On Contracts and Sandboxes for JavaScript

Matthias Keil and Peter Thiemann

University of Freiburg, Freiburg, Germany
{keilr,thiemann}@informatik.uni-freiburg.de

Abstract. JavaScript is the language of the web. It is used by more than 89% of all the websites. Most of them rely on third-party libraries for connecting to social networks, feature extensions, or advertisement. Some of these libraries are packaged with the application, but others are loaded at run time from origins of different trustworthiness, sometimes depending on user input. Thus, managing untrusted JavaScript code has become one of the key challenges of present research on JavaScript.
This work is about *TreatJS* and the *TreatJS-Sandbox*.
TreatJS is a language embedded, higher-order contract system for JavaScript which enforces contracts by run-time monitoring. Beyond providing the standard abstractions for building higher-order contracts (base, function, and object contracts), *TreatJS*'s novel contributions are its guarantee of a non-interfering contract execution, its systematic approach to blame assignment, its support for contracts in the style of union and intersection types, and its notion of a parameterized contract scope, which is the building block for composable run-time generated contracts that generalize dependent function contracts.
The *TreatJS-Sandbox* is a language-embedded sandbox for full JavaScript. It enables scripts to run in a configurable degree of isolation with fine-grained access control. It provides a transactional scope in which effects are logged for review by the access control policy. After inspection of the log, effects can be committed to the application state or rolled back.

1 Introduction

We present the design and implementation of *TreatJS*, a language embedded, higher-order contract system for JavaScript which enforces contracts by run-time monitoring. *TreatJS* supports most features of existing systems and a range of novel features that have not been implemented in this combination before. No source code transformation or change in the JavaScript run-time system is required. In particular, *TreatJS* is the first contract system for JavaScript that supports the standard features of contemporary contract systems (embedded contract language, JavaScript in flat contracts, contracts as projections, full interposition using JavaScript proxies) in combination with the following three novel points.

1. *Noninterference.* Contracts are guaranteed not to exert side effects on a contract abiding program execution. A predicate is an arbitrary JavaScript

function, which can access the state of the application program but which cannot change it. An exception thrown by a predicate is not visible to the application program.

2. *Dynamic contract construction.* Contracts can be constructed and composed at run time using contract abstractions *without compromising noninterference.* A contract abstraction may contain arbitrary JavaScript code; it may read from global state and it may maintain encapsulated local state. The latter feature can be used to construct recursive contracts lazily or to remember values from the prestate of a function for checking the postcondition.
3. *New contract operators.* Beyond the standard contract constructors (flat, function, pairs), *TreatJS* supports object, intersection, and union contracts. Furthermore, contracts can be combined arbitrarily with the boolean connectives: conjunction, disjunction, and negation.

2 TreatJS by Example

TreatJS is implemented as a library so that all aspects of a contract can be specified using the full JavaScript language. The library relies on JavaScript proxies to guarantee full interposition for contracts. It further exploits JavaScript's reflective features to run contracts in a sandbox environment, which guarantees that the execution of contract code does not modify the application state. No source code transformation or change in the JavaScript run-time system is required.

In *TreatJS*, contracts are first-class values that can be stored or further composed. They are dormant until they are asserted to a value.

We start out with explaining *TreatJS*'s notation for base contracts and function contracts, and then move on to discuss intersection and union contracts. The implementation of the system is available on the Web. [1].

2.1 Base Contracts

The *base contract* is the fundamental building block for all other contracts. It is defined by a predicate, that is, a function returning a boolean value. In JavaScript, any function can be used as a predicate, because any return value can be converted to boolean. For example, the function *typeOfNumber* can serve as a predicate that checks whether its argument is a number.

```
function typeOfNumber (arg) {
  return (typeof arg) === 'number';
};
```

To create a base contract from such a predicate, we apply the appropriate contract constructor to it.

```
var Num = Contract.Base (typeOfNumber);
```

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

Here, ***Contract*** is the object that encapsulates the *TreatJS* implementation. Its ***assert*** method attaches a contract to a *subject*. Attaching a base contract applies the predicate to the value. If the result is true, ***assert*** returns the original value. Otherwise, ***assert*** signals a contract violation which blames the subject. The following example demonstrates both outcomes.

```
Contract.assert (1, Num); // accepted, returns 1
Contract.assert ('a', Num); // violation, blame subject 'a'
```

Figure 2.1 defines a number of base contracts for later use. Analogous to *Num*, the contracts *Bool* and *Str* check the type of their argument. Contract *Any* is a contract that accepts any value.

```
var Bool = Contract.Base (function (arg) {
  return (typeof arg) === 'boolean';
});
var Str = Contract.Base (function (arg) {
  return (typeof arg) === 'string';
});
var Any = Contract.Base (function (arg) {
  return true;
});
```

Fig. 1. Some utility contracts.

2.2 Function Contracts

While a base contract can specify finitary properties of a function f (like $f(1) = 0$), a *function contract* is needed to specify that a function uniformly maps numbers to booleans. A function contract is built from one or more contracts, zero or more for the arguments and one for the result of the function. Asserting a function contract to a non-function value immediately signals a contract violation. Asserting it to a function creates a wrapper function that asserts the argument contracts to the arguments of each call of the function and the result contract to the return value of each call.

As a running example, we consider the function *plus*, which applies the plus operator + to its arguments and returns the result.

```
function plus(x, y) {
  return (x + y);
}
```

The function contract *PlusNum* restricts a function's argument to a number and asserts that the result is a number.

```
var PlusNum = Contract.AFunction ([Num,Num], Num);
var plusNum = Contract.assert (plus, PlusNum);
```

In general, a JavaScript function has no fixed arity and arguments are passed to a function in a special array like object. Thus, a standard function contract takes two arguments. The first argument is an object contract that maps an argument to a contract. The second argument is a contract for the function's return.

Contract.AFunction is the constructor for a simple function contract that takes an array of contracts for the arguments and a contract for the result of a function call as arguments.

The contracted function accepts any argument that satisfies the *Num* contract. If there is an argument that violates its contract, then the function contract raises an exception that blames the *context*, which is in this case the caller of the function that provides the wrong kind of argument. If the argument is ok, but the result contract fails, then blame is assigned to the *subject* (i.e., the function). Here are some examples that exercise *plusNum* as well as a broken version of it that returns a string.

```
plusNum (1, 2); // accepted, returns true
plusNum ('a', 'b'); // violation, blame context 'a'

function plusBroken (x) {
  return ('' + (x + y));
};
var plusNum2 = Contract.assert (plusBroken, PlusNum);
plusNum2 (1, 2); // violation, blame subject (function)
```

Higher-order contracts are also possible: the argument and result contracts may themselves be function contracts and so on, recursively. As an example, a function that takes a number and a numeric plus function as arguments and returns a number may be specified by the following contract.

```
var Add1Num =
  Contract.AFunction ([Num, PlusNum], Num);

function add1Broken (x, plus) {
  return plus(x, '1');
}
var add1BrokenNum = Contract.assert(add1Broken, Add1Num);
```

Higher-order contracts open up new ways for a function not to fulfill its contract. For example, the function *add1Broken* violates the contract *Add1Num*: the call *add1BrokenNum (1, plus)* signals a violation that blames the subject (the function) because it supplies the wrong kind of argument to its parameter *plus*.

Dually, a function that returns a function may be compromised. Consider the function *getAdd1* that fulfills the contract *GetAdd1*:

```
function getAdd1 (plus) {
  return function add1 (x) {
    return plus(x, 1);
  }
}
var GetAdd1 = Contract.AFunction ([PlusNum],
  Contract.AFunction ([Num], Num));
var add1Num = Contract.assert (getAdd1, GetAdd1) (plus);

add1Num (5); // accepted
add1Num ('a'); // violation, blame context 'a'
```

This example demonstrates that a function call that receives a suitable argument and returns a contract abiding result can still lead to a contract violation if the result is misused.

2.3 Intersection and Union Contracts

In the previous section, the function *plus* was contracted with *PlusNum* to restrict the arguments to numbers. Indeed, *plus* fulfills this contract so that we might say it has type $Num, Num \to Num$. However, the plus operator of JavaScript is overloaded and does not restrict its arguments to numbers: it works just as well if one argument is a string. Thus, *plus* also has type $Str, Str \to Str$.

TreatJS provides a corresponding constructor for intersection contracts.

```
var PlusStr = Contract.AFunction ([Str,Str], Str);
var PlusNumStr = Contract.Intersection (PlusNum, PlusStr);
var plusNumStr = Contract.assert (plus, PlusNumStr);
```

The function *plusNumStr* may be applied to number or string values and promises to return a either a number or a string, depending on its arguments. The context is blamed if it provides the function with an argument that does not fulfill the expectations. The subject is blamed if the function does not fulfill both constituent contracts.

Generally, the subject f of an intersection contract $C \cap D$ must fulfill both contracts C and D. If $C = C_1 \to C_2$ and $D = D_1 \to D_2$ are both function contracts, then any argument to f has to fulfill $C_1 \cup D_1$. Additionally, the context must be prepared to handle a value satisfying $C_2 \cup D_2$. In case the argument contracts overlap (i.e., $C_1 \cap D_1 \neq \emptyset$), then applying the function to an element in their intersection must yield a result that satisfies both, C_2 and D_2. As an example for the case where $C_1 \cap D_1 \neq \emptyset$, consider

```
var StrAny = Contract.AFunction ([Str,Any], Str)
var AnyStr = Contract.AFunction ([Any,Str], Str)
var PlusAny = Contract.Intersection (StrAny, AnyStr);
```

which is another valid typing for the *plus* function.

Just like intersections, union contracts are also applicable to functions. They also mimick union types as closely as possible. That is, a function satisfies a union of two function contracts if it satsifies either of them.

```
var TestPlus = Contract.Union(
  Contract.AFunction([PlusNum], Num),
  Contract.AFunction([PlusStr], Str));
```

A function which satisfied such a contract is either a function that accepts a plus function which satisfies *PlusNum* and returns a number or by one that accepts a plus function that satisfies *PlusStr* and returns a string value. As an example consider the *testPlus* function.

```
function testPlus (plus) {
  return plus(1, 2);
}
var testPlusNumStr = Contract.assert(testPlus, TestPlus);
```

Because the context do not know which kind or arguments *testPlus* supplies to its *plus* argument, he has to call *testPlus* with a plus function that satisfiers the intersection between *PlusNum* and *PlusStr*.

2.4 Dependent Contracts

A dependent contract is a contract on functions where the range portion depends on the function argument. The contract for the function's range can be created with a contract abstraction, a contract that returns a contract. This abstration is invoked with the caller's argument. so that the returned contract may refer to those values.

TreatJS's dependent contract operation only builds a range contract in this way; it does not check the domain as checking the domain may be achieved with a conjunction with another function contract.

For example, a dependent contract may be used to specify that the arguments type of function *add1* corresponds to the type of the functions return.

```
var SameType = Contract.SDependent(function(input) {
  return Contract.Base(output) {
    return (typeof input) === (typeof output).
  }
});
```

The contract receives the input arguments and returns a contract for the range that checks that the type of the input is identical to the type of the result. When calling a function contracted with the dependent contract *SameType*, the abstraction is invoked on the arguments and the resulting contract is imposed on the return value.

3 Sandboxing of Predicates

TreatJS is implemented as a library so that all aspects of a contract can be specified using the full JavaScript language. For example, the base contract *Num* checks its argument to be a number.

```
var Num = Contract.Base(function (arg) { {
  return (typeof arg) === 'number';
});
```

Asserting a base contracts to a value causes the predicate to be checked by applying the predicate to the value.

```
Contract.assert(1, Num); // accepted
```

However, predicates are attempted not to influence the program state in any way. A monitored program execution should either throw a contract violation or evaluate to the same result as without contracts.

TreatJS relies on the sandbox presented in this work to guarantee that the execution of contract code does not interfere with the contract abiding execution of the host program.

To illustrate, we use a modified *Num* contract.

```
var NumBroken = Contract.Base(function(arg) {
  type = (typeof arg);
  return type === 'number';
});
```

When asserting *NumBroken*, sandboxing intercepts the unintended write to the global variable *type* in the following code and throws an exception.

As read-only access to objects and functions is safe and useful in many contracts, TreatJS facilitates making external references visible inside of the sandbox. For example, the *Ary* contract below references the global object *Array*.

```
var Ary = Contract.With(
  {Array:Array},
  Contract.Base(function (arg) {
    return (arg instanceof Array);
}));
```

4 Transaction-based Sandboxing: A Primer

Today's state of the art in securing JavaScript application that include code from different origins is an all-or-nothing choice. Browsers apply protection mechanisms, such as the same-origin policy or the signed script policy, so that scripts either run in isolation or gain full access.

While script isolation guarantees noninterference with the function of the application as well as preservation of data integrity and confidentiality, there are scripts that must have access to part of the application state to function meaningfully. As all included scripts run with the same authority, the application script cannot exert fine-grained control over the use of data by an included script.

Transactional sandboxing is inspired by the idea of transaction processing in database systems and transactional memory. Each sandbox implements a transactional scope the content of which can be examined, committed, or rolled back. Its design is inspired by revocable references and SpiderMonkey's compartment concept. Our sandbox provides the following novel features:

1. *Language embedded.* The sandbox is implemented as a library in JavaScript. It handles the full JavaScript language (ES5) including its dynamic features. No source code transformation or change in the JavaScript run-time system is required.
2. *Full interposition.* Our sandbox adapts SpiderMonkey's compartment concept[2] and runs code in isolation to the application.
3. *Transaction-based sandboxing.* The sandbox provides a transactional scope. A proxy-based membrane makes objects accessible inside the sandbox, performs effect logging, and enables locally visible modifications. After inspection of the log, effects can be committed to the application state or rolled back.

The implementation of the system is available on the Web[3].

4.1 Cross-Sandbox Access

We consider operations on binary trees as defined by *Node* in Figure 2 along with some auxiliary functions. As an example, we perform operations on a tree consisting of one node and two leaves. All value fields are initially *0*.

```
var root = new Node(0, new Node(0), new Node(0));
```

Next, we create a new empty sandbox by calling the constructor *Sandbox*. Its first parameter acts as the global object of the sandbox environment. It is wrapped in a proxy to mediate all accesses and it is placed on top of the scope chain for code executing inside the sandbox. The seconds parameter is a configuration object. A sandbox is a first class value that can be used for several executions.

```
var sbx = new Sandbox(this, {/* some parameters */});
```

One use of a sandbox is to wrap invocations of function objects. To this end, the sandbox API provides methods *call*, *apply*, and *bind* analogous to methods from ***Function****.prototype*. For example, we may call *setValue* on *root* inside of *sbx*.

[2] SpiderMonkey creates one heap for each website, initially introduced to optimize garbage collection. All objects created by a website are only allowed to touch objects in the same compartment. Proxies are used as cross compartment wrappers to make objects accessible in other compartments.

[3] `https://github.com/keil/Sandbox`

```
1  function Node (value, left, right) {
2    this.value = value;
3    this.left = left;
4    this.right = right;
5  }
6  Node.prototype.toString = function () {
7    return (this.left?this.left + ”, ”:””) + this.value +(this.right?”, ”+this.right:””);
8  }
9  function heightOf (node) {
10   return Math.max(((node.left)?heightOf(node.left)+1:0), ((node.right)?heightOf(
          node.right)+1:0));
11 }
12 function setValue (node) {
13   if (node) {
14     node.value=heightOf(node);
15     setValue(node.left);
16     setValue(node.right);
17   }
18 }
```

Fig. 2. Implementation of *Node*. Each node object consists of a value field, a left node, and a right node. Its prototype provides a *toString* method that returns a string representation. Function *heightOf* computes the height of a node and function *setValue* replaces the value field of a node by its height, recursively.

```
21 sbx.call(setValue, this, root);
```

The first argument of *call* is a function object that is decompiled and redefined inside the sandbox. This step erases the function's free variable bindings and builds a new closure relative to the sandbox's global object. The second argument, the receiver object of the call, as well as the actual arguments of the call are wrapped in proxies to make these objects accessible inside of the sandbox.

The wrapper proxies mediate access to their target objects outside the sandbox. Reads are forwarded to the target unless there are local modifications. The return values are wrapped in proxies, again. Writes produce a *shadow value* that represents the sandbox-internal modification of an object. Initially, this modification is only visible to reads inside the sandbox.

Native objects, like the *Math* object in line 10, are also wrapped in a proxy, but their methods cannot be decompiled because there exists no string representation. Thus, native methods must either be trusted or forbidden. Fortunately, most native methods to not have side effects, so we chose to trust them.

Given all the wrapping and sandboxing, the call in line 21 did not modify the *root* object:

```
22 root.toString(); // returns 0, 0, 0
```

But calling *toString* inside the sandbox shows the effect.

```
sbx.call(root.toString, root); // return 0, 1, 0
```

4.2 Effect Monitoring

During execution, each sandbox records the effects on objects that cross the sandbox membrane. The resulting lists of *effect objects* are accessible through *sbx .effects*, *sbx.readeffects*, and *sbx.writeeffects* which contain all effects, read effects, and write effects, respectively. All three lists offer query methods to select the effects of a particular object.

```
sbx.call(heightOf, this, root);
var rects = sbx.effectsOf(this);
print(”;;; Effects of this”);
rects.foreach(function(i, e) {print(e)});
```

The code snippet above prints a list of all effects performed on *this*, the global object, by executing the *heightOf* function on *root*. The output shows the resulting accesses to *heightOf* and *Math*.

```
;;; Effects of this
(1425301383541) has [name=heightOf]
(1425301383541) get [name=heightOf]
(1425301383543) has [name=Math]
(1425301383543) get [name=Math]
...
```

The first column shows a timestamp, the second shows the name of the effect, and the last column shows the name of the requested parameter. The list does not contain write accesses to *this*. But there are write effects to *value* from the previous invocation of *setValue*.

```
var wectso = sbx.writeeffectsOf(root);
print(”;;; Write Effects of root”);
wectso.foreach(function(i, e) {print(e)});
```

```
;;; Write Effects of root
(1425301634992) set [name=value]
```

4.3 Inspecting a Sandbox

The state inside and outside of a sandbox may diverge for different reasons. We distinguish changes, differences, and conflicts.

A *change* indicates if the sandbox-internal value has been changed with respect to the outside value. A *difference* indicates if the outside value has been modified after the sandbox has concluded. For example, a difference to the previous execution of *setValue* arises if we replace the left leaf element by a new subtree of height 1 outside of the sandbox.

```
root.left = new Node(new Node(0), new Node(0));
```

Changes and differences can be examined using an API that is very similar to the effect API. There are flags to check whether a sandbox has changes or differences as well as iterators over them.

A *conflict* arises in the comparison between different sandboxes. Two sandbox environments are in conflict if at least one sandbox modifies a value that is accessed by the other sandbox later on. We consider only Read-After-Write and Write-After-Write conflicts.

To demonstrate conflicts, we define a function *appendRight*, which adds a new subtree on the right.

```
function appendRight (node) {
  node.right = Node('a', Node('b'), Node('c'));
}
```

To recapitulate, the global *root* is still unmodified and prints *0,0,0,0,0*, whereas the *root* in *sbx* prints *0,0,0,1,0*. Now, let's execute *appendRight* in a new sandbox *sbx2*.

```
var sbx2 = new Sandbox(this, {/* some parameters */});
sbx2.call(appendRight, this, root);
```

Calling *toString* in *sbx2* prints *0,0,0,0,b,a,c*. However, the sandboxes are *not* in conflict, as the following command show.

```
sbx.inConflictWith(sbx2); // returns false
```

While both sandboxes manipulate *root*, they manipulate different fields. *sbx* recalculates the field *value*, whereas *sbx2* replaces the field *right*. Neither reads data that has previously been written by the other sandbox. However, this situation changes if we call *setValue* again, which also modifies *right*.

```
sbx.call(setValue, this, root);
var cofts = sbx.conflictsWith(sbx2); // returns a list of conflicts
cofts.foreach(function(i, e) {print(e)});
```

It documents a read-after-write conflict:

```
Confict: (1425303937853) get [name=right]@SBX001 − (1425303937855) set [
    name=right]@SBX002
```

4.4 Transaction Processing

The *commit* operation applies select effects from a sandbox to its target. Effects may be committed one at a time by calling *commit* on each effect object or all at once by calling *commit* on the sandbox object.

```
sbx.commit();
root.toString(); // returns 0, 1, 0, 2, 0
```

The *rollback* operation undoes an existing manipulation and returns to its previous configuration before the effect. Again, rollbacks can be done on a per-effect basis or for the sandbox as a whole. However, a rollback did not remove the shadow object. Thus, after rolling back, the values are still shadow values in *sbx*.

```
51 sbx.rollabck();
52 root.toString(); // returns 0, 1, 0, 2, 0
53 sbx.call(toString, this, root); // returns 0, 0, 0, 0, 0
```

The *revert* operation resets the shadow object of a wrapped value. The following code snippet reverts the *root* object in *sbx*.

```
54 sbx.revertOf(root);
```

Now, *root*'s shadow object is removed and the origin is visible again in the sandbox. Calling *toString* inside of *sbx* returns *0,1,0,2,0*.

4.5 Transparent Sandboxing

Transparent sandboxing is a special mode of our sandbox. It deactivates the shadowing of write operations so that modifications apply directly to the target objects. As those modifications are performed inside the sandbox, write effects are still logged, so that they can be inspected and rolled back as usual. It can be enabled by changing the *transparent* flag in the sandbox configuration. Here is an example:

```
55 var tsbx = new Sandbox(this, {transparent:true});
56 tsbx.call(setValue, this, root);
```

Calling *toString* demonstrates the difference to the standard, non-transparent sandbox: All changes of line 56 are visible.

```
57 root.toString(); // returns 0, 1, 0, 2, 0
```

Calling *tsbx.rollback();* resets all modifications of *tsbx*. Afterwards, *root* prints *0,0,0,2,0*.

4.6 Pre-state Snapshot

The *snapshot* mode instructs the membrane to clone target objects at initialization time and to use the clone as shadow object. The snapshot enables to *rebase* the sandbox to its initialization state.

A snapshot can be triggered by including the object in the third argument of the sandbox constructor, the snapshot array.

```
58 var ssbx = new Sandbox(this, {/* some parameters */}, [root]);
```

The sandbox can be used as before.

```
59 ssbx.call(setValue, this, root);
60 ssbx.call(root.toString, root); // returns 0, 1, 0, 2, 0
```

Remember, the original *root* object prints *0,0,0,2,0*. Now, let's do some changes, for example by calling *setValue(root);*.

Both representations prints *0,1,0,2,0*. But if one would rebase the sandbox to its initial state, by calling *ssbx.rebase();*, the values go back to the version that exist at initialization time.

```
61 ssbx.call(root.toString, root); // returns 0, 0, 0, 2, 0
```

4.7 Wrapping

The methods *call* and *apply* are shortcuts. Internally, they call a *wrap* method to redefine the function inside of the sandbox and apply the corresponding method from ***Function***.*prototype* to it. The following example shows an alternative to the call in line 21.

```
62 sbx.wrap(setValue).call(this, root);
```

But *wrap* can also be used independently. One example is to obtained a sandboxed version of *root*.

```
63 var sbxroot = sbx.wrap(root);
```

The returned object is wrapped in the sandbox membrane and identical to the object visible inside of the sandbox. Each read access on *sbxroot* returns another sandbox object and each each write access causes an effect. All sandbox features like commit, rollback, and effect logging remain active.

Calling *toString* on *sbxroot* returns *0,1,0,2,0*. The method call illustrates that *sbxroot* is the modified object that occurs in *sbx*. Nevertheless, *sbxroot* can be used like any other object.

This feature allows us to extend an existing data structure with transactional features. For example, instead of defining *root* directly, a developer could define it as follow.

```
64 var sbx3 = new Sandbox(this, {/* some parameters */});
65 var root = sbx3.wrap(new Node(0, new Node(0), new Node(0)));
```

Proxies guarantee that the new *root* object performs as usual, for example when calling *setValue(root)*. But it enables to use all sandbox features in addition, e.g. to commit changes or to roll back.

5 Related Work

Contract Monitoring TreatJS [7] is a language embedded, dynamic, higher-order contract system implemented in JavaScript. Its development is based on a novel denotational semantics of contracts and on a blame calculus [6] that enables higher-order contract with unrestricted intersection and union of contracts. The specification for intersection and union contracts is strongly inspired by their type-theoretic counterparts. This connection tightly integrates statically and dynamically typed worlds which may be beneficial for future integration in a gradual type system.

Effect Monitoring JSConTest [2] is a framework that helps to investigate the effects of unfamiliar JavaScript code by monitoring the execution and by summarizing the observed access traces to access permission contracts. It comes with an algorithm [3] that infers a concise effect description from a set of access paths and it enables the programmer to specify the effects of a function using access permission contracts.

JSConTest2 [5] is a redesign and a reimplementation of JSConTest using JavaScript proxies. The new implementation addresses shortcomings of the previous version. In particular, the proxy-based implementation guarantees full interposition for the full language and for all code regardless of its origin, including dynamically loaded code and code injected via *eval*.

JavaScript Proxies Object equality becomes an issue for non-interference when the executed code ends up in a mixture between wrapper and target. The problem arises if an equality test between wrapper and target returns false instead of true. The work of Keil et al. [4] examines this problem and presents a modification of the underlying VM with respect to object equality and introduces new transparent proxies that fit better to this use case.

6 Conclusion

We presented *TreatJS*, a language embedded, dynamic, higher-order contract system for full JavaScript. *TreatJS* extends the standard abstractions for higher-order contracts with intersection and union contracts, boolean combinations of contracts, and parameterized contracts, which are the building blocks for contracts that depend on run-time values. *TreatJS* implements proxy-based sandboxing for all code fragments in contracts to guarantee that contract evaluation does not interfere with normal program execution. The only serious impediment to full noninterference lies in JavaScript's treatment of proxy equality, which considers a proxy as an individual object.

The *TreatJS-Sandbox* runs JavaScript code in a configurable degree of isolation with fine-grained access control. It provides a transactional scope in which effects are logged for inspection. Effects can be committed to the application state or rolled back.

Both systems are implemented as a JavaScript library. No source code transformation or adaption in the JavaScript run-time system is required. All aspects are accessible through a sandbox API.

References

1. John Boyland, editor. *ECOOP 2015 - Object-Oriented Programming - 29th European Conference*, volume ?, Prague, Czech Repulic, July 2015. LIPICS.
2. Phillip Heidegger, Annette Bieniusa, and Peter Thiemann. Access permission contracts for scripting languages. In John Field and Michael Hicks, editors, *Proc. 39th ACM Symp. POPL*, pages 111–122, Philadelphia, USA, January 2012. ACM Press.
3. Phillip Heidegger and Peter Thiemann. A heuristic approach for computing effects. In Judith Bishop and Antonio Vallecillo, editors, *Proc. 49th TOOLS*, volume 6705 of *LNCS*, pages 147–162, Zurich, Switzerland, June 2011. Springer.
4. Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. Transparent object proxies in JavaScript. In Boyland [1], pages 149–173.
5. Matthias Keil and Peter Thiemann. Efficient dynamic access analysis using JavaScript proxies. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 49–60, New York, NY, USA, 2013. ACM.
6. Matthias Keil and Peter Thiemann. Blame assignment for higher-order contracts with intersection and union. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 375–386, New York, NY, USA, 2015. ACM.
7. Matthias Keil and Peter Thiemann. Treatjs: Higher-order contracts for javascripts. In Boyland [1], pages 28–51.