Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

JSConTest: Contract-Driven Testing and Path Effect Inference for JavaScript

Phillip Heidegger[a] Peter Thiemann[a]

a. University of Freiburg, Germany

Abstract Program understanding is a major obstacle during program maintenance. In an object-oriented language, understanding an operation requires understanding its type and its effect on the object network. The effect is particularly important for scripting languages where there is neither class structure that restricts the shape of an object nor any other kind of access control.

We have designed and implemented JSConTest, a tool that provides a facility to annotate JavaScript programs with type and effect contracts and to create random tests out of the contracts. Run-time monitoring for contracts is implemented with a program transformation. The effect of an operation is described by access permissions, which abstract sets of access paths along which the operation reads or writes object properties. Type contracts can also be used to drive guided random testing of the program.

JSConTest contains an algorithm for computing access permissions from a set of access paths obtained by running the program. The main ingredient of the algorithm is a novel heuristic that produces precise and concise results without user interaction. It has been applied to a range of examples with encouraging results.

Keywords contracts, effects, scripting languages, access permissions, inference, JavaScript

1 Introduction

Software maintenance is a difficult task. During maintenance of a system, a programmer is asked to implement new features and/or to track down errors and fix them. For both activities, the programmer must understand the system sufficiently to come up with the right way of changing the code and to guarantee that no new errors are introduced by the changes. Our tool, JSConTest, contributes to both aspects of software maintenance for programs written in JavaScript. However, the underlying principles are transferable to other scripting languages like Python, PHP, and Ruby.

Phillip Heidegger, Peter Thiemann. *JSConTest: Contract-Driven Testing and Path Effect Inference for JavaScript*. In Journal of Object Technology, vol. 11, no. 1, 2012, pages 6:1–29.
doi:10.5381/jot.2012.11.1.a6

Compared to traditional languages, scripting languages accelerate the development process because of the flexibility gained by features like dynamic typing, weak typing, and meta programming. However, this flexibility makes it hard to understand the intended behavior of a system and to ensure that the system's functionality is not affected by a change.

Contracts with run-time monitoring [Mey97, FFF04, THF08] are a proven means to establish hints that help with understanding a system and with detecting changed behavior by adding specification elements to a programming language. For a scripting language, even a contract that just specifies a typing can be helpful, as demonstrated by the work on gradual typing [ST07] and with the construction of languages that enable the interoperation and the gradual migration of functionality between statically and dynamically typed parts of a program [WF09, MF09, BFN+09].

A common theme of these works is that the statically typed part of the code is shielded from the dynamically typed parts by type or contract annotations. The run-time monitoring of these annotations guarantees that the dynamically typed parts cannot corrupt the statically typed parts of the program.

Systematic testing is the generally accepted pragmatic means to establish faith that a system implements certain functionalities. There are many different kinds of testing that we cannot explore at this point [MS04], two of which are relevant to JSConTest: Unit testing is the main workhorse in agile development processes like test-driven development. Regression testing is geared at detecting changes in functionality after a change in the code base.

Our tool JSConTest [HT10a] is designed to help programmers maintain JavaScript programs by supporting problem detection, program understanding, unit testing, and regression testing.

- JSConTest provides a contract facility based on type signatures extended with effect specifications. This facility aids program understanding because programmers can express their current understanding about the type and effect of an operation and have it checked during test runs or program execution.

- Type contracts can be gradually introduced as the programmer explores the program. Later on, they serve as checked documentation.

- The effect portion of a type contract confines the side effects of an operation similar to a frame condition in program verification. This confinement simplifies reasoning and helps understanding an operation.

- JSConTest can conduct test runs with directed random testing where test cases generated from the type signatures and from hints extracted from the source code. Elsewhere [HBT12] we have shown that this style of testing in combination with effects is highly effective in detecting mutations that lead to changed behavior. Hence, these test cases are very well suited as regression tests.

- The types in contracts can be narrowed down to singleton types, which effectively turns the generated tests into unit tests.

- JSConTest comes with an algorithm that infers the effect of an operation from traces collected at run time. The inferred effect provides a starting point to investigate the meaning of an operation and to construct a contract that can be employed for conducting regression tests.

Why do we consider contracts with effects? The effects that we are proposing have the form of *access permissions*, which define the set of paths along which an operation is allowed to read and write object properties. Knowing the effect of an operation simplifies understanding and reusing the operation. For example, an operation that only explores paths matching the regular expression $(r|l)^*d$ can be safely applied to any object structure that contains a binary tree linked via r and l and which stores some data in its d properties. If we ascribe the operation a contract with these access permissions, then we can be certain that all other properties are left alone and that the tree is not restructured. JSConTest supports this style of contract and enforces it dynamically.

The effect inference of JSConTest is not a static program analysis, but it is based on data collected at run time. While this approach is incomplete, it can avoid some problems with static analysis. As a general problem, a static analysis also analyzes semantically dead code, which cannot be visited by a run-time approach. Regarding JavaScript-specific problems, a run-time approach can easily support properties accessed dynamically via the array indexing notation, an `eval` function, scope-distorting constructs like the `with` statement, and accesses via arbitrarily nested prototype chains, all of which are very hard to analyze statically.

1.1 Contribution

We have designed and implemented JSConTest, a tool that augments JavaScript with type contracts, access permission contracts, and contract monitoring. Beyond contract monitoring, JSConTest performs guided random testing, where test cases are automatically generated from the type contracts. To use contracts seamlessly in a maintenance situation, we further designed and implemented an algorithm that automatically computes a practically useful access permission for an operation from the access traces of this operation generated during one or more program executions.

The main contribution of the algorithm is the generation of concise, non-trivial access permission contracts, which we demonstrate with encouraging results on examples ranging from small data structure libraries to large benchmark programs.

This paper contains an overview of the JSConTest system excerpted from earlier work [HT10a]. Its main emphasis is the presentation of a significant overhaul of an earlier effect inference algorithm [HT11]. The revised algorithm is simpler and gives non-trivial, concise results without user intervention, whereas the previous algorithm required fiddling with the configuration options.

1.2 Outline

Section 2 contains a tour of JSConTest. It first motivates the design principles of JSConTest, then explores the use of type signatures as contracts with examples, and briefly explains its relation to testing. The section concludes with a brief description of the implementation and a discussion of application scenarios for JSConTest. Section 3 provides some background on access paths and access permissions. Section 4 presents the inference algorithm to compute access permissions from access traces and establishes its soundness. Section 5 evaluates the inference algorithm with examples. Section 6 discusses related work and Section 7 concludes.

Program 1 JavaScript - Predicate over an object.

```
1 /*c obj → bool */
2 function readyToShip(order) {
3   if (order.name && order.address && order.totalprice) {
4     return (order.totalprice > 20 || order.shippingCharges > 0);
5   }
6   return false;
7 }
```

2 A Tour of JSConTest

JSConTest is a tool for understanding, documenting, and testing JavaScript programs. It implements a contract system based on extended type signatures and an integrated random testing engine.

In this context, we consider a type signature as a loose specification of an operation. JSConTest regards a type signature as valid if it cannot find a counterexample in a given number of test cases, which are randomly generated from the type signature. Although this approach is incomplete, it nevertheless gathers evidence for the validity of the signature. Furthermore, JSConTest does not reject a program based on programming style (as some static systems propose [GSK10]). Second, it has a tight feedback loop, because each failing test case provides a concrete counterexample. Third, it is gradually applicable: Starting from a few operations with contracts, it is easy to add contracts step by step thus gradually expanding the specified part of a program.

2.1 Type Signatures as Contracts

JSConTest's contract language can express all types that are relevant to JavaScript programs: booleans, strings, numbers, objects, and first class functions. As an example, consider the function readyToShip (Program 1), which returns true if the object passed as the first parameter represents a valid order in an online shop. Otherwise, it returns false. To this end, it first checks the presence of the properties name, address, and totalprice. Then it checks that shipping charges are applied unless the total price is greater than 20.

With JSConTest, the programmer can specify the interface of an operation by attaching a special comment. In the example, the comment /*c obj → bool */ specifies that the function readyToShip takes an object as a parameter and returns a boolean value. Such a contract is similar to a type signature, in a notation adapted from functional programming languages like Haskell [Pey03] and ML [MTHM97].

The obj contract can be refined to specify a contract for each property. Such a contract has the form { $p_1 : c_1$, ..., $p_n : c_n$ } and describes an object with required properties p_1 to p_n where the property values have to fulfill the contracts c_1 to c_n. This style of contract admits further properties in the object, whereas the syntax {| $p_i : c_i$ |} disallows additional properties. So, the contract for readyToShip could be refined to /*c { name : string, address : string, totalprice : number } → bool */.

Although JavaScript arrays are just special objects, JSConTest supports a separate array contract of the form [c]. The array contract is homogeneous because all elements of the array are expected to match c (which may be top to accept any value).

Program 2 JavaScript - Contract for a function expression.

```
1 var g = function (y) /*c (int) → (int → bool) */{
2   return function(x) /*c int → bool */ {
3     return x === y;
4   };
5 };
```

JavaScript Primitives

$x \in$ identifier, $f \in$ float, $i \in$ integer, $s \in$ string, $b \in$ bool,
$prop \in$ property

Primitive contracts

p	::=	`undf` $\mid \top$	undefined, any value
	$\mid$	`bool` $\mid b$	boolean values
	$\mid$	`string` $\mid s$	string values
	$\mid$	`int` $\mid i \mid$ `[`i`;`i`]`	integers, integer range
	$\mid$	`number` $\mid f \mid$ `[`f`;`f`]`	floats, float range
	$\mid$	`obj` $\mid$ `fun`	object, function
	$\mid$	`js:`x	custom contract, JS scope

Composite contracts

c	::=	p	
	$\mid$	c`@numbers` $\mid$ c`@strings` $\mid$ c`@labels`	guided random testing
	$\mid$	$(c, \ldots, c) \rightarrow c\ (ap)^?$	functions, ap see $Figure\ 2$
	$\mid$	$c.(c, \ldots, c) \rightarrow c\ (ap)^?$	methods, ap see $Figure\ 2$
	$\mid$	`{`$p_1 : c_1, p_2 : c_2, \ldots, p_n : c_n(, \ldots)^?$`}`	objects
	$\mid$	`{|`$p_1 : c_1, p_2 : c_2, \ldots, p_n : c_n$`|}`	objects
	$\mid$	`[`c`]`	arrays

Annotations

a ::= `~noAsserts` $\mid$ `~noTests` $\mid$ `#Tests:`i

Contracts

t ::= `/*c` $c\ a^*$ (`|` $c\ a^*$)* `*/`

Figure 1 – Syntax of contracts.

Contracts may also be attached to inline function expressions. Program 2 contains an example.

Figure 1 presents the syntax of JSConTest's contract language. Beyond the contracts already discussed, there are singleton contracts for primitive data types — they only accept one particular value. There is also support for numeric ranges. Last, but not least, the contract system is user-extensible. A contract of the form js:x invokes a custom extension written by the programmer, where x is the name of a JavaScript function that is called by JSConTest to implement the user-defined contract. More than one contract per function can be specified by writing a list of contracts separated by the "|" character. For functions, the effect of such a specification is similar to an intersection type. For more details, refer to our previous work [HT10a].

Program 3 JavaScript - Contract for a predicate on two integers.

```
1 /*c (int,int) → bool */
2 function f(x,y) {
3   return (x != y && 2 * x == x + 10) ? "true" : false;
4 }
```

2.2 Contracts as Test Case Generators

JSConTest uses contracts in two different ways, for implementing contract monitoring and for generating random test cases. The latter is a natural extension of the former. For contract monitoring, each contract must provide a *checker* that asserts that a value adheres to the contract. For testing, each contract must provide a *generator* that produces random values that all fulfill the contract.

For primitive contracts, checkers and generators are straightforward. Generators produce uniformly distributed samples of the domain of the contract. For function contracts, checkers and generators for functions are needed. They can be defined in terms of the checkers and generators of the argument and result contracts:

Implementing $(A \to B)$.check	requires	A.generate and B.check
Implementing $(A \to B)$.generate	requires	A.check and B.generate

In particular, the generator for $(A \to B)$ produces a function that first invokes A.check on its argument x and signals failure if the check fails. Then it uses B.generate (potentially exploiting dependencies as already explained) to create an output value y for the function. Depending on the programmer's choice between pure or impure functions, this value can be memoized.

2.3 Guided Random Testing

Often it is useful to modify the random generator of a contract depending on the situation in which it is used. Consider the function f in Program 3. For some input constellations, the function returns a string instead of the expected boolean. It is unlikely that testing f with random integers discovers the contract violation, because the probability to return "true" is extremely small ($\approx 2^{-32}$) if the inputs are generated with a uniformly distributed random generator for x and y. To this end, JSConTest supports *guided random testing*.

The guided random generator is triggered by the annotation @numbers in the contract of Program 4. Its use increases the probability of finding a counterexample for f's contract to $p \approx \frac{1}{16}$. It relies on a simple static analysis which collects the number literals in the body of the function. Based on these numbers, JSConTest generates integer values either by using the random generator or by generating an expression tree (both cases with a probability of 0.5). The nodes in the expression trees are picked randomly and correspond to the basic arithmetic operations ($+$, $-$, $*$, $/$). The leaves are picked from the set $\{0, 1\}$ and the collected numbers (in this case, from $\{0, 1, 2, 10\}$) or from randomly generated integers (each case with a probability of 0.5). The algorithm chooses randomly between generating either a node or a leaf.

Program 5 contains the algorithm used to generate expression trees. To ensure termination of the algorithm, the probability to generate a node (the parameter p) is

Program 4 JavaScript - Contract for a predicate of two integers (annotated).

```
1 /*c (int@numbers,int@numbers) → bool */
2 function f(x,y) {
3   return (x != y && 2 * x == x + 10) ? "true" : false;
4 }
```

Program 5 Random Generator for Trees.

```
function rInt(constl)                    ▷ constl is a list of functions to create constants
  return genTree(constl, [+, −, ∗, \])
function genTree(ll,nl, p = 0.5)                          ▷ default value of p is 0.5
                          ▷ ll and nl are lists of functions which create leaves and notes
  if random() < p then
    genNode ← pick(nl)                          ▷ randomly picks a node generator
    arity ← genNode.arity
    params ← [1..arity].map(λi. genTree(ll, nl, p ∗ 0.9))
    return genNode(params[1], . . . , params[arity])
  else
    return pick(ll)()          ▷ randomly picks a leaf generator and executes it
```

reduced for each recursive call.[1] For the depth i the probability to generate a node is $p(node, i + 1) = 0.5 * 0.9^i$.

JSConTest generates random objects using an algorithm similar to genTree. Hence, in the default configuration JSConTest only generates tree structures for contracts. But it is easy to enable the algorithm to generate acyclic object graphs by including a function pickObject, say, that randomly picks a previously created object in the list of leaf generators. To support this operation, each leaf and node constructor is wrapped by a function that stores the created object in a global cache. From this cache pickObject can make its pick.

The approach to generate numbers from random expression trees turns out to work well even in complicated situations like in Program 6. This program contains a contract violation which is guarded by a diophantine equation. The guided random generator finds a solution to the equation in seconds.

The same approach works for generating strings and object labels. Attaching @strings to a string contract collects the string constants in the function body and makes them preferred outputs of the generator. The annotation @labels modifies the random generator for objects by collecting all property names from the function body. For example, finding the error in Program 7 with a uniformly distributed random generator for objects is hopeless. The annotation raises the probability to generate an object with properties p and quest, because these two properties are contained in the function body of h. JSConTest typically finds this defect in less than 10 test cases.

In general, it might be argued that even a very simple static analysis would be able to discover the problems in these example programs. However, a user of the static analysis would have to further investigate the program to check if the problem reported is a false positive. For example, it could be the case that the condition in

[1] The actual implementation is a slightly more complicated because the probability passed to the recursive call depends also on the arity of GenNode. We omit this technical detail here for simplicity.

Program 6 JavaScript - Complicated conditions.

```
1 /*c (int@numbers,int@numbers,int@numbers) → bool */
2 function DiophantineEquation(x,y,z) {
3   return ((x*3+5 == y*5+4) && (x*2−1 == z*9 − 1)) ? "true" : false;
4 }
```

Program 7 JavaScript - Object access.

```
1 /*c obj@labels → bool */
2 function h(x) {
3   return (x && x.p && x.quest) ? "true" : false;
4 }
```

Program 6 were not satisfiable so that the true-branch were dead code. In contrast, if JSConTest reports a problem, then it includes the test case exhibiting the problem as evidence.

2.4 Access Permission Contracts

Up to this point, we only discussed contracts specifying the functional behavior of an operation. However, many JavaScript operations perform side effects, which are not covered by a functional specification. To this end, JSConTest supports *access permission contracts* to specify the side effects of an operation.

An access permission contract consists of a set of path expressions, which start with variable name followed by a restricted regular expression over property names. A path expression specifies which properties of the object bound to the variable are readable or writeable. Figure 2 defines the syntax of access permission contracts.

As an example, consider an operation on binary trees. A node of the tree is implemented by an object that contains properties left, right, and value, where left and right contain objects representing the subtrees (or null) and value contains some value associated with the tree node. A method that computes the height and the balance[2] (Program 8) for all nodes of the tree may store information in a property balance for later reuse. The method only writes to balance, while it reads left and right. The contract with [this./left|right/*.balance] is a concise description of the side effect of this method. The subpart /left|right/* expresses that an arbitrary sequence of left and right properties may be traversed. An access permission grants write access to all paths that completely match a path expression. It also grants read access to all prefixes of a write path. Hence, reading the paths this, this.left, this.left.right, and so on is granted by the access permission.

The typing part of the contract for heightAndBalance states that the function accepts a tree and returns an integer. The tree contract js:tree is a user-defined contract that generates trees using the algorithm shown in Program 5 parameterized with suitable node and leaf constructors.

Program 9 contains an operation that cleans up a tree by deleting all temporary values stored in it. The type signature states that the function accepts a tree but does

[2]The balance of a binary tree is the difference between the height of the left and the right child.

r	$\in$	regular expressions	
ap	$::=$	`with` L `except` L	access permissions
L	$::=$	`[`P^{+}`]` $\mid$ `js:`x	access paths
P	$::=$	x`.`$(Pr)^{*}$ $\mid r$	path expressions
Pr	$::=$	$prop \mid r \mid$ `?` $\mid$ `*` $\mid Pr*$	properties, property sets

Figure 2 – Syntax of access permission contracts.

Program 8 JavaScript - Height and balance of a binary tree.

```
1 /*c js:tree.() → int with [this./left|right/*.balance] */
2 function heightAndBalance() {
3   var lh = this.left ? heightAndBalance.call(this.left) : 0;
4   var rh = this.right ? heightAndBalance.call(this.right) : 0;
5   this.balance = lh − rh;
6   return max(lh, rh) + 1;
7 }
```

Program 9 JavaScript - Clean a binary tree.

```
1 /*c js:tree.() → undf with [this./left|right/*.?] except [this./left|right/*.value?.←] */
2 function cleanUp() {
3   for (var property in this) {
4     if (property in {left: 0, right: 0}) {
5       cleanUp.call(this[property]);
6     } else if (property !== 'value') {
7         delete this[property];
8     }
9   }
10 }
```

not return any value, expressed by the type contract undf. Its access permission grants write access to all properties in the tree except left, right, and value. It is constructed from two parts. The first part grants write access to all properties in the tree, while the second part forbids writing to the properties left, right and value. The question mark behind the property name value makes it optional. The operation may delete all of the intermediate values stored in the tree, but it can neither change the structure of the tree nor the value of the nodes. The modifier ← at the end of the contract indicates that only the write permission should be restricted (see Section 3).

2.5 Working with JSConTest

JSConTest is designed to help programmers during the implementation, testing, and maintenance phases. We sketch two likely application scenarios and give an impression of working with the system in practice.

2.5.1 Test-Driven Development

In test-driven development the programmer writes test cases for an operation in lieu of a specification before starting to code it. Then, the body of the operation is constructed while trying to make the test cases succeed. Further test cases are added and the program is reworked and refactored until the functionality is complete.

Using JSConTest, the programmer would write a type contract to fix the interface of the operation before constructing the test cases. Then, test-driven development proceeds as usual. An an additional benefit, JSConTest can perform random testing of the type contract along with the tests that check the implemented functionality. Hence, test cases that just perform sanity checks can safely be omitted from the test suite.

Ideally, the programmer prescriptively constructs contracts with access permissions from the start. They can be used to enforce visibility restrictions like Java's private and protected in the context of a scripting language. If a test case requires extended access permissions, then the change to the contract serves as documentation for this fact. In any case, the programmer becomes aware of the failing access permission contract, which plays the role of a regression test.

Alternatively, the programmer can ignore the access permissions initially and employ our inference algorithm (see Section 4) to have them automatically computed from the code during the test runs. While this approach is not as disciplined as the prescriptive approach, it quickly provides meaningful documentation for an operation. If the contract is regularly augmented with the inferred access permission, then it also aids in regression testing.

2.5.2 Maintenance

This is the scenario already sketched in the introduction. In the best case, the system in maintenance has already been developed using JSConTest. In this case, the programmer can proceed as indicated in Subsubsection 2.5.1: for new functionality, state a test case and work on the code, using the contracts and access permissions as part of the regression test suite.

Access permissions are also helpful for tracking down software defects. As the access permissions clearly state dependencies on the program state, the operations that must be debugged or otherwise examined can be narrowed down more quickly.

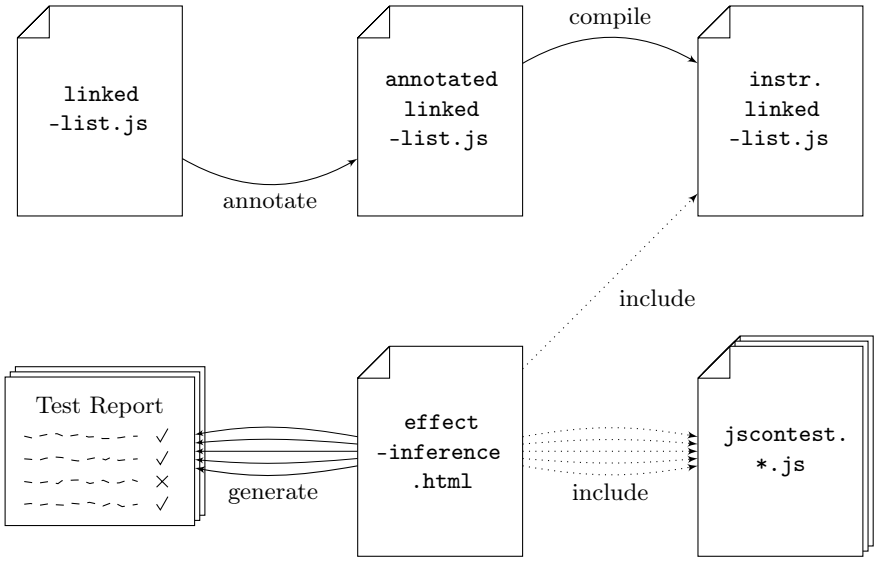

Figure 3 – Workflow.

After the defect has been narrowed down to a failing test case, then the permissions of a suspicious operation can be restricted to test if it is indeed the cause of the defect.

If the system does not (or not fully) employ contracts, then they can be gradually introduced to operations that are suspected to contribute to a defect. As before, the already existing contracts aid during regression testing when changing the code.

2.5.3 JSConTest Workflow

Figure 3 illustrates the use of JSConTest. Initially, the unit under test is annotated with contract specifications. After annotation, the resulting source file (Figure 3, annotated linked−list.js) is passed to the JSConTest compiler. The compiler transforms the program into an instrumented version (instr. linked−list.js) and links it into a runnable HTML template that includes the rest of the JSConTest framework. Executing this template in a browser generates the test cases from the contracts and produces a test report with concrete evidence in case of errors.

The resulting test report documents which of the contracts are fulfilled by the unit under test. Depending on the parameters passed to the compiler, the instrumented code may collect further run-time data, for instance, what properties are accessed during test execution. There are also event hooks to include user-specific functionality. For example, the effect inference algorithm (see Section 4) is implemented in JavaScript as such a hook.

3 Access Paths

To develop the inference algorithm for access contracts and prove its soundness, we formalize the notions of access paths and access permissions. An access path π is a list of property names (Figure 4). An access path is classified as a read path or a write path by writing $\mathbf{R}(\pi)$ or $\mathbf{W}(\pi)$. An access permission a is either empty 0, a path expression b, the union of two access permissions $a + a$, or the restriction of one

$$
\begin{array}{lcll}
p & \in & \mathit{Prop} & \text{property names} \\
\pi & ::= & \varepsilon \mid p.\pi & \text{access paths} \\
\gamma & ::= & \mathbf{R} \mid \mathbf{W} \mid \mathbf{N}_r \mid \mathbf{N}_w & \text{access classifiers} \\
\kappa & ::= & \gamma(\pi) & \text{classified access paths} \\
 & & & \\
P & \subseteq & \mathit{Prop} & \text{sets of property names} \\
b & ::= & \varepsilon \mid P.b \mid P^*.b & \text{path expressions} \\
a & ::= & 0 \mid b \mid a + a \mid a - a & \text{access permissions}
\end{array}
$$

Figure 4 – Syntax of access paths and access permissions.

$$
\mathbf{W}(\varepsilon) \prec \varepsilon \qquad \mathbf{R}(\varepsilon) \prec b \qquad \mathbf{N}_r(\pi) \prec \varepsilon \qquad \mathbf{N}_w(\varepsilon) \prec \emptyset.\varepsilon \qquad \mathbf{N}_w(\varepsilon) \prec \varepsilon
$$

$$
\frac{\gamma(\pi) \prec b \qquad p \in P}{\gamma(p.\pi) \prec P.b} \qquad \frac{\gamma(\pi) \prec b}{\gamma(\pi) \prec P^*.b} \qquad \frac{\gamma(\pi) \prec P^*.b \qquad p \in P}{\gamma(p.\pi) \prec P^*.b}
$$

$$
\frac{\mathbf{R}(\pi) \prec a_1 \qquad \mathbf{N}_r(\pi) \not\prec a_2}{\mathbf{R}(\pi) \prec a_1 - a_2} \qquad \frac{\mathbf{W}(\pi) \prec a_1 \qquad \mathbf{N}_w(\pi) \not\prec a_2}{\mathbf{W}(\pi) \prec a_1 - a_2}
$$

$$
\frac{\kappa \prec a_1}{\kappa \prec a_1 + a_2} \qquad \frac{\kappa \prec a_2}{\kappa \prec a_1 + a_2} \qquad \frac{(\forall \kappa \in K)\ \kappa \prec a}{K \prec a}
$$

Figure 5 – Matching paths with access permissions.

access permission by another $a - a$. A path expression b is a list where each element is either P or P^*, where P is a set of properties.

If Π is a set of paths, we say that Π is *prefix-closed*, if $\pi.p \in \Pi$ implies $\pi \in \Pi$. A pair (Π, Π') of sets of paths is *prefix-accessible*, if $\forall \pi.p \in \Pi' : \pi \in \Pi$ and Π is prefix-closed.

For an access permission a, the judgment $\kappa \prec a$ holds if the classified access path κ *matches* the access permission a. The derivation of a judgment for a restriction may introduce two further classifiers, $\mathbf{N}_r$ and $\mathbf{N}_w$. Figure 5 contains the inference rules for $\kappa \prec a$. Essentially, a single path step P in a permission is matched by a corresponding property $p \in P$ in the path. An iterated path step P^* is matched by a sequence of properties from P in the path. The five axioms on top of Figure 5 implement the different treatments of the four kinds of paths. A write path must be matched exactly by the permission, a read path may match any prefix of the permission, and a negative read path only requires that a path prefix is matched by the permission. The latter choice is required for the implementation of the restriction operator $a_1 - a_2$, where the second premise asks for $\mathbf{N}_r(\pi) \not\prec a_2$, that is, there should be no derivation of $\mathbf{N}_r(\pi) \prec a_2$. To remove a write permission from an access permission without influencing the corresponding read permission, $\mathbf{N}_w(\pi)$ also matches the empty set. This definition of the matching relation enforces prefix-accessibility for the read and write languages.

The rules for restriction may be surprising because of the preconditions $\mathbf{N}_r(\pi) \not\prec a_2$ and $\mathbf{N}_w(\pi) \not\prec a_2$, respectively. For the read case, this choice guarantees prefix-closedness: If a path restriction removes path π, then it should also remove all paths

of the form $\pi.\pi'$ because they are no longer reachable. The premise $\mathbf{N}_r(\pi) \not\prec a$ ensures exactly this implication. The condition $\mathbf{R}(\pi) \not\prec a$, on the other hand, removes all paths that match the access permission a, namely all prefixes of π.

The restriction of write accesses is best explained with an example. The base case for a negative write access permission contains two rules, $\mathbf{N}_w(\varepsilon) \prec \varepsilon$ and $\mathbf{N}_w(\varepsilon) \prec \emptyset.\varepsilon$. To motivate them, consider two example access permission $a_1 = \mathtt{w}.^* - \mathtt{w.l}$ and $a_2 = \mathtt{w}.^* - \mathtt{w.l}.@$. The first access permission a_1 removes both, read and write access to the path `w.l`, because it holds $\mathbf{N}_r(\mathtt{w.l}) \prec \mathtt{w.l}$ and $\mathbf{N}_w(\mathtt{w.l}) \prec \mathtt{w.l}$.

The second access permission a_2 removes the permission to write to the path `w.l`, but it allows reading from `w.l`, because it holds $\mathbf{N}_r(\mathtt{w.l}) \not\prec \mathtt{w.l}.\emptyset$ and $\mathbf{N}_w(\mathtt{w.l}) \prec \mathtt{w.l}.\emptyset$.

Define the *read language* of a contract a as $L_r(a) = \{\pi \mid \mathbf{R}(\pi) \prec a\}$ and its *write language* $L_w(a) = \{\pi \mid \mathbf{W}(\pi) \prec a\}$.

Lemma 3.1. *For each a, the read language $L_r(a)$ is prefix-closed.*

Proof. By induction on the definition of $\prec$ using the inductive hypothesis $\forall \pi, \pi' : (\mathbf{R}(\pi.\pi') \prec a \Rightarrow \mathbf{R}(\pi) \prec a) \wedge (\mathbf{N}_r(\pi.\pi') \not\prec a \Rightarrow \mathbf{N}_r(\pi) \not\prec a)$. □

Lemma 3.2. *For each a, the pair of languages $(L_r(a), L_w(a))$ is prefix-accessible.*

Proof. By induction as in the previous lemma. □

The formal access permissions of this section are an abstract representation for the syntax discussed in Subsection 2.4. An access permission for a variable x has the following general form:

$$\mathsf{with}\ [\mathsf{x}.w_1, \ldots, \mathsf{x}.w_n]\ \mathsf{except}\ [\mathsf{x}.e_1, \ldots, \mathsf{x}.e_m] \tag{1}$$

Translated to the formal syntax defined in Figure 2, this permission reads as follows:

$$a = (w_1 + \ldots + w_n) - e_1 - \ldots - e_m\ .$$

and thus the contract (1) grants access to x according to $L_r(a)$ and $L_w(a)$.

4 Inference of Access Permissions

The inference algorithm has the task to generate concise and correct access permissions from a set of classified access paths. These access paths are recorded during the execution of a JavaScrip program, often during test runs. The main challenge is that the inference problem has two trivially correct solutions.

Fact 4.1. *Let $K = \{\gamma_i(\pi_i) \mid i \in I\}$ be any finite set of classified access paths. Define a mapping β from classified access paths to path expressions by setting $\beta(\mathbf{W}(\pi)) := \pi$ and $\beta(\mathbf{R}(\pi)) := \pi.@$. The following two access permissions are certain to match K:*

- $K \prec \sum_i \beta(\gamma_i(\pi_i))$.
- $K \prec *$.

What's the problem with these two solutions? The sum of all classified paths is the smallest solution. It describes the finite regular language which contains exactly the access paths observed during the trial run. For many simple operations, this solution is perfectly adequate. However, if the traced operation processes a recursive data

structure, then this solution is sound but too specific, because the operation under test will be able to produce infinitely many more access paths, given different inputs. On the other hand, the permission $*$ is the largest solution and permits everything. This permission does not convey much information about the program, either.

This dilemma is a restricted version of a well known problem in machine learning. The task that we are looking at is learning a language from a finite set of positive examples, which has been shown to have no solution in a wide class of languages including the regular languages [Gol67, Ang80]. For that reason, we develop a heuristic approach.

4.1 Algorithm

The basic idea of our algorithm is to restrict the solution space suitably. It only considers access permissions of the form $\pi.P^*.\pi'$ where $P \subseteq \mathit{Prop}$ and π' may be empty.[3]

Why is that a useful restriction? If a data structure has only finitely many paths, like a fixed object tree, then all paths should be listed in the permission. If a data structure has infinitely many paths, then there is usually a fixed header part that has to be traversed. This header is followed by a regular structure, which is typically traversed by a loop or recursion in the program, like the link pointers in a list or the left/right pointers in a binary tree. At the end of a path, there may be a data object with some fixed access pattern, again. This kind of arrangement is quite typical for data structure libraries.

The algorithm infers the components of $\pi.P^*.\pi'$ as follows. 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'. The "interesting" prefixes should be chosen such that they traverse the header part of the data structure. Next, the algorithm computes a set of "interesting" suffixes in a similar way. The algorithm bundles the properties which are left after identifying the two interesting parts into the middle part P^* of the access permission.

Our algorithm proceeds in three steps. The first step builds a trie representation for a set of classified access paths. The next step extracts a set of interesting prefixes and suffixes from the set of classified access paths. The final step simplifies the set of contracts to remove some redundancies.

4.1.1 Building the Trie

A trie [Fre60] 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 tails of paths that start with p. If $p\backslash\Pi$ is non-empty, then the trie for Π includes $T(p\backslash\Pi)$ with an edge from r to the root node of $T(p\backslash\Pi)$.

For example, the path set $\Pi_{list} = \{\mathtt{l}, \mathtt{h}, \mathtt{h.d}, \mathtt{h.n}, \mathtt{h.n.d}, \mathtt{h.n.n}, \mathtt{h.n.n.d}\}$ is represented by the trie in Figure 6. The trie can also be considered a finite automaton recognizing the set Π with final states indicated by the double circles in the figure.

[3] If π' is empty and $P = \emptyset$, then $\pi.P^*.\pi'$ reduces to π.

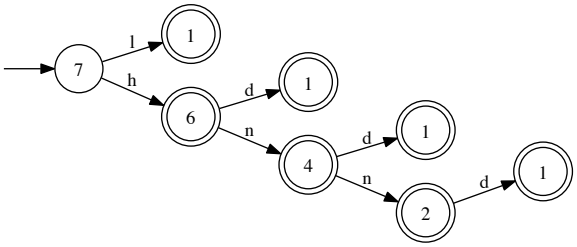

Figure 6 – Example trie.

$$\frac{|\pi| < n}{\pi \in \mathrm{IntPrefixes}_n(\Pi)} \qquad \frac{\pi \in \mathrm{IntPrefixes}_n(\Pi) \qquad \exists \pi' : \pi.q.\pi' \in \Pi \qquad \forall p \exists \pi' : \pi.p.\pi' \in \Pi \Rightarrow \mathrm{Prop}(\pi \setminus \Pi) \neq \mathrm{Prop}(\pi.p \setminus \Pi)}{\pi.q \in \mathrm{IntPrefixes}_n(\Pi)}$$

$$\frac{p \in \mathrm{Prop}(\pi) \qquad \pi \in \Pi}{p \in \mathrm{Prop}(\Pi)} \qquad p \in \mathrm{Prop}(p.\pi) \qquad \frac{p \in \mathrm{Prop}(\pi)}{p \in \mathrm{Prop}(q.\pi)}$$

Figure 7 – Interesting prefixes.

4.1.2 Interesting Prefixes

What prefixes are interesting? Or, alternatively, which properties should end up in the P part of the permission $\pi.P^*.\pi'$? We choose P to contain *loop properties* which are read inside of a loop or recursive function to traverse the heap to arbitrary depth. Loop properties are used multiple times in some paths and never in others. Figure 7 formalizes this idea as the set of interesting prefixes $\mathrm{IntPrefixes}_n(\Pi)$. It is parameterized by n to indicate that all prefixes of length less than n are interesting by default.

The intuition behind this inductive definition is to only extend an interesting prefix if the traversal of each single property results in a change of the set of reachable properties. Here, a property is reachable if it occurs in a set of paths (cf. $p \in \mathrm{Prop}(\Pi)$). Once a loop property is reached, the set of reachable properties remains stable. As $n = 1$ works nicely in practice, we define the shortcut $\mathrm{IntPrefixes}(\Pi) := \mathrm{IntPrefixes}_1(\Pi)$.

In our running example, a good access permission for the set of paths Π_{list} is $\mathtt{l}, \mathtt{h}.\mathtt{n}^*.\mathtt{d}$. Therefore, the set of interesting prefixes should contain `l` and `h`, but no paths containing `n`. The definition of interesting prefixes yields the expected result:

$$\mathrm{IntPrefixes}(\Pi_{list}) = \{\varepsilon, \mathtt{l}, \mathtt{h}\} \quad . \tag{2}$$

By definition, the set $\mathrm{IntPrefixes}_n(\Pi)$ is prefix-closed. For read paths, prefix-closedness results in redundant information because read permissions are also closed under prefix. Hence, we compute the prefix reduct by removing all paths that are proper prefixes of other paths.

$$\mathrm{Reduct}(\Pi) = \{\pi \in \Pi \mid (\forall p)\pi.p \notin \Pi\}$$

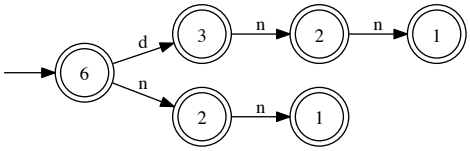

Figure 8 – Reversed suffix trie.

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.

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

4.1.3 Interesting Suffixes

Having come up with candidates for the π component of an access permission $\pi.P^*.\pi'$, we now need to compute the part $P^*.\pi'$, for each such π. First, we compute for each interesting prefix its set of suffixes. As $\mathrm{Reduct}(\mathrm{IntPrefixes}(\Pi_{list})) = \{\mathtt{l}, \mathtt{h}\}$, the sets of suffixes in our running example are:

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

For each of these sets, we 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,

$$\mathrm{Suffixes}(\Sigma) = \overleftarrow{\mathrm{IntPrefixes}_2(\overleftarrow{\Sigma})}$$

where $\overleftarrow{\Sigma} = \{\overleftarrow{\pi} \mid \pi \in \Sigma\}$ and $\overleftarrow{\pi}$ is the reverse of a path π. In this case, we choose $l = 2$ to ensure that all suffixes with a length of 1 are added to the set of interesting suffixes. The intuition behind this choice is that the last few properties in a path access the actual data and thus should be mentioned explicitly in the path expression. As an example, consider a recursive function that traverses a binary search tree, following pointers l and r. The actual data is stored in another property, say, v. In such a case, the path expression $\{\mathtt{l}, \mathtt{r}\}^*.\mathtt{v}$ expresses the access pattern better than $\{\mathtt{l}, \mathtt{r}, \mathtt{v}\}^*$.

Going back to the example, Figure 8 shows the trie containing the reversed suffixes of $\mathtt{h}\backslash\Pi_{list}$. From this trie, it is easy to see that the interesting suffixes of $\mathtt{h}\backslash\Pi_{list}$ are $\{\varepsilon, \mathtt{d}, \mathtt{n}\}$, whereas there is only one respective suffix of $\mathtt{l}\backslash\Pi_{list}$, namely ε.

4.1.4 Extracting the Loop Properties

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 with respect to the interesting suffixes 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

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

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

Figure 9 – Loop properties

The middle language is described by the set P of properties that occur in it.

For the running example, Figure 9 contains a table with the four cases to consider. It displays the computations in the left column and the resulting access permissions in the right column. This result is not entirely satisfactory because it contains redundancies. The path expression $\mathtt{h}.\{\mathtt{n},\mathtt{d}\}^*$ clearly subsumes $\mathtt{h}.\mathtt{n}^*.\mathtt{d}$ and $\mathtt{h}.\mathtt{n}^*.\mathtt{n}$, but the latter two permissions are more informative and thus preferable. Unfortunately, even together, they do not cover the access path $\mathtt{h}$, which is only covered by $\mathtt{h}.\{\mathtt{n},\mathtt{d}\}^*$.

The source of the problem is that the set $\{\varepsilon,\mathtt{d},\mathtt{n}\}$ 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 to 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 Program 10), the case $(\mathtt{h}\backslash\Pi_{list})$ with suffix ε yields the access permission $\mathtt{h}$. 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 $\{\mathtt{l},\mathtt{h},\mathtt{h}.\mathtt{n}^*.\mathtt{d},\mathtt{h}.\mathtt{n}^*.\mathtt{n}\}$.

4.1.5 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 $\{\mathtt{l},\mathtt{h},\mathtt{h}.\mathtt{n}^*.\mathtt{d},\mathtt{h}.\mathtt{n}^*.\mathtt{n}\}$. As this set only contains read permissions, which are closed under prefix, it follows that permission $\mathtt{h}$ is subsumed by $\mathtt{h}.\mathtt{n}^*.\mathtt{d}$ and $\mathtt{h}.\mathtt{n}^*.\mathtt{n}$ so that the result is equivalent to (the simpler set) $\{\mathtt{l},\mathtt{h}.\mathtt{n}^*.\mathtt{d},\mathtt{h}.\mathtt{n}^*.\mathtt{n}\}$.

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

$$
\textsc{SubEps}\ \vdash \varepsilon \subseteq b
\qquad
\textsc{SubPPS}\ \frac{\vdash b \subseteq P'^*.b' \qquad P \subseteq P'}{\vdash P.b \subseteq P'^*.b'}
\qquad
\textsc{SubSZero}\ \frac{\vdash P.b \subseteq b'}{\vdash P.b \subseteq P'^*.b'}
\qquad
\textsc{SubPSPS}\ \frac{\vdash b \subseteq b' \qquad P \subseteq P'}{\vdash P^*.b \subseteq P'^*.b'}
$$

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

Lemma 4.2. *If* $\mathbf{R}(\pi) \prec b$ *and* $\vdash b \subseteq b'$*, then* $\mathbf{R}(\pi) \prec b'$.

Proof. By induction over the sum of the length of the derivation of $\prec$, the length of the derivation of $\subseteq$ and the length of the path π.

□

Program 10 Building access permissions.

```
function BuildPermissions(π, Σ)
                          ▷ π is a prefix, Σ is the corresponding suffix language
    R ← ∅                                          ▷ result set of path expressions
    Σ0 = Suffixes(Σ)                               ▷ set of interesting suffixes of Σ
    for all σ ∈ Σ0 do
        if σ is proper suffix of an element of Σ0 then
            if σ ∈ Σ then
                R = R ∪ {π.σ}
        else
            let P = Prop(Σ/σ) in
            if P = ∅ then                          ▷ middle language is empty
                R = R ∪ {π.σ}
            else
                R = R ∪ {π.P*.σ}
    return R

function PermissionsFromPathSet(Π0, Π)
                                ▷ Π0 set of prefixes of Π, sampled set of paths
    R ← ∅                                          ▷ result set of path expressions
    for all π ∈ Π0 do
        R = R ∪ BuildPermissions(π, π\Π)
    return R
```

Program 11 Simplification.

```
function Simplify(R, W)          ▷ sets of path permissions (Reading, Writing)
    while (∃b, b′) b ∈ R ∧ (b′ ∈ R ∧ b ≠ b′ ∨ b′ ∈ W)∧ ⊢ b ⊆ b′ do
        R ← R − {b}
    return (R,W)
```

Given this relation, simplification removes all read path permissions that are subsumed by other (read or write) path permissions as specified in Program 11. In the example, h can be removed from the read path permissions because $\vdash h \subseteq h.n^*.d$.

This subsumption relation is very simple. It can be replaced by any other sound subsumption relation if more precision is required. It is sufficient for all examples we considered because the shape of the subsuming permissions is always similar.

4.1.6 Putting it Together

Program 12 summarizes the overall algorithm as it has been presented so far. In our previous work [HT11] the inference algorithm has some parameters because it was based on a definition of interesting prefixes and interesting suffixes that depends on an integer value. The algorithm in this work yields satisfactory results without requiring any kind of tuning parameters.

Program 12 a

```
function Main(Π^r, Π^w)                          ▷ Π^r read paths, Π^w write paths
    Π^r_0 ← IntPrefixes(Π^r)                     ▷ interesting prefixes of Π^r
    Π^w_0 ← IntPrefixes(Π^w)                     ▷ interesting prefixes of Π^w
    R ← PermissionsFromPathSet(Reduct(Π^r_0), Π^r)
    W ← PermissionsFromPathSet(ReductW(Π^w_0), Π^w)
    (R, W) ←Simplify(R, W)
    return R.@ + W
```

lgorithm]Overall inference]algorithm.

4.2 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 phase, building the trie, is trivially sound. The third simplification phase is sound by lemma 4.2. 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 $\Pi_0 = \text{Reduct}(\text{IntPrefixes}_{l,d}(\Pi))$ 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'.

- *Case* ($\exists^{=1}\pi_0 \in \Pi_0 : \pi_0$ is prefix of π):

  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 $\pi_1 \prec P^*.\sigma$.

  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.

  Otherwise the middle language is $P = \text{Prop}(\Sigma/\sigma)$. There are two cases:

  - *Case* $P = \emptyset$: It holds that $\Sigma/\sigma = \{\varepsilon\}$[4]. Therefore, $\sigma = \pi_1$, a contradiction.
  - *Case* $P \neq \emptyset$: It holds that $\Sigma/\sigma \neq \{\varepsilon\}$. We conclude that $\pi_1 \prec P^*.\sigma$, a contradiction.

  Hence, all cases are matched.

- *Case* ($\exists\pi' \in \Pi_0 : \pi$ is prefix of π'): In this case, π' will be prefix of an access path $\pi'.b$ with $\pi \prec \pi'.b$.

4.3 Special Cases

There are two special cases of property accesses that lead to trie nodes with an extremely high branching degree. The first case is an object that 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

[4] $\Sigma/\sigma = \emptyset$ is not possible for $\Sigma \neq \emptyset$.

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 also leads to nodes with high branching degrees, but it 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, merges these subtries, and relabels the remaining edge to the merged successor trie with a wildcard pseudo property name ?. The threshold for the high branching degree is user-configurable and defaults to 20.

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.

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

The contract js:ll describes the receiver object. It refers to a user-provided JavaScript function that generates and checks a certain kind of lists, top stands for any value, and undf is the undefined value, which is returned when no return value is specified.

To run the effect inference, it is sufficient to add an empty effect specification to the contract as in /*c js:ll.(top) → undf 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 the invocation of the function. Applying the JSConTest compiler to these contracts results in instrumented code that monitors all property accesses.

When the compiled code executes it records thousands of property accesses which violate the empty effect annotation. From this raw data, our effect inference computes concise access permissions using the algorithm in Section 4. 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.

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

The computed effect for remove is also interesting:

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

The function remove deletes a value from the list. To this end, it compares this value with all data properties reachable via _head and a sequence of next properties, as indicated with the first path expression. 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.[6] 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 a similar example implementing binary search trees. For these two examples, the algorithm infers a precise effect annotation.

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 [HT11]. This example uncovered a number of new points for our inference algorithm, in particular, that a special treatment for arrays and objects used as hash tables is required (see Subsubsection 2.5.3). This treatment is also covered in a micro benchmark in the webpage.

The Deltablue benchmark from the Google V8 benchmark suite is our largest example. We annotate the source code with contracts and let the inference algorithm compute the access permissions. For example, the remove method of an ordered collection in the Deltablue benchmark has the following contract:

```
1 OrderedCollection.prototype.remove = function(elm)
2     /*c js:oc.(obj) → undf with [this.elms.length.@,this.elms.#] */
3     { ... }
```

The occurrence of the ♯ pseudo property reveals that the ordered collection is implemented by an array elms. The access to its length property is read-only, because the JavaScript engine automatically adjusts the length property of arrays when they grow or shrink. Our implementation does not catch this internal write operation. An implementation in the browser would catch it, but it is not clear if that would be useful.

A manual inspection of the access permissions computed by our algorithm indicates that the contracts are generally informative and precise descriptions of the functions' side effects. For some of the functions, we had developed access permissions by hand before the inference algorithm was available. To our surprise, the inferred access permissions turn out to be better than the hand-written ones in some cases. The reason is that the human annotator did not always distinguish between read-only and read/write accesses and sometimes forgot to add the @ at the end of a path expression. The annotator also disliked path expressions with long paths and shortened the path by stuffing more into the loop properties than strictly necessary. The Deltablue benchmark makes heavy use of arrays to store collections and it does not rely on list of tree-like structure. So there is no recursive traversal and the loop properties introduced by the human annotator introduce a lot of imprecision.

In our previous work [HT11], we used the same set of examples except the Deltablue benchmark. The evaluation in that paper was based on the old inference algorithm

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

with interactive parameter tuning. The new algorithm computes essentially the same access permissions like the old one, but without human intervention. We conclude that the new definition of an interesting prefix in this work yields a much improved heuristic for the inference of access permissions.

One key point observed in the case studies is that it is important for the inference to run with tight type contracts and/or a test suite with high coverage. Tightness of the contract is required because a loose contract causes the generation of entirely random test cases. It is unlikely that these random test cases completely explore the access path pattern of a function. Similarly, 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 access traces. The evaluation back end and the inference algorithm would remain the same.

6 Related Work

Purely manual testing per se does not guarantee any kind of coverage criterion and its effectiveness depends highly on the experience of the tester and on the system of the chosen approach to testing. Hence, manual testing should be backed up by further kinds of testing. Random testing [BM83, Ham94] is one promising candidate, which is surprisingly effective [FM00], but which does not give guarantees with respect to coverage [OH96, DN84]. However, there are a number of approaches and tools that support random testing and that employ various means for improving coverage.

JCrasher [CS04] is a black-box random testing tool for the Java programming language. It analyzes a set of classes with the goal to find a crashing program fragment involving methods of these classes. It constructs fragments by applying methods with random parameters to randomly constructed objects and then using these objects as a basis for randomly generating further method calls. There is no further specification of contract needed for JCrasher as the failure criterion is a program crash.

In contrast, JSConTest can test against user-specified contracts and can also do run-time monitoring. JSConTest improves coverage by performing a limited amount of white-box testing by collecting constants from the code to perform guided testing.

The QuickCheck library [CH00] for Haskell, a purely functional programming language, enables the statement of properties of program constructs, which are then automatically tested. Test cases are randomly generated from the types of the variables in properties. Additionally, programmers can specify their own generators. In contrast, JSConTest derives its test cases from contracts, which can be more expressive than types, and it is only geared to test contracts (although it could be extended to test properties as well). JSConTest handles test case generation for imperative JavaScript objects, which go beyond functions and primitive data. Another difference is that QuickCheck performs pure black-box testing whereas JSConTest's inclusion of program analysis information places it on the brink to white-box testing.

DoubleCheck [Eas09] is an adaptation of QuickCheck to the ACL2 language implemented in the PLT programming environment[FFF+97][7]. It is used as a verification aid to generate counterexamples for properties of programs that ACL2 cannot prove

[7]The DrScheme teaching languages also provide QuickCheck-style testing of contracts.

right away. The idea is to restate these properties guided by the counterexamples. PLT-Redex also comes with a random testing facility that has detected errors in semantics specifications [KF09].

RUTE-J [AHLL06] is a framework that enables writing unit tests for Java that make use of some portion of randomness. It can randomize a list of method calls as well as input data and it performs minimization of failing test cases.

Randoop [Pac09] is a tool for directed random testing of Java classes. It generates test cases in a similar way as JCrasher, but additionally uses the test outcomes as feedback to avoid creating useless or outright erroneous tests.

Similarly, the ARTOO system [CLOM08] performs adaptive random testing for Eiffel. It adapts previous ideas from the ART approach [CKMT10] to an object-oriented setting. Its underlying idea is that tests are more effective if they evenly cover the parameter space of the method under test. Its execution requires a distance metric on the input values.

A highly effective approach to randomized testing is the DART system [GKS05]. It performs what has been coined concolic testing: it combines running concrete test cases with symbolic execution of the underlying code. Guided by the outcome of concrete test cases it generates symbolic predicates for the branches taken in the computation. It employs theorem proving to systematically falsify these predicates and thus attempts to cover all branch alternatives, which is often successful. JSConTest is inspired by this system, but relies on a much more lightweight approach (collecting constants), which requires a larger number of test cases, for increasing the coverage.

A different approach to generating test cases is bounded exhaustive testing, which systematically enumerates all inputs below a certain size threshold. This approach is implemented, for example, in the Smallcheck system for Haskell [RNL08] and also in the Korat system for Java [BKM02]. The idea here is that counterexamples are usually small and that the exhaustive tests give some guarantees, at least for finite structures like functions over finite domains. This approach is complementary to the random testing approach presently chosen by JSConTest.

There are a number of JavaScript testing frameworks, for example, JSUnit[8], JsTester[9], FireUnit[10], JSCoverage[11], JSMock[12], and rhinounit[13]. However, these frameworks are in the tradition of unit testing frameworks like JUnit[14]. Their focus is on automating the execution of unit tests, but not on the creation of these tests. In contrast, JSConTest only requires the manual construction of interface specifications. Also, JSConTest is currently restricted to functional testing of the JavaScript code, it does not test the interactive behavior (GUI testing), nor the interface to web services via XmlHTTPRequest. These extensions are left to future work.

JSConTest is inspired by, but complementary to work on type analysis for JavaScript [JMT09, HT10b, HT09]. The focus of these works is to determine the type safety of JavaScript programs by static analysis (abstract interpretation and constraint-based analysis, respectively). Neither work supports type specifications, nor test case generation.

Static effect analysis in programming languages has some history already. Initial

[8] http://www.jsunit.net/
[9] http://jstester.sourceforge.net/
[10] http://fireunit.org/
[11] http://siliconforks.com/jscoverage/
[12] http://jsmock.sourceforge.net/
[13] http://code.google.com/p/rhinounit/
[14] http://junit.org/

efforts by Gifford and Lucassen [GL86] 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 [TJ94, TT97]. Such an effect describes reading, writing, and allocation in terms of regions. An important goal in these works is automatic effect inference [BT01], because regions and effects are deemed as analysis results in a phase of a compiler.

Path related properties are investigated by Deutsch [Deu92] 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 [GB99] 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 [Ska05] also considers effects of object-oriented programs, but his effects are traces of operations. He proves 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. 4.1 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 [Gol67, Ang80]. Clearly, the class of regular languages qualifies. Better results can be achieved by restricting the view to "simple examples" [Den01] or to more restricted kinds of languages [FOC98].

Transformation of JavaScript programs is a well-studied topic in work on enforcing and analyzing security properties. For example, Maffeis and coworkers [MMT09] achieve isolation properties between mashed-up scripts using filters, rewriting, and wrapping. Chugh and coworkers [CMJL09] present (among others) a dynamic information flow analysis based on wholesale rewriting. Yu and coworkers [YCIS07] perform rewriting guided by a security policy. BrowserShield [RDW+07] relies on similar techniques to attain safety. As detailed elsewhere [HBT12], 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

JSConTest is a JavaScript tool for program maintenance and understanding. It enables a programmer to gradually introduce partial type and effect specifications in a program. Both specifications take the form of contracts that are monitored at run time. An effect specification restricts the paths that may be accessed from the free variables in scope.

JSConTest contains an effect inference algorithm that computes access permissions from sample runs of the program. The resulting effect specifications are concise and

precise. They are as good as manually determined ones and sometimes better. Thus, effect inference appears to be a useful tool to analyze JavaScript programs and enhance their contracts with effect information.

The current JSConTest distribution is available for download along with all examples from this paper.[15]

References

[AHLL06] James H. Andrews, Susmita Haldar, Yong Lei, and Felix Chun Hang Li. Tool support for randomized unit testing. In *Proceedings of the 1st International Workshop on Random testing*, RT '06, pages 36–45. ACM, 2006. doi:10.1145/1145735.1145741.

[Ang80] Dana Angluin. Inductive inference of formal languages from positive data. *Information and Control*, 45(2):117–135, 1980. doi:10.1016/S0019-9958(80)90285-5.

[BFN+09] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: robust, concurrent, extensible scripting on the jvm. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 117–136. ACM, 2009. doi:10.1145/1640089.1640098.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 123–133. ACM, 2002. doi:10.1145/566172.566191.

[BM83] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22:229–245, September 1983. doi:10.1147/sj.223.0229.

[BT01] Lars Birkedal and Mads Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, 58:299–392, 2001. doi:10.1016/S0304-3975(00)00025-6.

[CH00] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279. ACM, 2000. doi:10.1145/351240.351266.

[CKMT10] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83:60–66, January 2010. doi:10.1016/j.jss.2009.02.022.

[CLOM08] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: adaptive random testing for object-oriented software. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 71–80. ACM, 2008. doi:10.1145/1368088.1368099.

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

[CMJL09] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 50–62. ACM, 2009. doi:10.1145/1542476.1542483.

[CS04] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for java. *Journal of Software—Practice & Experience*, 34:1025–1050, September 2004. doi:10.1002/spe.602.

[Den01] François Denis. Learning regular languages from simple positive examples. *Journal of Machine Learning*, 44:37–66, July 2001. doi:10.1023/A:1010826628977.

[Deu92] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proc. IEEE International Conference on Computer Languages 1992*, pages 2–13, Oakland, CA, April 1992. IEEE.

[DN84] J. Duran and S. Ntafos. An evaluation of random testing. *Transactions on Software Engineering*, 10(4):438–444, 1984. doi:10.1109/TSE.1984.5010257.

[Eas09] Carl Eastlund. Doublecheck your theorems. In *Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications*, ACL2 '09, pages 42–46. ACM, 2009. doi:10.1145/1637837.1637844.

[FFF+97] Robert B. Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Drscheme: A pedagogic programming environment for scheme. In *Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs: Including a Special Trach on Declarative Programming Languages in Education*, PLILP '97, pages 369–388, London, UK, 1997. Springer-Verlag.

[FFF04] Robert B. Findler, Matthew Flatt, and Matthias Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In Martin Odersky, editor, *Proceedings of the 2004 International Conference on Object-Oriented Technology*, ECOOP'04, pages 614–639. Springer Berlin / Heidelberg, 2004.

[FM00] Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, pages 6–6. USENIX Association, 2000. URL: http://dl.acm.org/citation.cfm?id=1267102.1267108.

[FOC98] Laura Firoiu, Tim Oates, and Paul R. Cohen. Learning deterministic finite automaton with a recurrent neural network. In *Proceedings of the 4th International Colloquium on Grammatical Inference*, pages 90–101, London, UK, 1998. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=645517.655778.

[Fre60] Edward Fredkin. Trie memory. *Communications of the ACM*, 3:490–499, September 1960. doi:10.1145/367390.367400.

[GB99] Aaron Greenhouse and John Boyland. An object-oriented effects system. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pages 205–229, London, UK, 1999. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=646156.679836.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223. ACM, 2005. doi:10.1145/1065010.1065036.

[GL86] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 28–38. ACM, 1986. doi:10.1145/319838.319848.

[Gol67] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967. URL: http://www.isrl.uiuc.edu/~amag/langev/paper/gold67limit.html.

[GSK10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 126–150. Springer-Verlag, 2010. URL: http://dl.acm.org/citation.cfm?id=1883978.1883988.

[Ham94] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[HBT12] Phillip Heidegger, Annette Bieniusa, and Peter Thiemann. Access permission contracts for scripting languages. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 111–122. ACM, 2012. doi:10.1145/2103656.2103671.

[HT09] Phillip Heidegger and Peter Thiemann. Recency types for dynamically-typed object-based languages. In *2009 International Workshop on Foundations of Object-Oriented Languages*, FOOL'09, Savannah, Georgia, USA, 2009. URL: http://www.cs.cmu.edu/~aldrich/FOOL09/heidegger-abstract.html.

[HT10a] Phillip Heidegger and Peter Thiemann. Contract-driven testing of javascript code. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, TOOLS'10, pages 154–172. Springer-Verlag, 2010. URL: http://dl.acm.org/citation.cfm?id=1894386.1894395.

[HT10b] Phillip Heidegger and Peter Thiemann. Recency types for analyzing scripting languages. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 200–224. Springer-Verlag, 2010. URL: http://dl.acm.org/citation.cfm?id=1883978.1883992.

[HT11] Phillip Heidegger and Peter Thiemann. A heuristic approach for computing effects. In *Proceedings of the 49th International Conference on Objects, Models, Components, Patterns*, TOOLS'11, pages 147–162. Springer-Verlag, 2011. URL: http://dl.acm.org/citation.cfm?id=2025896.2025908.

[JMT09] Simon H. Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255. Springer-Verlag, 2009. doi:10.1007/978-3-642-03237-0_17.

[KF09] Casey Klein and Robert Bruce Findler. Randomized testing in PLT Redex. In *Workshop on Scheme and Functional Programming 2009*, Boston, MA, USA, 2009.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[MF09] Jacob Matthews and Robert B. Findler. Operational semantics for multi-language programs. *ACM Transactions on Programming Languages and Systems*, 31:12:1–12:44, April 2009. doi:10.1145/1190215.1190220.

[MMT09] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating javascript with filters, rewriting, and wrappers. In *Proceedings of the 14th European Symposium on Research in Computer Security*, ESORICS'09, pages 505–522. Springer-Verlag, 2009. URL: http://dl.acm.org/citation.cfm?id=1813084.1813126.

[MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[OH96] A. Jefferson Offutt and J. Huffman Hayes. A semantic model of program faults. In *1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 195–200, San Diego, CA, USA, January 1996. doi:10.1145/229000.226317.

[Pac09] Carlos Pacheco. *Directed Random Testing*. Ph.D., MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, USA, June 2009.

[Pey03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[RDW+07] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Transactions on the Web*, 1, September 2007. doi:10.1145/1281480.1281481.

[RNL08] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proceedings of the first ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 37–48. ACM, 2008. doi:10.1145/1411286.1411292.

[Ska05] Christian Skalka. Trace effects and object orientation. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, pages 139–150. ACM, 2005. doi:10.1145/1069774.1069787.

[ST07] Jeremy Siek and Walid Taha. Gradual typing for objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, pages 2–27. Springer-Verlag, 2007. doi:10.1007/978-3-540-73589-2_2.

[THF08] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 395–406. ACM, 2008. doi:10.1145/1328438.1328486.

[TJ94] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111:245–296, June 1994. doi:10.1109/LICS.1992.185530.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132:109–176, February 1997. doi:10.1145/543552.512563.

[WF09] Philip Wadler and Robert B. Findler. Well-typed programs can't be blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 1–16. Springer-Verlag, 2009. doi:10.1007/978-3-642-00590-9_1.

[YCIS07] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 237–249. ACM, 2007. doi:10.1145/1190216.1190252.

About the authors

Phillip Heidegger is a PhD student at the University of Freiburg since 2007.
http://www.informatik.uni-freiburg.de/~linkenhe/.

Peter Thiemann is a full professor at the University of Freiburg and head of the programming languages research group since 1999.
http://www.informatik.uni-freiburg.de/~thiemann.