TreatJS: Higher-Order Contracts for JavaScript

Matthias Keil and Peter Thiemann

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

Abstract

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 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.

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. The impact of contracts on execution speed is evaluated using the Google Octane benchmark.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Higher-Order Contracts, JavaScript, Proxies

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.999

1 Introduction

A contract specifies the interface of a software component by stating obligations and benefits for the component's users. Customarily contracts comprise invariants for objects and components as well as pre- and postconditions for individual methods. Prima facie such contracts may be specified using straightforward assertions. But further contract constructions are needed for contemporary languages with first-class functions and other advanced abstractions. These facilities require higher-order contracts as well as ways to dynamically construct contracts that depend on run-time values.

Software contracts were introduced with Meyer's *Design by Contract*™ methodology [39] that stipulates the specification of contracts for all components of a program and the monitoring of these contracts while the program is running. Since then, the contract idea has taken off and systems for contract monitoring are available for many languages [33, 1, 37, 32, 12, 22, 11, 10] and with a wealth of features [35, 31, 7, 20, 46, 16, 2]. Contracts are particularly important for dynamically typed languages as these languages only provide memory safety and dynamic type safety. Hence, it does not come as a surprise that the first higher-order contract systems were devised for Scheme and Racket [24], out of the need to

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 999–1023

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

create maintainable software. Other dynamic languages like JavaScript[1], Python[2], Ruby[3], PHP[4], and Lua[5] have followed suit.

Many contract systems [33, 12, 22, 10, 35, 46, 16, 2] are *language-embedded*: contracts are first-class values constructed through some library API. This approach is advantageous because it does not tie the contract system to a particular implementation, it neither requires users to learn a separate contract language nor do implementors have to develop specialized contract tools. As the contract system can be distributed as a library, it is easily extensible.

But there are also disadvantages because the contract execution may get entangled with the application code. For example, every contract system supports "flat" contracts which assert that a predicate holds for a value. In most language-embedded systems, the predicate is just a host-language function returning a boolean value. Unlike a real predicate, such a function may have side effects that change the behavior of the application code.

Contributions

We present the design and implementation of *TreatJS*, a language embedded, higher-order contract system for JavaScript [21] 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 [47]) 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. Our guarantees are explained in detail in Section 4.3.
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.

The discussion of related work in Section 6 contains a detailed comparison with other systems. The implementation of the system is available on the Web[6].

---

[1] `http://kinsey.no/projects/jsContract/`, `https://github.com/disnet/contracts.js`
[2] `http://legacy.python.org/dev/peps/pep-0316/`
[3] `https://github.com/egonSchiele/contracts.ruby`
[4] `https://github.com/wick-ed/php-by-contract`
[5] `http://luaforge.net/projects/luacontractor/`
[6] `http://proglang.informatik.uni-freiburg.de/treatjs/`

Overview

The rest of this paper is organized as follows: Section 2 introduces *TreatJS* from a programmer's point of view. Section 3 specifies contract monitoring and Section 4 explains the principles underlying the implementation. Section 5 reports our experiences from applying *TreatJS* to a range of benchmark programs. Section 6 discusses related work and Section 7 concludes.

A technical report [36] extends this paper by an appendix with further technical details, examples, and a formalization of contracts and contract monitoring.

2 A *TreatJS* Primer

The design of *TreatJS* obeys the following rationales.

- *Simplicity and orthogonality.* A core API provides the essential features in isolation. While complex contracts may require using the core API, the majority of contracts can be stated in terms of a convenience API that *TreatJS* provides on top of the core.
- *Non-interference.* Contract checking does not interfere with contract abiding executions of the host program.
- *Composability.* Contracts can be composed arbitrarily.

A series of examples explains how contracts are written and how contract monitoring works. The contract API includes *constructors* that build contracts from other contracts and auxiliary data as well as an ***assert*** function that attaches a contract to a JavaScript value.

Our discussion focuses on general design issues for contracts and avoids JavaScript specifics where possible. Contracts.js [18] provides contracts tailored to the idiosyncrasies of JavaScript's object system—these may be added to *TreatJS* easily.

2.1 Base Contracts

The base contract (aka flat contract) is the fundamental building block for all other contracts. It is defined by a predicate and asserting it to a value immediately sets it in action. We discuss it for completeness—all contract libraries that we know of provide this functionality, but they do not guarantee noninterference like *TreatJS* does.

In JavaScript, any function can be used as a predicate, because any return value can be converted to boolean. JavaScript programmers speak of *truthy* or *falsy* about values that convert to true or false. Thus, a predicate holds for a value if applying the function evaluates to a truthy value.

For example, the function *typeOfNumber* checks its argument to be a number. We apply the appropriate contract constructor to create a base contract from it.

```
1 function typeOfNumber (arg) {
2   return (typeof arg) === 'number';
3 };
4 var typeNumber = Contract.Base(typeOfNumber);
```

Contract is the object that encapsulates the *TreatJS* implementation. Its ***assert*** function attaches a contract to a value. Attaching a base contract causes the predicate to be checked immediately. If the predicate holds, ***assert*** returns the original value. Otherwise, ***assert*** signals a contract violation blaming the *subject*.

In the following example, the first ***assert*** returns *1* whereas the second ***assert*** fails.

```
var typeBoolean = Contract.Base(function (arg) {
  return (typeof arg) === 'boolean';
});
var typeString = Contract.Base(function (arg) {
  return (typeof arg) === 'string';
});
var isArray = Contract.With({Array:Array},
  Contract.Base(function (arg) {
    return (arg instanceof Array);
}));
```

Listing 1 Some utility contracts.

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

Listing 1 defines a number of base contracts for later use. Analogous to *typeNumber*, the contracts *typeBoolean* and *typeString* check the type of their argument. Contract *isArray* checks if the argument is an array. Its correct implementation requires the ***With*** operator, which will be explained in Section 2.4.

2.2 Higher-Order Contracts

The example contracts in Subsection 2.1 are geared towards values of primitive type, but a base contract may also specify properties of functions and other objects. However, base contracts are not sufficiently expressive to state interesting properties of objects and functions. For example, a contract should be able to express that a function maps a number to a boolean or that a certain field access on an object always returns a number.

2.2.1 Function Contracts

Following Findler and Felleisen [24], a function contract is built from zero or more contracts for the arguments and one contract for the result of the function. Asserting the function contract amounts to asserting the argument contracts to the arguments of each call of the function and to asserting the result contract to the return value of each call. Asserting a function contract to a value immediately signals a contract violation if the value is not a function. Nevertheless, we call a function contract *delayed*, because asserting it to a function does not immediately signal a contract violation.

As a running example, we develop several contracts for the function *cmpUnchecked*, which compares two values and returns a boolean.

```
function cmpUnchecked(x, y) {
  return (x > y);
}
```

Our first contract restricts the arguments to numbers and asserts that the result of the comparison is a boolean.

```
var cmp = Contract.assert(cmpUnchecked,
            Contract.AFunction([typeNumber,typeNumber],typeBoolean));
```

AFunction is the convenience constructor for a function contract. Its first argument is an array with n contracts for the first n function arguments and the last argument is the contract for the result. This contract constructor is sufficient for most functions that take a fixed number of arguments.

The contracted function accepts arguments that satisfies contract *typeNumber* and promises to return a value that satisfies *typeBoolean*. If there is call with an argument that violates its contract, then the function contract raises an exception blaming the *context*, which is the caller of the function that provides the wrong kind of argument. If the argument is ok but the result fails, then blame is assigned to the *subject* (i.e., the function itself). Here are some examples that exercise *cmp*.

```
22 cmp(1,2); // accepted
23 cmp('a','b'); // violation, blame the context
```

To obtain a subject violation we use a broken version of *cmpUnchecked* that sometimes returns a string.

```
24 var cmpBroken = function(x, y) {
25   return (x>0 && y>0) ? (x > y) : 'error';
26 }
27 var faultyCmp = Contract.assert(cmpBroken,
28                                 Contract.AFunction([typeNumber,typeNumber],typeBoolean));
29 faultyCmp(0,1); // violation, blame the subject
```

Higher-order contracts may be defined in the usual way and their blame reporting in *TreatJS* follows Findler and Felleisen [24]. For example, a function *sort*, which takes an array and a numeric comparison function as arguments and which returns an array, may be specified by the following contract, which demonstrates nesting of function contracts.

```
30 var sortNumbers = Contract.AFunction([isArray, cmp], isArray);
```

Higher-order contracts open up new ways for a function not to fulfill its contract. For example, *sort* may violate the contract by calling its comparison function (contracted with *cmp*) with non-numeric arguments. Generally, the context is responsible to pass an argument that satisfies its specification to the function and to use the function's result according to its specification. Likewise, the function is responsible for the use of its arguments and in case the arguments meet their specification to return a value that conforms to its specification.

In general, a JavaScript function has no fixed arity and arguments are passed to the function in a special array-like object, the *arguments* object. Thus, the core contract ***Function*** takes two arguments. The first argument is an object contract (cf. Subsubsection 2.2.2) that maps an argument index (starting from zero) to a contract. The second argument is the contract for the function's return value. Thus, ***AFunction*** creates an object contract from the array in its first argument and passes it to ***Function***.

Using the core ***Function*** contract is a bit tricky because it exposes the unwary contract writer to some JavaScript internals. The contract ***Function****(isArray, typeNumber)* checks whether the arguments object is an array (which it is not), but it does *not* check the function's arguments. As a useful application of this feature, the following contract *twoArgs* checks that a function is called with exactly two arguments.

```
31 var lengthTwo = Contract.Base(function (args) {
32   return (args.length == 2);
33 });
34 var Any = Contract.Base (function() { return true; });
```

```
35 var twoArgs = Contract.Function(lengthTwo, any);
```

2.2.2 Object Contracts

Apart from base contracts that are checked immediately and delayed contracts for functions, *TreatJS* provides contracts for objects. An object contract is defined by a mapping from property names to contracts. Asserting an object contract to a value immediately signals a violation if the value is not an object. The contracts in the mapping have no immediate effect. However, when reading a property of the contracted object, the contract associated with this property is asserted to the property value. Similarly, when writing a property, the new value is checked against the contract. This way, each value read from a property and each value that is newly written into the property is guaranteed to satisfy the property's contract. Reads and writes to properties not listed in an object contract are not checked.

The following object contract indicates that the *length* property of an object is a number. The constructor ***AObject*** expects the mapping from property names to contracts as a JavaScript object.

```
36 var arraySpec = Contract.AObject({length:typeNumber});
```

Any array object would satisfy this contract. Each access to the *length* property of the contracted array would be checked to satisfy *typeNumber*.

Blame assignment for property reads and writes is inspired by Reynolds [43] interface for a reference cell: each property is represented as a pair of a getter and a setter function. Both, getter and setter apply the same contract, but they generate different blame. If the contract fails in the getter, then the *subject* (i.e., the object) is blamed. If the contract fails in the setter, then the *context* (i.e., the assignment) is blamed. The following example illustrates this behavior.

```
37 var faultyObj = Contract.assert({length:'1'}, arraySpec);
38 faultyObj.length; // violation, blame the subject
39 faultyObj.length='1'; // violation, blame the context
```

An object contract may also serve as the domain portion in a function contract. It gives rise to yet another equivalent way of writing the contract from Line 21.

```
40 Contract.Function(
41   Contract.AObject([typeNumber, typeNumber]), typeBoolean);
```

Functions may also take an intersection (cf. Section 2.3) of a function contract and an object contract to address properties of functions and *this*. There is also a special ***Method*** contract that includes a contract specification for *this*.

2.3 Combination of Contracts

Beyond base, function, and object contracts, *TreatJS* provides the intersection and union of contracts as well as the standard boolean operators on contracts: conjunction (***And***), disjunction (***Or***), and negation (***Not***). The result of an operator on contracts is again a contract that may be further composed.

For space reasons, we only discuss intersection and union contracts, which are inspired by the corresponding operators in type theory. If a value has two types, then we can assign it an *intersection type* [13]. It is well known that intersection types are useful to model overloading and multiple inheritance.

As an example, we revisit *cmpUnchecked*, which we contracted with *cmpNumbers* in Section 2.2.1 to ensure that its arguments are numbers. As the comparison operators are overloaded to work for strings, too, the following contract is appropriate.

```
42 Contract.Intersection(
43   Contract.AFunction([typeNumber, typeNumber], typeBoolean),
44   Contract.AFunction([typeString, typeString], typeBoolean));
```

This contract blames the context if the contracted function is applied to arguments that fail both domain contracts, that is, *[typeNumber, typeNumber]* and *[typeString, typeString]*. The subject is blamed if a function call does not fulfill the range contract that corresponds to a satisfied domain contract.

This interpretation coincides nicely with the meaning of an intersection type. The caller may apply the function to arguments both satisfying either *typeNumber* or *typeString*. In general, the argument has to satisfy the union of *typeNumber* and *typeString*. For disjoint arguments the intersection contract behaves identically to the disjunction contract.

As in type theory, the union contract is the dual of an intersection contract. Exploiting the well-known type equivalence $(A \to C) \wedge (B \to C) = (A \vee B) \to C$ [5], we may rephrase the above contract with a union contract, which accepts either a pair of numbers or a pair of strings as function arguments:

```
45 Contract.Function(
46   Contract.Union(
47     Contract.AObject([typeNumber, typeNumber]),
48     Contract.AObject([typeString, typeString])),
49  typeBoolean);
```

Next, we consider the union of two function contracts.

```
50 var uf = Contract.Union(
51   Contract.AFunction([typeNumber,typeNumber], typeBoolean),
52   Contract.AFunction([typeString,typeString], typeBoolean));
```

Asserting this contract severely restricts the domain of a function. An argument is only acceptable if it is acceptable for all function contracts in the union. Thus, the context is blamed if it provides an argument that does not fulfill both constituent contracts. For example, *uf* requires an argument that is both a number and a string. As there is no such argument, any caller will be blamed.

For a sensible application of a union of function contracts, the domains should overlap:

```
53 Contract.Union(
54   Contract.AFunction([typeNumber, typeNumber], typeBoolean),
55   Contract.AFunction([typeNumber, typeNumber], typeString));
```

This contract is satisfied by a function that either always returns a boolean value or by one that always returns an string value. It is *not* satisfied by a function that alternates between both return types between calls. A misbehaving function is blamed on the first alternation.

2.4 Sandboxing Contracts

All contracts of *TreatJS* guarantee noninterference: Program execution is not influenced by the evaluation of a terminating predicate inside a base contract. That is, a program with

contracts is guaranteed to read the same values and write to the same objects as without contracts. Furthermore, it either signals a contract violation or returns a results that behaves the same as without contracts.

To achieve this behavior, predicates must not write to data structures visible outside of the predicate. For this reason, predicate evaluation takes place in a sandbox that hides all external bindings and places a write protection on objects passed as parameters.

To illustrate, we recap the *typeNumber* contract from Line 4. Without the sandbox we could abstract the target type of *typeNumber* with a function and build base contracts by applying the function to different type names as in the following attempt:

```
56 function badTypeOf(type) {
57   return Contract.Base(function(arg) {
58     return (typeof arg) === type;
59   });
60 }
61 var typeNumberBad=badTypeOf('number');
62 var typeStringBad=badTypeOf('string');
```

However, this code fragment does not work as expected. The implementation method for our sandbox reopens the closure of the anonymous function in line 57 and removes the binding for *type* from the contract's predicate. Both *typeNumberBad* and *typeStringBad* would be stopped by the sandbox because they try to access the (apparently) global variable *type*. This step is required to guarantee noninterference, because the syntax of predicates is not restricted in their expressiveness and programmers may do arbitrary things, including communicating via global variables or modifying data outside the predicate's scope.

In general, read-only access to data (functions and objects) is safe and many useful contracts (e.g., the *isArray* contract from Line 14 references the global variable *Array*) require access to global variables, so a sandbox should permit regulated access.

Without giving specific permission, the sandbox rejects *any* access to the *Array* object and signals a sandbox violation. To grant read permission, a new contract operator ***With*** is needed that makes an external reference available inside the sandbox. The ***With*** operator takes a *binding object* that maps identifiers to values and a contract. Evaluating the resulting contract installs the binding in the sandbox environment and then evaluates the constituent contract with this binding in place. Each value passed into the sandbox (as an argument or as a binding) is wrapped in an identity preserving membrane [47] to ensure read-only access to the entire object structure.

The ***With*** constructor is one approach to build parameterized contracts by providing a form of dynamic binding.

```
63 var typeOf = Contract.Base(function(arg)  {
64   return (typeof arg) === type;
65 });
66 var typeNumber=Contract.With({type:'number'},typeOf);
67 var typeString=Contract.With({type:'string'},typeOf);
```

For aficionados of lexical scope, contract constructors, explained in the next subsection, are another means for implementing parameterized contracts.

2.5 Contract Constructors

While sandboxing guarantees noninterference, it prohibits the formation of some useful contracts. For example, the range portion of a function contract may depend on the

arguments or a contract may enforce a temporal property by remembering previous function calls or previously accessed properties. Implementing such a facility requires that predicates should be able to store data without affecting normal program execution.

TreatJS provides a *contract constructor* ***Constructor*** for building a parameterized contract. The constructor takes a function that maps the parameters to a contract. This function is evaluated in a sandbox, like a predicate. Unlike a predicate, the function may contain contract definitions and must return a contract. Each contract defined inside the sandbox is associated with the same sandbox environment and shares the local variables and the parameters visible in the function's scope. No further sandboxing is needed for the predicates / base contracts defined inside the sandbox. The returned contract has no ties to the outside world and thus the included predicates will not be evaluated in the sandbox again. If such a predicate is called, the encapsulated sandbox environment can be used to store data for later use and without affecting normal program execution.

In the next example, a contract constructor builds a base contract from the name of a type. The constructor provides a lexically scoped alternative to the approach in Line 63.

```
68 var Type = Contract.Constructor(function(type) {
69   return Contract.Base(function(arg) {
70     return (typeof arg) === type;
71   });
72 });
```

To obtain the actual contract we apply the constructor to parameters with the method ***Contract.construct***(*Type, 'number'*) or by using the ***construct*** method of the constructor.

```
73 var typeNumber = Type.construct('number');
74 var typeString = Type.construct('string');
```

Let's consider yet another contract for a compare function. For this contract, we only want the contract of the comparison to state that the two arguments have the same type.

```
75 Contract.Constructor(function() {
76   var type;
77   var getType = Contract.Base(function (arg) {
78     return type = (typeof arg);
79   });
80   var checkType = Contract.Base(function (arg) {
81     return type === (typeof arg);
82   });
83   var typeBoolean = Contract.Base(function (arg) {
84     return (typeof arg) === 'boolean';
85   });
86   return Contract.AFunction([getType, checkType], typeBoolean);
87 });
```

This code fragment defines a constructor with zero parameters (viz. the empty parameter list in Line 75). As there are no parameters, this example only uses the constructor to install a shared scope for several contracts. The contract *getType* saves the type of the first argument. The comparison function has to satisfy a function contract which compares the type of the second arguments with the saved type.

2.6 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 constructor. This constructor is invoked with the caller's argument. Additionally, it is possible to import pre-state values in the scope of the constructor 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 by conjunction with another function contract. By either pre- or postcomposing the other contract, the programmer may choose between picky and lax semantics for dependent contracts (cf. [29]).

For example, a dependent contract *PreserveLength* may specify that an array processing function like *sort* (Line 30) preserves the length of its input. The constructor receives the arguments (input array and comparison function) of a function call and returns a contract for the range that checks that the length of the input array is equal to the length of the result.

```
88 var PreserveLength = Contract.Dependent(
89   Contract.Constructor(function(input, cmp) {
90     return Contract.Base(function (result) {
91       return (input.length === result.length);
92     });
93 }));
```

3 Contract Monitoring

This section explains how contract monitoring works and how the outcome of a contract assertion is determined by the outcome of its constituents. For space reasons we focus on the standard contract types (base, function, and object contracts) with intersection and union; we describe the boolean operators in the supplemental material.

3.1 Contracts and Normalization

A contract is either an immediate contract, a delayed contract, an intersection between an immediate and a delayed contract, or a union of contracts. Immediate contracts may be checked right away when asserted to a value whereas delayed contracts need to be checked later on. Only a base contract is immediate.

A delayed contract is a function contract, a dependent contract, an object contract, or an intersection of delayed contracts. Intersections are included because all parts of an intersection must be checked on each use of the contracted object: a call to a function or an access to an object property.

The presence of operators like intersection and union has severe implications. In particular, a failing base contract must not signal a violation immediately because it may be enclosed in an intersection. Reporting the violation must be deferred until the enclosing operator is sure to fail.

To achieve the correct behavior for reporting violations, monitoring *normalizes* contracts before it starts contract enforcement. Normalization separates the immediate parts of a contract from its delayed parts so that each immediate contract can be evaluated directly,

whereas the remaining delayed contracts wrap the subject of the contract in a proxy that asserts the contract when the subject is used.

To expose the immediate contracts, normalization first pulls unions out of intersections by applying the distributive law suitably. The result is a union of intersections where the operands of each intersection are either immediate contracts or function contracts. At this point, monitoring can check all immediate contracts and set up proxies for the remaining delayed contracts. It remains to define the structure needed to implement reporting of violations (i.e., blame) that is able to deal with arbitrary combinations of contracts.

3.2 Callbacks

To assert a contract correctly, its evaluation must connect each contract with the enclosing operations and it must keep track of the evaluation state of these operations. In general, the signaling of a violation depends on a combination of failures in different contracts.

This connection is modeled by so-called *callbacks*. They are tied to a particular contract assertion and link each contract to its next enclosing operation or, at the top-level, its assertion. A callback linked to a source-level assertion is called *root callback*. Each callback implements a constraint that specifies the outcome of a contract assertion in terms of its constituents.

A callback is implemented as a method that accepts the result of a contract assertion. The method updates a shared property, it evaluates the constraint, and passes the result to the enclosing callback.

Each callback is related to one specific contract occurrence in the program; there is at least one callback for each contract occurrence and there may be multiple callbacks for a delayed contract (e.g., a function contract). The callback is associated with a record that defines the blame assignment for the contract occurrence. This record contains two fields, *subject* and *context*. The intuitive meaning of the fields is as follows. If the *subject* field is false, then the contract fails blaming the subject (i.e., the value to which the contract is asserted). If the *context* field is false, then the contract fails blaming the context (i.e., the user of the value to which the contract is asserted).

3.3 Blame Calculation

The fields in the record range over $\mathbb{B}_4$, the lattice underlying Belnap's four-valued logic [6], which is intended to deal with incomplete or inconsistent information. The set $\mathbb{B}_4 = \{\bot, \mathsf{f}, \mathsf{t}, \top\}$ of truth values forms a lattice modeling accumulated knowledge about the truth of a proposition. Thus, a truth value may be considered as the set of classical truth values $\{\mathit{true}, \mathit{false}\}$ that have been observed so far for a proposition. For instance, contracts are valued as $\bot$ before they are evaluated and $\top$ signals potentially conflicting outcomes of repeated checking of the same contract.

As soon as a base contract's predicate returns, the contract's callback is applied to its outcome. A function translates the outcome to a truth value according to JavaScript's idea of *truthy* and *falsy*, where *false*, *undefined*, *null*, *NaN*, and *""* is interpreted as false. Exceptions thrown during evaluation of a base contract are captured and count as $\top$.

A new contract assertion signals a violation if a root callback maps any field to f or $\top$. Evaluation continues if only internal fields have been set to f or $\top$.

3.4 Contract Assertion

Contract monitoring starts when calling the assert function with a value and a contract.

The top-level assertion first creates a new root callback that may signal a contract violation later on and an empty sandbox object that serves as the context for the internal contract monitoring. The sandbox object carries all external values visible to the contract.

Asserting a base contract to a value wraps the value to avoid interference and applies the predicate to the wrapped value. Finally, it applies the current callback function to the predicate's outcome.

Asserting a delayed contract to an object results in a proxy object that contains the current sandbox object, the associated callback, and the contract itself. It is an error to assert a delayed contract to a primitive value.

Asserting a union contract first creates a new callback that combines the outcomes of the subcontracts according to the blame propagation rules for the union. Then it asserts the first subcontract (with a reference to one input of the new callback) to the value before it asserts the second subcontract (with a reference to the other input) to the resulting value.

Asserting a ***With*** contract first wraps the values defined in the binding object and builds a new sandbox object by merging the resulting values and the current sandbox object. Then it asserts its subcontract.

3.5 Application, Read, and Assignment

Function application, property read, and property assignment distinguish two cases: either the operation applies directly to a non-proxy object or it applies to a proxy. If the target of the operation is not a proxy object, then the standard operation is applied.

If the target object is a proxy with a delayed contract, then the contract is checked when the object is used as follows.

A function application on a contracted function first creates a fresh callback that combines the outcomes of the argument and range contract according to the propagation rules for functions. Then it asserts the domain contract to the argument object with reference to the domain input of the new callback before it applies the function to the result. After completion, the range contract is applied to the function's result with reference to the range input of the callback.

A function application on a function contracted with a dependent contract first applies the contract constructor to the argument and saves the resulting range contract. Next, it applies the function to the argument and asserts the computed range contract to the result.

A property access on a contracted object has two cases depending on the presence of a contract for the accessed property. If a contract exists, then the contract is asserted to the value after reading it from the target object and before writing it to the target object. Otherwise, the operation applies directly to the target object.

Property write creates a fresh callback that inverts the responsibility of the contract assertion (the context has to assign a value according to the contract).

An operation on a proxy with an intersection contract asserts the first subcontract to the value before it asserts the second subcontract to the resulting value. Both assertions are connected to one input channel of a new callback that combines there outcomes according to the rules for intersection.

All contract assertions forward the sandbox object in the proxy to the subsequent contract assertion.

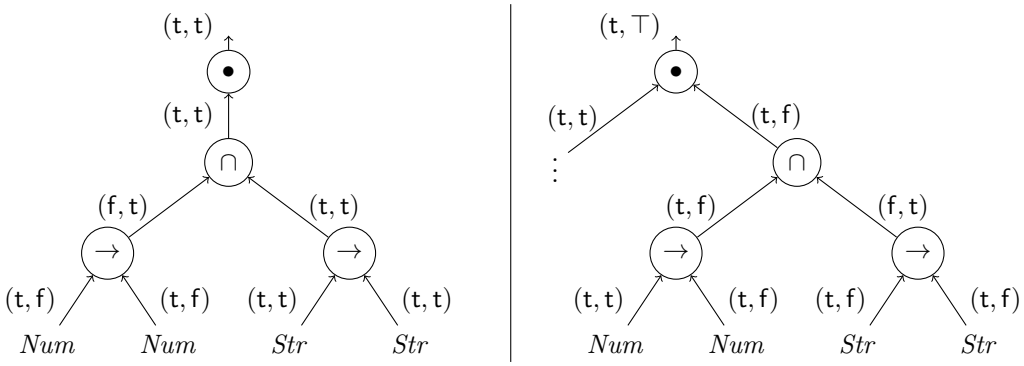

Figure 1 Blame Calculation of $addOne = \lambda x.(x{+}'1')$ contracted with $(Num \to Num) \cap (Str \to Str)$. The left side shows the callback graph after applying *addOne* to the string *'1'* (first call). The right side shows the graph constructed after applying *addOne* to the number *1* (second call). Each node is a callback. Each edge an input channel. The labeling next to the arrow shows the record (*context*, *subject*). The root callback (•) collects the outcome of all delayed contract assertions.

3.6 Sandbox Encapsulation

The sandbox ensures noninterference with the actual program code by stopping evaluation of the predicate if it attempts to modify data that is visible outside the contract.[7]

To ensure noninterference, the subject of a base contract, the argument of a dependent contract as well as the value passed by a contract constructor are all wrapped in a membrane to ensure that the contract's code cannot modify them in any way.

To wrap a non-proxy object, the object is packaged in a fresh proxy along with the current sandbox object. This packaging ensures that further access to the wrapped object uses the current sandbox object. If the object points to a contracted object, the wrap operation continues with the target object, before adding all contracts from the contracted object. A primitive value is not wrapped.

A property read on a sandboxed object forwards the operation to the target and wraps the resulting behavior. An assignment to a sandboxed object is not allowed, thus it signals a sandbox violation.

The application of a sandboxed function recompiles the function by adding the given sandbox object to the head of the scope chain. Finally, it calls the function. The function's argument and its result are known to be wrapped in this case.

3.7 Contract Satisfaction

The blame assignment for a function contract is calculated from the blame assignment for the argument and result contracts, which are available through the records ι_d and ι_r. A function does not fulfill a contract if it does not fulfill its obligations towards its argument $\iota_d.context$ **or** if the argument fulfills its contract, but the result does not fulfills its contract $\iota_d.subject \Rightarrow \iota_r.subject$. The first part arises for higher-order functions, which may pass illegal arguments to their function-arguments. The second part corresponds to partial correctness of the function with respect to its contract.

A function's context (caller) fulfills the contract if it passes an argument that fulfills its

[7] The sandbox cannot ensure termination of the predicate, of course.

contract $\iota_d.subject$ **and** it uses the function result according to its contract $\iota_r.context$. The second part becomes non-trivial with functions that return functions.

An object (subject) does not fulfill an object contract if a property access returns a value that does not fulfill the contract. An object's context (caller) does not fulfills the contract if it assigns an illegal value to a contracted property **or** it does not uses the objects return according to its contract.

The outcome of read access on a contracted property $\iota_c.subject$ is directly related to the parent callback and does not need a special constraint. A write to a property guarded with contract C generates blame like a call to a function with contract $C \to Any$. (Any accepts any value.)

The blame assignment for an intersection contract is defined from its constituents at ι_r and ι_l. A subject fulfills an intersection contract if it fulfills both constituent contracts: $\iota_r.subject \wedge \iota_l.subject$. A context, however, only needs to fulfill one of the constituent contracts: $\iota_r.context \vee \iota_l.context$.

Dually to the intersection rule, the blame assignment for a union contract is determined from its constituents at ι_l and ι_r. A subject fulfills a union contract if it fulfills one of the constituent contracts: $\iota_l.subject \vee \iota_r.subject$. A context, however, needs to fulfill both constituent contracts: $\iota_l.context \wedge \iota_r.context$, because it does not known which contract is fulfilled by the subject.

Figure 1 illustrates the working of callbacks. After applying *addOne* to *'1'*, the first function contract ($Num \to Num$) would fail blaming the context, whereas the second contract ($Str \to Str$) succeeds. Because the context of an intersection may choose which side to fulfill, the intersection is satisfied.

However, the second call which applies *addOne* to *1* raises an exception. The first function contract fails, blaming the subject, whereas the second contract fails, blaming the context. Because the subject of an intersection has to fulfill both contracts, the intersection fails, blaming the subject.

4 Implementation

The implementation is based on the JavaScript Proxy API [47, 48], a part of the ECMAScript 6 draft standard. This API is implemented in Firefox since version 18.0 and in Chrome V8 since version 3.5. Our development is based on the SpiderMonkey JavaScript engine.

4.1 Delayed Contracts

Delayed contracts are implemented using JavaScript Proxies [47, 48], which guarantees full interposition by intercepting all operations. The assertion of a delayed contract wraps the subject of the contract in a proxy. The handler for the proxy contains the contract and implements traps to mediate the use of the subject and to assert the contract. No source code transformation or change in the JavaScript run-time system is required.

4.2 Sandboxing

Our technique to implement sandboxing relies on all the evil and bad parts of JavaScript: the *eval* function and the ***with*** statement. The basic idea is as follows. The standard implementation of the *toString* method of a user-defined JavaScript function returns a string that contains the source code of that function. When *TreatJS* puts a function (e.g., a predicate) in a sandbox, it first *decompiles* it by calling its *toString* method. Applying *eval*

to the resulting string creates a fresh variant of that function, **but** it dynamically rebinds the free variables of the function to whatever is currently in the scope at the call site of *eval*.

JavaScript's ***with*** *(obj){ ... body ... }* statement modifies the current environment by placing *obj* on top of the scope chain while executing *body*. With this construction, which is somewhat related to dynamic binding [30], any property defined in *obj* shadows the corresponding binding deeper down in the scope chain. Thus, we can add and shadow bindings, but we cannot remove them. Or can we?

It turns out that we can also abuse ***with*** to *remove* bindings! The trick is to wrap the new bindings in a proxy object, use ***with*** to put it on top of the scope chain, and to trap the binding object's *hasOwnProperty* method. When JavaScript traverses the scope chain to resolve a variable reference *x*, it calls *hasOwnProperty(x)* on the objects of the scope chain starting from the top. Inside the ***with*** statement, this traversal first checks the proxied binding object. If its *hasOwnProperty* method always returns true, then the traversal stops here and the JavaScript engine sends all read and write operations for free variables to the proxied binding object. This way, we obtain full interposition and the handler of the proxied binding object has complete control over the free variables in *body*.

The ***With*** contract is *TreatJS*'s interface to populate this binding object. The operators for contract abstraction and dependent contracts all take care to stitch the code fragments together in the correct scope. To avoid the frequent decompilation and *eval* of the same code, our implementation caches the compiled code where applicable.

No value is passed inside the sandbox without proper protection. Our protection mechanism is inspired by *Revocable Membranes* [47, 44]. A membrane serves as a regulated communication channel between two worlds, in this case between an object/ a function and the rest of a program. A membrane is essentially a proxy that guards all read operations and—in our case—stops all writes. If the result of a read operation is an object, then it is recursively wrapped in a membrane before it is returned. Access to a property that is bound to a *getter* function needs to decompile the *getter* before its execution. Care is taken to preserve object identities when creating new wrappers (our membrane is *identity preserving*).

We employ membranes to keep the sandbox apart from normal program execution thus guaranteeing noninterference. In particular, we encapsulate objects passed through the membrane, we enforce write protection, and we withhold external bindings from a function.

4.3 Noninterference

The ideal contract system should not interfere with the execution of application code. That is, as long as the application code does not violate any contract, the application should run as if no contracts were present. Borrowing terminology from security, this property is called noninterference (NI) [27]: with the assumption that contract code runs at a higher level of security than application code, the low security application code should not be able to observe the results of the high-level contract computation.

Looking closer, we need to distinguish internal and external sources of interference. Internal sources of interference arise from executing unrestricted JavaScript code in the predicate of a base contract. This code may try to write to an object that is visible to the application, it may throw an exception, or it may not terminate. Our implementation restricts all write operations to local objects using sandboxing. It captures all exceptions and turns them into an appropriate contract outcome. A timeout could be used to transform a contract that may not terminate into an exception, alas, such a timeout cannot be implemented in

JavaScript.[8]

External interference arises from the interaction of the contract system with the language. Two such issues arise in a JavaScript contract system, exceptions and object equality.

Exceptions arise when a contract failure is encoded by a contract exception, as it is done in Eiffel, Racket, and contracts.js. If an application program catches exceptions, then it may become aware of the presence of the contract system by observing an exception caused by a contract violation. Our implementation avoids this problem by reporting the violation and then using a JavaScript API method to quit JavaScript execution[9].

Object equality becomes an issue because function contracts as well as object contracts are implemented by some kind of wrapper. The problem arises if a wrapper is different (i.e., not pointer-equal) from the wrapped object so that an equality test between wrapper and wrapped object or between different wrappers for the same object (read: tests between object and contracted object or between object with contract A and object with contract B) in the application program returns false instead of true.

Our implementation uses JavaScript proxies to implement wrappers. Unfortunately, JavaScript proxies are always different from their wrapped objects and the only safe way to change that is by modifying the proxy implementation in the JavaScript VM. See our companion paper [34] for more discussion. There are proposals based on preprocessing all uses of equality to proxy-dereferencing equality, for example using SweetJS [19], but they do not work in combination with `eval` and hence do not provide full interposition.

5 Evaluation

This section reports on our experience with applying contracts to select programs. We focus on the influence of contract assertion and sandboxing on the execution time.

All benchmarks were run on a machine with two AMD Opteron Processor with 2.20 GHz and 64 GB memory. All example runs and timings reported in this paper were obtained with the SpiderMonkey JavaScript engine.

5.1 Benchmark Programs

To evaluate our implementation, we applied it to JavaScript benchmark programs from the Google Octane 2.0 Benchmark Suite[10]. Octane 2.0 consists of 17 programs that range from performance tests to real-world web applications (Figure 2), from an OS kernel simulation to a portable PDF viewer. Each program focuses on a special purpose, for example, function and method calls, arithmetic and bit operations, array manipulation, JavaScript parsing and compilation, etc.

Octane reports its result in terms of a score. The Octane FAQ[11] explains the score as follows: "*In a nutshell: bigger is better. Octane measures the time a test takes to complete and then assigns a score that is inversely proportional to the run time.*" The constants in this computation are chosen so that the current overall score (i.e., the geometric mean of the individual scores) matches the overall score from earlier releases of Octane and new

[8] The JavaScript *timeout* function only schedules a function to run when the currently running JavaScript code—presumably some event handler—stops. It cannot interrupt a running function.

[9] This aspect is customizable because the API method is not generally available. It can easily be overwritten to report a violation elsewhere or to throw an exception.

[10] `https://developers.google.com/octane/`

[11] `https://developers.google.com/octane/faq`

benchmarks are integrated by choosing the constants so that the geometric mean remains the same. The rationale is to maintain comparability.

5.2 Methodology

To evaluate our implementation, we wrote a source-to-source compiler that first modifies the benchmark code by wrapping each function expression[12] in an additional function. In a first run, this additional function wraps its target function in a proxy that, for each call to the function, records the data types of the arguments and of the function's return value. This recording distinguishes the basic JavaScript data types *boolean*, *null*, *undefined*, *number*, *string*, *function*, and *object*. Afterwards, the wrapper function is used to assert an appropriate function contract to each function expression. These function contracts are built from the types recorded during the first phase. If more than one type is recorded at a given program point, then the disjunction of the individual type contracts is generated.

All run-time measurements were taken from a deterministic run, which requires a predefined number of iterations, and by using a warm-up run.

5.3 Results

Figure 2 contains the scores of all benchmark programs in different configurations, which are explained in the figure's caption. As expected, all scores decrease when adding contracts. The impact of a contract depends on the frequency of its application. A contract on a heavily used function (e.g., in *Richards*, *DeltaBlue*, or *Splay*) causes a significantly higher decrease of the score. These examples show that the run-time impact of contract assertion depends on the program and on the particular value that is monitored. While some programs like *Richards*, *DeltaBlue*, *RayTrace*, and *Splay* are heavily affected, others are almost unaffected: *Crypto*, *NavierStokes*, and *Mandreel*, for instance.

In several cases the execution with contracts (or with a particular feature) is faster than without. All such fluctuations in the score values are smaller than the standard deviation over several runs of the particular benchmark.

For better understanding, Figure 3 lists some numbers of internal counters. The numbers indicate that the heavily affected benchmarks (*Richards*, *DeltaBlue*, *RayTrace*, *Splay*) contain a very large number of internal contract assertions. Other benchmarks are either not affected (*RegExp*, *zlib*) or only slightly affected (*Crypto*, *pdf.js*, *Mandreel*) by contracts.

For example, the *Richards* benchmark performs 24 top-level contract assertions (these are all calls to ***Contract.assert***), 1.6 billion internal contract assertions (including top-level assertions, *delayed* contract checking, and predicate evaluation), and 936 million predicate executions. The sandbox wraps about 4.7 billion elements, but performs only 4 decompile operations. Finally, contract checking performs 3.4 billion callback update operations.

Because of the fluctuation in slightly affected benchmark programs the following discussion focuses on benchmarks that were heavily impacted. Thus, we ignore the benchmark programs *Crypto*, *RegExp*, *pdf.js*, *Mandreel*, *zlib*.

In a first experiment, we turn off predicate execution and return *true* instead of the predicate's result. This splits the performance impact into the impact caused by the contract system (proxies, callbacks, and sandboxing) and the impact caused by evaluating predicates. From the score values we find that the execution of the programmer provided predicates

[12] Function expressions are all expressions of the form ***function****(..){..}*.

Benchmark	F	S	w/o C	w/o D	w/o M	w/o P	B
Richards	0.391	0.519	0.582	0.782	0.781	0.903	11142
DeltaBlue	0.276	0.360	0.409	0.544	0.544	0.625	17462
Crypto	11888	12010	11912	11914	11986	11979	11879
RayTrace	1.09	1.45	1.82	2.51	2.51	3.02	23896
EarleyBoyer	5135	5292	5126	5205	5233	5242	5370
RegExp	1208	1181	1205	1199	1212	1178	1207
Splay	20.6	27.8	31.2	42.5	42.5	49.7	9555
SplayLatency	73.1	99.7	109	151	151	177	6289
NavierStokes	6234	7159	7924	9176	8943	9456	12612
pdf.js	9191	9257	9548	9156	9222	9152	9236
Mandreel	12555	12542	12586	12549	12346	12431	12580
MandreelLatency	18741	18883	18741	18883	19027	18955	19398
Gameboy Emulator	6.80	9.07	10.8	14.9	14.9	17.7	23801
Code loading	6245	6785	6937	7372	7335	7533	9324
Box2DWeb	3.57	4.67	5.72	7.80	7.82	9.19	12528
zlib	29108	28708	29025	29047	28926	29063	29185
TypeScript	187	248	290	400	396	463	11958

Figure 2 Scores for the Google Octane 2.0 Benchmark Suite (bigger is better). Column **F** (*Full*) contains the scores for running with sandboxed contract assertion. Column **S** (*System only*) contains the score values for running TreatJS without predicate evaluation (all predicates are set to *true*) but with all internal components (callback, decompile, membrane). Column **w/o C** (*without callback*) shows the scores from a full run (with predicates) but without callback updates. Column **w/o D** (*without decompile*) shows the scores without recompiling functions. Column **w/o M** (*without membrane*) lists the scores with contract assertion but without sandboxing (and thus without decompile). Column **w/o P** (*without predicate*) shows the score values of raw contract assertions without predicate evaluation and thus without sandboxing, decompile, and callback updates. The last column **B** (*Baseline*) gives the baseline scores without contract assertion.

causes a slowdown of 9.20% over all benchmarks (difference between **F** and **S**). The remaining slowdown is caused by the contract system itself. The subsequent detailed treatment of the score values splits the impact into its individual components.

Comparing columns **F** and **w/o C** shows that callback updates cause an overall slowdown of 4.25%. This point includes the recalculation of the callback constraints as explained in Section 3.7.

The numbers also show that decompiling functions has negligible impact on the execution time. Decompiling decreases the score by 6.29% over all benchmarks (compare columns **w/o C** and **w/o D**)[13]. This number is surprisingly low when taking into account that predicate evaluation includes recompiling all predicates on all runs as explained in Section 4.

Comparing the scores in columns **w/o D** and **w/o M** indicates that the membrane, as it is used by the sandbox, does not contribute significantly to the run-time overhead. It does not decrease the total scores.

Finally, after deactivating predicate execution, we see that pure predicate handling causes a slowdown of approximately 1.76% (this is the impact of the function calls). In contrast to column **S**, column **w/o P** shows the score values of the programs without sandboxing, without recompiling, and without callback updates, whereas in column **S** sandboxing, recompilation,

[13] Function recompilation can be safely deactivated for the benchmarks without changing the outcome because our generated base contracts are guaranteed to be free of side effects.

Benchmark	Contract			Sandbox		Callback
	A	I	P	M	D	
Richards	24	1599377224	935751200	4678756000	4	3351504000
DeltaBlue	54	2319477672	1340451212	6702256060	5	4744203248
Crypto	1	5	3	15	3	13
RayTrace	42	687240082	509234422	2546172110	4	2190186074
EarleyBoyer	3944	89022	68172	340860	6	309120
RegExp	0	0	0	0	0	0
Splay	10	11620663	7067593	35337965	5	26231845
SplayLatency	10	11620663	7067593	35337965	5	26231845
NavierStokes	51	48334	39109	195545	5	177197
pdf.js	3	15	9	45	4	39
Mandreel	7	57	28	140	4	128
MandreelLatency	7	57	28	140	4	128
Gameboy Emulator	3206	141669753	97487985	487439925	5	399084085
Code loading	5600	34800	18400	92000	4	70400
Box2DWeb	20075	172755100	112664947	563324735	5	469141435
zlib	0	0	0	0	0	0
TypeScript	4	12673644	8449090	42245450	2	33796350

Figure 3 Statistic from running the Google Octane 2.0 Benchmark Suite. Column **A** (*Assert*) shows the numbers of top-level contract assertions. Column **I** (*Internal*) contains the numbers of internal contract assertions whereby column **P** (*Predicate*) lists the number of predicate evaluations. Column **M** (*Membrane*) shows the numbers of wrap operation and column **D** (*Decompile*) show the numbers of decompile operations. The last column **Callback** gives the numbers of callback updates.

and callback updates remain active.

From the score values we find that the overall slowdown of sandboxed contract checking vs. a baseline without contracts amounts to a factor of 7136, approximately. The dramatic decrease of the score values in the heavily affected benchmarks is simply caused by the tremendous number of checks that arise during a run.

For example, in the *Splay* benchmark, the insert, find, and remove functions on trees are contracted. These functions are called every time a tree operation is performed. As the benchmark runs for 1400 iterations on trees with 8000 nodes, there is a considerable number of operations, each of which checks a contract. It should be recalled that every contract check performs at least two *typeof* checks.

Expressed in absolute time spans, contract checking causes a run time deterioration of 0.17ms for every single predicate check. For example, the contracted *Richards* requires 152480 seconds to complete and performs 935751200 predicate checks. Its baseline requires 8 seconds. Thus, contract checking requires 152472 seconds. That gives 0.16ms per predicate check.

Google claims that Octane "measure[s] the performance of JavaScript code found in large, real-world web applications, running on modern mobile and desktop browsers"[14]. For an academic, this is as realistic as it gets.

However, there are currently no large JavaScript application with contracts that we could use to benchmark, so we had to resort to automatic generation and insertion of contracts. These contracts may end up in artificial and unnatural places that would be avoided by an

[14] `https://developers.google.com/octane/`

efficiency conscious human developer. Thus, the numbers that we obtain give insight into the performance of our contract implementation, but they cannot be used to predict the performance impact of contracts on realistic programs with contracts inserted by human programmers. The scores of the real-world programs (*pdf.js*, *Mandreel*, *Code Loading*) among the benchmarks provide some initial evidence in this direction: their scores are much higher and they are only slightly effected by contract monitoring. But more experimentation is needed to draw a statistically valid conclusion.

6 Related Work

Contract Validation

Contracts may be validated statically or dynamically. Purely static frameworks (e.g. ESC/-Java [25]) transform specifications and programs into verification conditions to be verified by a theorem prover. Others [50, 45] rely on symbolic execution to prove adherence to contracts. However, most frameworks perform run-time monitoring as proposed in Meyer's work.

Higher-Order Contracts

Findler and Felleisen [24] first showed how to construct contracts and contract monitors for higher-order functional languages. Their work has attracted a plethora of follow-up works that range from semantic investigations [8, 23] over deliberations on blame assignment [15, 49] to extensions in various directions: contracts for polymorphic types [2, 7], for affine types [46], for behavioral and temporal conditions [17, 20], etc. While the cited semantic investigations consider noninterference, only Disney and coworkers [20] give noninterference a high priority and propose an implementation that enforces it. The other contract monitoring implementations that we are aware of, do not address noninterference or restrict their predicates.

Embedded Contract Language

Specification Languages like JML [38] state behavior in terms of a custom contract language or in terms of annotations in comments. An embedded contract language exploits the language itself to state contracts. Thus programmers need not learn a new language and the contract specification can use the full power of the language. Existing compilers and tools can be used without modifications.

Combinations of Contracts

Over time, a range of contract operators emanated, many of which are inspired by type operators. There are contract operators analogous to (dependent) function types [24], product types, sum types [33], as well as universal types [2]. Racket also implements restricted versions of conjunctions and disjunctions of contracts (see below). However, current systems do not support contracts analogous to union and intersection types nor do they support full boolean combination of contracts (negation is missing).

Dimoulas and Felleisen [14] propose a contract composition, which corresponds to a conjunction of contracts. But their operator is restricted to contracts of the same type. Before evaluating a conjunction it lifts the operator recursively to base contracts where it finally builds the conjunction of the predicate results.

Racket's contract system [26, Chapter 7] supports boolean combinations of contracts. Conjunctions of contracts are decomposed and applied sequentially [45]. Disjunctions of flat contracts are transformed so that the first disjunct does not cause a blame immediately if its predicate fails. However, Racket places severe restrictions on using disjunction with higher-order contracts and restricts negation to base contracts. A disjunction must be resolved by first-order choice to at most one higher-order contract; otherwise it is rejected at run time.

Proxies

The JavaScript proxy API [47] enables a developer to enhance the functionality of objects easily. JavaScript proxies have been used for Disney's JavaScript contract system, contracts.js [18], to enforce access permission contracts [35], as well as for other dynamic effects systems, meta-level extension, behavioral reflection, security, or concurrency control [40, 4, 9].

Sandboxing JavaScript

The most closely related work to our sandbox mechanism is the work of Arnaud et al. [3]. They provide features similar to our sandbox mechanism. Both approaches focus on access restriction and noninterference to guarantee side effect free assertions of contracts.

Our sandbox mechanism is inspired by the design of access control wrappers which is used for revocable references and membranes [47, 42]. In memory-safe languages, a function can only cause effects to objects outside itself if it holds a reference to the other object. The authority to affect the object can simply be removed if a membrane revokes the reference which detaches the proxy from its target.

Our sandbox works in a similar way and guarantees read-only access to target objects, but redirects write operations. Write access is completely forbidden and raises an exception. However, the restrictions affect only values that cross the border between the global execution environment and a predicate execution. Values that are defined and used in one side, e.g. local values, were not restricted. Write access to those values is fine.

Other approaches implement restrictions by filtering and rewriting untrusted code or by removing features that are either unsafe or that grant uncontrolled access. The Caja Compiler [28, 41], for example, compiles JavaScript code in a sanitized JavaScript subset that can safely be executed in normal engines. However, some static guarantees do not apply to code created at run time. For this reason Caja restricts dynamic features and adds run-time checks that prevent access to unsafe function and objects.

7 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 impact of contracts on the execution time varies widely depending on the particular functions that are under contract and on the frequency with which the functions are called.

While some programs' run time is heavily impacted, others are nearly unaffected. We believe that if contracts are carefully and manually inserted with the purpose of determining interface specifications and finding bugs in a program, their run time will mostly be unaffected. But more experimentation is needed to draw a statistically valid conclusion.

References

1 Parker Abercrombie and Murat Karaorman. jContractor: Design by contract for Java. `http://jcontractor.sourceforge.net/`, 2003.

2 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In Thomas Ball and Mooly Sagiv, editors, *Proceedings 38th Annual ACM Symposium on Principles of Programming Languages*, pages 201–214, Austin, TX, USA, January 2011. ACM Press.

3 Jean-Baptiste Arnaud, Marcus Denker, Stéphane Ducasse, Damien Pollet, Alexandre Bergel, and Mathieu Suen. Read-only execution for dynamic languages. In Jan Vitek, editor, *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, volume 6141 of *Lecture Notes in Computer Science*, pages 117–136, Málaga, Spain, June 2010. Springer.

4 Thomas H. Austin, Tim Disney, and Cormac Flanagan. Virtual values for language extension. In Cristina Videira Lopes and Kathleen Fisher, editors, *OOPSLA*, pages 921–938, Portland, OR, USA, 2011. ACM.

5 Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de' Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

6 Nuel Belnap. A useful four-valued logic. In J. Michael Dunn and George Epstein, editors, *Modern Uses of Multiple-Valued Logic*, volume 2 of *Episteme*, pages 5–37. Springer Netherlands, 1977.

7 João Filipe Belo, Michael Greenberg, Atsushi Igarashi, and Benjamin C. Pierce. Polymorphic contracts. In Gilles Barthe, editor, *Proceedings of the 20th European Symposium on Programming*, volume 6602 of *Lecture Notes in Computer Science*, pages 18–37, Saarbrücken, Germany, March 2011. Springer-Verlag.

8 Matthias Blume and David McAllester. Sound and complete models of contracts. *Journal of Functional Programming*, 16:375–414, July 2006.

9 Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA*, pages 331–344. ACM, 2004.

10 Lorenzo Caminiti. Contract++, contract programming library for C++. `http://sourceforge.net/projects/contractpp/`, August 2012.

11 Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for Web services. *ACM Transactions on Programming Languages and Systems*, 31(5), 2009.

12 Olaf Chitil. Practical typed lazy contracts. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*. ACM, September 2012.

13 Mario Coppo and M. Dezani-Ciancaglini. A new type-assignment for λ-terms. *Archiv. Math. Logik*, 19(139-156), 1978.

14 Christos Dimoulas and Matthias Felleisen. On contract satisfaction in a higher-order world. *ACM Transactions on Programming Languages and Systems*, 33(5):16, 2011.

15 Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: No more scapegoating. In Thomas Ball and Mooly Sagiv, editors, *Proceedings 38th Annual ACM Symposium on Principles of Programming Languages*, pages 215–226, Austin, TX, USA, January 2011. ACM Press.

16 Christos Dimoulas, Riccardo Pucella, and Matthias Felleisen. Future contracts. In António Porto and Francisco J. López-Fraguas, editors, *Principles and Practice of Declarative Programming, PPDP 2009*, pages 195–206, Coimbra, Portugal, September 2009. ACM.

17 Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *European Symposium on Programming*, volume 7211 of *Lecture Notes in Computer Science*, Tallinn, Estland, April 2012. Springer-Verlag.

18 Tim Disney. contracts.js. `https://github.com/disnet/contracts.js`, April 2013.

19 Tim Disney, Nathan Faubion, David Herman, and Cormac Flanagan. Sweeten your JavaScript: Hygienic macros for ES5. In Andrew P. Black and Laurence Tratt, editors, *DLS*, pages 35–44, Portland, OR, USA, October 2014. ACM.

20 Tim Disney, Cormac Flanagan, and Jay McCarthy. Temporal higher-order contracts. In Olivier Danvy, editor, *Proceedings International Conference on Functional Programming 2011*, pages 176–188, Tokyo, Japan, September 2011. ACM Press, New York.

21 ECMAScript Language Specification, December 2009. ECMA International, ECMA-262, 5th edition.

22 Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2103–2110, Sierre, Switzerland, 2010. ACM.

23 Robert Bruce Findler and Matthias Blume. Contracts as pairs of projections. In Philip Wadler and Masami Hagiya, editors, *Proceedings of the 8th International Symposium on Functional and Logic Programming FLOPS 2006*, pages 226–241, Fuji Susono, Japan, April 2006. Springer-Verlag.

24 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In Simon Peyton-Jones, editor, *Proceedings International Conference on Functional Programming 2002*, pages 48–59, Pittsburgh, PA, USA, October 2002. ACM Press, New York.

25 Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 2002. ACM Press.

26 Matthew Flatt, Robert Bruce Findler, and PLT. *The Racket Guide*, v.6.0 edition, March 2014. `http://docs.racket-lang.org/guide/index.html`.

27 Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

28 google-caja: A source-to-source translator for securing JavaScript-based web content. `http://code.google.com/p/google-caja/`, (as of 2011).

29 Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Proceedings 37th Annual ACM Symposium on Principles of Programming Languages*, pages 353–364, Madrid, Spain, January 2010. ACM Press.

30 David R. Hanson and Todd A. Proebsting. Dynamic variables. In *Proceedings of the 2001 Conference on Programming Language Design and Implementation*, pages 264–273, Snowbird, UT, USA, June 2001. ACM Press, New York, USA.

31 Phillip Heidegger, Annette Bieniusa, and Peter Thiemann. Access permission contracts for scripting languages. In John Field and Michael Hicks, editors, *Proceedings 39th Annual ACM Symposium on Principles of Programming Languages*, pages 111–122, Philadelphia, USA, January 2012. ACM Press.

32 Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In Norman K. Meyrowitz, editor, *Conference Object Oriented Programming, Systems, Languages, and Applications / European Conference on Object-Oriented Programming*, pages 169–180, Ottawa, Canada, October 1990.

33 Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In Philip Wadler and Masami Hagiya, editors, *Proceedings of the 8th International Symposium on Functional and Logic Programming FLOPS 2006*, pages 208–225, Fuji Susono, Japan, April 2006. Springer-Verlag.

34 Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. Transparent object proxies for JavaScript. In John Boyland, editor, *ECOOP*, volume ?, Prague, Czech Repulic, July 2015. LIPICS. to appear.

35 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.

36 Matthias Keil and Peter Thiemann. TreatJS: Higher-order contracts for JavaScript. Technical report, Institute for Computer Science, University of Freiburg, 2015.

37 Reto Kramer. iContract — the Java design by contract tool. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 295–307, Santa Barbara, CA, USA, 1998. IEEE Computer Society.

38 Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188, Norwell, MA, USA, 1999. Kluwer Academic Publishers.

39 Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

40 Mark S. Miller, Tom Van Cutsem, and Bill Tulloh. Distributed electronic rights in JavaScript. In Matthias Felleisen and Philippa Gardner, editors, *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 1–20, Rome, Italy, March 2013. Springer.

41 Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized JavaScript. `http://google-caja.googlecode.com`, 2008. Google White Paper.

42 Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006. AAI3245526.

43 John C. Reynolds. Gedanken - a simple typeless language based on the principle of completeness and the reference concept. *Commun. ACM*, 13(5):308–319, 1970.

44 T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: Run-time support for reasonable interposition. In Gary T. Leavens and Matthew B. Dwyer, editors, *OOPSLA*, pages 943–962. ACM, 2012.

45 Sam Tobin-Hochstadt and David Van Horn. Higher-order symbolic execution via contracts. In Gary T. Leavens and Matthew B. Dwyer, editors, *OOPSLA*, pages 537–554. ACM, 2012.

46 Jesse A. Tov and Riccardo Pucella. Stateful contracts for affine types. In Andrew D. Gordon, editor, *ESOP 2010*, volume 6012 of *Lecture Notes in Computer Science*, pages 550–569. Springer-Verlag, 2010.

47 Tom Van Cutsem and Mark S. Miller. Proxies: Design principles for robust object-oriented intercession APIs. In William D. Clinger, editor, *DLS*, pages 59–72. ACM, 2010.

48 Tom Van Cutsem and Mark S. Miller. Trustworthy proxies - virtualizing objects with invariants. In Giuseppe Castagna, editor, *ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 154–178, Montpellier, France, July 2013. Springer.

49 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Proceedings 18th European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16, York, UK, March 2009. Springer.

50 Dana N. Xu, Simon L. Peyton Jones, and Koen Claessen. Static contract checking for Haskell. In Benjamin Pierce, editor, *Proceedings 36th Annual ACM Symposium on Prin-*

ciples of Programming Languages, pages 41–52, Savannah, GA, USA, January 2009. ACM Press.