$$\mathcal{L}[\![\nabla_\ell(\mathcal{C})]\!] \ = \ \bigcap_{p \in \mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C})]\!] \tag{13}$$

**Theorem 1** (Containment).

$$\begin{aligned} \mathcal{C} \sqsubseteq \mathcal{C}' \ \Leftarrow \ & (\forall \ell \in \mathit{first}(\mathcal{C} \sqsubseteq \mathcal{C}'))\ \nabla_\ell(\mathcal{C}) \sqsubseteq \nabla_\ell(\mathcal{C}') \\ & \wedge\ (\nu(\mathcal{C}) \Rightarrow \nu(\mathcal{C}')) \end{aligned} \tag{14}$$

### 7.2.3 Containment Semantics

Based on the containment theorem (Theorem 1) and the syntactic derivative, we present an algorithm that approximates the containment relation of contracts.

To recapitulate, a path $\mathcal{P}$ is an element of the language defined by a contract $\mathcal{C}$ iff the derivative of contract $\mathcal{C}$ w.r.t. path $\mathcal{P}$ is nullable $\nu(\partial_{\mathcal{P}}(\mathcal{C}))$. If a contract $\mathcal{C}$ is not contained in a contract $\mathcal{C}'$, then there exists at least one access path $\mathcal{P} \in \mathcal{L}[\![\mathcal{C}]\!]$ that derives a non-nullable contract from $\mathcal{C}'$.

Define containment expressions by $\phi \ ::= \ \mathcal{C} \sqsubseteq \mathcal{C}'$ and let a context $\Gamma$ be a set of previous visited containment expressions. The context lookup $\phi \in \Gamma$ determines if an expression is already calculated in this chain. The derivation of a containment expression $\nabla_\ell(\mathcal{C} \sqsubseteq \mathcal{C}')$ is forwarded to its subcontracts as $\nabla_\ell(\mathcal{C}) \sqsubseteq \nabla_\ell(\mathcal{C}')$.

The decision procedure is defined by a judgment of the form $\Gamma \vdash \phi : \{\top, \bot\}$. The evaluation of expression $\phi$ in context $\Gamma$ results either in true $\top$ or false $\bot$. In Figure 16, rule (C-DISPROVE) shows the generalized disproving axiom. If the first contract $\mathcal{C}$ is nullable and the second contract $\mathcal{C}'$ is not nullable, then there exists at least one element–the empty access path $\mathcal{E}$–in the language of $\mathcal{L}[\![\mathcal{C}]\!]$ which is not element of $\mathcal{L}[\![\mathcal{C}']\!]$. This condition is sufficient to disprove the inequality, so the rule returns false. Rule (C-DELETE) returns true if the evaluated expression is already subsumed by the context. Further derivatives of $\phi$ would not contribute new information. (C-UNFOLD-TRUE) and (C-UNFOLD-FALSE) applies only if $\phi$ is not in the context. It applies all derivatives according to $\mathsf{first}(\phi)$ and conjoins them together.

(C-DISPROVE)
$$\frac{\nu(\mathcal{C}) \qquad \neg\nu(\mathcal{C}')}{\Gamma \vdash \mathcal{C} \sqsubseteq \mathcal{C}' \ : \ \bot}$$

(C-DELETE)
$$\frac{\phi \in \Gamma}{\Gamma \vdash \phi \ : \ \top}$$

(C-UNFOLD-TRUE)
$$\frac{\phi \notin \Gamma \qquad \forall \ell \in \mathsf{first}(\phi) : \langle \Gamma, \phi \rangle \vdash \nabla_\ell(\phi) \ : \ \top}{\Gamma \vdash \phi \ : \ \top}$$

(C-UNFOLD-FALSE)
$$\frac{\phi \notin \Gamma \qquad \exists \ell \in \mathsf{first}(\phi) : \langle \Gamma, \phi \rangle \vdash \nabla_\ell(\phi) \ : \ \bot}{\Gamma \vdash \phi \ : \ \bot}$$

**Figure 16.** Unfolding axioms and rules.

$$\text{(C-IDENTITY)}\quad \frac{}{\Gamma \vdash \mathcal{C} \sqsubseteq \mathcal{C} \;:\; \top} \qquad \text{(C-PROOF-EDGE)}\quad \frac{\mathsf{emp}(\mathcal{C}) \vee \mathsf{unv}(\mathcal{C}')}{\Gamma \vdash \mathcal{C} \sqsubseteq \mathcal{C}' \;:\; \top} \qquad \text{(C-NULLABLE)}\quad \frac{\nu(\mathcal{C}')}{\Gamma \vdash \mathcal{E} \sqsubseteq \mathcal{C}' \;:\; \top}$$

**Figure 17.** Prove axioms.

$$\text{(C-DISPROVE-EMPTY)}\quad \frac{\neg\mathsf{emp}(\mathcal{C}) \qquad \mathsf{emp}(\mathcal{C}')}{\Gamma \vdash \mathcal{C} \sqsubseteq \mathcal{C}' \;:\; \bot} \qquad \text{(C-DISPROVE-BLANK)}\quad \frac{\mathsf{ind}(\mathcal{C}) \vee \mathsf{unv}(\mathcal{C}) \qquad \mathsf{bl}(\mathcal{C}')}{\Gamma \vdash \mathcal{C} \sqsubseteq \mathcal{C}' \;:\; \bot}$$

**Figure 18.** Disprove axioms.

$$
\begin{aligned}
\lfloor \mathcal{C}* \rfloor &= \begin{cases} \mathcal{E}, & \mathsf{emp}(\mathcal{C}) \ \vee\ \mathsf{bl}(\mathcal{C}) \\ \lfloor \mathcal{C} \rfloor *, & \text{otherwise} \end{cases} \\
\lfloor \mathcal{C}{+}\mathcal{C}' \rfloor &= \begin{cases} \emptyset, & \mathsf{emp}(\mathcal{C}) \wedge \mathsf{emp}(\mathcal{C}') \\ @, & \mathsf{bl}(\mathcal{C}) \wedge \mathsf{bl}(\mathcal{C}') \\ \lfloor \mathcal{C} \rfloor, & \mathcal{C} \sqsupseteq \mathcal{C}' \\ \lfloor \mathcal{C}' \rfloor, & \mathcal{C} \sqsubseteq \mathcal{C}' \\ \lfloor \mathcal{C} \rfloor + \lfloor \mathcal{C}' \rfloor, & \text{otherwise} \end{cases} \\
\lfloor \mathcal{C}\&\mathcal{C}' \rfloor &= \begin{cases} \emptyset, & \mathsf{emp}(\mathcal{C}) \vee \mathsf{emp}(\mathcal{C}') \\ @, & \mathsf{bl}(\mathcal{C}) \vee \mathsf{bl}(\mathcal{C}') \\ \lfloor \mathcal{C} \rfloor, & \mathcal{C} \sqsubseteq \mathcal{C}' \\ \lfloor \mathcal{C}' \rfloor, & \mathcal{C} \sqsupseteq \mathcal{C}' \\ \lfloor \mathcal{C} \rfloor \& \lfloor \mathcal{C}' \rfloor, & \text{otherwise} \end{cases} \\
\lfloor \mathcal{C}.\mathcal{C}' \rfloor &= \begin{cases} \emptyset, & \mathsf{emp}(\mathcal{C}) \\ @, & \mathsf{bl}(\mathcal{C}) \\ \lfloor \mathcal{C} \rfloor . \lfloor \mathcal{C}' \rfloor, & \text{otherwise} \end{cases}
\end{aligned}
$$

**Figure 19.** Reduction rules.

**Theorem 2** (Correctness)**.**

$$\Gamma \vdash \mathcal{C} \sqsubseteq \mathcal{C}' \;:\; \top \;\Rightarrow\; \mathcal{C} \sqsubseteq \mathcal{C}' \qquad (15)$$

In addition to the rules from Figure 16, we add auxiliary axioms to detect trivially consistent (inconsistent) inequalities early (Figures 17 for consistent inequalities and 18 for inconsistent ones). They decide containment directly without unfolding. The axioms rely on four functions that operate on the contract syntax: bl, emp, ind, and unv. Each of them is correct, but not complete. For instance, the function emp can only approximate whether the language of a contract of the form $\mathcal{C}\&\mathcal{C}'$ denotes the empty set. The following definition specifies these functions, their actual definitions are straightforward and thus elided.

**Definition 7.** *A contract $\mathcal{C}$ is . . .*

***blank*** *if $\mathcal{L}[\![\mathcal{C}]\!] = \{\iota\}$, let $bl(\mathcal{C}) = \top$ imply $\mathcal{C}$ is blank;*
***empty*** *if $\mathcal{L}[\![\mathcal{C}]\!] = \emptyset$, let $emp(\mathcal{C}) = \top \Rightarrow$ imply $\mathcal{C}$ is empty;*
***indifferent*** *if $\mathcal{L}[\![\mathcal{C}]\!] = \mathcal{A}$, let $ind(\mathcal{C}) = \top$ imply $\mathcal{C}$ is indifferent;*
***universal*** *if $\mathcal{L}[\![\mathcal{C}]\!] = \mathcal{A}^*$, let $unv(\mathcal{C}) = \top$ imply $\mathcal{C}$ is universal.*

#### 7.2.4 Reductions Rules

Finally, we apply the preceding machinery to define a reduction function $\lfloor \cdot \rfloor$ on contracts in Figure 19. Reduction produces an equivalent contract, which is smaller than its input.

Literals and empty contracts are not further reducible. A Kleene star contract $\mathcal{C}*$ is reduced to the empty contract $\mathcal{E}$ if the subcontract is either empty or blank. The disjunction contract $\mathcal{C}{+}\mathcal{C}'$ is reduced to the empty set $\emptyset$ if both contracts are empty or it reduces to the empty literal @ if both contracts are blank. If one of the subcontracts is subsumed by the other one, the subsuming contract is used. Similarly, the conjunction contract $\mathcal{C}\&\mathcal{C}'$ is reduced to the empty set $\emptyset$ or the empty literal @ if one of the subcontracts is empty or blank. If one contract subsumes the other, then the subsumed contract is used. The concatenation $\mathcal{C}.\mathcal{C}'$ is reduced to the empty set $\emptyset$ if the first subcontract is empty or it is reduced to the empty literal @ if the first sub-contract is blank.

## 8. Implementation

The implementation is based on the JavaScript Proxy API [14, 43], a proposed addition to the JavaScript standard. This API is implemented in Firefox since version 18.0 and in Chrome V8 since version 3.5. We developed the implementation using the *SpiderMonkey JavaScript-C 1.8.5 (2011-03-31)* JavaScript engine.

### 8.1 Description

The implementation provides a proxy handler `AccessHandler` that overrides all trap functions. The traps implement the access control mechanism as well as path monitoring. They either interrupt the operation, if it is not permitted, or forward it to the target object. They maintain the path set and contract data structures using the fly-weight pattern to minimize memory consumption.

Our framework can easily be included in existing JavaScript software projects. Its functionality is encapsulated in a facade whose interface–the function `permit`–can be used to wrap objects.

The framework provides two evaluation modes, *Observer Mode* and *Protector Mode*. The *Observer Mode* performs only path and violation logging without changing the semantics of the underlying program. Thus, if a program reads multiple properties along a prohibited path, then each individual read is logged as a violation. For example, suppose an object is protected by the contract `'b+c'`. Reading property `a` results in a violation with access path `a` and a subsequent read of `a.b` results in a violation of `a.b`, and so on.

The *Protector Mode* follows the scripting-language philosophy as implemented in the rest of JavaScript. If a read access violates the contract of an object, the value `undefined` is returned instead of an abnormal termination. Forbidden write accesses are simply omitted. Thus, only top-level violations are visible.

Our framework comes with a JavaScript-based GUI. Included in a web page, the interface shows all accessed paths as well as all incurred contract violations. A heuristic allows us to generate short effect descriptions from the gathered path sets using the approach reported elsewhere [22, 24].

### 8.2 Limitations

Because of the browser's sandbox, JSConTest2 cannot directly protect DOM objects with access permission contracts. The security mechanism forbids to replace the references to the `window` and `document` objects by suitably contracted proxies. This deficiency can be partially addressed by embedding an entire script in a scope which substitutes the global object by a suitable proxy.

The use of proxies for access control has one unfortunate consequence: the equality operators == and === do not work correctly, anymore. Dependening on the access path, the same target object may have different access rights and hence distinct proxies that enforce these rights. Comparing these distinct proxies returns false even though the underlying target is the same. Similarly, an unwrapped target object may be compared with its contracted version, which should be true, but yields false.

Here is an example illustrating the problem.

```
var ch = { c : 42 }
```

```
2 var root = __APC.permit ('a.@+b.c', { a : ch, b : ch })
3 var same_acc = (root.a === root.b)
4 var same_unw = (ch === root.b)
```

With our implementation, both same_acc and same_unw are false although they are true without the **permit** operation.

Unfortunately, there is no easy way to address this shortcoming. One possibility is to assign each target a unique proxy, which requires a potentially unintuitive merge of different access contract. Another idea would be to trap the equality operation, which is not supported by the proxy API. However, neither the unique proxy nor trapping the equality operation would solve the problem with comparing the proxy with its target as in same_unw (just consider === as a method call on the unwrapped target object `ch` in line 4).

The best solution would be to provide two proxy-aware equality functions and replace all uses of == and === by these functions. This solution would require some light rewriting of the source code (also at run time to support eval), which is much less intrusive than the rewriting of the original JSConTest implementation. Currently, we do not supply this rewriting because none of the programs we examined in our evaluation were affected by the problem.

## 9. Evaluation

This section reports on our experiences with applying JSConTest2 to selected programs. All benchmarks were run on a MacBook Pro with a 2 GHz Intel Core i7 processor with 8 GB memory. All example runs and timings reported in this paper were obtained with version 23.0a2 (2013-05-21) of the *Firefox Aurora* browser.

### 9.1 Benchmark Programs

To evaluate our implementation, we applied it to a range of JavaScript programs: the Google V8 Benchmark Suite[2] and a selection of benchmarks accompanying the TAJS system [28].

The Google V8 Benchmark Suite consists of a webpage with several JavaScript programs, which are listed in Figure 20. The benchmarks range from about 400 to 5000 lines of code implementing an OS kernel simulation, constraint solving, encryption, ray tracing, parsing, regular expression operations, benchmarking data structures, and solving differential equations. The suite was originally composed to evaluate the performance of JavaScript engines. It is designed to stress various aspects of the implementation of a JavaScript engine, but the programs it contains are not necessarily representative of the typical programs run in a browser.

The TAJS benchmarks consist of JavaScript programs and dumped Web pages collected in the wild to test the static analysis system TAJS. To easily run the tests in the Aurora web browser, we selected all programs that came packaged with a webpage: 3dmodel, countdown, oryx, ajaxtabscontent, arkanoid, ball_pool, bunnyhunt, gamespot, google_pacman, jscalc, jscrypto, logo, mceditor, minesweeper, msie9, simple_calc, wala. The selection further contains programs like a calculator or a simple browser game as well as libraries extending the functionality of JavaScript like jQuery, a linked list data type, or an MD5 hashing library. We also applied our system to a number of dumped web pages like *youtube*, *twitter*, or *imageshack*.

### 9.2 Methodology

To evaluate our implementation with the Google Benchmarks, we manually examined their source code, identified frequently used objects, and marked them with an empty contract **@**. Each access to those objects generated an access violation, which was logged.

In the TAJS benchmarks, we looked for interesting objects and functions, non-locally used data, and uses of external libraries like

[2] `http://v8.googlecode.com/svn/data/benchmarks/v7/run.html`

| **Benchmark** | **Full** | **Without logging** | **Contracts only** | **Baseline** |
|---|---|---|---|---|
| Richards | 22.5min | 18.6min | 3.3sec | 2.3sec |
| DeltaBlue | 9.8sec | 9.5sec | 3.3sec | 2.3sec |
| Crypto | 4.2h | 2.5h | 2.6min | 4.4sec |
| RayTrace | 1.2h | 1.1h | 1.6min | 2.3sec |
| EarleyBoyer | 4.4sec | 4.4sec | 4.4sec | 4.3sec |
| RegExp | 2.4sec | 2.4sec | 2.4sec | 2.4sec |
| Splay | - | - | 2.3sec | 2.3sec |
| NavierStokes | 2.3sec | 2.3sec | 2.3sec | 2.3sec |

**Figure 20.** Google V8 Benchmark Suite.

jQuery. In a first run, we augmented these objects with a universal contract (e.g. **?***) to monitor the accessed properties. Based on the generated protocol we prepared customized contracts to protect these objects. To exercise the customized contracts, we extended the source code with additional, nonconforming operations to provoke violations.

### 9.3 Results

With our initial implementation, contract enforcement for programs in the Google V8 Benchmark Suite was not possible, because the browser quickly ran out of memory. Our reimplementation based on the ideas described in Section 7 enabled us to cut down memory consumption dramatically.[3] The reduction in memory use of path logging comes at the expense of higher computational cost. The reimplemented system successfully applies contract enforcement to all programs in the Google V8 Benchmark Suite; for the *Splay* benchmark, we have no numbers for path set collection and logging because it did not terminate within four hours.

Figure 20 contains the run times for all V8 benchmark programs in different configurations. The column *Full* contains the run time for contract enforcement, path set collection, and log output. The effect heuristic to condense the resulting set of paths to a short effect description is disabled. The column *Without logging* shows the time used for contract enforcement and path set collection, but without logging. The column *Contracts only* shows the time for contract enforcement, without any path set generation. The last column *Baseline* shows the baseline for a run without JSConTest2.

Using forwarding proxies instead of normal objects did not have a measurable effect: in addition to the above configurations, we ran the benchmarks with forwarding proxies, where the handler intercepts all operations but the trap functions forward the operation to the target object, as shown in Figure 1. The resulting run times exhibit no measurable difference to the numbers in column *Baseline*.

In most benchmarks, the run-time difference between contract enforcement and the baseline is negligible, so monitoring is cheap. The exceptions are *Crypto* and *RayTrace* where the contract is applied to the main API object.

The run times for the programs in the V8 Benchmark Suite range from few seconds up to four hours when running with contract monitoring fully engaged. This run time depends on the objects chosen for contract monitoring: contracting heavily used objects causes more overhead (viz. *Crypto* and *RayTrace*), contracting the root of a tree (for example in *Splay*) also causes overhead because the membrane implementation creates a shadow tree populated with proxies while the program runs.

Unfortunately, the most expensive benchmark (*Splay*) increases the size of the trie structure in a way that the contract implementation was not able to handle efficiently. Further optimizations like condensing paths are required to run this benchmark to completion.

[3] Unfortunately, we did not find a way to measure memory consumption.

The second most expensive benchmark (*Crypto*) produces more than 5GB output of logged paths, depending on the selected object. For comparison, the benchmark *Richards* requires approximately 22.5 minutes to calculate slightly more than 1GB of logged paths. In *Crypto*, a significant percentage of the memory consumption and the computation time is due to the path recovery at the end of the run. This mechanism flattens a trie structure to a list of paths, which removes all sharing from the structure. It accounts for much of the difference between columns *Full* and *Without logging* in Figure 20.

These examples show that the run time impact of monitoring is highly dependent on the program and on the particular values that are monitored. While some programs are heavily affected (*Crypto*, *Richards*, *RayTrace*, *Splay*), others are almost unaffected: *EarleyBoyer*, *RegEx*, *NavierStokes*.

The numbers also show that the path logging accounts for most of the run-time overhead: the biggest fraction of the total run time is used for path generation, which comprises appending of trie structures and merging tries. The remaining time is spent for path reconstruction, logging, and log output.

The evaluation of contracts themselves is negligible in many cases, but occasionally it may create an overhead of 35x (*Crypto*) to 41x (*RayTrace*). On the other hand, these programs are an artificial selection to stress the JavaScript implementation.

The run times of the more realistic collection of TAJS benchmark programs are all much shorter (less than one second) than the run times for the V8 benchmarks. Furthermore, the difference between the run times of the four configurations listed in Figure 20 is negligible for the TAJS benchmarks. These findings indicate that contract monitoring seems feasible for realistic programs.

### 9.4 General Observations

The benchmarks show that the most time-consuming parts are path logging and contract derivation. Their overhead is influenced by several factors: the number and frequency of proxy calls and the length of access chains.

The length of the access chains determines the number of derivation steps and the size of the trie structure. Further, the number of nested proxies influences the number of merge operations and can cause a blowup of the data structures. Path extension and computing the contract derivative is more expensive on merged handlers. Also, the structure of the contract affects the performance. Wide and complex contracts require more derivation steps than deep contracts. In addition, a derived contract sometimes gets bigger than the original contract. For example, the contract $a{*}{+}a{*}.a{*}$ is the result of deriving $a{*}.a{*}$ by $a$. All these factors contribute to the time and space complexity of contract monitoring.

## 10. Related Work

JSConTest [22, 24] is a framework for logging side effects and enforcing path-based access permission contracts. It comes with an algorithm [23] that infers a concise effect description from a set of access paths. Access permission contract enable the specification of effects to restrict the access to the object graph by defining a set of permitted access paths. JSConTest is based on an offline code transformation. Its implementation is restricted to a subset of the language, it does not scale to large programs, and it is hard to guarantee full interposition.

Access permission contracts are closely related to extended regular expression. Permissions are computed from the iterated derivative of a contracts by the current access path. Derivatives of extended regular expressions and their properties are well known from the literature [3, 10, 39]. Computing contract subsumption is related to solving regular expression inequalities and checking regular expression equivalence, which has been addressed in several places [2, 25, 31]. Most approaches rely on NFA checking [7, 32] or on rewriting [1, 4, 42]. For checking contract subsumption, we adapted Antimirov's approach to obtain a reasonably fast algorithm. We extended Antimirov's algorithm to an infinite alphabet and to extended regular expressions including negation and intersection.

The JavaScript Reflection API [14, 43] enables developers to easily enhance the functionality of objects and functions. The implementation of proxies opens up the means to fully interpose operations applied to objects and functions calls. Proxies have already been used for dynamic effects systems [26]. Other common uses for proxies, e.g. [5, 8, 9, 15, 37, 44], are meta-level extension, behavioral reflection, security, or concurrency control.

There are further proposals to limit effects on heap-allocated objects both statically and dynamically. An *effect system* is a static analysis that partitions the heap into disjoint regions and annotates the type of a heap reference with the region in which the reference points [18]. Although initially developed for functional languages, region-based effects have been transposed to object-oriented languages [19]. A notable proposal targeting Java is the type and effect system of DPJ [6]. DPJ targets parallel execution and provides by default a deterministic semantics.

Also, specification languages like JML [11, 33] include a mechanism for specifying side effects, the `assignable` clause. While the JML toolchain supports verification as well as run-time monitoring [12, 34, 35], `assignable` clauses are not widely used, partly because their semantics has not been formally and unanimously defined until recently [35], and partly because support for `assignable` clauses is present in only a few tools that perform run-time monitoring for JML [36] and then not always in full generality[12].

Our system may also be useful to guarantee security aspects like confidentiality or integrity of information. In JavaScript, static approaches are often lacking because of the dynamicity of the language. However, the approaches range from static and dynamic control of information flow control [13, 21, 29] over restricting the functionality [38] to the isolations of scopes [40].

## 11. Conclusion

We successfully applied JavaScript proxies to the implementation of effect logging and dynamic enforcement of access permission contracts, which specify the allowed side effects using access paths in the object graph. The implementation avoids the shortcomings of an earlier implementation in the JSConTest system, which is based on an offline code transformation. The proxy-based approach handles the full JavaScript language, including the `with`-statement, `eval`, and arbitrary dynamic code loading techniques. Contrary to the earlier implementation, the proxy-based approach guarantees full interposition.

This reimplementation presents a major step towards practical applicability of access permission contracts. The run-time overhead and the additional memory consumption of pure contract enforcement is negligible. Hence, we believe that this implementation can provide encapsulation in realistic applications, as demonstrated with our examples and case studies. Full effect logging, on the other hand, incurs quite some overhead, but we regard it primarily as a tool for program understanding and debugging.

## References

[1] M. Almeida, N. Moreira, and R. Reis. Antimirov and mosses's rewrite system revisited. *Int. J. Found. Comput. Sci.*, 20(4):669–684, 2009.

[2] V. M. Antimirov. Rewriting regular inequalities (extended abstract). In H. Reichel, editor, *FCT*, volume 965 of *Lecture Notes in Computer Science*, pages 116–125. Springer, 1995.

[3] V. M. Antimirov. Partial derivates of regular expressions and finite automata constructions. In *STACS*, pages 455–466, 1995.

[4] V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. In *Developments in Language Theory*, pages 195–209, 1993.

[5] T. H. Austin, T. Disney, and C. Flanagan. Virtual values for language extension. In C. V. Lopes and K. Fisher, editors, *OOPSLA*, pages 921–938, Portland, OR, USA, 2011. ACM. ISBN 978-1-4503-0940-0.

[6] R. L. Bocchino Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In S. Arora and G. T. Leavens, editors, *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 97–116, Orlando, Florida, USA, 2009. ACM Press, New York. ISBN 978-1-60558-766-0.

[7] F. Bonchi and D. Pous. Checking NFA equivalence with bisimulations up to congruence. In R. Giacobazzi and R. Cousot, editors, *POPL*, pages 457–468. ACM, 2013.

[8] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In J. M. Vlissides and D. C. Schmidt, editors, *OOPSLA*, pages 331–344. ACM, 2004.

[9] J. Brant, B. Foote, R. E. Johnson, and D. Roberts. Wrappers to the rescue. In E. Jul, editor, *ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 396–417. Springer, 1998.

[10] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[11] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005. ISSN 1433-2779. .

[12] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, Apr. 2003. TR #03-09.

[13] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In M. Hind and A. Diwan, editors, *PLDI*, pages 50–62. ACM, 2009.

[14] T. V. Cutsem and M. S. Miller. Proxies: design principles for robust object-oriented intercession APIs. In W. D. Clinger, editor, *DLS*, pages 59–72. ACM, 2010. ISBN 978-1-4503-0405-4.

[15] P. T. Eugster. Uniform proxies for Java. In P. L. Tarr and W. R. Cook, editors, *OOPSLA*, pages 139–152. ACM, 2006.

[16] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C.-C. Hung, editors, *SAC*, pages 2103–2110, Sierre, Switzerland, 2010. ACM. ISBN 978-1-60558-639-7.

[17] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, Sept. 1960.

[18] D. Gifford and J. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conf. on Lisp and Functional Programming*, pages 28–38, Cambridge, Massachusetts, United States, 1986. ACM Press.

[19] A. Greenhouse and J. Boyland. An object-oriented effects system. In R. Guerraoui, editor, *13th European Conference on Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 205–229, Lisbon, Portugal, June 1999. Springer-Verlag. ISBN 3-540-66156-5.

[20] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In T. D'Hondt, editor, *ECOOP*, volume 6183 of *Lecture Notes in Computer Science*, pages 126–150. Springer, 2010.

[21] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In S. Chong, editor, *CSF*, pages 3–18. IEEE, 2012.

[22] P. Heidegger and P. Thiemann. Contract-driven testing of JavaScript code. In J. Vitek, editor, *TOOLS (48)*, volume 6141 of *Lecture Notes in Computer Science*, pages 154–172, Málaga, Spain, June 2010. Springer. ISBN 978-3-642-13952-9.

[23] P. Heidegger and P. Thiemann. A heuristic approach for computing effects. In J. Bishop and A. Vallecillo, editors, *TOOLS (49)*, volume 6705 of *Lecture Notes in Computer Science*, pages 147–162, Zurich, Switzerland, June 2011. Springer. ISBN 978-3-642-21951-1.

[24] P. Heidegger, A. Bieniusa, and P. Thiemann. Access permission contracts for scripting languages. In *POPL*, pages 111–122, Philadelphia, USA, Jan. 2012. ACM Press.

[25] F. Henglein and L. Nielsen. Regular expression containment: coinductive axiomatization and computational interpretation. In POPL 2011 [41], pages 385–398. ISBN 978-1-4503-0490-0.

[26] J. hoon (David) An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for Ruby. In POPL 2011 [41], pages 459–472. ISBN 978-1-4503-0490-0.

[27] E. International. *Standard ECMA-262*, volume 5. 2009.

[28] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium, SAS '09*, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255, Los Angeles, CA, USA, Aug. 2009. Springer-Verlag.

[29] S. Just, A. Cleary, B. Shirley, and C. Hammer. Information flow analysis for javascript. In *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients*, PLASTIC '11, pages 9–18, New York, NY, USA, 2011. ACM.

[30] M. Keil and P. Thiemann. Efficient access analysis using JavaScript proxies. Technical report, Institute for Computer Science, University of Freiburg, 2013.

[31] V. Komendantsky. Regular expression containment as a proof search problem. In *PSATTT'11: International Workshop on Proof-Search in Axiomatic Theories and Type Theories*, Wroclaw, Pologne, 2011. Germain Faure, Stéphane Lengrand, Assia Mahboubi.

[32] A. Krauss and T. Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *J. Autom. Reasoning*, 49(1):95–106, 2012.

[33] 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, Norwell, MA, USA, 1999. Kluwer Academic Publishers.

[34] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, 2005.

[35] H. Lehner. *A Formal Definition of JML in Coq and its Application to Runtime Assertion Checking*. PhD thesis, ETH Zurich, Switzerland, 2011.

[36] H. Lehner and P. Müller. Efficient runtime assertion checking of assignable clauses with datagroups. In D. S. Rosenblum and G. Taentzer, editors, *FASE*, volume 6013 of *Lecture Notes in Computer Science*, pages 338–352, Paphos, Cyprus, 2010. Springer. ISBN 978-3-642-12028-2.

[37] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[38] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Safe active content in sanitized JavaScript. Technical report, Tech. Rep., Google, Inc, 2008.

[39] S. Owens, J. H. Reppy, and A. Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, 2009.

[40] P. H. Phung and L. Desmet. A two-tier sandbox architecture for untrusted JavaScript. In *Proceedings of the Workshop on JavaScript Tools*, JSTools '12, pages 1–10, New York, NY, USA, 2012. ACM.

[41] POPL 2011. *Proceedings 38th Annual ACM Symposium on Principles of Programming Languages*, Austin, TX, USA, Jan. 2011. ACM Press. ISBN 978-1-4503-0490-0.

[42] G. Rosu and M. Viswanathan. Testing extended regular language membership incrementally by rewriting. In R. Nieuwenhuis, editor, *RTA*, volume 2706 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 2003.

[43] T. Van Cutsem and M. S. Miller. On the design of the ECMAScript reflection API. Technical report, Technical Report VUB-SOFT-TR-12-03, Vrije Universiteit Brussel, 2012.

(GET-PROXY-OBSERVER)
$$\frac{\begin{array}{c}\langle \xi', \mathcal{P}, \mathcal{C}\rangle = \mathcal{H}(\xi) \qquad \mathcal{C} \not\vdash_{\mathcal{R}} \mathit{str} \\ \mathcal{H}, \xi' \vdash_{\mathsf{Get}} \mathit{str} \Downarrow \mathcal{H}' \mid c \mid \mathcal{M}\end{array}}{\mathcal{H}, \xi \vdash_{\mathsf{Get}} \mathit{str} \Downarrow \mathcal{H}' \mid c \mid \mathcal{M} \blacktriangleleft_{\mathcal{R}} (\mathcal{P}.\mathit{str}, \mathcal{C})}$$

(GET-MEMBRANE-OBSERVER)
$$\frac{\begin{array}{c}\langle \xi', \mathcal{P}, \mathcal{C}\rangle = \mathcal{H}(\xi) \qquad \mathcal{C} \not\vdash_{\mathcal{R}} \mathit{str} \\ \mathcal{H}, \xi' \vdash_{\mathsf{Get}} \mathit{str} \Downarrow \mathcal{H}' \mid \xi'' \mid \mathcal{M} \\ P = \langle \xi'', \mathcal{P}.\mathit{str}, \partial_{\mathit{str}}(\mathcal{C})\rangle \qquad \xi''' \notin \mathit{dom}(\mathcal{H}')\end{array}}{\mathcal{H}, \xi \vdash_{\mathsf{Get}} \mathit{str} \Downarrow \mathcal{H}'[\xi''' \mapsto P] \mid \xi''' \mid \mathcal{M} \blacktriangleleft_{\mathcal{R}} (\mathcal{P}.\mathit{str}, \mathcal{C})}$$

PUT-PROXY-OBSERVER)
$$\frac{\begin{array}{c}\langle \xi', \mathcal{P}, \mathcal{C}\rangle = \mathcal{H}(\xi) \qquad \mathcal{C} \not\vdash_{\mathcal{W}} \mathit{str} \\ \mathcal{H}, \xi' \vdash_{\mathsf{Put}} \mathit{str}, v \Downarrow \mathcal{H}' \mid v' \mid \mathcal{M}\end{array}}{\mathcal{H}, \xi \vdash_{\mathsf{Put}} \mathit{str}, v \Downarrow \mathcal{H}' \mid v' \mid \mathcal{M} \blacktriangleleft_{\mathcal{W}} (\mathcal{P}.\mathit{str}, \mathcal{C})}$$

**Figure 21.** Inference rules for *Observer Mode*.

(GET-PROXY-PROTECTOR)
$$\frac{\langle \xi', \mathcal{P}, \mathcal{C}\rangle = \mathcal{H}(\xi) \qquad \mathcal{C} \not\vdash_{\mathcal{R}} \mathit{str}}{\mathcal{H}, \xi \vdash_{\mathsf{Get}} \mathit{str} \Downarrow \mathcal{H}' \mid \textbf{undefined} \mid \mathcal{M} \blacktriangleleft_{\mathcal{R}} (\mathcal{P}.\mathit{str}, \mathcal{C})}$$

PUT-PROXY-PROTECTOR)
$$\frac{\langle \xi', \mathcal{P}, \mathcal{C}\rangle = \mathcal{H}(\xi) \qquad \mathcal{C} \not\vdash_{\mathcal{W}} \mathit{str}}{\mathcal{H}, \xi \vdash_{\mathsf{Put}} \mathit{str}, v \Downarrow \mathcal{H}' \mid v \mid \mathcal{M} \blacktriangleleft_{\mathcal{W}} (\mathcal{P}.\mathit{str}, \mathcal{C})}$$

**Figure 22.** Inference rules for *Protector Mode*.

[44] E. Wernli, P. Maerki, and O. Nierstrasz. Ownership, filters and crossing handlers: flexible ownership in dynamic languages. In A. Warth, editor, *DLS*, pages 83–94. ACM, 2012.

## A. Crashing rules

This section presents the formal semantics of path monitoring and contract enforcement in case of a violated contract. The rules extend the set of inference rules of section 6.

The implementation covers two different types of violation treatment. Figure 21 and 22 contain its evaluation rules. The *Observer Mode* performs only path and violation logging without any interruption. $\mathcal{M} \blacktriangleleft_{\mathcal{R}} (\mathcal{P}, \mathcal{C})$ and $\mathcal{M} \blacktriangleleft_{\mathcal{W}} (\mathcal{P}, \mathcal{C})$ extends the monitor to log a violation. The *Protector Mode* returns `undefined` for an access instead of an abnormal termination or omits the operation.

## B. Extended Membrane

Section 7 introduces the necessity of merged proxies to avoid inefficient chains of nested proxy calls.

Figure 23 extends the inference rules from section 6 with path-tries and merged handlers. The single path $\mathcal{P}$ in a handler $H$ gets changed into a trie $\mathcal{T}$. We write $\mathcal{P} \in \mathcal{T}$ if path $\mathcal{P}$ is represented by $\mathcal{T}$. The operator $\oplus$ appends property $p$ to all path-endings in trie $\mathcal{T}$. A trie $\mathcal{T}' = (\mathcal{T} \oplus (p.\mathcal{P}'))$ is equivalent to $\mathcal{T}' = ((\mathcal{T} \oplus p) \oplus \mathcal{P}')$, appending $p.\mathcal{P}'$ to $\mathcal{T}$. $(\mathcal{T} \uplus \mathcal{T}') = (\mathcal{T} \oplus \mathcal{P}) \mid \forall \mathcal{P} \in \mathcal{T}'$ denotes the union of the tries $\mathcal{T}$ and $\mathcal{T}'$.

Further, the definition of monitor $\mathcal{M}$ is extended by $\mathcal{M} \lhd_{\mathcal{R}} \mathcal{T}$, extending monitor $\mathcal{M}$ with all paths $\mathcal{P} \in \mathcal{T}$. The definitions of $\mathcal{M} \lhd_{\mathcal{W}} \mathcal{T}$, $\mathcal{M} \blacktriangleleft_{\mathcal{R}} (\mathcal{T}, \mathcal{C})$, and $\mathcal{M} \blacktriangleleft_{\mathcal{W}} (\mathcal{T}, \mathcal{C})$ and the extensions to the crashing rules (Figure 21 ans 22) are analogous.

(GET-TRIEPROXY)
$$\frac{\begin{array}{c}\langle \xi', \mathcal{T}, \mathcal{C}\rangle = \mathcal{H}(\xi) \qquad \mathcal{C} \vdash_{\mathcal{R}} \mathit{str} \\ \mathcal{H}, \xi' \vdash_{\mathsf{Get}} \mathit{str} \Downarrow \mathcal{H}' \mid c \mid \mathcal{M}\end{array}}{\mathcal{H}, \xi \vdash_{\mathsf{Get}} \mathit{str} \Downarrow \mathcal{H}' \mid c \mid \mathcal{M} \lhd_{\mathcal{R}} (\mathcal{T} \oplus \mathit{str})}$$

(GET-TRIEMEMBRANE-NONEXISTING)
$$\frac{\begin{array}{c}\langle \xi', \mathcal{T}, \mathcal{C}\rangle = \mathcal{H}(\xi) \qquad \mathcal{C} \vdash_{\mathcal{R}} \mathit{str} \\ \mathcal{H}, \xi' \vdash_{\mathsf{Get}} \mathit{str} \Downarrow \mathcal{H}' \mid \xi'' \mid \mathcal{M} \qquad \xi'' = \langle o, f, \pi\rangle \\ P = \langle \xi'', (\mathcal{T} \oplus \mathit{str}), \partial_{\mathit{str}}(\mathcal{C})\rangle \qquad \xi''' \notin \mathit{dom}(\mathcal{H}')\end{array}}{\mathcal{H}, \xi \vdash_{\mathsf{Get}} \mathit{str} \Downarrow \mathcal{H}'[\xi''' \mapsto P] \mid \xi''' \mid \mathcal{M} \lhd_{\mathcal{R}} (\mathcal{T} \oplus \mathit{str})}$$

(GET-TRIEMEMBRANE-EXISTING)
$$\frac{\begin{array}{c}\langle \xi', \mathcal{T}, \mathcal{C}\rangle = \mathcal{H}(\xi) \qquad \mathcal{C} \vdash_{\mathcal{R}} \mathit{str} \\ \mathcal{H}, \xi' \vdash_{\mathsf{Get}} \mathit{str} \Downarrow \mathcal{H}' \mid \xi'' \mid \mathcal{M} \qquad \xi'' = \langle \xi''', \mathcal{T}', \mathcal{C}'\rangle \\ P = \langle \xi''', ((\mathcal{T} \oplus \mathit{str}) \uplus \mathcal{T}'), \partial_{\mathit{str}}(\mathcal{C}) \& \mathcal{C}'\rangle \qquad \xi'''' \notin \mathit{dom}(\mathcal{H}')\end{array}}{\mathcal{H}, \xi \vdash_{\mathsf{Get}} \mathit{str} \Downarrow \mathcal{H}'[\xi'''' \mapsto P] \mid \xi'''' \mid \mathcal{M} \lhd_{\mathcal{R}} (\mathcal{T} \oplus \mathit{str})}$$

PUT-TRIEPROXY)
$$\frac{\begin{array}{c}\langle \xi', \mathcal{T}, \mathcal{C}\rangle = \mathcal{H}(\xi) \qquad \mathcal{C} \vdash_{\mathcal{W}} \mathit{str} \\ \mathcal{H}, \xi' \vdash_{\mathsf{Put}} \mathit{str}, v \Downarrow \mathcal{H}' \mid v' \mid \mathcal{M}\end{array}}{\mathcal{H}, \xi \vdash_{\mathsf{Put}} \mathit{str}, v \Downarrow \mathcal{H}' \mid v' \mid \mathcal{M} \lhd_{\mathcal{W}} (\mathcal{T} \oplus \mathit{str})}$$

**Figure 23.** Inference rules for extended membranes.

| | | | | | |
|---|---|---|---|---|---|
| $\mathsf{bl}(@)$ | $=$ | $\top$ | $\mathsf{bl}(\mathcal{E})$ | $=$ | $\bot$ |
| $\mathsf{bl}(?)$ | $=$ | $\bot$ | $\mathsf{bl}(\mathcal{C}*)$ | $=$ | $\bot$ |
| $\mathsf{bl}(r)$ | $=$ | $\bot$ | $\mathsf{bl}(\mathcal{C}+\mathcal{C}')$ | $=$ | $\mathsf{bl}(\mathcal{C}) \wedge \mathsf{bl}(\mathcal{C}')$ |
| $\mathsf{bl}(!r)$ | $=$ | $\bot$ | $\mathsf{bl}(\mathcal{C}\&\mathcal{C}')$ | $=$ | $\mathsf{bl}(\mathcal{C}) \vee \mathsf{bl}(\mathcal{C}')$ |
| $\mathsf{bl}(\emptyset)$ | $=$ | $\bot$ | $\mathsf{bl}(\mathcal{C}.\mathcal{C}')$ | $=$ | $\mathsf{bl}(\mathcal{C})$ |

**Figure 24.** The bl function.

| | | | | | |
|---|---|---|---|---|---|
| $\mathsf{emp}(@)$ | $=$ | $\bot$ | $\mathsf{emp}(\mathcal{E})$ | $=$ | $\bot$ |
| $\mathsf{emp}(?)$ | $=$ | $\bot$ | $\mathsf{emp}(\mathcal{C}*)$ | $=$ | $\bot$ |
| $\mathsf{emp}(r)$ | $=$ | $\bot$ | $\mathsf{emp}(\mathcal{C}+\mathcal{C}')$ | $=$ | $\mathsf{emp}(\mathcal{C}) \vee \mathsf{emp}(\mathcal{C}')$ |
| $\mathsf{emp}(!r)$ | $=$ | $\bot$ | $\mathsf{emp}(\mathcal{C}\&\mathcal{C}')$ | $=$ | $\mathsf{first}(\mathcal{C}\&\mathcal{C}') = \emptyset$ |
| $\mathsf{emp}(\emptyset)$ | $=$ | $\top$ | $\mathsf{emp}(\mathcal{C}.\mathcal{C}')$ | $=$ | $\mathsf{emp}(\mathcal{C}) \vee \mathsf{emp}(\mathcal{C}')$ |

**Figure 25.** The emp function.

| | | | | | |
|---|---|---|---|---|---|
| $\mathsf{ind}(@)$ | $=$ | $\bot$ | $\mathsf{ind}(\mathcal{C}*)$ | $=$ | $\mathsf{ind}(\mathcal{C})$ |
| $\mathsf{ind}(?)$ | $=$ | $\top$ | $\mathsf{ind}(\mathcal{C}+\mathcal{C}')$ | $=$ | $\mathsf{ind}(\mathcal{C}) \vee \mathsf{ind}(\mathcal{C}')$ |
| $\mathsf{ind}(r)$ | $=$ | $\bot$ | $\mathsf{ind}(\mathcal{C}\&\mathcal{C}')$ | $=$ | $\mathsf{ind}(\mathcal{C}) \wedge \mathsf{ind}(\mathcal{C}')$ |
| $\mathsf{ind}(!r)$ | $=$ | $\bot$ | | | |
| $\mathsf{ind}(\emptyset)$ | $=$ | $\bot$ | $\mathsf{ind}(\mathcal{C}.\mathcal{C}')$ | $=$ | $\begin{cases}\top, & \mathcal{C} = \mathcal{E} \wedge \mathsf{ind}(\mathcal{C}') \\ \top, & \mathsf{ind}(\mathcal{C}) \wedge \mathcal{C}' = \mathcal{E} \\ \bot, & \mathit{otherwise}\end{cases}$ |
| $\mathsf{ind}(\mathcal{E})$ | $=$ | $\bot$ | | | |

**Figure 26.** The ind function.

## C. Auxiliary Functions

This section contains the full definitions of the four auxiliary functions from section 7.2.3.

**Definition 8** (Blank). *A contract* $\mathcal{C}$ *is* blank *if* $\mathcal{L}[\![\mathcal{C}]\!] = \{\iota\}$. *The function bl* : $\mathcal{C} \to \{\top, \bot\}$ *(Figure 24) checks if* $\mathcal{C}$ *is blank.*

**Lemma 8** (Blank). $\mathit{bl}(\mathcal{C}) = \top \Rightarrow \mathcal{L}[\![\mathcal{C}]\!] = \iota$

$$\begin{array}{lcllcl}
\mathsf{unv}(@) & = & \bot & \mathsf{unv}(\mathcal{C}*) & = & \mathsf{unv}(\mathcal{C}) \vee \mathsf{ind}(\mathcal{C}) \\
\mathsf{unv}(?) & = & \bot & \mathsf{unv}(\mathcal{C}+\mathcal{C}') & = & \mathsf{unv}(\mathcal{C}) \vee \mathsf{unv}(\mathcal{C}') \\
\mathsf{unv}(r) & = & \bot & \mathsf{unv}(\mathcal{C}\&\mathcal{C}') & = & \mathsf{unv}(\mathcal{C}) \wedge \mathsf{unv}(\mathcal{C}') \\
\mathsf{unv}(!r) & = & \bot & & & \\
\mathsf{unv}(\emptyset) & = & \bot & & & \\
\mathsf{unv}(\mathcal{E}) & = & \bot & \mathsf{unv}(\mathcal{C}.\mathcal{C}') & = & \begin{cases} \top, & \mathcal{C} = \mathcal{E} \wedge \mathsf{unv}(\mathcal{C}') \\ \top, & \mathsf{unv}(\mathcal{C}) \wedge \mathcal{C}' = \mathcal{E} \\ \top, & \mathsf{unv}(\mathcal{C}) \wedge \mathsf{unv}(\mathcal{C}') \\ \bot, & otherwise \end{cases}
\end{array}$$

**Figure 27.** The unv function.

**Definition 9** (Empty). *A contract $\mathcal{C}$ is* empty *if $\mathcal{L}[\![\mathcal{C}]\!] = \emptyset$. The function $emp : \mathcal{C} \to \{\top, \bot\}$ (Figure 25) checks if $\mathcal{C}$ is empty.*

**Lemma 9** (Empty). $emp(\mathcal{C}) = \top \Rightarrow \mathcal{L}[\![\mathcal{C}]\!] = \emptyset$

**Definition 10** (Indifferent). *A contract $\mathcal{C}$ is* indifferent *if $\mathcal{L}[\![\mathcal{C}]\!] = \mathcal{A}$. The function $ind : \mathcal{C} \to \{\top, \bot\}$ (Figure 26) checks if $\mathcal{C}$ is indifferent.*

**Lemma 10** (Indifferent). $ind(\mathcal{C}) = \top \Rightarrow \mathcal{L}[\![\mathcal{C}]\!] = \mathcal{A}$

**Definition 11** (Universal). *A contract $\mathcal{C}$ is* universal *if $\mathcal{L}[\![\mathcal{C}]\!] = \mathcal{A}^*$. The function $unv : \mathcal{C} \to \{\top, \bot\}$ (Figure 27) checks if $\mathcal{C}$ is universal.*

**Lemma 11** (Universal). $unv(\mathcal{C}) = \top \Rightarrow \mathcal{L}[\![\mathcal{C}]\!] = \mathcal{A}^*$

## D. Semantic containment

*Proof of Lemma 3.* A contract $\mathcal{C}$ is subset of another contract $\mathcal{C}'$ iff for all paths $\mathcal{P} \in \mathcal{L}[\![\mathcal{C}]\!]$ the derivation of $\mathcal{C}'$ w.r.t. path $\mathcal{P}$ is nullable. For all $\mathcal{P} \in \mathcal{A}^*$ it holds that $\mathcal{P} \in \mathcal{L}[\![\mathcal{C}']\!]$ iff $\nu(\partial_{\mathcal{P}}(\mathcal{C}'))$. It is trivial to see that

$$\begin{aligned}
& \mathcal{C} \sqsubseteq \mathcal{C}' && (16)\\
\Leftrightarrow\ & \mathcal{L}[\![\mathcal{C}]\!] \subseteq \mathcal{L}[\![\mathcal{C}']\!] && (17)\\
\Leftrightarrow\ & \forall \mathcal{P} \in \mathcal{L}[\![\mathcal{C}]\!] : \mathcal{P} \in \mathcal{L}[\![\mathcal{C}]\!] && (18)\\
\Leftrightarrow\ & \forall \mathcal{P} \in \mathcal{L}[\![\mathcal{C}]\!] : \nu(\partial_{\mathcal{P}}(\mathcal{C})) && (19)
\end{aligned}$$

holds. □

*Proof of Lemma 4.* A contract $\mathcal{C}$ is subset of another contract $\mathcal{C}'$ iff for all properties $p$ in $\mathsf{next}(\mathcal{C})$ the derivation of $\mathcal{C}$ w.r.t. property $p$ is subset of the derivation of $\mathcal{C}'$ w.r.t. $p$. By lemma 2 we obtain

$$\mathcal{L}[\![\partial_p(\mathcal{C})]\!] = p^{-1}\mathcal{L}[\![\mathcal{C}]\!] \qquad (20)$$

and this leads to

$$\{\mathcal{E} \mid \nu(\mathcal{C})\} \cup \{p.\mathcal{P} \mid p \in \mathsf{next}(\mathcal{C}), \mathcal{P} \in p^{-1}\mathcal{L}[\![\mathcal{C}]\!]\} = \mathcal{L}[\![\mathcal{C}]\!] \qquad (21)$$

Claim holds because

$$\begin{aligned}
& \mathcal{C} \sqsubseteq \mathcal{C}' && (22)\\
\Leftrightarrow\ & \mathcal{L}[\![\mathcal{C}]\!] \subseteq \mathcal{L}[\![\mathcal{C}']\!] && (23)\\
\Leftrightarrow\ & \forall \mathcal{P} \in \mathcal{L}[\![\mathcal{C}]\!] :\ \mathcal{P} \in \mathcal{L}[\![\mathcal{C}']\!] && (24)\\
\Leftrightarrow\ & \mathcal{E} \in \mathcal{L}[\![\mathcal{C}]\!] \Rightarrow \mathcal{E} \in \mathcal{L}[\![\mathcal{C}']\!] \wedge && (25)\\
& \forall p, \mathcal{P} :\ p.\mathcal{P} \in \mathcal{L}[\![\mathcal{C}]\!] \Rightarrow \nu(\partial_{p.\mathcal{P}}(\mathcal{C}')) && (26)\\
\Leftrightarrow\ & \nu(\mathcal{C}) \Rightarrow \nu(\mathcal{C}') \wedge && (27)\\
& \forall p \in \mathsf{next}(\mathcal{C}), \forall \mathcal{P} :\ p.\mathcal{P} \in \mathcal{L}[\![\mathcal{C}]\!] \Rightarrow \nu(\partial_{\mathcal{P}}(\partial_p(\mathcal{C}'))) && (28)\\
\Leftrightarrow\ & \nu(\mathcal{C}) \Rightarrow \nu(\mathcal{C}') \wedge && (29)\\
& \forall p \in \mathsf{next}(\mathcal{C}), \forall \mathcal{P} \in \mathcal{L}[\![\partial_p(\mathcal{C})]\!] :\ \nu(\partial_{\mathcal{P}}(\partial_p(\mathcal{C}'))) && (30)\\
\Leftrightarrow\ & \nu(\mathcal{C}) \Rightarrow \nu(\mathcal{C}') \wedge && (31)\\
& \forall p \in \mathsf{next}(\mathcal{C}) :\ \mathcal{L}[\![\partial_p(\mathcal{C})]\!] \subseteq \mathcal{L}[\![\partial_p(\mathcal{C}')]\!] && (32)\\
\Leftrightarrow\ & \nu(\mathcal{C}) \Rightarrow \nu(\mathcal{C}') \wedge && (33)\\
& \forall p \in \mathsf{next}(\mathcal{C}) :\ \partial_p(\mathcal{C}) \sqsubseteq \partial_p(\mathcal{C}') && (34)
\end{aligned}$$

□

## E. Syntactic derivative

*Proof of Lemma 5.* $\forall \ell$ :

$$\mathcal{L}[\![\nabla_\ell(\mathcal{C})]\!] \subseteq \bigcap_{p \in \mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C})]\!] \qquad (35)$$

Proof by induction on $\mathcal{C}$.

**Case** $\mathcal{C} = \emptyset$: Claim holds because $\mathcal{L}[\![\nabla_\ell(\emptyset)]\!] = \mathcal{L}[\![\partial_\ell(\emptyset)]\!] = \emptyset$.
**Case** $\mathcal{C} = \mathcal{E}$: Claim holds because $\mathcal{L}[\![\nabla_\ell(\mathcal{E})]\!] = \mathcal{L}[\![\partial_\ell(\mathcal{E})]\!] = \emptyset$.
**Case** $\mathcal{C} = r$:
**Subcase** $\ell \sqsubseteq_r r$: Claim holds because $\mathcal{L}[\![\nabla_\ell(r)]\!] = \mathcal{L}[\![\partial_p(r)]\!] = \mathcal{E} \mid \forall p \in \mathcal{L}[\![\ell]\!]$.
**Subcase** $\ell \not\sqsubseteq_r r$: Claim holds because $\mathcal{L}[\![\nabla_p(r)]\!] = \emptyset$.
**Case** $\mathcal{C} = \mathcal{C}'*$: By induction

$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!] \overset{\mathsf{IH}}{\subseteq} \bigcap_{p \in \mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}')]\!] \qquad (36)$$

holds.We obtain that

$$\begin{aligned}
\forall p :\ \mathcal{L}[\![\partial_p(\mathcal{C}'*)]\!] &= \mathcal{L}[\![\partial_p(\mathcal{C}').\mathcal{C}'*]\!] && (37)\\
\forall \ell :\ \mathcal{L}[\![\nabla_\ell(\mathcal{C}'*)]\!] &= \mathcal{L}[\![\nabla_\ell(\mathcal{C}').\mathcal{C}'*]\!] && (38)
\end{aligned}$$

holds. Claim holds because

$$\begin{aligned}
& \forall \ell :\ \mathcal{L}[\![\nabla_\ell(\mathcal{C}'*)]\!] && (39)\\
&= \mathcal{L}[\![\nabla_\ell(\mathcal{C}').\mathcal{C}'*]\!] && (40)\\
&\overset{\mathsf{IH}}{\subseteq} \{\mathcal{P}.\mathcal{P}' \mid \mathcal{P} \in \bigcap_{p \in \mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}')]\!], \mathcal{P}' \in \mathcal{L}[\![\mathcal{C}'*]\!]\} && (41)\\
&= \bigcap_{p \in \mathcal{L}[\![\ell]\!]} \{\mathcal{P}.\mathcal{P}' \mid \mathcal{P} \in \mathcal{L}[\![\partial_p(\mathcal{C}')]\!], \mathcal{P}' \in \mathcal{L}[\![\mathcal{C}'*]\!]\} && (42)\\
&= \bigcap_{p \in \mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}'*)]\!] && (43)
\end{aligned}$$

**Case** $\mathcal{C} = \mathcal{C}'+\mathcal{C}''$: By induction

$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!] \overset{\mathsf{IH}}{\subseteq} \bigcap_{p \in \mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}')]\!] \qquad (44)$$

$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!] \overset{\mathsf{IH}}{\subseteq} \bigcap_{p \in \mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \qquad (45)$$

holds. We obtain that

$$\mathcal{L}[\![\partial_\ell(\mathcal{C}'+\mathcal{C}'')]\!] \;=\; \mathcal{L}[\![\partial_\ell(\mathcal{C}')]\!] \cup \mathcal{L}[\![\partial_\ell(\mathcal{C}'')]\!] \tag{46}$$
$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}'+\mathcal{C}'')]\!] \;=\; \mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!] \cup \mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!] \tag{47}$$

holds. Claim holds because

$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}'+\mathcal{C}'')]\!] \tag{48}$$
$$= \mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!] \cup \mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!] \tag{49}$$
$$\overset{\mathsf{IH}}{\subseteq} \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}')]\!] \cup \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{50}$$
$$\subseteq \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}')]\!] \cup \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{51}$$
$$= \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}'+\mathcal{C}'')]\!] \tag{52}$$

**Case** $\mathcal{C} = \mathcal{C}'\&\mathcal{C}''$: By induction

$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!] \;\overset{\mathsf{IH}}{\subseteq}\; \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}')]\!] \tag{53}$$
$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!] \;\overset{\mathsf{IH}}{\subseteq}\; \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{54}$$

holds. We obtain that

$$\mathcal{L}[\![\partial_\ell(\mathcal{C}'\&\mathcal{C}'')]\!] \;=\; \mathcal{L}[\![\partial_\ell(\mathcal{C}')]\!] \cap \mathcal{L}[\![\partial_\ell(\mathcal{C}'')]\!] \tag{55}$$
$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}'\&\mathcal{C}'')]\!] \;=\; \mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!] \cap \mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!] \tag{56}$$

holds. Claim holds because

$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}'\&\mathcal{C}'')]\!] \tag{57}$$
$$= \mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!] \cap \mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!] \tag{58}$$
$$\overset{\mathsf{IH}}{\subseteq} \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}')]\!] \cap \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{59}$$
$$= \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}')]\!] \cap \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{60}$$
$$= \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}'\&\mathcal{C}'')]\!] \tag{61}$$

**Case** $\mathcal{C} = \mathcal{C}'.\mathcal{C}''$: By induction

$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!] \;\overset{\mathsf{IH}}{\subseteq}\; \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}')]\!] \tag{62}$$
$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!] \;\overset{\mathsf{IH}}{\subseteq}\; \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{63}$$

holds.
**Subcase** $\nu(\mathcal{C})$: We obtain that

$$\forall p:\; \mathcal{L}[\![\partial_\ell(\mathcal{C}'.\mathcal{C}'')]\!] \;=\; \mathcal{L}[\![\partial_p(\mathcal{C}').\mathcal{C}'']\!] \cup \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{64}$$
$$\forall \ell:\; \mathcal{L}[\![\nabla_\ell(\mathcal{C}'.\mathcal{C}'')]\!] \;=\; \mathcal{L}[\![\nabla_\ell(\mathcal{C}').\mathcal{C}'']\!] \cup \mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!] \tag{65}$$

holds. Claim holds because

$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}'.\mathcal{C}'')]\!] \tag{66}$$
$$= \mathcal{L}[\![\nabla_\ell(\mathcal{C}').\mathcal{C}'']\!] \cup \mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!] \tag{67}$$
$$\overset{\mathsf{IH}}{\subseteq} \{\mathcal{P}.\mathcal{P}' \mid \mathcal{P} \in \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}')]\!], \mathcal{P}' \in \mathcal{L}[\![\mathcal{C}'']\!]\} \tag{68}$$
$$\cup \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{69}$$
$$\subseteq \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \{\mathcal{P}.\mathcal{P}' \mid \mathcal{L}[\![\partial_p(\mathcal{C}')]\!], \mathcal{P}' \in \mathcal{L}[\![\mathcal{C}'']\!]\} \tag{70}$$
$$\cup \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{71}$$
$$= \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}').\mathcal{C}'']\!] \cup \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{72}$$
$$= \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \partial_p(\mathcal{C}'.\mathcal{C}'') \tag{73}$$

**Subcase** $\neg\nu(\mathcal{C})$: We obtain that

$$\forall p:\; \mathcal{L}[\![\partial_\ell(\mathcal{C}'.\mathcal{C}'')]\!] \;=\; \mathcal{L}[\![\partial_p(\mathcal{C}').\mathcal{C}'']\!] \tag{74}$$
$$\forall \ell:\; \mathcal{L}[\![\nabla_\ell(\mathcal{C}'.\mathcal{C}'')]\!] \;=\; \mathcal{L}[\![\nabla_\ell(\mathcal{C}').\mathcal{C}'']\!] \tag{75}$$

holds. Claim holds because

$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}'.\mathcal{C}'')]\!] \tag{76}$$
$$= \mathcal{L}[\![\nabla_\ell(\mathcal{C}').\mathcal{C}'']\!] \tag{77}$$
$$\overset{\mathsf{IH}}{\subseteq} \{\mathcal{P}.\mathcal{P}' \mid \mathcal{P} \in \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}')]\!], \mathcal{P}' \in \mathcal{L}[\![\mathcal{C}'']\!]\} \tag{78}$$
$$= \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \{\mathcal{P}.\mathcal{P}' \mid \mathcal{P} \in \mathcal{L}[\![\partial_p(\mathcal{C}')]\!], \mathcal{P}' \in \mathcal{L}[\![\mathcal{C}'']\!]\} \tag{79}$$
$$= \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}').\mathcal{C}'']\!] \tag{80}$$
$$= \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \partial_p(\mathcal{C}'.\mathcal{C}'') \tag{81}$$

□

*Proof of Lemma 7.* $\forall \ell \in \mathsf{first}(\mathcal{C})$ :

$$\mathcal{L}[\![\nabla_\ell(\mathcal{C})]\!] \;=\; \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C})]\!] \tag{82}$$

Suppose that $\forall p, p' \in \mathcal{L}[\![\ell]\!] :\; \partial_p(\ell.\mathcal{C}) = \partial_{p'}(\ell.\mathcal{C})$.

Proof by induction on $\mathcal{C}$. The cases for $\emptyset$, $\mathcal{E}$, $r$, $\mathcal{C}\&\mathcal{C}'$ are analogous to the cases in the proof of lemma 5. All occurences of $\overset{\mathsf{IH}}{\subseteq}$ can be replaced by $\overset{\mathsf{IH}}{=}$.

**Case** $\mathcal{C} = \mathcal{C}'+\mathcal{C}''$: By induction

$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!] \;\overset{\mathsf{IH}}{=}\; \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}')]\!] \tag{83}$$
$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!] \;\overset{\mathsf{IH}}{=}\; \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{84}$$

holds. We obtain that

$$\mathcal{L}[\![\partial_\ell(\mathcal{C}'+\mathcal{C}'')]\!] \;=\; \mathcal{L}[\![\partial_\ell(\mathcal{C}')]\!] \cup \mathcal{L}[\![\partial_\ell(\mathcal{C}'')]\!] \tag{85}$$
$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}'+\mathcal{C}'')]\!] \;=\; \mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!] \cup \mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!] \tag{86}$$

holds. Claim holds because

$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}'+\mathcal{C}'')]\!] \tag{87}$$
$$= \mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!] \cup \mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!] \tag{88}$$
$$\overset{\mathsf{IH}}{=} \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}')]\!] \cup \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{89}$$
$$= \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}')]\!] \cup \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{90}$$
$$= \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}'+\mathcal{C}'')]\!] \tag{91}$$

**Case** $\mathcal{C} = \mathcal{C}'.\mathcal{C}''$: By induction

$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!] \overset{\mathsf{IH}}{=} \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}')]\!] \tag{92}$$
$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!] \overset{\mathsf{IH}}{=} \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{93}$$

holds.

**Subcase** $\nu(\mathcal{C})$: We obtain that

$$\forall p:\ \mathcal{L}[\![\partial_\ell(\mathcal{C}'.\mathcal{C}'')]\!] \;=\; \mathcal{L}[\![\partial_p(\mathcal{C}').\mathcal{C}'']\!] \cup \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{94}$$
$$\forall \ell:\ \mathcal{L}[\![\nabla_\ell(\mathcal{C}'.\mathcal{C}'')]\!] \;=\; \mathcal{L}[\![\nabla_\ell(\mathcal{C}').\mathcal{C}'']\!] \cup \mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!] \tag{95}$$

holds. Claim holds because

$$\mathcal{L}[\![\nabla_\ell(\mathcal{C}'.\mathcal{C}'')]\!] \tag{96}$$
$$= \mathcal{L}[\![\nabla_\ell(\mathcal{C}').\mathcal{C}'']\!] \cup \mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!] \tag{97}$$
$$\overset{\mathsf{IH}}{=} \{\mathcal{P}.\mathcal{P} \mid \mathcal{P} \in \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}')]\!], \mathcal{P} \in \mathcal{L}[\![\mathcal{C}'']\!]\} \tag{98}$$
$$\cup \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{99}$$
$$= \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \{\mathcal{P}.\mathcal{P} \mid \mathcal{L}[\![\partial_p(\mathcal{C}')]\!], \mathcal{P} \in \mathcal{L}[\![\mathcal{C}'']\!]\} \tag{100}$$
$$\cup\, \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{101}$$
$$= \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \mathcal{L}[\![\partial_p(\mathcal{C}').\mathcal{C}'']\!] \cup \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \tag{102}$$
$$= \bigcap_{p\in\mathcal{L}[\![\ell]\!]} \partial_p(\mathcal{C}'.\mathcal{C}'') \tag{103}$$

**Subcase** $\neg\nu(\mathcal{C})$: Analogus to the case in the proof of lemma 5.

□

## F. Syntactic containment

Before proving the syntactic containment we state an auxiliary lemma. For simplification, the literals @, ?, and $!r$ are collapsed into a single regular expression literals $r$.

**Lemma 12** (Path-preservation)**.** *$\forall p, \mathcal{P}$:*

$$\mathcal{P} \in \mathcal{L}[\![\partial_p(\mathcal{C})]\!] \;\Rightarrow\; \exists \ell \in \mathit{first}(\mathcal{C}):\ \mathcal{P} \in \mathcal{L}[\![\nabla_\ell(\mathcal{C})]\!] \tag{104}$$

*Proof of Lemma 12.* Suppose $\mathcal{L}[\![\partial_p(\mathcal{C})]\!] \neq \emptyset$. Show $\exists \ell \in \mathsf{first}(\mathcal{C}):\ \mathcal{P} \in \mathcal{L}[\![\nabla_\ell(\mathcal{C})]\!]$. Proof by induction on $\mathcal{C}$.

**Case** $\mathcal{C} = \emptyset$, $\mathsf{first}(\mathcal{C}) = \{\emptyset\}$: Contradicts assumption.
**Case** $\mathcal{C} = \mathcal{E}$, $\mathsf{first}(\mathcal{C}) = \{\}$: Contradicts assumption.
**Case** $\mathcal{C} = r$, $\mathsf{first}(\mathcal{C}) = \{r\}$:
We obtain that $p \in \mathcal{L}[\![r]\!] \;\Rightarrow\; \partial_p(r) = \mathcal{E}$. Claim holds because $\mathsf{first}(\mathcal{C}) = \{r\}$, $\nabla_r(r) = \mathcal{E}$, and thus $\mathcal{P} = \epsilon$ and $\epsilon \in \mathcal{L}[\![\mathcal{E}]\!]$.

**Case** $\mathcal{C} = \mathcal{C}'*$, $\mathsf{first}(\mathcal{C}) = \mathsf{first}(\mathcal{C}')$:
We obtain that $\mathcal{P} \in \mathcal{L}[\![\partial_p(\mathcal{C}'*)]\!] = \mathcal{L}[\![\partial_p(\mathcal{C}').\mathcal{C}'*]\!] \neq \emptyset$. By induction $\exists \ell' \in \mathsf{first}(\mathcal{C}'):\ \mathcal{P}' \in \mathcal{L}[\![\nabla_{\ell'}(\mathcal{C}')]\!]$. The chain holds because $\mathsf{first}(\mathcal{C}'*) = \mathsf{first}(\mathcal{C}')$ and $\nabla_\ell(\mathcal{C}'*) = \nabla_\ell(\mathcal{C}').\mathcal{C}'*$ and $\mathcal{P}' \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!], \mathcal{P}'' \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}'*)]\!]$ implies $\mathcal{P} = \mathcal{P}'.\mathcal{P}'' \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}'*)]\!]$.
**Case** $\mathcal{C} = (\mathcal{C}'+\mathcal{C}'')$, $\mathsf{first}(\mathcal{C}) = \mathsf{first}(\mathcal{C}') \cup \mathsf{first}(\mathcal{C}'')$:
We obtain that $\mathcal{P} \in \mathcal{L}[\![\partial_p(\mathcal{C}'+\mathcal{C}'')]\!] = \mathcal{L}[\![\partial_p(\mathcal{C}')]\!] \cup \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!] \neq \emptyset$. By induction $\exists \ell' \in \mathsf{first}(\mathcal{C}'):\ \mathcal{P}' \in \mathcal{L}[\![\nabla_{\ell'}(\mathcal{C}')]\!]$ and $\exists \ell'' \in \mathsf{first}(\mathcal{C}''):\ \mathcal{P}'' \in \mathcal{L}[\![\nabla_{\ell''}(\mathcal{C}'')]\!]$. The chain holds because $\mathsf{first}(\mathcal{C}'+\mathcal{C}'') = \mathsf{first}(\mathcal{C}') \cup \mathsf{first}(\mathcal{C}'')$ and $\nabla_\ell(\mathcal{C}'+\mathcal{C}'') = \nabla_\ell(\mathcal{C}')+\nabla_\ell(\mathcal{C}'')$ and $\mathcal{P} \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!]$ or $\mathcal{P} \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!]$ implies $\mathcal{P} \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}'+\mathcal{C}'')]\!]$.
**Case** $\mathcal{C} = (\mathcal{C}'\&\mathcal{C}'')$, $\mathsf{first}(\mathcal{C}) = \{\ell' \sqcap_r \ell'' \mid \ell' \in \mathsf{first}(\mathcal{C}'), \ell'' \in \mathsf{first}(\mathcal{C}'')\}$:
We obtain that $\mathcal{P} \in \mathcal{L}[\![\partial_p(\mathcal{C}'\&\mathcal{C}'')]\!] = \mathcal{L}[\![\partial_p(\mathcal{C}')]\!] \cap \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!]$ implies $\mathcal{P} \in \mathcal{L}[\![\partial_p(\mathcal{C}')]\!]$ and $\mathcal{P} \in \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!]$. By induction $\exists \ell' \in \mathsf{first}(\mathcal{C}'):\ \mathcal{P} \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!]$ and $\exists \ell'' \in \mathsf{first}(\mathcal{C}''):\ \mathcal{P} \in \mathcal{L}[\![\nabla_{\ell''}(\mathcal{C}'')]\!]$. Let $\ell = \ell' \sqcap_r \ell'' \in \mathsf{first}(\mathcal{C}'\&\mathcal{C}'')$. If $p \in \mathcal{L}[\![\ell']\!]$ and $p \in \mathcal{L}[\![\ell']\!]$ then $p \in \mathcal{L}[\![\ell]\!]$. The chain holds because $\mathsf{first}(\mathcal{C}'\&\mathcal{C}'') = \{\ell' \sqcap_r \ell'' \mid \ell' \in \mathsf{first}(\mathcal{C}'), \ell'' \in \mathsf{first}(\mathcal{C}'')\}$ and $\nabla_\ell(\mathcal{C}'\&\mathcal{C}'') = \nabla_\ell(\mathcal{C}')\&\nabla_\ell(\mathcal{C}'')$, and $\mathcal{P} \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!]$ and $\mathcal{P} \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!]$ implies $\mathcal{P} \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}'\&\mathcal{C}'')]\!]$.
**Case** $\mathcal{C} = (\mathcal{C}'.\mathcal{C}'')$:
**Subcase** $\nu(\mathcal{C}')$, $\mathsf{first}(\mathcal{C}) = \mathsf{first}(\mathcal{C}') \cup \mathsf{first}(\mathcal{C}'')$:
We obtain that $\mathcal{P} \in \mathcal{L}[\![\partial_p(\mathcal{C}'.\mathcal{C}'')]\!] = \mathcal{L}[\![\partial_p(\mathcal{C}').\mathcal{C}'']\!] \cup \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!]$ implies $\mathcal{P} \in \mathcal{L}[\![\partial_p(\mathcal{C}').\mathcal{C}'']\!]$ or $\mathcal{P} \in \mathcal{L}[\![\partial_p(\mathcal{C}'')]\!]$. By induction $\exists \ell' \in \mathsf{first}(\mathcal{C}'):\ \mathcal{P}' \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!]$ and $\exists \ell'' \in \mathsf{first}(\mathcal{C}''):\ \mathcal{P}'' \in \mathcal{L}[\![\nabla_{\ell''}(\mathcal{C}'')]\!]$. The chain holds because $\mathsf{first}(\mathcal{C}'.\mathcal{C}'') = \mathsf{first}(\mathcal{C}') \cup \mathsf{first}(\mathcal{C}'')$ and $\nabla_\ell(\mathcal{C}'.\mathcal{C}'') = (\nabla_\ell(\mathcal{C}').\mathcal{C}'')+\nabla_\ell(\mathcal{C}'')$, and $\mathcal{P}' \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!]$ and $\mathcal{P}'' \in \mathcal{L}[\![\mathcal{C}'']\!]$ implies $\mathcal{P} = \mathcal{P}'.\mathcal{P}'' \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}'.\mathcal{C}'')]\!]$ or $\mathcal{P} = \epsilon.\mathcal{P}'' \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}'.\mathcal{C}'')]\!]$.
**Subcase** $\neg\nu(\mathcal{C}')$, $\mathsf{first}(\mathcal{C}) = \mathsf{first}(\mathcal{C}')$:
We obtain that $\mathcal{P} \in \mathcal{L}[\![\partial_p(\mathcal{C}'.\mathcal{C}'')]\!] = \mathcal{L}[\![\partial_p(\mathcal{C}').\mathcal{C}'']\!]$ implies $\mathcal{P} \in \mathcal{L}[\![\partial_p(\mathcal{C}').\mathcal{C}'']\!]$. By induction $\exists \ell' \in \mathsf{first}(\mathcal{C}'):\ \mathcal{P}' \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!]$. The chain holds because $\mathsf{first}(\mathcal{C}'.\mathcal{C}'') = \mathsf{first}(\mathcal{C}')$ and $\nabla_\ell(\mathcal{C}'.\mathcal{C}'') = \nabla_\ell(\mathcal{C}').\mathcal{C}''$, and $\mathcal{P}' \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!]$ and $\mathcal{P}'' \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}'')]\!]$ implies $\mathcal{P} = \mathcal{P}'.\mathcal{P}'' \in \mathcal{L}[\![\nabla_\ell(\mathcal{C}'.\mathcal{C}'')]\!]$.

□

*Proof of Theorem 1.* The proof is by contraposition. If $\mathcal{C} \not\sqsubseteq \mathcal{C}'$ then $\exists \ell \in \mathsf{first}(\mathcal{C}):\ \nabla_\ell(\mathcal{C}) \not\sqsubseteq \nabla_\ell(\mathcal{C}')$ or $\neg(\nu(\mathcal{C}) \Rightarrow \nu(\mathcal{C}'))$.
We obtain that:

$$\mathcal{C} \not\sqsubseteq \mathcal{C}' \;\Leftrightarrow\; \mathcal{L}[\![\mathcal{C}]\!] \not\subseteq \mathcal{L}[\![\mathcal{C}']\!] \tag{105}$$
$$\Leftrightarrow\; \exists \mathcal{P} \in \mathcal{L}[\![\mathcal{C}]\!] \backslash \mathcal{L}[\![\mathcal{C}']\!] \tag{106}$$

**Case** $\mathcal{P} = \epsilon$:
Claim holds because $\neg(\nu(\mathcal{C}) \Rightarrow \nu(\mathcal{C}'))$.
**Case** $\mathcal{P} \neq \epsilon$:
It must be that $\mathcal{P} = p.\mathcal{P}'$ with $p \in \mathsf{next}(\mathcal{C}) = \mathcal{L}[\![\mathsf{first}(\mathcal{C})]\!]$. Therefore $\exists \ell \in \mathsf{first}(\mathcal{C}):\ p \in \mathcal{L}[\![\ell]\!]$.
**Subcase** $p \notin \mathsf{next}(\mathcal{C}')$:
Claim holds by Lemma 5 and 7 because $\exists \ell \in \mathsf{first}(\mathcal{C}):\ \nabla_\ell(\mathcal{C}) \neq \emptyset$ and $\nabla_\ell(\mathcal{C}') = \emptyset$ implies that $\nabla_\ell(\mathcal{C}) \not\sqsubseteq \nabla_\ell(\mathcal{C}')$.
**Subcase** $p \in \mathsf{next}(\mathcal{C}')$:
By Lemma 5 and 7 claim holds because
$\mathcal{P}' \in \mathcal{L}[\![\partial_p(\mathcal{C})]\!] \backslash \mathcal{L}[\![\partial_p(\mathcal{C}')]\!]$ implies that
$\mathcal{P}' \in \mathcal{L}[\![\nabla_\ell(\mathcal{C})]\!] \backslash \mathcal{L}[\![\nabla_\ell(\mathcal{C}')]\!]$

□

## G. Correctness

*Proof of Theorem 2.* If $\Gamma \vdash \mathcal{C} \sqsubseteq \mathcal{C}' : \top$ than $\mathcal{C} \sqsubseteq \mathcal{C}'$ Proof is by induction in the derivation of $\Gamma \vdash \phi : \{\top, \bot\}$

**Case** (C-DELETE):
    Obtaining the rule (C-UNFOLD-TRUE) and (C-UNFOLD-FALSE) the result of $\phi = \mathcal{C} \sqsubseteq \mathcal{C}'$ is the conjunction of its derivative w.r.t. the first literals. If $\phi \in \Gamma$ than $\phi$ is already part of the conjunction.

**Case** (C-DISPROVE), (C-UNFOLD-TRUE); (C-UNFOLD-FALSE):
    Claim holds by theorem 1

□