A Heuristic Approach for Computing Effects

Phillip Heidegger and Peter Thiemann

University of Freiburg, Germany

Abstract. The effect of an operation on an object network can be described by the access paths along which the function reads or writes object properties. Abstracted to access path permissions, the effect can serve as part of the operation's documentation, augmenting a type signature or a contract for the operation. Statically determining such an effect is a challenging problem, in particular in a dynamic language which represents objects by hash tables like the current breed of scripting languages.
In this work, we propose an analysis that computes access permissions describing the effect of an operation from a set of access paths obtained by running the program. The main ingredient of the analysis is a novel heuristic to abstract a set of access paths to a concise access permission. Our analysis is implemented as part of JSConTest, a testing framework for JavaScript. It has been applied to a range of examples with encouraging results.

1 Introduction

For a program in an untyped scripting language like JavaScript, maintenance and understanding can be a nightmare. Given a function or method, it is often not clear which types of arguments are required to make the function work as expected and which types of values are returned. A first step towards understanding such an operation is thus to find a type signature for it.

However, a type signature only describes the functional behavior of an operation, but its side effects are equally important. In most object-oriented languages side effects are limited thanks to data encapsulation. The situation is different in a scripting language like JavaScript: Objects lack any kind of encapsulation, so that an operation can arbitrarily explore and modify the object graph starting from any object in scope.

The goal of this work is thus to provide a concise description of the way that an operation accesses and modifies the object graph. This information can be vital for program understanding and program maintenance.

Our approach is to describe the effect of an operation on the object graph by the set of access paths along which the function and its callees read or write object properties. These paths can start from any object accessible to the operation, that is, it either has to be passed as an argument or it must be bound to a global variable. Reads and writes in objects that are created within the operation do not matter for the effect as they are not observable from the outside.

As the set of access paths is potentially infinite, it cannot usefully serve as a high-level description of an operation's effect. Instead, we approximate sets of access paths by concise access path permissions. Such a permission can be attached to any variable in scope and can thus become part of the operation's documentation in addition to a type signature or a contract. Permissions are easy to understand because they are structured like file paths with wildcards.

In a statically typed language, it would be feasible to compute the effect of an operation statically. In a scripting language with dynamic types and where objects also serve as hash tables and arrays, computing an access permission statically would be much harder, if possible at all, because the description of a permission may depend on particular values like strings and indexes. A manual effect annotation is, of course, possible, but too time consuming.

The main contribution of this work is a heuristic analysis that learns access path permissions from access paths sampled from running JavaScript programs. This information can be used to enhance type-signature-based contracts as proposed in our previous work [12]. Because a static analysis of the effects is not feasible, we perform a dynamic analysis which collects access paths during runs of the program. The heuristic extracts concise access path permissions from the collected path sets. The extraction procedure is user configurable so that the results can be refined interactively.

Our analysis is implemented and available as part of JSConTest,[1] a testing framework for JavaScript. It has been applied to a range of examples with encouraging results.

2 Testing Effects

Previous work of the authors [12] proposes a contract framework for JavaScript. It permits the specification of contracts which are similar to type signatures and provides the facilities to perform contract monitoring as well as contract-based testing. This contract system is value-oriented in the sense that a contract specifies restrictions on the values that are passed to a method and returned from it. However, a value-oriented contract misses an important facet of the semantics of a method because a type signature does not specify its side effects.

Subsequent work [11] extends the contract language with access permissions that restrict the side effects that a method is allowed to perform. An access permission explicitly states the set of paths (sequences of property accesses) that a method may access from the objects in scope. Being able to state such permissions is important in a language like JavaScript, where a side effect is often the raison d'être of an operation. For such an operation, a value-oriented contract is insufficient as the following example code shows:

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

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

The type signature /∗c (string) → undefined ∗/ fully describes the functional behavior of redirectTo: its argument should be a string and it returns the value **undefined** as there is no explicit return statement. However, the interesting information about the function is that it changes the location property of the window object, which has the further effect of redirecting the web browser to a new page. To specify this effect, our extended contract language enables us to add an *access permission* to the above contract:

... **with** [window.location]

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

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

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

The paper further reports two case studies to validate the significance of access permission contracts. The results demonstrate that contracts with effects can detect 6-13% more programming errors than contracts without effects.

While these results are encouraging, their preparation is tedious. Functional contracts are mostly straightforward to write and can be finalized in a few iterations of testing with the framework, but careful manual scrutiny is required to come up with concise and useful effect annotations. The main problem is the dynamic nature of JavaScript, which permits non-obvious control flows (e.g., callback functions or method invocation through several levels of prototypes) as well as non-obvious data accesses when object properties are addressed using the array notation as in obj[prop]. Furthermore, from an interprocedural perspective, it is not straightforwardly possible to compose effect annotations of callees to the effect annotation of the caller.

For these reasons, we propose to record all access paths by running test cases on the program after constructing the type-signature contracts. These access paths are generated by our framework by setting all effect annotations to $\emptyset$ and recording all access violations. From the collected access paths, we compute a set of access permissions by abstracting the recorded paths to a restricted regular expression.

This abstraction is guided by a heuristic because there is no easy way to define a best abstraction of a finite language to a regular expression. As each finite language is regular, there is always a (potentially huge) regular expression specifying the language of observed access paths exactly. On the other hand, every language is contained in the regular language $.*$. As both extremes are useless, the goal of the heuristic is to find a regular language that includes the observed access paths but which also includes further likely access paths exhibited by the same program.

Fig. 1 Syntax of access paths and access permissions.

$p \in \mathit{Prop}$	property names	$P \subseteq \mathit{Prop}$	set of property names
$\pi ::= \varepsilon \mid p.\pi$	access paths	$b ::= \varepsilon \mid P.b \mid P*.b$	path permissions
$\gamma ::= \mathbf{R} \mid \mathbf{W}$	access classifiers	$a ::= \emptyset \mid b \mid a + a$	access permissions
$\kappa ::= \gamma(\pi)$	classified access path	$? = \mathit{Prop}$,	$@ = \emptyset \subseteq \mathit{Prop}$

For that reason, our inference algorithm is based on the intuition[2] that objects have an fixed structure a few levels of properties deep, followed by a traversal of a recursive structure (repeated list or tree links), and ending in objects with fixed structure. Thus, we have chosen a particular result template for an access permission. The inferred permissions are either concrete paths of small lengths or they start with a few concrete path elements, followed by an arbitrary sequence of path elements, and then finish with a few concrete path elements. The number of concrete initial and final path elements are parameters of the algorithm, which can be modified by the user to interactively find a satisfactory permission. The underlying algorithm guarantees the soundness of the resulting permission.

The arbitrary list of path elements in the middle can be further refined to enumerate the properties that can be repeated.

3 Inference Algorithm

This section first formally defines the syntax and semantics of access paths and access permissions and states some of their properties. Then, it describes the three phases of the inferences algorithm: trie building, extraction of access permissions, and simplification. Finally, it considers some special cases which are covered by the implementation, but which are not reflected in the formal development.

3.1 Access Paths and Access Permissions

Fig. 1 defines the syntax of access paths and access permissions. An access path is a sequence of property names. It is classified with an access classifier γ as either a read path or a write path yielding a classified access path κ.

A path permission extends an access path by admitting a set P of properties in each step. A component in a path permission may also be $P*$ to match any sequence of property names in P. An access permission is either empty, a path permission, or the union of two access permissions. We abbreviate the path component $?*$ to $*$.

While the definitions of path permissions and access paths inductively add path elements only to the left ends, we also decompose permissions and paths

[2] which is supported by our examples, but not yet empirically validated.

Fig. 2 Matching access permissions.

$$\mathbf{W}(\varepsilon) \prec \varepsilon \qquad \mathbf{R}(\varepsilon) \prec b \qquad \frac{\gamma(\pi) \prec b \qquad p \in P}{\gamma(p.\pi) \prec P.b} \qquad \frac{\gamma(\pi) \prec b}{\gamma(\pi) \prec P{*}.b}$$

$$\frac{\gamma(\pi) \prec P{*}.b \qquad p \in P}{\gamma(p.\pi) \prec P{*}.b} \qquad \frac{\kappa \prec a_1}{\kappa \prec a_1 + a_2} \qquad \frac{\kappa \prec a_2}{\kappa \prec a_1 + a_2} \qquad \frac{(\forall \kappa \in K)\ \kappa \prec a}{K \prec a}$$

from the right as in $\pi = \pi'.p$ or even consider the infix "." as concatenation operator as in the permission $\pi.*.\pi'$. We write $|\pi|$ for the length of a path and say that π' *is a prefix of* π if $\pi = \pi'.\pi''$, for some π''. Dually, π' *is a suffix of* π if $\pi = \pi''.\pi'$, for some π''. A set of paths Π is prefix-closed (suffix-closed) if $\pi \in \Pi$ implies that $\pi' \in \Pi$, for each prefix (suffix) of π.

We define the semantics of access permissions using the inference rules in Fig. 2. Let K be a set of classified access paths. A classified access path κ (or a set K of those) matches an access permission a, if the judgment $\kappa \prec a$ ($K \prec a$) is derivable from the inference rules. Property names in the permission must be matched exactly in the path, whereas $*$ components in the permission match any sequence of property names. The component @ matches no property. When the path is exhausted ($\pi = \varepsilon$), matching distinguishes read and write paths. While a read path is accepted with any remaining permission, a write path requires the permission to be exhausted, too. With this convention, a permission ending in @ specifies a set of read paths without giving write permission. In summary, write accesses $\mathbf{W}(\pi)$ must be matched entirely by a path permission whereas read accesses $\mathbf{R}(\pi)$ just need to be extensible to a full match. Hence, the set of read access paths is closed under prefixes.

Lemma 1. *1. If* $\mathbf{R}(\pi.p) \prec a$*, then* $\mathbf{R}(\pi) \prec a$*.*
2. If $\mathbf{W}(\pi.p) \prec a$*, then* $\mathbf{R}(\pi) \prec a$*.*
3. $\mathbf{W}(\pi) \not\prec b.@$*.*

3.2 Algorithm

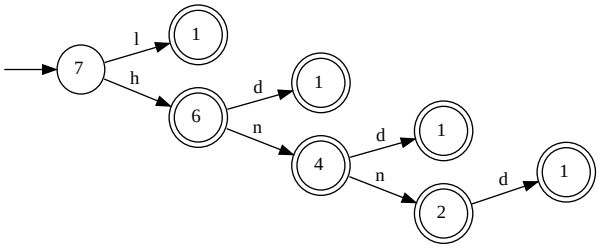
The task of the access path inference algorithm is thus to map a set of classified access paths to a set of reasonable path permissions. This task is akin to the problem of learning a (regular) language from a set of positive examples.[3] The problem of this task is that there is no best solution. For example, there are always two trivial path permissions that match a given classified path set:

Lemma 2. *Let* $K = \{\gamma_i(\pi_i) \mid i \in I\}$*.*

1. Let $b_i = \pi_i$ *considered as a path permission. Then* $K \prec \sum_i b_i$*.*
2. $K \prec *$*.*

[3] A negative example would be an impossible access path.

Fig. 3 Example trie.



For that reason, we have devised an algorithm based on a heuristic that computes reasonable results for a range of interesting examples.

Our algorithm has three phases. The first phase collects access paths in a trie data structure. This data structure enables efficient operations during the second phase. The second phase extracts access permissions from the trie. The third phase simplifies the resulting access permissions. The first two phases keep read and write paths separate because there are subtle differences in their handling due to the prefix closure of read accesses.

Building the Trie For our purposes, a trie [7] is a rooted, directed graph where each node is labeled with an integer and each edge is labeled with a property name. The trie $T(\Pi)$ represents a set of access paths Π as follows. The root node r is labeled with the number of paths $|\Pi|$. For each property p, let $p\backslash\Pi = \{\pi \mid p.\pi \in \Pi\}$ be the set of path tails of paths that start with p. If $p\backslash\Pi$ is non-empty, then the trie for Π includes $T(p\backslash\Pi)$ where there is an edge from r to the root node of $T(p\backslash\Pi)$.

For example, the path set $\Pi_{list} = \{l, h, h.d, h.n, h.n.d, h.n.n, h.n.n.d\}$ is represented by the trie in Fig. 3. The trie can also be considered a finite automaton recognizing the set Π with final states indicated by the double circles in the figure.

Extracting Access Permissions The goal of the extraction algorithm is to create access permissions of one of the forms π or $\pi.P*.\pi'$ where $P \subseteq Prop$ and π' may be empty. The initial component π is determined by computing a set of "interesting" prefixes from a set of paths Π, where π is a prefix of Π if there exists some $\pi' \in \Pi$ such that π is a prefix of π'.

Given two integers $l \geq 0$ and $d \geq 1$, we consider a path as (l, d)-interesting with respect to a path set Π if it is a prefix of Π and it is either shorter than the *base length* l or it has a *branching degree* less than or equal to d above length l. Here, the branching degree of a path $\mathrm{BDeg}_{\Pi}(\pi)$ is the number of properties q such that $\pi.q$ is a prefix of some path in Π. The (l, d)-interesting prefixes of Π are formalized by $\mathrm{Prefixes}_{l,d}(\Pi)$, which is simple to compute from $T(\Pi)$.

$$\begin{aligned}
\mathrm{BDeg}_{\Pi}(\pi) &= |\{q \mid (\exists \pi')\ \pi.q.\pi' \in \Pi\}| \\
\mathrm{Prefixes}_{l,d}(\Pi) &= \{\, p_1 \ldots p_n \mid \\
&\qquad (\exists \pi)\ p_1 \ldots p_n.\pi \in \Pi, \\
&\qquad (\forall j \in \{l, \ldots, n-1\})\ \mathrm{BDeg}_{\Pi}(p_1 \ldots p_j) \leq d\}
\end{aligned}$$

To continue the example from the preceding subsection,

$$\begin{aligned}
\mathrm{Prefixes}_{0,1}(\Pi_{list}) &= \{\varepsilon\} \\
\mathrm{Prefixes}_{1,1}(\Pi_{list}) &= \{\varepsilon, l, h\} \\
\mathrm{Prefixes}_{2,1}(\Pi_{list}) &= \{\varepsilon, l, h, h.d, h.n\} \\
\mathrm{Prefixes}_{0,2}(\Pi_{list}) &= \{\varepsilon, l, h, h.d, h.n, h.n.d, h.n.n, h.n.n.d\} \\
&= \mathrm{Prefixes}_{k,2}(\Pi_{list}) \qquad (\forall k)
\end{aligned}$$

At this point, we distinguish the treatment of read paths from the treatment of write paths. As read paths are closed under taking the prefix, we may compute the prefix reduct by removing all paths that are proper prefixes of other paths.

$$\mathrm{Reduct}(\Pi) = \{\pi \in \Pi \mid (\forall \pi')\ |\pi'| > 0 \Rightarrow \pi.\pi' \notin \Pi\}$$

Continuing the example further:

$$\begin{aligned}
\mathrm{Reduct}(\mathrm{Prefixes}_{0,1}(\Pi_{list})) &= \{\varepsilon\} \\
\mathrm{Reduct}(\mathrm{Prefixes}_{1,1}(\Pi_{list})) &= \{l, h\} \\
\mathrm{Reduct}(\mathrm{Prefixes}_{2,1}(\Pi_{list})) &= \{l, h.d, h.n\} \\
\mathrm{Reduct}(\mathrm{Prefixes}_{0,2}(\Pi_{list})) &= \{l, h.d, h.n.d, h.n.n.d\}
\end{aligned}$$

For write paths, a more conservative reduction must be applied. Only those proper prefixes can be removed that are not members of the underlying original set. Let Π be a set of prefixes of Π_0.

$$\mathrm{ReductW}(\Pi, \Pi_0) = \mathrm{Reduct}(\Pi) \cup (\Pi \cap \Pi_0)$$

Given an interesting prefix π of path set Π, we now construct the left quotient of Π with respect to π, i.e., the set of suffixes

$$\pi \backslash \Pi = \{\pi' \mid \pi.\pi' \in \Pi\}$$

Technically, we construct this set in time linear in the length of π by returning the subtrie of the trie $T(\Pi)$ obtained by following the path π.

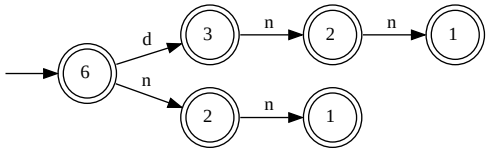
If we continue the example with $\mathrm{Reduct}(\mathrm{Prefixes}_{1,1}(\Pi_{list})) = \{l, h\}$, we obtain the following sets of suffixes:

$$\begin{aligned}
l \backslash \Pi_{list} &= \{\varepsilon\} \\
h \backslash \Pi_{list} &= \{\varepsilon, d, n, n.d, n.n, n.n.d\}
\end{aligned}$$

For each of these sets, we now consider the set of interesting suffixes, where "interesting" is defined in the same way as for prefixes. Technically, we just reverse all path suffixes and apply the interesting-prefixes algorithm. That is,

$$\mathrm{Suffixes}_{l,d}(\Sigma) = \overleftarrow{\mathrm{Prefixes}_{l,d}(\overleftarrow{\Sigma})}$$

Fig. 4 Reversed suffix trie.



where $\overleftarrow{\Sigma} = \{\overleftarrow{\pi} \mid \pi \in \Sigma\}$ and $\overleftarrow{\pi}$ is the reverse of a path π.

Going back to the example, Fig. 4 shows the trie containing the reversed suffixes of $h\backslash\Pi_{list}$. From this trie, it is easy to see that the $(0,1)$-interesting suffixes of $h\backslash\Pi_{list}$ are $\{\varepsilon, d, n\}$, whereas there is only one respective suffix of $l\backslash\Pi_{list}$, namely ε.

The final step of the algorithm considers for each pair of interesting prefix and interesting suffix the remaining part in the middle. The *right quotients* of the suffix language yield exactly this remaining part. The right quotient Π/π of a language with respect to a path π is defined dually to the left quotient by

$$\Pi/\pi = \{\pi' \mid \pi'.\pi \in \Pi\}$$

To abstract the resulting middle language, we restrict the algorithm to two choices. Either ε, if the middle language is $\{\varepsilon\}$, or $P*$ in all other cases.

In the example, we need to consider four cases, with the computation shown left and the resulting access permission shown in the right column:

$$\begin{array}{lll}
(l\backslash\Pi_{list})/\varepsilon & = \{\varepsilon\} & \mapsto l \\
(h\backslash\Pi_{list})/\varepsilon & = h\backslash\Pi_{list} & \mapsto h.\{n,d\}* \\
(h\backslash\Pi_{list})/d & = \{\varepsilon, n, n.n\} & \mapsto h.n*.d \\
(h\backslash\Pi_{list})/n & = \{\varepsilon, n\} & \mapsto h.n*.n
\end{array}$$

This result is not entirely satisfactory because $h.\{n,d\}*$ clearly subsumes $h.n*.d$ and $h.n*.n$, but the latter two permissions are more informative and thus preferable. Unfortunately, even together, they do not cover the access path h, which is only covered by $h.*$.

The source of the problem is that the set $\{\varepsilon, d, n\}$ is suffix-closed. For prefixes, we apply the prefix reduction because the semantics of access paths is prefix-closed. However, we cannot just apply suffix reduction as the example shows: If the suffix (in this case ε) is actually an element of the underlying set $h\backslash\Pi_{list}$, then dropping the suffix would be incorrect.

The solution is to treat the suffixes which would be removed by suffix reduction but which are elements of the underlying set specially and drop the rest. The special treatment is simple: we just declare their middle language to be $\{\varepsilon\}$. With this treatment (specified in function BuildPermissions in Fig. 5), the case $(h\backslash\Pi_{list})$ with suffix ε yields the access permission h. The function has to be called for each interesting prefix with the corresponding suffix language (function PermissionsFromPathSet).

The final result of this phase applied to the running example is the set of access permissions $\{l, h, h.n*.d, h.n*.n\}$.

Fig. 5 Building access permissions.

function BUILDPERMISSIONS(π, Σ, sl, sd)
  ▷ π is a prefix, Σ corresponding suffix language, sl, sd suffix length and degree
  $R \leftarrow \emptyset$ ▷ result set of access permissions
  $\Sigma_0 \leftarrow \text{Suffixes}_{sl,sd}(\Sigma)$ ▷ set of interesting suffixes of Σ
  for all $\sigma \in \Sigma_0$ **do**
    if σ is proper suffix of an element of Σ_0 **then**
      if $\sigma \in \Sigma$ **then**
        $R = R + \pi.\sigma$
    else
      if $\Sigma/\sigma = \{\varepsilon\}$ **then** ▷ middle language is empty
        $R = R + \pi.\sigma$
      else
        $R = R + \pi.P*.\sigma$ ▷ P is set of properties in Σ/σ
  return R

function PERMISSIONSFROMPATHSET(Π_0, Π, sl, sd)
  ▷ Π_0 set of prefixes of Π, sampled set of paths, sl, sd suffix length and degree
  $R \leftarrow \emptyset$ ▷ result set of access permissions
  for all $\pi \in \Pi_0$ **do**
    $R = R + \text{BUILDPERMISSIONS}(\pi, \pi \backslash \Pi, sl, sd)$
  return R

Simplifying Access Permissions The result of the previous phase is not as concise as it could be. It may still generate redundant access permissions. Consider the result of the example $\{l, h, h.*.d, h.*.n\}$. As this set only contains read permissions, which are closed under prefix, it follows that permissions h is subsumed by $h.*.d$ and $h.*.n$, so that the result is equivalent to (the simpler set) $\{l, h.*.d, h.*.n\}$.

To perform this simplification, we first define a subsumption relation $\subseteq$ on path permissions.

$$\vdash \varepsilon \subseteq b \qquad \frac{\vdash b \subseteq P'*.b' \qquad P \subseteq P'}{\vdash P.b \subseteq P'*.b'} \qquad \frac{\vdash P.b \subseteq b'}{\vdash P.b \subseteq P'*.b'}$$

$$\frac{\vdash b \subseteq b' \qquad P \subseteq P'}{\vdash P*.b \subseteq P'*.b'}$$

This relation is sound in the sense that it reflects the semantic subset relation on sets of accepted access paths.

Lemma 3. *If* $\mathbf{R}(\pi) \prec b$ *and* $\vdash b \subseteq b'$*, then* $\mathbf{R}(\pi) \prec b'$*.*

Given this relation, simplification just removes all read path permissions that are subsumed by other (read or write) path permissions as specified in Fig. 6. In the example, clearly $\vdash h \subseteq h.n*.d$, so that h can be removed from the read path permissions.

Fig. 6 Simplification.

function SIMPLIFY(R, W) ▷ sets of path permissions, R for reading, W for writing
  while $(\exists b, b')\ b \in R \wedge (b' \in R \wedge b \neq b' \vee b' \in W) \wedge \vdash b \subseteq b'$ **do**
    $R \leftarrow R - b$
  return (R,W)

Fig. 7 Overall algorithm.

function MAIN($\Pi^r, \Pi^w, pl = 1, pd = 1, sl = 0, sd = 1$)
  ▷ Π^r read paths, Π^w write paths
  ▷ pl, pd prefix length and degree, sl, sd suffix length and degree
  $\Pi_0^r \leftarrow \text{Prefixes}_{pl,pd}(\Pi^r)$ ▷ interesting prefixes of Π^r
  $\Pi_0^w \leftarrow \text{Prefixes}_{pl,pd}(\Pi^w)$ ▷ interesting prefixes of Π^w
  $R \leftarrow$ PERMISSIONSFROMPATHSET(Reduct(Π_0^r), Π^r, sl, sd)
  $W \leftarrow$ PERMISSIONSFROMPATHSET(ReductW(Π_0^w), Π^w, sl, sd)
  $(R, W) \leftarrow$SIMPLIFY(R, W)
  return $R.@ + W$

Putting it Together Fig. 7 summarizes the overall algorithm as explained up to this point. The parameters that determine the length and degree for the computation of interesting prefixes and suffixes have default values that yield good results in our experiments. In addition, our implementation makes them accessible through the user interface for experimentation, on a global as well as on a per-function basis.

3.3 Special Cases

There are two special cases of property accesses that lead to extremely high branching degrees. The first case is that an object is used as an array. The symptom of this case is the presence of accesses to numeric properties. Our implementation assumes that arrays contain homogeneous data and collapses all numeric property names to a single *pseudo property name* ♯. This collapsing already happens when the trie is constructed from the access paths.

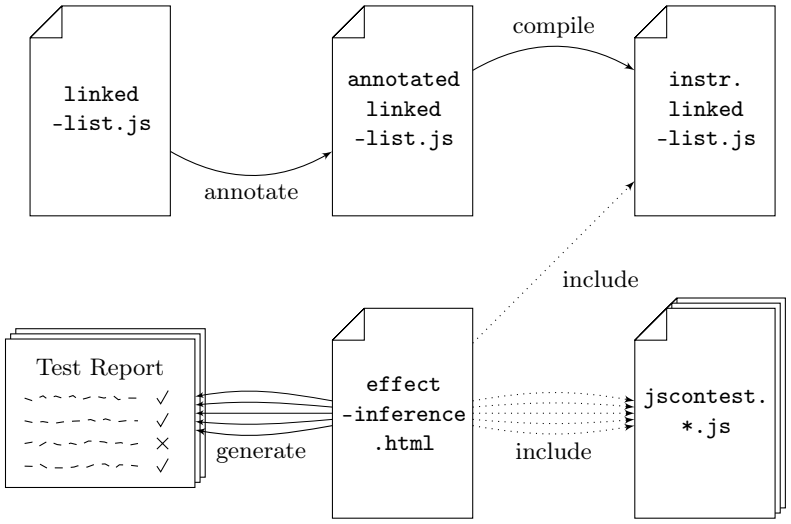
Similarly, an object might be used as a hash table. This use leads to the same high branching degrees as array accesses, but cannot be reliably detected at trie construction time. Instead, the implementation makes a pre-pass over the trie that detects nodes with a high number of successors (e.g., set with the parameter HIGH_DEGREE which defaults to 20), merges these subtries, and relabels the remaining edge to the merged successor trie with a wildcard pseudo property name ?.

As the rest of the algorithm does not depend on the actual form of the property names, the introduction of these pseudo property names is inconsequential.

3.4 Soundness

To establish the soundness of the algorithm, we need to prove that each element of the original path set is matched by the extracted access permission. The first

Fig. 8 Overview over JSConTest



phase, building the trie, is trivially sound. The third simplification phase is sound by Lemma 3. It remains to consider the second phase. We only examine the case for read paths with write paths handled similarly.

Suppose $\pi \in \Pi$, the initial set of access paths. As $\Pi_0 = \text{Reduct}(\text{Prefixes}_{l,d}(\Pi))$ is prefix free, there are two possibilities. Either, there is exactly one element $\pi_0 \in \Pi_0$ such that π_0 is a prefix of π, or there is at least one element $\pi' \in \Pi_0$ such that π is a prefix of π'.

In the second case, π' will be prefix of an access path $\pi'.b$ with $\pi \prec \pi'.b$.

In the first case, it remains to show that π_0 is extended to an access path that matches $\pi = \pi_0.\pi_1$. Let Σ_0 be the set of interesting suffixes of $\Sigma = \pi_0 \backslash \Pi$. By construction, $\pi_1 \in \Sigma$. We need to show that there is an element $\sigma \in \Sigma_0$ where either $\pi_1 = \sigma$ or $\pi_1 \prec *.\sigma$.

For a contradiction, suppose that neither is the case and let σ be the maximal suffix of π_1 in Σ_0 (such σ must exist). If σ is a proper suffix of an element of Σ_0 and $\sigma \in \Sigma$, then $\sigma = \pi_1$, a contradiction. If $\Sigma/\sigma = \{\varepsilon\}$, then $\sigma = \pi_1$, a contradiction. If $\Sigma/\sigma \neq \{\varepsilon\}$, then $\pi_1 \prec *.\sigma$, a contradiction.

Hence, all cases are matched.

4 Implementation

Our effect inference algorithm is implemented as part of the JSConTest system for contract-based testing of JavaScript programs. JSConTest supports a typical workflow for unit testing, which starts with augmenting the unit under test with a specification of the tests that should be performed. Then JSConTest generates

the test cases from the contracts and produces a test report from the outcomes. The test report either contains concrete evidence that some part of the desired behavior of the unit under test is incorrect or, if all tests pass, it increases the confidence that the unit under test behaves according to its specification.

Figure 8 illustrates this workflow. First, the tester specifies the desired properties of the program under test by annotating functions with contracts. The resulting annotated source file (Fig. 8, annotated linked−list.js) is passed to the JSConTest compiler. The compiler generates an instrumented version of the program (instr. linked−list.js). To test a JavaScript program inside a browser, a HTML file is needed to start the JSConTest framework and include the necessary files. The result of execution is a test report that documents which of the contracts are fulfilled by the unit under test. Depending on the parameters passed to the JSConTest compiler, the instrumented code does not only report contract violations, but also collects run-time data, for instance, what properties are accessed during test execution. As the JSConTest run-time framework is event-driven, it is possible to extend it to execute arbitrary algorithms on the collected data and thus create comprehensive test reports instead of just reporting raw data to the user.

In this work we make use of this feature and let the JSConTest compiler generate code that reports all property accesses and invokes a handler for doing effect inference. As the effect inference is an interactive process, which depends on a number of interactively modifiable parameters, the test report is not just a static page with the test results, but a dynamic interface that interacts with the inference algorithm.

5 Evaluation

To evaluate the inference algorithm, we applied it to a few examples and compared the computed access permissions with manually constructed permissions.

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

The first step towards effect inference is to come up with contracts for each of the functions. The result is a source file annotated as in this code snippet:

```
1 /*c js:ll.(top) → undefined */
2 function add(data) { ... }
3 /*c js:ll.(top) → top */
4 function item(index) { ... }
5 /*c js:ll.(top) → top */
6 function remove(index) { ... }
```

In these contracts, js:ll describes the receiver object, the parenthesized phrase the types of the arguments, and the phrase following the → the result type. In

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

particular, js:ll refers to JavaScript function that generates and checks a certain kind of lists, top stands for any value, and undefined is the undefined value, which is returned when no return value is given.

The JSConTest compiler picks up the contracts in the special comments, generates code for assertions derived from the contracts, and creates a test suite for checking the contracts. This setup enables the tester to test the input/output behavior of all functions using directed random testing as explained in our previous work [12].

In the current version of JSConTest it is furthermore possible to infer the effects of the functions as follows. To obtain a first impression what properties are accessed by the different functions, it is sufficient to add the empty effect to the contract as in the contract /∗c js:ll.(top) → undefined **with** [] ∗/ for the add function. This augmented contract states that the function with this contract is not allowed to change anything in the heap that already exists before invocation of the function. Extending the remaining functions' contracts in the same way and applying the JSConTest compiler again results in instrumented code that monitors all property accesses.

When the compiled code executes in a browser, it collects, as a side effect, thousands of property accesses which violate the empty effect annotation. From this raw data, our effect inference computes concise access permissions. The syntax of these permissions is inspired by the syntax of file paths. For example, the computed effect for add is

this._head, this._head.next∗, this._length

which means that add only accesses objects via its this pointer, it reads and writes the _head and _length properties, and it reads and writes a next property that is reachable via _head followed by a sequence of next properties as indicated by next∗. All three path permissions are write permissions that implicitly permit reading all prefixes of any path leading to a permitted write.

The computed effect for remove is also interesting:

this._head.next∗.data.@, this._head.next∗, this._length

The function remove deletes a given value from the list. To this end, it compares this value with all data properties reachable via _head and a sequence of (all next) properties, as indicates with the first access path. Its ending in @ indicates a read-only path. Furthermore, remove changes next pointers and modifies the _length property of this.

Full details of this example are available on the project homepage of JSConTest.[5] It presents the outcomes of four examples complete with the annotated source code, the instrumented source code, and a web page to execute the example locally.

On the webpage, there is another similar example implementing binary search trees. For these two examples the algorithm infers a precise effect annotation.

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

As a larger example, which is also detailed on the webpage, we consider the Richards benchmark from the Google V8 benchmark suite. After annotating its source code with contracts as outlined above, the effect inference algorithm automatically obtains informative results albeit less precise than the manually determined effects that we used in our previous work [11]. This example uncovered a number of new points for our inference algorithm, in particular, that special treatment for arrays and objects used as hash tables is required (see Sec. 3.3). This treatment is also covered in a micro benchmark in the webpage.

6 Related Work

Effect analysis in programming languages has some history already. Initial efforts by Gifford and Lucassen [8] perform a mere side-effect analysis which captures allocation as well as reading from and writing to variables. Subsequent work extends this approach to effects on memory regions which abstract sets of heap-allocated objects [16, 17]. Such an effect describes reading, writing, and allocation in terms of regions. An important goal in these works is automatic effect inference [2], because regions and effects are deemed as analysis results in a phase of a compiler.

Path related properties are also investigated by Deutsch [5], but with the main goal of analyzing aliasing. His framework is based on abstract interpretation and offers unique abstract domains that provide very precise approximations of path properties.

In object-oriented languages, the focus of work on regions and effects is much more on documentation and controlling the scope of effects than on uncovering optimization opportunities. Greenhouse and Boyland [10] transpose effects to objects. One particular point of their effect system is that it preserves data abstraction by not mentioning the particular field names that are involved in an effect, but by instead declaring effect regions that encompass groups of fields (even across classes) and by being able to have abstract regions. In contract, our work is geared towards the scripting language JavaScript, which provides no data abstraction facilities and where the actual paths are important documentation of an operation that aids program understanding.

Skalka [15] also considers effects of object-oriented programs, but his effects are traces of operations. The goal of his is to prove that all traces generated by a program are safe with respect to some policy. Data access is not an issue in this work.

The learning algorithm in Sec. 3.2 abstracts a set of access paths to a set of access permissions, which are modeled after file paths with wildcards. The more general problem is learning a language from positive examples, which has been shown to be impossible, as soon as a class of languages contains all of the finite languages and at least one infinite language [9, 1]. Clearly, the class of regular languages qualifies. Better results can be achieved by restricting the view to "simple examples" [4] or to more restricted kinds of languages [6].

Transformation of JavaScript programs is a well-studied topic in work on enforcing and analyzing security properties. For example, Maffeis and coworkers [13] achieve isolation properties between mashed-up scripts using filters, rewriting, and wrapping. Chugh and coworkers [3] present (among others) a dynamic information flow analysis based on wholesale rewriting. Yu and coworkers [18] perform rewriting guided by a security policy. BrowserShield [14] relies on similar techniques to attain safety. As detailed in our submitted work [11], extensive rewriting has a significant performance impact and gives rise to subtle semantic problems. These problems are shared among all transformation-based tools.

7 Conclusion

The current version of JSConTest induces access path permissions from sample test runs. In many cases, the resulting permissions are as good as manually determined ones. In the few remaining cases, interactive tweaking of the parameters is required to obtain good results. Thus, effect inference appears to be a useful tool to analyze JavaScript programs and enhance their contracts with effect information.

Effect inference or effect learning removes much of the tedium of declaring effect annotations for a given program. However, it is important for then inference to run with tight contracts or/and a test suite with high coverage, since the inference algorithm can only find a accurate effect annotation, if all aspects of the code under test are explored.

Tightness of the contract is required because a loose contract essentially causes the generation of entirely random test cases. It is unlikely that these random test cases discover the access path pattern of a function. For that reason, some of our examples rely on custom contracts that generate random values in the shape expected by the function.

Similarly, if the coverage of a test session is low, then it is likely that some paths through the input data are never traversed. Thus, high coverage increases the probability that all access paths are exercised.

One way to circumvent these restrictions is to observe the program running in the wild and collect and evaluate the resulting trace data. To be most effective and efficient, this approach would require instrumenting a JavaScript engine to collect the required access traces. Our evaluation back end and inference algorithm, however, would remain the same.

References

1. Angluin, D.: Inductive inference of formal languages from positive data. Information and Control 45(2), 117–135 (1980)
2. Birkedal, L., Tofte, M.: A constraint-based region inference algorithm. Theor. Comput. Sci. 58, 299–392 (2001)
3. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for JavaScript. In: Proc. 2009 ACM Conf. PLDI. pp. 50–62. ACM, Dublin, Ireland (Jun 2009), http://doi.acm.org/10.1145/1542476.1542483

4. Denis, F.: Learning regular languages from simple positive examples. Machine Learning 44(1/2), 37–66 (2001)
5. Deutsch, A.: A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In: Proc. IEEE International Conference on Computer Languages 1992. pp. 2–13. IEEE, Oakland, CA (Apr 1992)
6. Firoiu, L., Oates, T., Cohen, P.R.: Learning deterministic finite automaton with a recurrent neural network. In: Honavar, V., Slutzki, G. (eds.) ICGI. Lecture Notes in Computer Science, vol. 1433, pp. 90–101. Springer (1998)
7. Fredkin, E.: Trie memory. Commun. ACM 3, 490–499 (Sep 1960), http://doi.acm.org/10.1145/367390.367400
8. Gifford, D., Lucassen, J.: Integrating functional and imperative programming. In: Proc. 1986 ACM Conf. on Lisp and Functional Programming. pp. 28–38 (1986)
9. Gold, E.M.: Language identification in the limit. Information and Control 10(5), 447–474 (1967)
10. Greenhouse, A., Boyland, J.: An object-oriented effects system. In: Guerraoui, R. (ed.) 13th ECOOP. LNCS, vol. 1628, pp. 205–229. Springer, Lisbon, Portugal (Jun 1999)
11. Heidegger, P., Bieniusa, A., Thiemann, P.: Access permission contracts (Dec 2010), submitted for publication, http://proglang.informatik.uni-freiburg.de/jscontest/
12. Heidegger, P., Thiemann, P.: Contract-driven testing of javascript code. In: Proceedings of the 48th international conference on Objects, models, components, patterns. pp. 154–172. TOOLS'10, Springer, Malaga, Spain (Jun 2010), http://portal.acm.org/citation.cfm?id=1894386.1894395
13. Maffeis, S., Mitchell, J.C., Taly, A.: Isolating JavaScript with filters, rewriting, and wrappers. In: ESORICS'09: Proceedings of the 14th European Conference on Research in Computer Security. pp. 505–522. Springer-Verlag, Saint-Malo, France (2009)
14. Reis, C., Dunagan, J., Wang, H.J., Dubrovsky, O., Esmeir, S.: BrowserShield: Vulnerability-driven filtering of dynamic HTML. ACM Trans. Web 1(3), 11 (2007)
15. Skalka, C.: Trace effects and object orientation. In: Proceedings of the ACM Conference on Principles and Practice of Declarative Programming. Lisbon, Portugal (Jul 2005)
16. Talpin, J.P., Jouvelot, P.: The type and effect discipline. Information and Computation 111(2), 245–296 (1994)
17. Tofte, M., Talpin, J.P.: Region-based memory management. Information and Computation 132(2), 109–176 (1997)
18. Yu, D., Chander, A., Islam, N., Serikov, I.: JavaScript instrumentation for browser security. In: Felleisen, M. (ed.) Proc. 34th ACM Symp. POPL. pp. 237–249. ACM Press, Nice, France (Jan 2007)